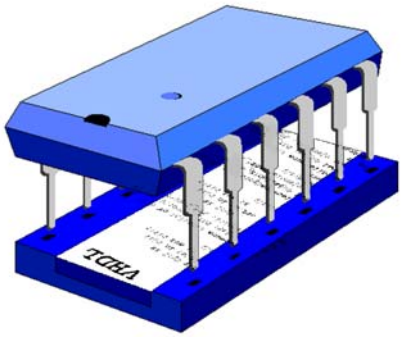
Introducción a la Programación en VHDL



Facultad de Informática
Universidad Complutense de Madrid

Marcos Sánchez-Élez
marcos@fis.ucm.es

Índice

I. Introducción 5
II. Elementos Básicos de VHDL 8
2.1 Entity 8
2.2 Architecture 9
2.3 Identificadores 10
2.4 Operadores 14
III. Estructura Básica de un Archivo fuente en VHDL 16
3.1 Sentencias Concurrentes 16
3.2 Sentencias Condicionales 18
3.3 Sentencia process 20
3.4 Descripción Estructural 25
3.5 Ejemplos 28
IV. Simulación en VHDL 31
4.1 Fases de la simulación 31
4.2 Sentencias de simulación 33
4.3 Plantillas para simulación en VHDL 34
V. Descripción de Lógica Secuencial 38
5.1 Hardware Secuencial 38
5.2 Contadores 41
5.3 Ejemplos 44
VI. Diseño de una Máquina de Estados 46
VII. Funciones, Procedimientos y Paquetes 49
7.1 Ejemplo 50
VIII. Ejemplo: Diseño de una Memoria RAM 52
Apéndices 54

I. Introducción

VHDL es un lenguaje de descripción de circuitos electrónicos digitales que utiliza distintos niveles de abstracción. El significado de las siglas VHDL es *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language*. Esto significa que VHDL permite acelerar el proceso de diseño.

VHDL no es un lenguaje de programación, por ello conocer su sintaxis no implica necesariamente saber diseñar con él. VHDL es un lenguaje de descripción de hardware, que permite describir circuitos síncronos y asíncronos. Para realizar esto debemos:

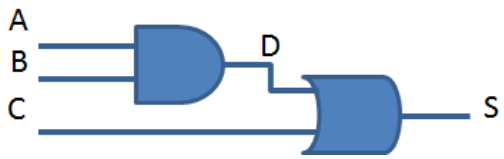
- Pensar en puertas y biestables, no en variables ni funciones.
- Evitar bucles combinacionales y relojes condicionados.
- Saber qué parte del circuito es combinacional y cuál secuencial.

¿Por qué usar un lenguaje de descripción hardware?

- Poder descubrir problemas en el diseño antes de su implementación física.
- La complejidad de los sistemas electrónicos crece exponencialmente, es necesaria una herramienta que trabaje con el ordenador.
- Permite que más de una persona trabaje en el mismo proyecto.

En particular VHDL permite tanto una descripción de la estructura del circuito (descripción a partir de subcircuitos más sencillos), como la especificación de la funcionalidad de un circuito utilizando formas familiares a los lenguajes de programación.

La misión más importante de un lenguaje de descripción HW es que sea capaz de simular perfectamente el comportamiento lógico de un circuito sin que el programador necesite imponer restricciones (ver ejemplo 1). En el ejemplo, una ejecución del código utilizando las reglas básicas de cualquier lenguaje de programación al uso daría dos resultados diferentes sobre la misma descripción del circuito. Esto es debido a que en HW todos los circuitos trabajan a la vez para obtener el resultado (todo se ejecuta en paralelo) mientras que en software el orden de las instrucciones delimita la actualización de las variables (ejecución secuencial de las instrucciones). Un lenguaje de descripción HW, VHDL o cualquier otro de los existentes en el mercado, nos debe dar el mismo resultado en simulación para los dos programas del ejemplo 1.



Prog 1	Prog 2
D = A and B; S = D or C;	S = D or C; D = A and B

Simulación Hardware		
t = 0ns	**t = 5ns**	**t = 10ns**
A = 0	A = 0	A = 1
B = 0	B = 1	B = 1
C = 0	C = 0	C = 0

	Simulación Software	
	Prog 1	Prog 2
S(5 ns)	0	0
S(10 ns)	1	0

Ejemplo 1. Simulación incorrecta de un circuito

Los circuitos descritos en VHDL pueden ser simulados utilizando herramientas de simulación que reproducen el funcionamiento del circuito descrito. Para la realización de la simulación existe un estándar aprobado por el *ieee*, en el cual se explican todas las expresiones propias de VHDL y cómo se simulan. Además, existen herramientas que transforman una descripción VHDL en un circuito real (a este proceso se le denomina síntesis). La sintaxis para síntesis y su implementación final, aunque sigue unas normas generales, depende en gran medida de la herramienta de síntesis seleccionada.

En este manual utilizaremos la herramienta de síntesis proporcionada de manera gratuita por Xilinx (Xilinx ISE Web Pack), que se puede conseguir en la siguiente dirección URL: http://www.xilinx.com/support/download/index.htm. Todos los ejemplos del manual que presenten una codificación que sea particular para la herramienta de Xilinx aparecerán en un recuadro similar a éste.

CONSEJO

A lo largo de este manual se utilizarán recuadros como este para recalcar los consejos para una programación eficiente en VHDL. Estos consejos son una serie de normas básicas que ayudan a que los resultados de la simulación sean independientes de la forma de programación y el código desarrollado pueda ser sintetizado, y por lo tanto, implementado físicamente en una plataforma, con el mínimo esfuerzo.

Webs y Noticias Relacionadas con la programación en VHDL y sus herramientas de simulación y síntesis

www.edacafe.com

Espacio web dedicado a difundir las noticias relacionadas con el mundo del diseño de circuitos. Tiene un foro particular de VHDL (problemas, herramientas gratuitas …)

www.eda.org/vasg/

"Welcome to the VHDL Analysis and Standardization Group (VASG). The purpose of this web site is to enhance the services and communications between members of the VASG and users of VHDL. We've provided a number of resources here to help you research the current and past activities of the VASG and report language bugs, LRM ambiguities, and suggest improvements to VHDL …"

www.cadence.com

"Cadence Design Systems is the world's largest supplier of EDA technologies and engineering services. Cadence helps its customers break through their challenges by providing a new generation of electronic design solutions that speed advanced IC and system designs to volume …"

www.xilinx.com

"In the world of digital electronic systems, there are three basic kinds of devices: memory, microprocessors, and logic. Memory devices store random information such as the contents of a spreadsheet or database. Microprocessors execute software instructions to perform a wide variety of tasks such as running a word processing program or video game. Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform"

II. Elementos Básicos de VHDL

Un sistema digital está descrito por sus entradas y sus salidas y la relación que existe entre ellas.

En el caso de VHDL por un lado se describirá el aspecto exterior del circuito: entradas y salidas; y por otro la forma de relacionar las entradas con las salidas. El aspecto exterior, cuántos puertos de entrada y salida tenemos, es lo que denominaremos **entity**. Y la descripción del comportamiento del circuito **architecture**, toda *architecture* tiene que estar asociada a una *entity*.

Además, aunque no es estrictamente necesario, podemos definir también las bibliotecas y paquