

Introducción al lenguaje VHDL

Very High Speed Integrated Circuit
Hardware Description Language
(VHSIC HDL)

Fernando Nuño García



Área de Tecnología Electrónica
Universidad de Oviedo

ÍNDICE

1.- Elementos de VHDL

2.- Unidades de diseño

- Entidad
- Arquitectura
- Configuración

3.- Métodos de diseño

- Flujo de datos
- Diseño Jerárquico
- Diseño Secuencial
 - Sentencias secuenciales
 - Asignaciones a señales y a variables
 - Sentencias de control de flujo
- Sentencia concurrente WITH-SELECT
- Sentencia concurrente WHEN-ELSE

4.- Bibliotecas y paquetes

- Etapas en el diseño con VHDL
- Bibliotecas
- Paquetes

5.- Ejemplos de Diseño

1.- Elementos de descripción en VHDL

* Bibliotecas (*Libraries*)

Almacenan los elementos de diseño: tipo de datos, operadores, componentes, objetos, funciones,...

Esos elementos se organizan en Paquetes (*Packages*): son unidades de almacenamiento de elementos y tienen que hacerse "visibles" para poder utilizarlos

Hay 2 bibliotecas que siempre son visibles por defecto: **std** (la standard) y **work** (la de trabajo) y que no es necesario declarar

* Entidades (*Entities*)

Es el modelo de **interfaz de un circuito con el exterior** mediante unos terminales de entrada y de salida

* Arquitecturas (*Architectures*)

Es la especificación del funcionamiento de una Entidad

Elementos del lenguaje VHDL

- * **Identificadores:** letras, números y _
- * **Delimitadores:** caracteres utilizados en operaciones y sentencias =>, :=, <=, ...
- * **Comentarios:** precedidos de -- y hasta fin de línea
- * **Tipos de Datos y Subtipos:** bit, enteros, reales, ...
- * **Valores Literales:** símbolos que indican un valor
'Z' (carácter), "abcde" (cadenas), B"1001001"
(binarios), X"CF3" (valores hexadecimales),...
- * **Operadores**
 - Aritméticos (+, -, *, /, **, mod, rem, abs)
 - Relacionales (=, /=, >, <, >=, <=)
 - Lógicos (and, nand, or, nor, xor, xnor, not)
- * **Objetos:** elementos para almacenar información
 - Constantes (no cambian en toda la descripción)
 - Variables (valores que pueden cambiar)
 - Señales (representan conexiones o terminales)
- * **Atributos:** características de elementos
'EVENT, 'LEFT, 'RIGHT, 'HIGH, 'LOW, ...

2.- Unidades de Diseño en VHDL y sintaxis

* Entidad(*Entity*)

- Es el bloque constructivo básico en VHDL
- Su declaración equivale a definir un elemento con sus terminales externos de conexión (ports):

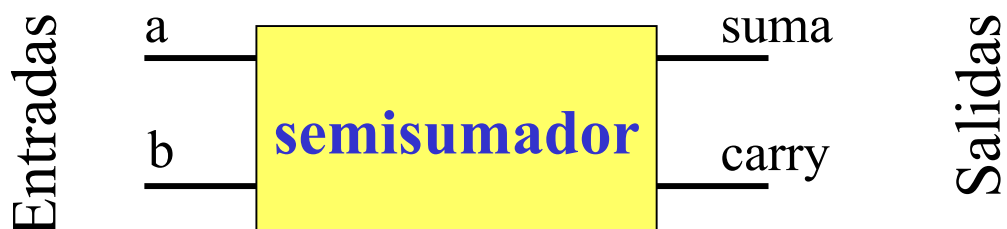
```
ENTITY nombre_entidad IS  
  PORT (port_list);  
END nombre_entidad;
```

Sintaxis

port_list ("lista de puertos"): debe definir **nombre**, **modo** (dirección) y **tipo** de cada uno de ellos

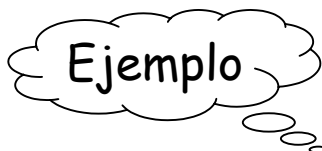
Ejemplo

```
ENTITY semisumador IS  
  PORT(a,b :    IN    BIT;  
        suma, carry : OUT  BIT);  
END semisumador;
```



Cada **puerto** actúa como una señal que es "visible" a la hora de describir el funcionamiento del elemento

- El **MODO** define si se puede sólo leer, sólo escribir o leer y escribir simultáneamente
- El **TIPO** de dato define el conjunto de valores que puede tomar una señal. Los datos pueden ser:
 - a) Escalares: sólo pueden tener un valor asociado
 - b) Matriciales (arrays): varios elementos mismo tipo
 - c) Registros (records): varios elem. y distintos tipos



Todos escalares

```
ENTITY contador IS
PORT ( clk      : IN BIT;
      reset     : IN BIT;
      q         : OUT INTEGER RANGE 0 TO 15);
END contador;
```

MODOS POSIBLES DE LOS "PORTS"

- in

Modo por defecto de los puertos
Sólo pueden ser leídos, nunca escritos

- out

Permite ser escrito pero no leído

- inout

Permite lectura y escritura

- buffer

Idéntico a inout, pero sólo admite una fuente de escritura. Sólo admite la conexión a una fuente de señal

Las señales (terminales físicos internos) de una entidad no tienen ningún tipo de problema para su conexión a puertos de una entidad o componente

Pueden ser leídas y escritas

TIPOS DE DATOS STANDARD

(en la biblioteca STD en *Package Standard*)

• Datos escalares standard:

Bit:	'0' '1'
Boolean:	FALSE TRUE
Character:	Conjunto ASCII: 'a' 'b'...
Integer:	Enteros: 137 -MAXINT...MAXINT
Real:	Reales: 37.4 -MAXREAL...MAXREAL
Time:	Tiempo: 100ps, 25ns 0...MAXTIME fs Unidades: (ps, ns, us, ms, sec, min,hr)

• Datos compuestos standard:

Bit_vector: vector de bits. P.e. "0010"

String: tira de caracteres. P.e. "Hola"

Tipos de Datos Enumerados

- Son tipos de datos escalares definidos por el usuario, pueden definirse con caracteres o con unos nombres literales elegidos a conveniencia

Sintaxis de la declaración:

```
TYPE nombre_tipo IS definición_de_tipo;
```

Ejemplo de declaración para señales:

```
ARCHITECTURE rtl OF fsm IS
    TYPE estado IS (reset,libre,ocupado);
    SIGNAL anterior, siguiente: estado;
BEGIN
    --Sentencias de la arquitectura
    ....
END rtl;
```

Para asignar a los puertos de una entidad tipos enumerados, se deben haber declarado éstos en un "paquete" (**package**) y utilizar la biblioteca correspondiente.

Subtipos de Datos

Para los datos escalares, es posible definir un **rango restringido** de datos de un tipo determinado definido con anterioridad: los subtipos

Los subtipos pueden definirse con una declaración específica:

```
SUBTYPE nombre_subtipo IS tipo_base  
    RANGE rango_valores;
```

o bien a la hora de definir el tipo de una determinada señal o puerto:

```
puerto : MODO tipo_base RANGE rango_valores;
```

```
señal : tipo_base RANGE rango_valores;
```

Ejemplos de subtipos o rangos reducidos:

```
SUBTYPE valor_bus IS INTEGER RANGE 0 TO 255;  
SUBTYPE mi_logica IS STD_ULOGIC RANGE 'x' TO 'z';
```

```
.....  
PORT ( Qa : OUT INTEGER RANGE 0 TO 7);
```

```
.....
```

Matrices (Arrays) (I)

- Son tipos de datos compuestos, cada array es una colección de datos del **mismo tipo**
- El **rango** se define cuando se declara el array. Se pueden declarar índices enteros cualesquiera, ascendentes con **TO** o descendentes con **DOWNTO**

Sintaxis:

TYPE nombre **IS ARRAY** (rango) **OF** tipo_elementos;

Ejemplo:

Declaraciones de tipo

```
TYPE nibble IS ARRAY (3 DOWNTO 0) OF BIT;  
TYPE ram IS ARRAY (0 TO 31) OF INTEGER  
RANGE 0 TO 255;
```

```
SIGNAL a_bus : nibble;  
SIGNAL ram_0 : ram;
```

Declaraciones de señales

Los arrays con elementos de tipo carácter como **bit_vector** o **string** admiten asignaciones completas al vector entre comillas dobles: " "

```
a_bus <= "0000";  
loc_bus := "10101010";
```

Matrices (Arrays) (II)

- Bit_vector y string son los 2 tipos de arrays definidos de manera standard que no es necesario declarar

Ejemplos de declaración de señales:

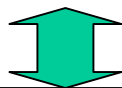
```
SIGNAL z_bus : BIT_VECTOR(3 DOWNT0 0);  
SIGNAL c_bus : BIT_VECTOR(1 TO 4);
```

- Los arrays también pueden ser de 2 ó más dimensiones

```
TYPE t_2d IS ARRAY (3 downto 0, 1 downto 0) OF integer;  
SIGNAL x_2d : t_2d;  
.....  
x_2d(3,1) <= 4;
```

- Se pueden realizar asignaciones entre arrays si son del mismo tipo de datos y del mismo tamaño.
Se mantiene el orden definido en el rango:

```
z_bus <= c_bus;
```



Asignaciones equivalentes:

```
z_bus(3) <= c_bus(1);  
z_bus(2) <= c_bus(2);  
z_bus(1) <= c_bus(3);  
z_bus(0) <= c_bus(4);
```

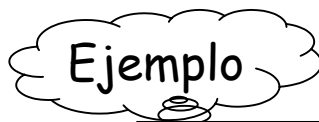
por posición
no por índice

* Arquitectura (*Architecture*)

• Es la segunda unidad de diseño y define la funcionalidad o comportamiento de una determinada **Entidad**

Sintaxis

```
ARCHITECTURE nombre_arq OF nombre_ent IS
    declaraciones
BEGIN
    sentencias concurrentes
END nombre_arq;
```



Nombre de entidad

```
ARCHITECTURE ex1 OF conc IS
    SIGNAL z,a,b,c,d : BIT;
BEGIN
    d <= a OR b;
    z <= c OR d;
END ex1;
```

Declaración de señales

Asignaciones concurrentes

Las sentencias son las que describen cómo se generan las señales externas (ports) de salida de la entidad en función de las entradas: diseño de la entidad

Las **Declaraciones** en una Arquitectura anticipan términos que van a aparecer, éstos pueden ser:

Objetos (**Constantes**, **Variables**, **Señales**), Componentes, Tipos, Subtipos, Atributos, Funciones, Procedimientos,...

Las **Señales** son objetos que representan terminales físicos. Son "locales" a la arquitectura, no son entradas ni salidas de la entidad que se está describiendo.

No es necesario declarar como señales los puertos (ports) de la Entidad, se pueden manejar por igual dentro de la arquitectura

Sintaxis

SIGNAL nombre : tipo;

o bien

SIGNAL nombre : tipo := valor_inicial;

Si no se les asigna un valor inicial, toman el "más a la izquierda" de sus valores posibles (primero en el orden de valores).

Ejemplo

```
SIGNAL i : integer range 0 to 3;  
-- i se inicializará con 0
```

En general: ¿Qué son los Objetos?

Sintaxis declaración

“Objeto” identificadores: tipo [:=expresión];

• Constantes (*Constant*)

- Almacena un único valor durante toda la descripción
 - Deben inicializarse al ser declaradas (exc.diferidas)
- Ej.: `CONSTANT V37: bit_vector(7 downto 0):="0100011";`

• Variables (*Variable*)

- Almacenan valores que pueden cambiar en la descrip.
 - Sólo pueden declararse y utilizarse dentro de procesos o subprogramas
 - Se les puede asignar un valor inicial, si no se hace así, el descriptor le hace una asignación al menor valor
 - La asignación se realiza con **:=**
- Ej.: `VARIABLE Estado: bit:=0; --definición`
 `Estado:= 1; --asignación`

• Señales (*Signal*)

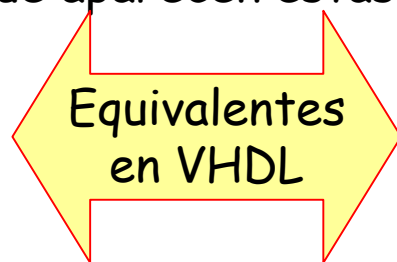
- También almacenan valores que pueden cambiar
- Representan las **conexiones** o **terminales físicos** presentes en un circuito
- Se declaran en sentencias concurrentes y pueden aparecer también en procesos
- La asignación se realiza con **<=**

Ej.: `SIGNAL b_datos: bit_vector(15 downto 0);--definición`
 `b_datos<=X"3FB4"; --asignación`

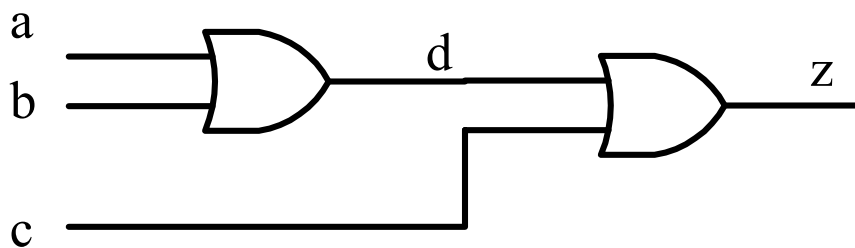
Sentencias concurrentes: se ejecutan en "paralelo", de manera simultánea y no importa el orden en que se escriban dentro de la arquitectura

Asignaciones concurrentes: el resultado no depende del orden en que aparecen éstas

$d \leq a \text{ OR } b;$
 $z \leq c \text{ OR } d;$



$z \leq c \text{ OR } d;$
 $d \leq a \text{ OR } b;$



VHDL: ventajas del *software* pero modelando de manera correcta el *hardware*

Los programas se ejecutan de manera secuencial, un circuito físico funciona concurrentemente:

Todos los elementos están activos de manera simultánea e interaccionan entre sí a lo largo del tiempo

VHDL también admite descripciones en las que el código se ejecuta según un algoritmo secuencial (**PROCESOS**)

Operadores (I)

Operadores lógicos

- Están predefinidos para los tipos de datos:

bit, bit_vector, boolean, std_logic,
vectores de booleanos y vectores de std_logic

- Son los siguientes:

and, or, nand, nor, xor y not

y devuelven un dato del mismo tipo

Operadores relacionales

- Igualdad (=) y desigualdad (/=) están definidos para todos los tipos y devuelven un valor booleano

- El resto, están definidos para todos los tipos escalares y para arrays unidimensionales:

> mayor	< menor
>= mayor o igual	<= menor o igual

Operadores (II)

- Se pueden comparar arrays de diferentes tamaños, se alinean por la izquierda y se comparan posiciones

```
ARR1 <= "0011";  
ARR2 <= "01";  
-- Comparando (ARR1 < ARR2) devolverá TRUE
```

- El operador **&** se emplea para concatenar (unir) arrays o añadir nuevos elementos a un array:

```
Z_BUS(1 downto 0) <= '0' & B_BIT;  
BYTE <= A_BUS & B_BUS;
```

Operadores aritméticos

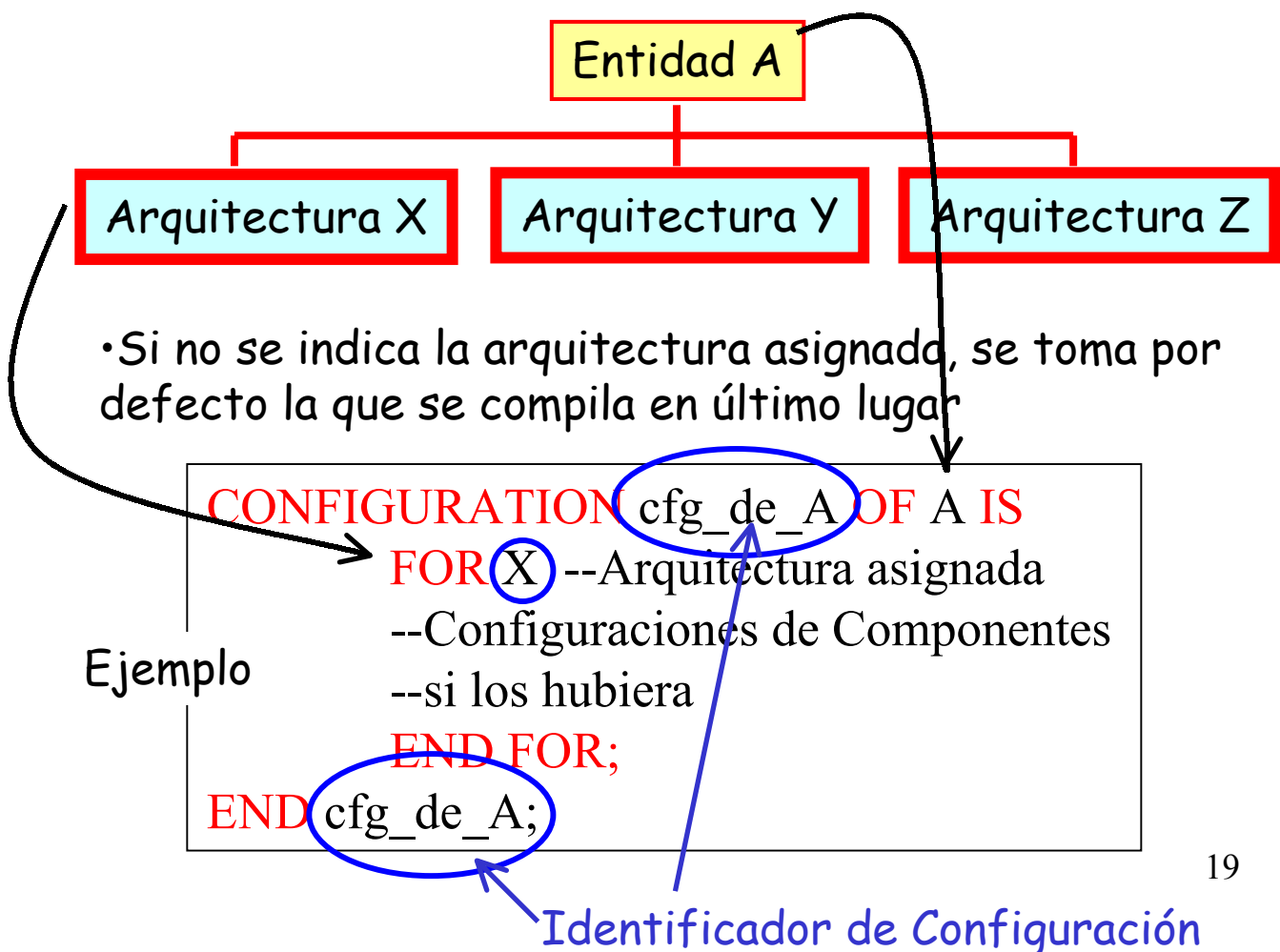
- Suma (+), resta (-), multiplicación (*) y división (/) están definidos para **enteros** y **reales**. Los **2 operandos han de ser del mismo tipo** y el resultado también

- Otros operadores numéricos:

abs	(valor absoluto)
mod	(módulo)
**	(potenciación)
rem	(resto de división entera)

* Configuraciones(*Configuration*)

- Una entidad puede tener varias arquitecturas posibles aunque lo más usual es que aparezca sólo una en la descrip.
- Una **Configuración** indica qué arquitectura debe utilizarse para sintetizar una entidad en caso de que haya varias posibles en el fichero de descripción
- También pueden indicar qué componentes de una arquitectura se asocian con otras entidades externas



3.- Métodos de Diseño en VHDL

Diseño
de Entidades:

Flujo de datos

Jerárquico

Secuencial

- Flujo de datos:

Se expresa el comportamiento de las señales de salida a partir de las señales de entrada mediante asignaciones concurrentes

- Jerárquico:

Descripción estructural (*structural modeling*) en la que se descompone en los componentes del sistema y se indican sus interconexiones

- Secuencial:

Se emplean sentencias secuenciales y no concurrentes: PROCESOS. Se ejecutan en un orden determinado, se finaliza la ejecución de una sentencia antes de pasar a la siguiente.

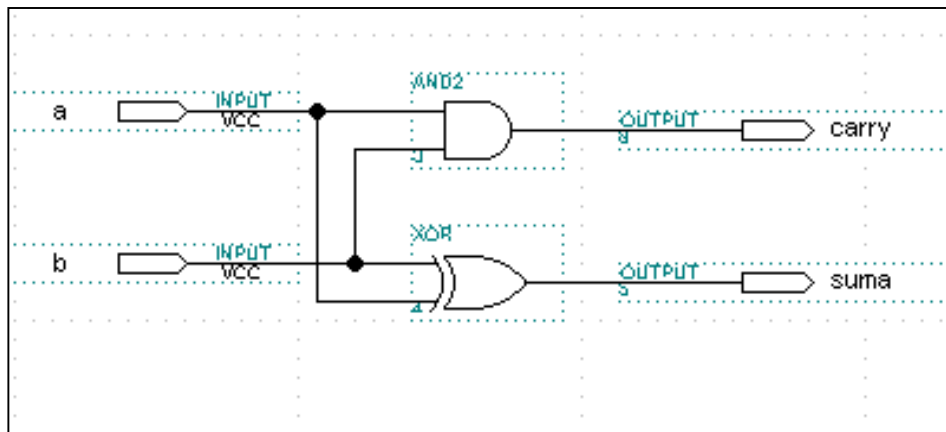
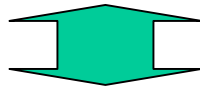
Diseño por Flujo de datos

Se emplean asignaciones para las señales con expresiones empleando operadores aritméticos y lógicos

Ejemplo

```
ENTITY semisumador IS
PORT ( a,b : IN BIT ;
      suma,carry: OUT BIT);
END semisumador;

ARCHITECTURE semi OF semisumador IS
BEGIN
    suma <= a XOR b;
    carry <= a AND b;
END semi;
```



Este método se aconseja para entidades simples

Diseño Jerárquico

- Se descompone la entidad en su estructura de elementos más simples: **COMPONENTES**
- Los componentes se declaran dentro de la Arquitectura, antes de la sentencia BEGIN de la misma.
- La declaración es similar a la de una **ENTIDAD**, no se define el comportamiento, sólo los terminales externos

```
COMPONENT nombre_componente  
  PORT (port_list);  
END nombre_componente;
```

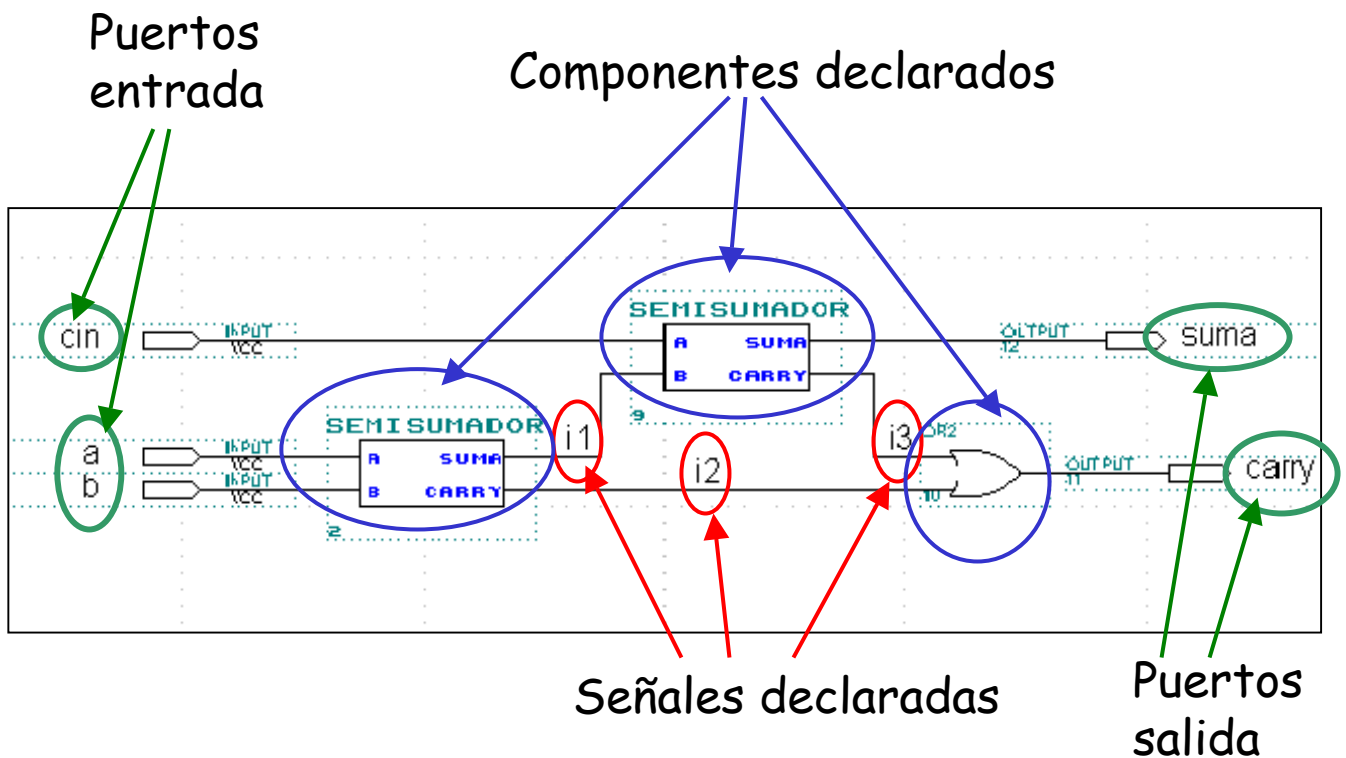
Sintaxis

- El comportamiento debe definirse fuera de la arquitectura donde se declara. Definiendo una **Entidad** de igual nombre que el **Componente** y con una arquitectura asociada al mismo
- Una vez declarados los componentes, se pueden utilizar dentro de las sentencias concurrentes de la arquitectura asociando **señales** a entradas y salidas de los mismos según el orden de declaración

```
etiqueta : nombre_componente  
  PORT MAP (lista_asociación_ports);
```

Sintaxis

Ejemplo: Diseño jerárquico de un sumador de 1 bit



```
ENTITY sumador IS
PORT( a, b, cin : IN      BIT;
      suma, carry : OUT BIT);
END sumador;
```

```
ARCHITECTURE suma OF sumador IS
```

```
SIGNAL i1, i2, i3 : BIT;
```

```
-- Declaración del componente semisumador
COMPONENT semisumador
PORT (a, b : IN BIT; suma, carry : OUT BIT);
END COMPONENT;
-- Puerta or como componente
COMPONENT puerta_or
PORT (a, b : IN BIT; z : OUT BIT);
END COMPONENT;
```

```
BEGIN
    u1 : semisumador PORT MAP (a, b, i1, i2);
    u2 : semisumador PORT MAP (i1, cin, suma, i3);
    u3 : puerta_or PORT MAP (i2, i3, carry);
END suma;
```

Señales locales

Declaración
Componentes

Arquitectura
conectando
componentes

23

sigue...

...continuación

semisumador	--Definición del componente semisumador ENTITY semisumador IS PORT (a, b : IN BIT; suma, carry : OUT BIT); END semisumador;	Declaración del componente como Entidad
	ARCHITECTURE semi OF semisumador IS BEGIN suma <= a XOR b; carry <= a AND b; END semi;	Definición de su comportamiento
puerta_or	--Definición del componente puerta_or ENTITY puerta_or IS PORT (a, b : IN BIT; z : OUT BIT); END puerta_or;	Puerta OR como entidad
	ARCHITECTURE puerta_or OF puerta_or IS BEGIN z <= a OR b; END puerta_or;	Definición de comportamiento

- La declaración de entidad para un componente y la definición de su arquitectura puede aparecer antes o después de la declaración de ese componente en un diseño jerárquico

- En el ejemplo anterior se podría haber sustituido la línea:
u3 : puerta_or PORT MAP (i2, i3, carry);
 por:
carry <= i2 OR i3; (asignación)
 sin necesidad de definir el componente puerta_or

Diseño Secuencial

- Si se emplea este procedimiento de diseño, todas las sentencias se sitúan entre la cabecera **PROCESS** y la terminación **END-PROCESS**
- Se define así una zona de código VHDL en el que las sentencias se ejecutan de **manera secuencial** (en un orden determinado) y por tanto **no concurrente**
- Los procesos se definen siempre dentro de una **arquitectura**. Pueden aparecer varios procesos en una arquitectura, **interactuando entre sí de forma concurrente**

```
etiqueta_opcional : PROCESS (lista sensibilidad opcional)
    declaraciones
BEGIN
    sentencias secuenciales
END PROCESS etiqueta_opcional;
```

Sintaxis

- Un proceso se **ejecuta**:
 - a).- Cuando cambia alguna de las señales que aparecen en **la lista de sensibilidad (EVENTO)**
 - b).- Cuando se da una condición por la que se estaba esperando (**WAIT...**)

Ejemplo

Lista de sensibilidad

```
Puerta_OR: PROCESS (a, b)
BEGIN
  IF a='1' or b='1' THEN
    z <= '1';
  ELSE
    z <= '0';
  END IF;
END PROCESS;
```

Sentencia condicional

• Cuando un proceso no presenta una lista de sensibilidad equivale a un bucle sin fin (el proceso se repite de forma continua). En ese caso, el proceso sólo se suspende cuando aparece una sentencia **WAIT**:

* Se reinicia el proceso cuando se da una condición **WAIT UNTIL** condición;

* Se reinicia cuando se produce un evento en la lista **WAIT ON** lista_ señales;

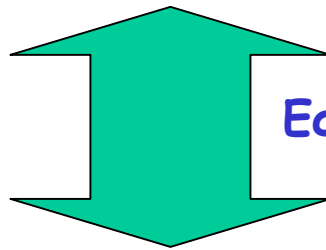
* Se reinicia una vez transcurrido el tiempo especificado **WAIT FOR** tiempo;

* El proceso se suspende indefinidamente **WAIT**;

¡ SI EL PROCESO TIENE **LISTA DE SENSIBILIDAD** NO PUEDE TENER SENTENCIAS **WAIT** !

Ejemplo con sentencia WAIT

```
PROCESS
BEGIN
  IF (hora_alarma = hora_actual) THEN
    sonido_alarma <= '1';
  ELSE
    sonido_alarma <= '0';
  END IF;
  WAIT ON hora_alarma, hora_actual;
END PROCESS;
```



Equivalentes

Ejemplo con lista de sensibilidad

```
PROCESS (hora_alarma, hora_actual)
BEGIN
  IF (hora_alarma = hora_actual) THEN
    sonido_alarma <= '1';
  ELSE
    sonido_alarma <= '0';
  END IF;
END PROCESS;
```

Tipos de Sentencias **secuenciales** en un PROCESO

• Sentencias de sincronización

Espera (*wait*)_

• Asignaciones secuenciales de señales y variables:

Las asignaciones son secuenciales si se encuentran dentro de un proceso. *Es diferente la asignación de variables y de señales*

Las variables se actualizan en el acto (cuando aparece la asignación dentro del proceso)

La asignación secuencial de señales sólo tiene efecto cuando se suspende (se detiene) el proceso

• Sentencias de control de flujo de ejecución

if-then-else: condicionales

for-loop: bucles de repetición

case: selección en función de valor

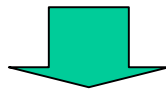
Asignaciones secuenciales a señales (I)

- Si una señal tiene asignaciones dentro de un proceso y está en la lista de sensibilidad, puede ocasionar que se reactive el proceso una vez que éste se ha suspendido.
- Un proceso puede necesitar varias ejecuciones en la fase de simulación para que todas sus señales se actualicen al valor final

Ejemplo:

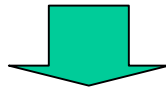
```
PROCESS (B,M)
BEGIN
  M<=B;
  Z<=M;
END PROCESS;
```

B=M=Z=0 al inicio



Evento B=1

B=M=1 & Z=0 final 1^{er} proceso



Reactivación por evento en M

B=M=Z=1 final 2^o proceso

- Las señales asignadas en un proceso "con reloj" representan **las salidas de los biestables o registros**

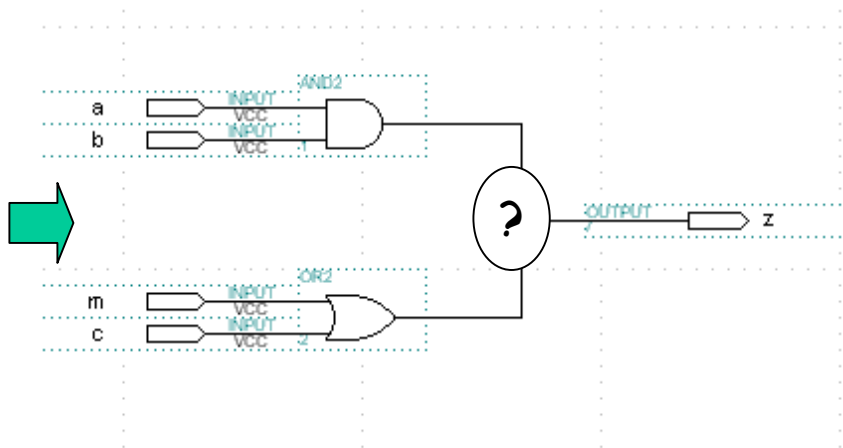
```
PROCESS
BEGIN
  WAIT UNTIL CLK = '1';
  Q<=D;
END PROCESS;
```

Asignaciones secuenciales a señales (II)

- Si hay más de una asignación secuencial dentro de un proceso, sólo tiene efecto la última que aparece
- Si existen varias asignaciones concurrentes, se da la situación de varias puertas o "drivers" intentando "manejar" eléctricamente el mismo punto. La señal debe ser del tipo Std_Logic (de la biblioteca IEEE) para que pueda existir una resolución de conflictos eléctricos.

Asignaciones concurrentes

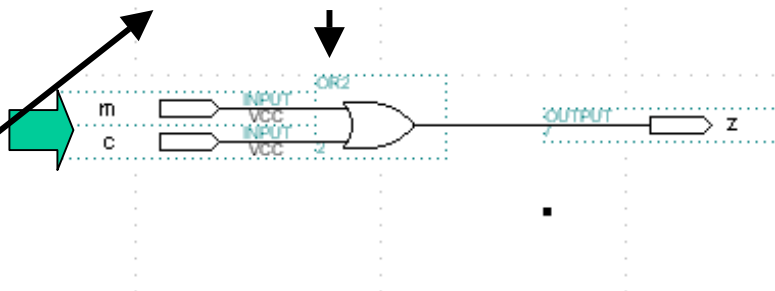
```
z<=a AND b;  
z<=m OR c;
```



Asignaciones secuenciales

```
PROCESS (b,m)  
BEGIN  
  z<=a AND b;  
  z<=m OR c;  
END PROCESS;
```

Sólo tiene efecto la última



Atributos

• Los atributos **se aplican a una señal** para obtener información específica sobre ella: señal **'ATRIBUTO'**

Atributos de función:

señal **'EVENT'**:

devuelve TRUE si la señal cambia

señal **'LAST_VALUE'**:

devuelve valor de la señal antes del último evento

señal **'DRIVING_VALUE'**:

devuelve el valor actual de la señal

Atributos de valor:

señal_escalar **'LEFT' / 'RIGHT'**:

devuelve el valor más a la izquierda / derecha

señal_escalar **'HIGH' / 'LOW'**:

devuelve el mayor / menor valor

señal_array **'LEFT' / 'RIGHT'**:

devuelve el índice izquierdo / derecho del array

señal_array **'HIGH' / 'LOW'**:

devuelve el índice superior / inferior del array

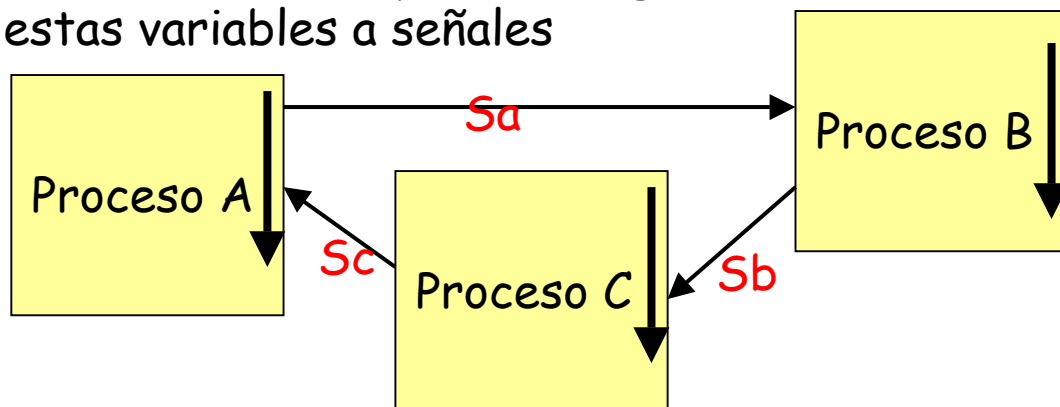
Atributos de rango:

señal_array **'RANGE'**:

devuelve el rango del array
(como se haya definido)

Variables

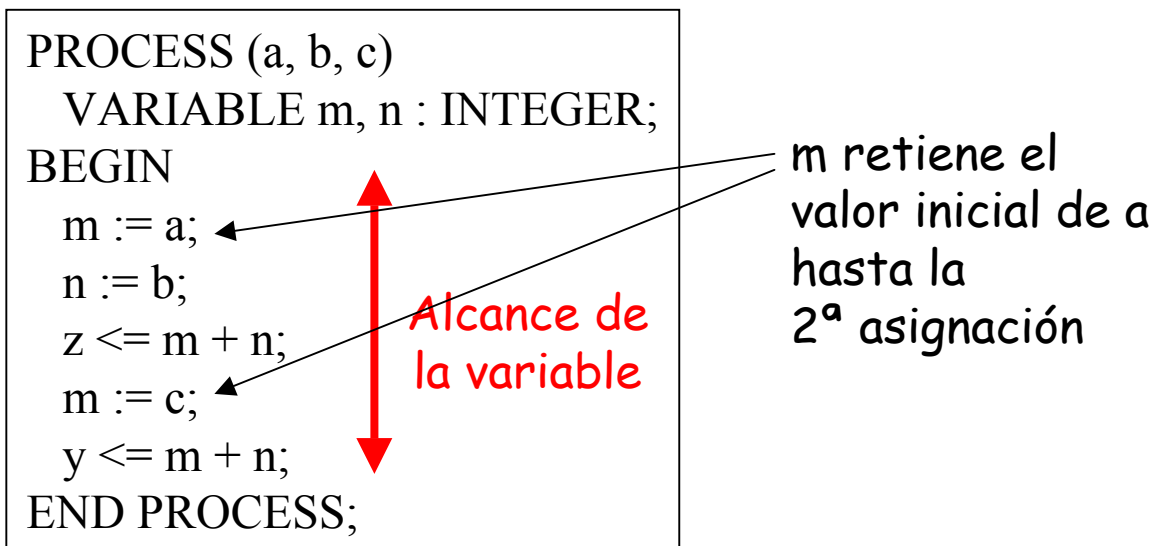
- Las variables son objetos de VHDL que sólo pueden ser declaradas y usadas dentro de un PROCESO.
- Las **variables** se distinguen de las **señales** en que su asignación **es inmediata** como en el "software clásico" (no al final de un proceso como sucede con las señales)
- Para la asignación de variables se utiliza **`:=`**, para distinguir de la asignación de señales en la que se utiliza **`<=`**
- Las variables sólo pueden usarse **dentro del PROCESO** en el que han sido declaradas
- Se pueden asignar señales a variables y variables a señales, siempre que sean del mismo tipo.
- Como las variables tienen alcance sólo dentro del Proceso en el que se declaran, para la "transferencia" entre Procesos será preciso asignar los valores finales de estas variables a señales



Variables (II)

• Las variables se emplean dentro de los procesos para almacenar valores intermedios dado que se actualizan de manera inmediata

Ejemplo:



Si los valores al entrar en el proceso son:

$a=7$ $b=8$ $c=2$

a la detención del proceso las salidas son:

$z=15$ $y=10$

se actualizan ambas señales al final del proceso pero con diferentes valores (secuenciales)

Sentencias de control de flujo de ejecución

- Sentencia **IF**:

Permite escoger qué sentencia o sentencias deben ejecutarse en función de una **CONDICIÓN**

La expresión de la condición debe dar como resultado un valor booleano

- Sentencia **FOR LOOP**:

Permite ejecutar un grupo de sentencias un **NÚMERO DE VECES** fijo

- Sentencia **CASE**:

Permite escoger qué grupo de sentencias deben ejecutarse en función del rango de valores de una **EXPRESIÓN**

Sentencia condicional **IF** (I)

Permite ejecutar un grupo de sentencias secuenciales si se da una determinada condición.

Sintaxis posibles:

- 1.-

```
IF condición_1 THEN
    Sentencias secuenciales
END IF;
```

 Si se da la condición
- 2.-

```
IF condición_1 THEN
    Grupo 1 de sentencias secuenciales
ELSE
    Grupo 2 de sentencias secuenciales
END IF;
```

 Si se da la condición
y si no se da
- 3.-

```
IF condición_1 THEN
    Grupo 1 de sentencias secuenciales
ELSIF condición_2 THEN
    Grupo 2 de sentencias secuenciales
.....
ELSIF condición_n THEN
    Grupo n de sentencias secuenciales
END IF;
```

 Si se da alguna condición se ejecuta su grupo

Sentencia condicional **IF** (II)

Si hay varias condiciones ciertas, la primera que se cumple marca el resultado final.

Una condición es cualquier expresión booleana.

Ejemplo para biestable Latch:

```
PROCESS (en, d)
BEGIN
  IF (en = '1') THEN q <= d;
  END IF;
END PROCESS;
```

En la expresión booleana puede aparecer la propia condición de evento en una determinada señal, con la sintaxis: **señal'EVENT**

Ejemplo de biestable D activo por flanco de subida:

```
PROCESS (clock, d)
BEGIN
  IF (clock'EVENT AND clock='1') THEN
    q <= d;
  END IF;
END PROCESS;
```

Hubo cambio en el reloj y está a 1

Sentencia de repetición **FOR LOOP**

Se pueden repetir las sentencias un número fijo de veces, empleando un índice que va tomando los valores del rango especificado

Sintaxis:

```
FOR parámetro IN rango LOOP  
    sentencias secuenciales  
END LOOP;
```

El parámetro debe ser de tipo discreto.

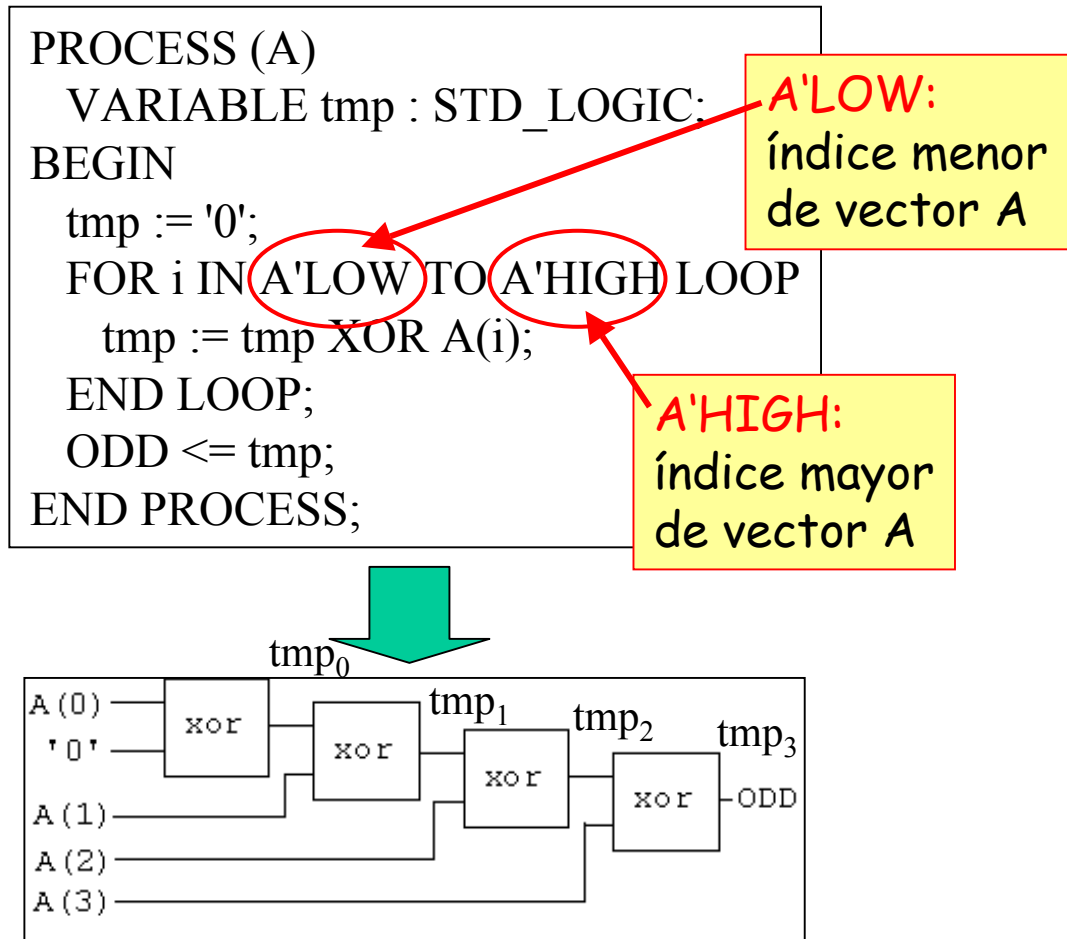
No es necesario declarar el parámetro, no se puede modificar dentro del bucle

Ejemplo de bucle de repetición

```
PROCESS (a)  
BEGIN  
    z <= "0000";  
    FOR i IN 0 TO 3 LOOP  
        IF (a = i) THEN  
            z(i) <= '1';  
        END IF;  
    END LOOP;  
END PROCESS;
```

Ejemplo:

Variable tmp para proceso que detecta paridad impar



La señal ODD saldrá con el valor '1' si el número total de "unos" existentes en el vector A(de 4 bits) es impar y tendrá el valor '0' al acabar el proceso si el número total de "unos" es par

Las variables declaradas en un proceso no pierden su valor al finalizar el mismo, sino que lo mantienen hasta que se vuelve a ejecutar

Sentencia de selección **CASE** (I)

Permite escoger entre grupos de sentencias secuenciales en función de la evaluación de una determinada expresión.

La expresión puede ser de tipo discreto o tipo vector de una dimensión.

Sintaxis:

```
CASE expresión IS
  WHEN valor_1 =>
    sentencias secuenciales
  WHEN valor_2 =>
    sentencias secuenciales
  .....
  WHEN valor_n =>
    sentencias secuenciales
END CASE;
```

Los valores de selección no pueden solaparse: aparecer en dos grupos distintos de manera simultánea

En los valores se puede especificar un **rango** de variación. Se debe realizar la asignación para todos los valores posibles de la expresión, pudiendo agruparse al final el "resto" de posibilidades con la palabra clave: "**OTHERS**"

Sentencia de selección **CASE** (II)

Ejemplo correcto:

```
CASE int IS
  WHEN 0      => z <= a;
  WHEN 1 TO 3 => z <= b;
  WHEN 4|6|8  => z <= c;
  WHEN OTHERS => z <= 'x';
END CASE;
```

Unión de valores

Rango

Ejemplo erróneo:

```
CASE int IS
  WHEN 0      => z <= a;
  WHEN 1 TO 3 => z <= b;
  WHEN 2|6|8  => z <= c;
  WHEN OTHERS => z <= 'x';
END CASE;
```

Cuando int=2 aparece solapamiento

Ejemplo con vector de bits

```
CASE sel IS
  WHEN "01" => z <= a;
  WHEN "10" => z <= b;
  WHEN OTHERS => z <= 'x';
END CASE;
```

Con vectores, no es válido el uso de rangos

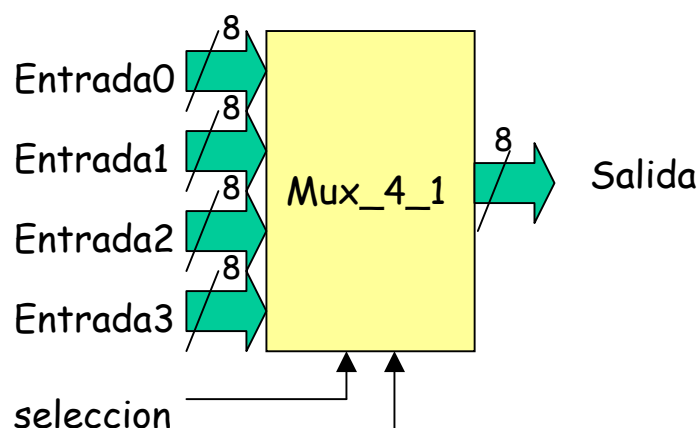
Sel es una señal de tipo
bit_vector(0 to 1)

Con la sentencia case y condicionales internos, se pueden realizar máquinas de estados finitos: la expresión identifica el estado y dentro de las sentencias secuenciales se puede cambiar éste

Ejemplo completo: multiplexor 4 a 1 (8 líneas/entrada)

```
ENTITY mux_4_1 IS
  PORT(
    seleccion      : IN  BIT_VECTOR (1 downto 0);
    Entrada0, Entrada1, Entrada2, Entrada3 : IN  BIT_VECTOR (0 to 7);
    Salida : OUT  BIT_VECTOR(0 to 7));
END mux_4_1;

ARCHITECTURE mux_4_1 OF mux_4_1 IS
  BEGIN
    PROCESS (seleccion, Entrada0, Entrada1, Entrada2, Entrada3)
      BEGIN
        CASE seleccion IS
          WHEN "00" =>
            Salida <= Entrada0;
          WHEN "01" =>
            Salida <= Entrada1;
          WHEN "10" =>
            Salida <= Entrada2;
          WHEN "11" =>
            Salida <= Entrada3;
        END CASE;
      END PROCESS;
    END mux_4_1;
```



Sentencia concurrente **WITH** (I)

Es una asignación por coincidencia entre un valor y una expresión a evaluar

Sintaxis:

```
WITH expresión SELECT  
    señal <= expresión_1 WHEN valor_1 ,  
        expresión_2 WHEN valor_2 ,  
        .....  
        expresión_n WHEN valor_n;
```

Si la expresión toma el valor_1 se hace la primera asignación, si el valor_2 la segunda y así sucesivamente

Esta sentencia es muy similar a la sentencia *CASE*, con un matiz de diferenciación importante:

- la sentencia **WITH** es concurrente
- la sentencia **CASE** es secuencial (en proceso)

Las condiciones de selección y los valores posibles tiene normas idénticas a las indicadas para la "CASE":

Se pueden especificar rangos y uniones

No puede haber "solape" en los valores

Se puede agrupar el resto: **OTHERS**


Sentencia concurrente **WITH (II)**

Ejemplo

```
WITH int SELECT
  z <= a WHEN 0,
    b WHEN 1 TO 3,
    c WHEN 4|6|8,
    d WHEN OTHERS;
```

Esta sentencia se puede emplear para síntesis de tablas de verdad de circuitos combinacionales.

```
ENTITY bcd_7seg IS
  PORT( codigo      : IN      BIT_VECTOR (3 downto 0);
        segmentos   : OUT     BIT_VECTOR (7 downto 0));
END bcd_7seg;
--Displays de ánodo común se activan con nivel bajo
--Orden de Segmentos de 7 a 0:  a-b-c-d-e-f-g-dp
ARCHITECTURE convertidor OF bcd_7seg IS
BEGIN
  WITH codigo SELECT
    segmentos <="00000011" WHEN "0000",
              "10011111" WHEN "0001",
              "00100101" WHEN "0010",
              "00001101" WHEN "0011",
              "10011001" WHEN "0100",
              "01001001" WHEN "0101",
              "01000001" WHEN "0110",
              "00011111" WHEN "0111",
              "00000001" WHEN "1000",
              "00011001" WHEN "1001",
              "11111111" WHEN OTHERS;
END convertidor;
```



Ejemplo
convertidor
BCD-7segmentos

Sentencia concurrente **WHEN-ELSE**

También es una asignación concurrente condicional similar a **WITH**

Sintaxis:

```
señal <= valor_1 WHEN condición_1 ELSE  
        valor_2 [WHEN condición_2][ELSE  
        ..... ] [ELSE  
        valor_n WHEN condición_n ]  
        [ELSE UNAFFECTED];
```

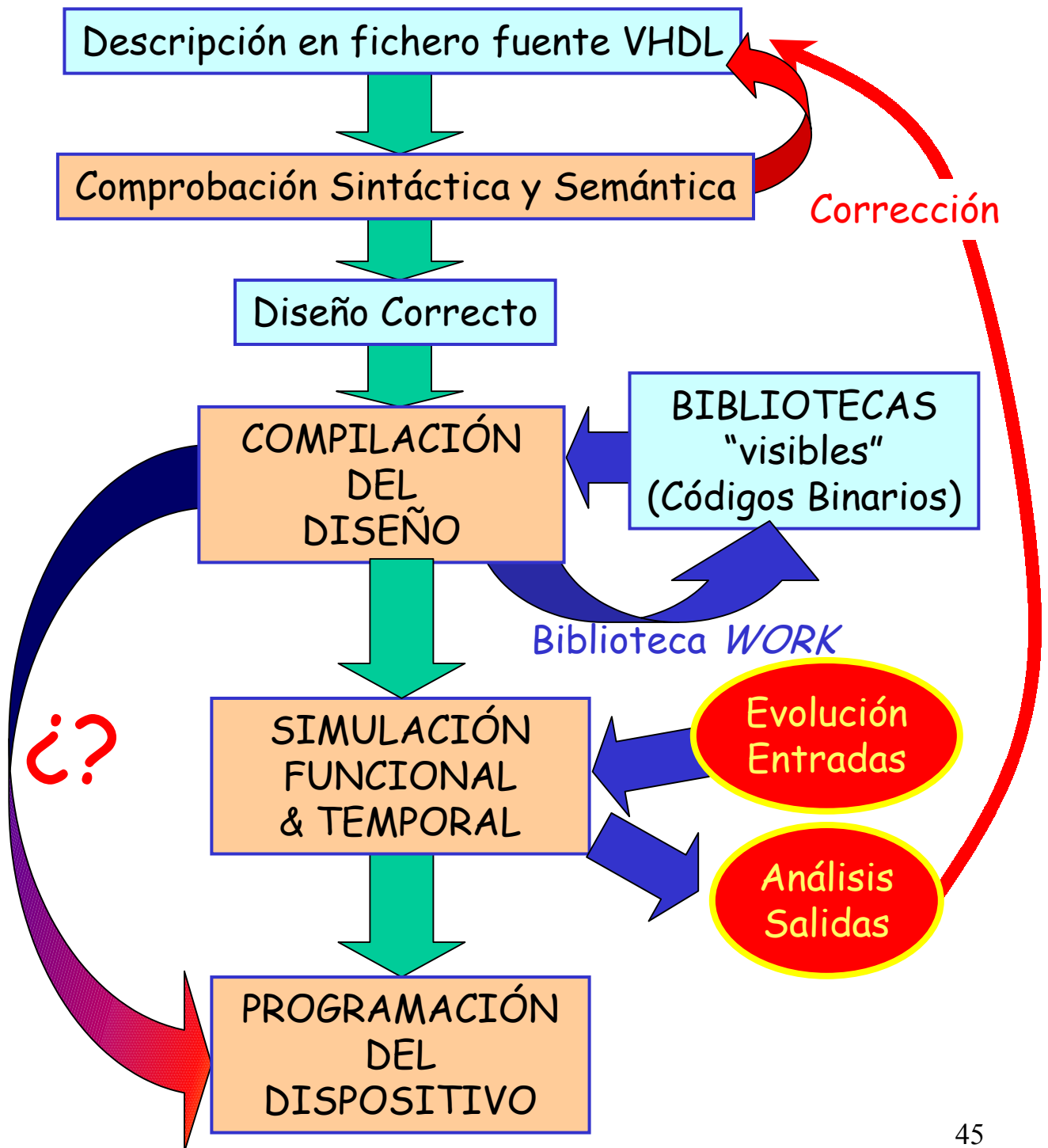
La señal recibe una asignación u otra en función de las condiciones tras **WHEN**. Si no hay cambio en la señal se puede utilizar la palabra clave **UNAFFECTD**.

Ejemplo

```
ENTITY sumador_2bits IS  
  PORT(  A,B: IN BIT;  
         SUMA_CARRY: OUT BIT_VECTOR(1 DOWNT0 0));  
END sumador_2bits;  
ARCHITECTURE sumar OF sumador_2bits IS  
BEGIN  
  SUMA_CARRY <= "00" WHEN (A='0' AND B='0') ELSE  
                "10" WHEN (A='0' AND B='1') ELSE  
                "10" WHEN (A='1' AND B='0') ELSE  
                "01" WHEN (A='1' AND B='1');  
END sumar;
```

4.- Bibliotecas y paquetes

Etapas en el Diseño con VHDL:



SIMULACIÓN DEL DISEÑO

- No es un paso obligado para grabar un dispositivo pero sí conveniente para verificar el diseño y reducir tiempos de desarrollo
- Se trata de imponer unas evoluciones a las entradas de la entidad principal (estímulos) y obtener la evolución que seguirían las salidas con el diseño descrito
- Se puede realizar mediante:

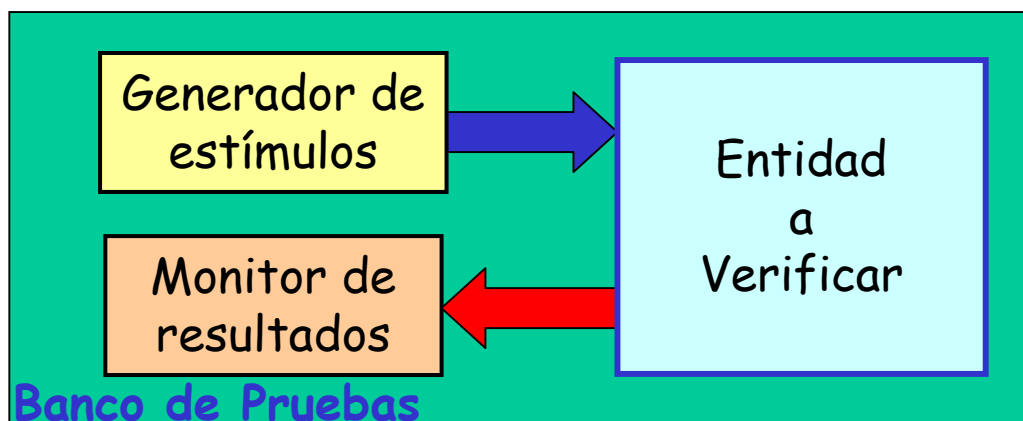
☀ a) Un editor de formas de onda (*waveform editor*)

Fichero de evolución de entradas

b) Banco de pruebas en VHDL (*test bench*)

Son entidades de diseño que engloban:

- Entidad a testear
- Generador de estímulos (en VHDL, tablas, ficheros)
- Monitor de resultados (tabla o fichero)



BIBLIOTECAS

- Las **Bibliotecas** son directorios de **ficheros compilados** (en binario) que contienen componentes, funciones, tipos de datos, objetos,...

- Tipos de Bibliotecas:

- **Biblioteca de trabajo** (work es el nombre lógico) que almacena las unidades que modelan el diseño actual

- **Bibliotecas de recursos** que contienen elementos frecuentes para facilitar la compartición y utilización del mismo código en varios diseños.

Se pueden distinguir aquí dos subtipos:

- ***B.normalizadas** (STD, IEEE, ...) y

- ***B.definidas por el usuario** (diseños compilados)

- En la biblioteca work (fichero binario) se compilan y almacenan por separado todas las **unidades de diseño del modelo** y las **externas que se referencian**

- Unidades de Diseño:

- Entidad

- Arquitectura

- Configuración

- Paquete

- Cuerpo de Paquete



Unidades primarias

Unidades secundarias

- Todo modelo o diseño necesita al menos de dos unidades: una entidad y una arquitectura
- Toda unidad primaria debe compilarse antes que sus correspondientes unidades secundarias
- Una unidad primaria referenciada dentro de otra unidad debe ser compilada antes que esta última
- Los elementos de las bibliotecas de recursos se suelen agrupar en unidades denominadas **paquetes** ("packages")
- Dentro de un mismo paquete pueden existir **elementos diversos** como: tipos de datos, componentes, funciones, etc.
- Para **poder utilizar los elementos** presentes en un paquete (hacerlos visibles al diseño) se utilizan las sentencias:

Sintaxis:

LIBRARY nombre_biblioteca;

USE nombre_biblioteca.nombre_paquete.nombre_elemento;

Si se quiere tener acceso a todos los elementos del paquete en nombre_elemento se pone la palabra reservada **all**

LIBRARY STD,WORK;
USE STD.standard.all;

Sentencias existentes siempre por defecto en todos los diseños

PAQUETES

- Son unidades de diseño en las que se declaran y describen tipos, objetos, funciones y componentes
- Los paquetes se pueden descomponer en dos unidades de diseño:
 - a) Declaración del paquete (visibilidad externa)

```
PACKAGE identificador IS
    --Declaración de
    --tipos, subtipos,
    --subprogramas, componentes, funciones
END [PACKAGE] identificador;
```

- b) Cuerpo del paquete (implementación del mismo)

```
PACKAGE BODY identificador IS
    --Definición de
    --tipos, subtipos,
    --subprogramas, componentes, funciones
END [PACKAGE BODY] identificador;
```

Ejemplo de paquete definido en el propio diseño, corresponde a un tipo de datos definido por el usuario, también se debe hacer "visible"

```
PACKAGE comidas_pkg IS
  -- Declaración de Tipo enumerado en un "package"
  TYPE sesion IS (desayuno, comida, merienda, cena);
END comidas_pkg;

-- Uso de la Librería de trabajo
USE work.comidas_pkg.all;

-- Uso del tipo enumerado definido
ENTITY comidas IS
  PORT( anterior    : IN    sesion;
        siguiente   : OUT   sesion);
END comidas;

ARCHITECTURE comidas OF comidas IS
BEGIN
  WITH anterior SELECT
    siguiente <= comida WHEN desayuno,
                  merienda WHEN comida,
                  cena    WHEN merienda,
                  desayuno WHEN cena;
END comidas;
```

5.- Ejemplos de diseño

Estructura típica de un fichero de descripción en VHDL

```
library < nombre_biblioteca >;  
use < nombre_biblioteca.nombre_paquete.nombre_elemento >;  
  
entity < nombre_entidad > is  
    port( -- listado de puertos, modo y tipo );  
end < nombre_entidad > ;  
  
architecture < nombre_arq. > of < nombre_entidad > is  
    -- Declaración de señales internas  
    -- Declaración de tipos de datos de usuario  
    -- Declaración de componentes  
  
begin--Cuerpo de la arquitectura  
    --Asignaciones concurrentes  
    --Conexionado de componentes  
    --Procesos  
end < nombre_arq. >;
```

En MAX+PLUS II debe aparecer,
en la descripción VHDL
una Entidad con el mismo nombre
que el fichero del diseño

DESCRIPCIÓN DE CIRCUITOS COMBINACIONES

•Con asignaciones concurrentes simples y condicionales

$Y \leftarrow A \text{ and } B$; --asignación simple

$Z \leftarrow B \text{ when } S='1' \text{ else } A$; -- condicional

Hay que evitar la "realimentación combinacional"
porque representa "memoria asíncrona":

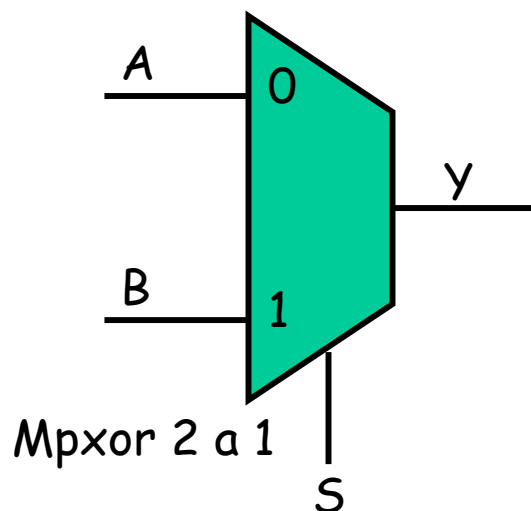
$Y \leftarrow Y \text{ nand } X$;

•Con procesos

-Las señales leídas dentro de un proceso deben aparecer en su lista de sensibilidad

-Si a una señal se le hace una asignación condicional hay que asegurar que se ha definido el valor para todas las condiciones

```
PROCESS(A,B,S)
BEGIN
  IF (S='1') THEN
    Y<=B;
  ELSE
    Y<=A;
  END IF;
END PROCESS;
```



DESCRIPCIÓN DE CIRCUITOS SECUENCIALES

• Para biestables y registros: usaremos procesos en los que la señal de reloj sea la única que actúe por flanco

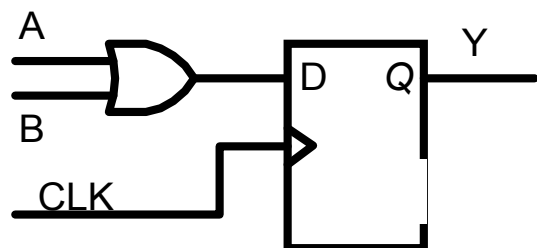
if (clk'EVENT and clk='1'); --flanco de subida

wait until clk='1'; -- también flanco de subida

• Todas las asignaciones de señales dentro de un proceso con reloj y tras un flanco equivalen a la presencia de biestables síncronos

• Un proceso con reloj puede incluir lógica combinatorial en las demas entradas de los biestables

```
PROCESS
BEGIN
    WAIT UNTIL CLK='1'
    Y<= A OR B;
END PROCESS;
```



• Si además de reloj hay entradas asíncronas, éstas deben aparecer en la lista de sensibilidad

```
PROCESS(RST, CLK)
```

Biestable D con Reset

```
BEGIN
```

```
    IF RST='1' THEN Q<='0';
```

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

```
        Q<=D;
```

```
    END IF;
```

```
END PROCESS;
```

Ejemplo 1:

Biestable j-k con reloj activo por flanco de bajada

```
--Entidad biestable j-k
ENTITY biestable_j_k IS
    PORT(j, k, clock      : IN      BIT;
          Q                : OUT    BIT);
END biestable_j_k;

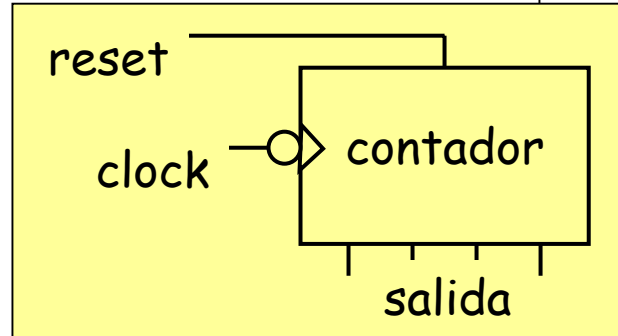
--Arquitectura de la entidad
ARCHITECTURE J_K OF biestable_j_k IS
    SIGNAL QSAL : BIT;
BEGIN
    PROCESS(clock)
    BEGIN
        IF (clock'EVENT AND clock='0') THEN
            IF (j='1' AND k='1') THEN
                QSAL <= not QSAL;
            END IF;
            IF (j='1' AND k='0') THEN
                QSAL <= '1';
            END IF;
            IF (j='0' AND k='1') THEN
                QSAL <= '0';
            END IF;
        END IF;
        Q <= QSAL;
    END PROCESS;
END J_K;
```

La condición
clock'EVENT
no sería necesaria
(es redundante)
por estar clock en
la lista de
sensibilidad
del proceso

Comentario: Hay que declarar una señal auxiliar QSAL porque el port Q se ha declarado como salida y no se podría leer no sería válido: Q<= not Q

Ejemplo 2: contador de 4 bits con reset

-- Contador de 4 bits con entrada de reloj por flanco de bajada
-- y reset asíncrono de nivel alto



-- Declaración de la entidad

ENTITY conta_rs IS

PORT(clock, reset: IN
salida : OUT

END conta_rs;

BIT;

INTEGER RANGE 0 TO 15);

-- Declaración de la arquitectura

ARCHITECTURE Contador_de_4 OF conta_rs IS

BEGIN

-- Declaración del proceso

PROCESS (clock, reset)

-- Variable local cuenta de 4 bits

VARIABLE cuenta :INTEGER RANGE 0 TO 15;

BEGIN

IF (reset='1') THEN

cuenta:=0;

ELSIF (clock'EVENT AND clock='0') THEN

cuenta:=cuenta+1;

END IF;

salida <= cuenta;

END PROCESS;

END Contador_de_4;

Ejemplo 3: contador ascendente/descendente

```
-- Contador de 4 bits con entrada de reloj (flanco de subida),
-- y señales de enable (parar o activar) y a_d para cuenta
-- ascendente o descendente

-- Declaración de la entidad
ENTITY Contador_asc_desc IS
    PORT( clock, enable, asdes : IN  BIT;
          salida : OUT  INTEGER RANGE 0 TO 15);
END Contador_asc_desc;

-- Declaración de la arquitectura
ARCHITECTURE Contador_de_4 OF Contador_asc_desc IS
BEGIN
-- Declaración del proceso
    PROCESS (clock)
        -- Variable local cuenta de 4 bits
        VARIABLE cuenta :INTEGER RANGE 0 TO 15;
        BEGIN
            IF (clock'EVENT AND clock='1') THEN
                IF (enable = '1' AND asdes = '1') THEN
                    cuenta:=cuenta+1;
                ELSIF (enable = '1' AND asdes = '0') THEN
                    cuenta:=cuenta-1;
                END IF;
            END IF;
            salida <= cuenta;
        END PROCESS;
    END Contador_de_4;
```

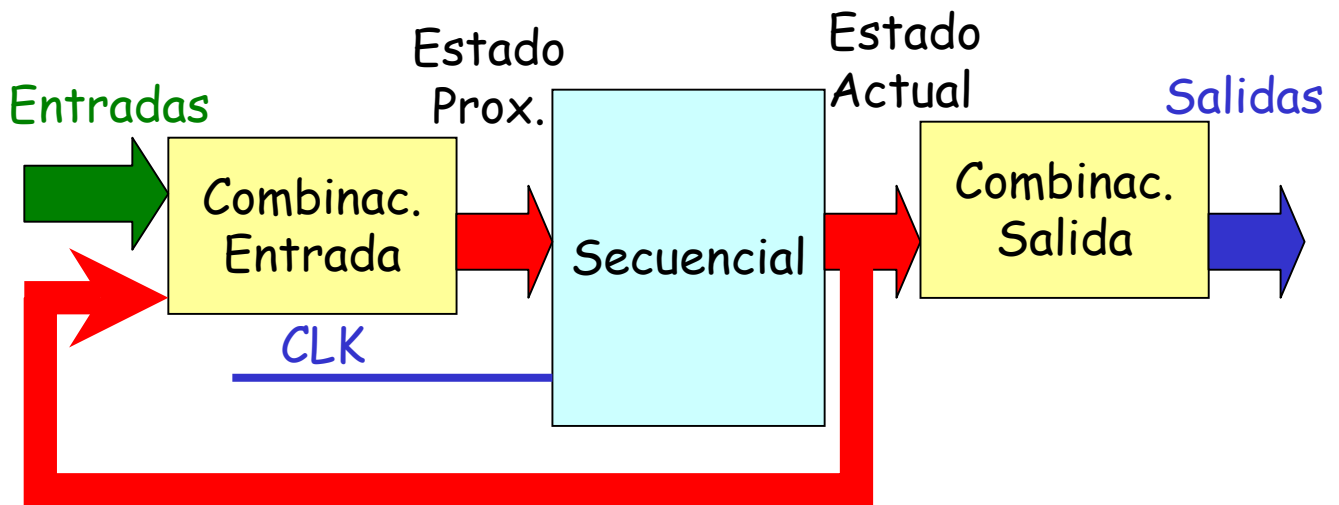

DESCRIPCIÓN DE MÁQUINAS DE ESTADO

- VHDL permite la descripción algorítmica de máquinas de estado a alto nivel
- Debe especificar únicamente el diagrama o la tabla de estados
- El diseñador se evita las tareas de generar la tabla de transiciones de estado, la codificación de estados y la tabla de excitaciones y salidas

Pasos en el diseño:

- 1.- Se parte del diagrama o de la tabla de estados y se define un tipo enumerado formado por los nombres de los estados identificados
- 2.- Se define un proceso que puede ser sensible sólo a la señal de reloj si se trata de una máquina de Moore o a la señal de reloj y a las entradas si es una máquina de Mealy
- 3.- Dentro del proceso, se determina el ESTADO siguiente en función del estado actual y de las entradas
- 4.- Finalmente se determinan las SALIDAS en función del ESTADO (máquina de Moore) o del ESTADO y las ENTRADAS (máquina de Mealy)

MÁQUINA DE MOORE



Descripción VHDL

- Combinacional de Entrada + Secuencial:

Proceso sensible al reloj y al reset de inicialización

Process(CLK, Reset_asíncrono)

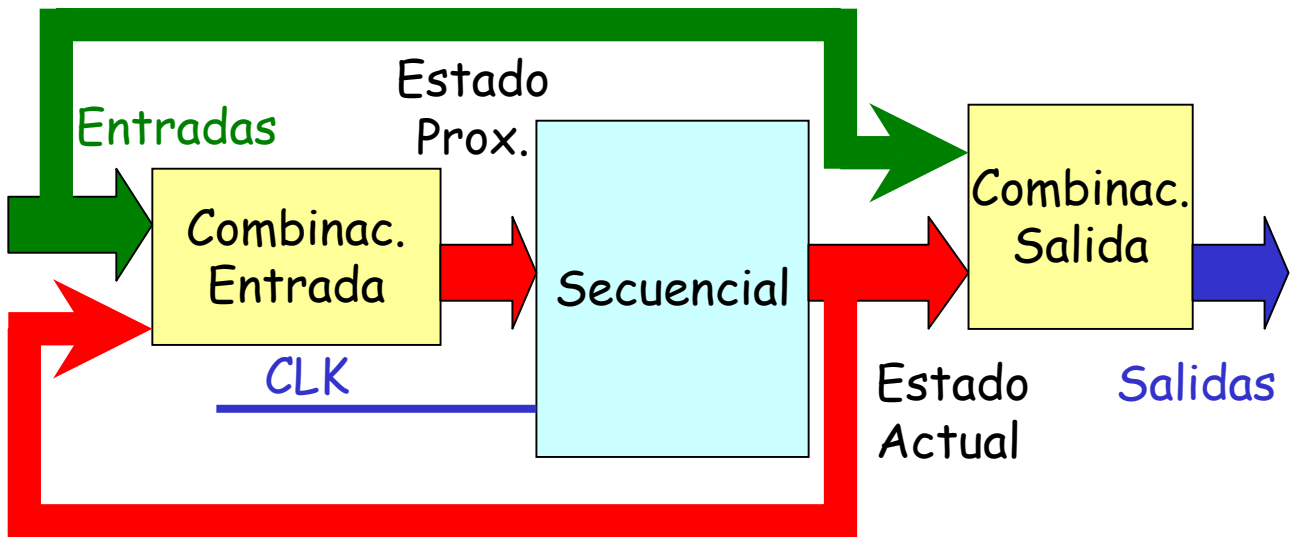
Estado Próximo <= f(Entradas, Estado actual)

- Combinacional de Salida

Asignaciones dependientes del Estado actual

Salidas <= g(Estado actual)

MÁQUINA DE MEALY



Descripción VHDL

- Combinacional de Entrada + Secuencial:

Proceso sensible al reloj y al reset de inicialización

Process(CLK, Reset_asíncrono)

Estado Próximo <= f(Entradas, Estado actual)

- Combinacional de Salida

Proceso sensible al Estado actual y a las Entradas

Process(Entradas, Estado actual)

Salidas <= g(Entradas, Estado actual)

Ejemplo: Puerta de apertura y cierre automático

Se trata de realizar la síntesis de un circuito digital que permita generar las señales de Abrir (S1) y Cerrar (S2) del mecanismo de una Puerta Automática.

Como entradas se dispone de 3 señales digitales:

P: presencia de persona en la plataforma de acceso (a 1)

C: puerta totalmente cerrada (si está a 1)

A: puerta totalmente abierta (si está a 1)

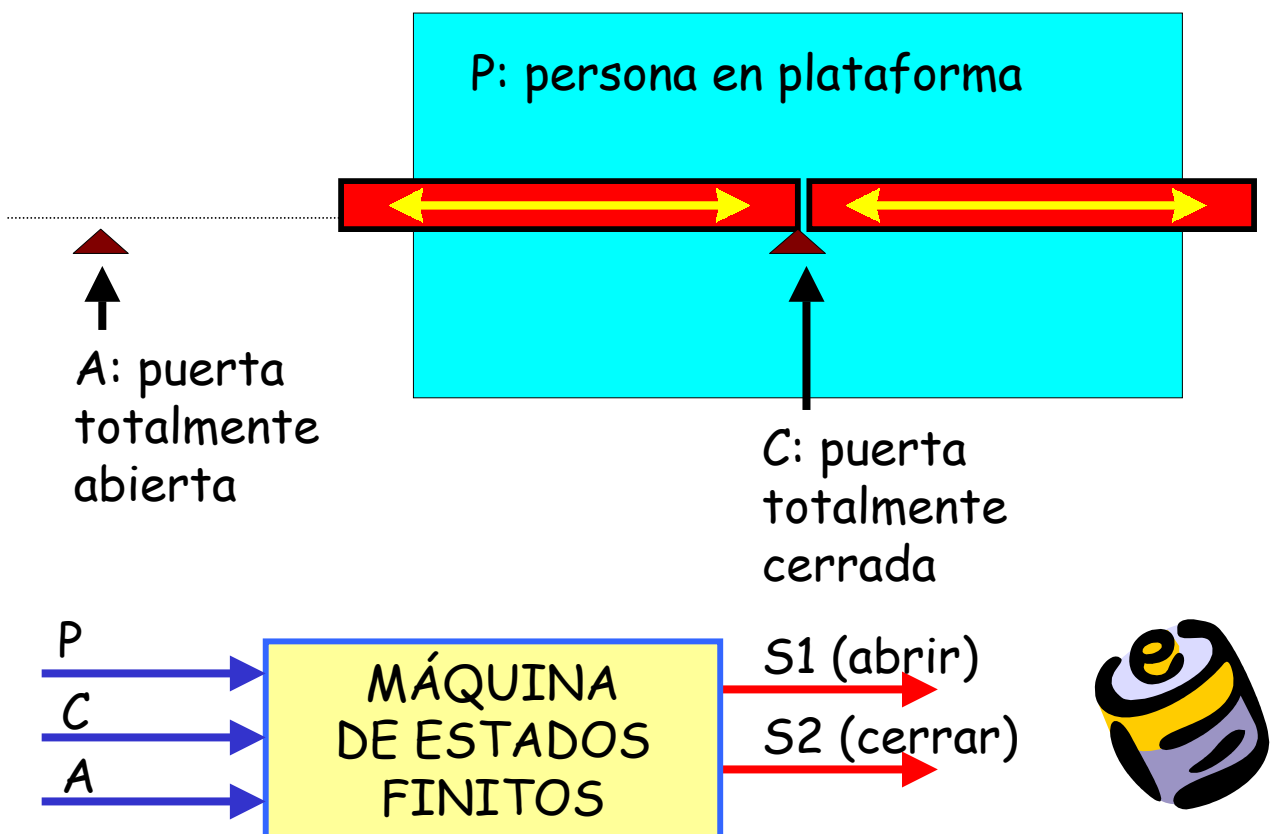


Tabla de Estados:

Combinación Entradas: PCA (Persona-Cerrada-Abierta)						Salidas		Descripción de los estados
						Abrir	Cerrar	
000	001	010	100	101	110	S1	S2	Estado
-	-	①	-	-	2	0	0	E1: Plat. vacía y puerta cerrada
-	-	1	3	-	②	1	0	E2: Con persona y puerta cerrada
5	-	-	③	4	-	1	0	E3: Con persona y abriendo
-	6	-	-	④	-	0	0	E4: Con persona y abierta
⑤	6	-	3	-	-	1	0	E5: Sin persona y abriendo
7	⑥	-	-	4	-	0	1	E6: Sin persona y abierta
⑦	-	1	3	-	-	0	1	E7: Sin persona y cerrando



--Diseño de una máquina de estados finitos de Moore
 --para la apertura y cierre de una puerta automática

ENTITY puerta IS

PORT(

--Entrada de reloj para C. Secuencial Síncrono

CLK : IN BIT;

--Resto de Entradas

Plataforma, FC_Cerrada, FC_Abierta : IN BIT;

Reset : IN BIT;

--Defino una entrada de reset para inicializar

--P:persona en plataforma, C:puerta cerrada, A:puerta abierta

--Salidas:

Abrir, Cerrar : OUT BIT);

--S1: Abrir, S2: Cerrar

END puerta;

continúa...

...continuación

```
ARCHITECTURE automatica OF puerta IS
--Definimos un tipo con los estados posibles
TYPE Estados_posibles IS (E1,E2,E3,E4,E5,E6,E7);
--Declaramos una señal que puede tomar cualquiera de los estados posibles
SIGNAL ESTADO: Estados_posibles;
BEGIN
--Defino un proceso sensible al reloj y al reset para modificar el Estado

PROCESS(CLK, Reset)
BEGIN
IF (Reset='0') THEN
    ESTADO <= E1; --Reset asíncrono de nivel activo bajo
ELSIF (CLK'EVENT AND CLK='1') THEN

    CASE ESTADO IS
        WHEN E1 => --Estando en el estado 1
            IF (Plataforma='1' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
                ESTADO <= E2;
            END IF;
        WHEN E2 => --Estando en el estado 2
            IF (Plataforma='0' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
                ESTADO <= E1;
            ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
                ESTADO <= E3;
            END IF;
        WHEN E3 => --Estando en el estado 3
            IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
                ESTADO <= E5;
            ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
                ESTADO <= E4;
            END IF;
        WHEN E4 => --Estando en el estado 4
            IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
                ESTADO <= E6;
            END IF;
        WHEN E5 => --Estando en el estado 5
            IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
                ESTADO <= E6;
            ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
                ESTADO <= E3;
            END IF;
    END CASE;
END IF;
END PROCESS;
```

continúa...

...continuación

```
WHEN E6 => --Estando en el estado 6
    IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        ESTADO <= E7;
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
        ESTADO <= E4;
    END IF;
WHEN E7 => --Estando en el estado 7
    IF (Plataforma='0' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
        ESTADO <= E1;
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        ESTADO <= E3;
    END IF;

END CASE;
END IF;
END PROCESS;

--Ahora en función del Estado, genero las salidas
--con dos sentencias WITH concurrentes con el Proceso

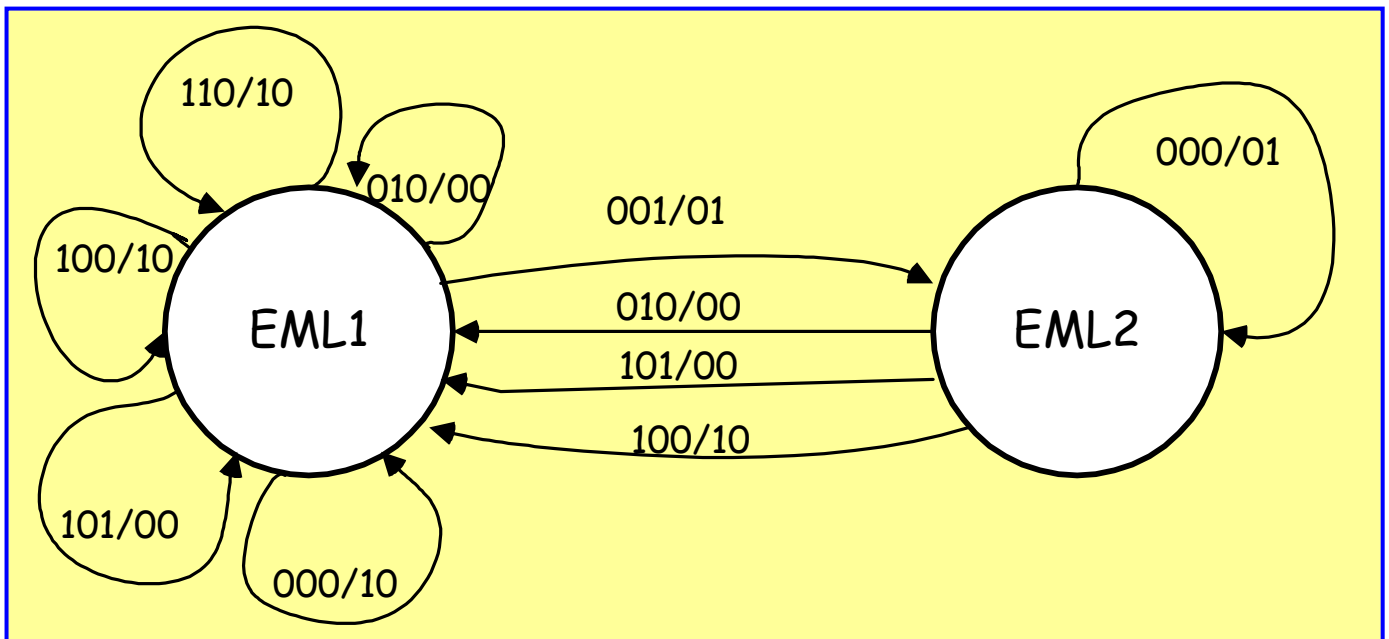
WITH ESTADO SELECT --Para la señal de abrir
    Abrir <= '0' WHEN E1|E4|E6|E7,
    '1' WHEN OTHERS;

WITH ESTADO SELECT --Para la señal de cerrar
    Cerrar <= '1' WHEN E6|E7,
    '0' WHEN OTHERS;

END automatica;
```

Fin descripción

Diagrama de Estados de Mealy de la puerta automática



--Diseño de una máquina de estados finitos de Mealy
 --para la apertura y cierre de una puerta automática

ENTITY puerta_Mealy IS

PORT(

--Entrada de reloj para C. Secuencial Síncrono

CLK : IN BIT;

--Resto de Entradas

Plataforma, FC_Cerrada, FC_Abierta : IN BIT;

Reset : IN BIT; --Defino una entrada de reset para inicializar

--P:persona en plataforma, C:puerta cerrada, A:puerta abierta

--Salidas:

Abrir, Cerrar : OUT BIT);

--S1: Abrir, S2: Cerrar

END puerta_Mealy;

continúa...

...continuación

```
ARCHITECTURE automatica_dos OF puerta_Mealy IS
    --Definimos un tipo con los estados posibles
    TYPE Estados_posibles IS (EML1,EML2);
    --Declaramos una señal que puede tomar cualquiera de los estados posibles
    SIGNAL ESTADO: Estados_posibles;
BEGIN
    --Defino un proceso sensible al reloj y al reset para modificar el Estado
    PROCESS(CLK, Reset)
    BEGIN
        IF (Reset='0') THEN
            ESTADO <= EML1; --Reset asíncrono de nivel activo bajo
        ELSIF (CLK'EVENT AND CLK='1') THEN
            CASE ESTADO IS
                WHEN EML1 => --Estando en el estado 1 de Mealy
                    IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
                        ESTADO <= EML2;
                    END IF;
                WHEN EML2 => --Estando en el estado 2 de Mealy
                    IF (Plataforma='0' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
                        ESTADO <= EML1;
                    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
                        ESTADO <= EML1;
                    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
                        ESTADO <= EML1;
                    END IF;
            END CASE;
        END IF;
    END PROCESS;
```

continúa...

...continuación

```
--Se define ahora un proceso sensible al estado y a las entradas
--concurrente con el anterior
PROCESS(ESTADO,Plataforma,FC_Cerrada,FC_Abierta)
BEGIN
CASE ESTADO IS
WHEN EML1 =>
    IF (Plataforma='0' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
        Abrir <= '0';
        Cerrar <= '0';
    ELSIF (Plataforma='1' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
        Abrir <= '1';
        Cerrar <= '0';
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        Abrir <= '1';
        Cerrar <= '0';
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
        Abrir <= '0';
        Cerrar <= '0';
    ELSIF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        Abrir <= '1';
        Cerrar <= '0';
    ELSIF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
        Abrir <= '0';
        Cerrar <= '1';
    END IF;
WHEN EML2 =>
    IF (Plataforma='0' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        Abrir <= '0';
        Cerrar <= '1';
    ELSIF (Plataforma='0' AND FC_Cerrada='1' AND FC_Abierta='0') THEN
        Abrir <= '0';
        Cerrar <= '0';
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='1') THEN
        Abrir <= '0';
        Cerrar <= '0';
    ELSIF (Plataforma='1' AND FC_Cerrada='0' AND FC_Abierta='0') THEN
        Abrir <= '1';
        Cerrar <= '0';
    END IF;
END CASE;
END PROCESS; --Final del proceso para las salidas
END automatica_dos;
```

Fin descripción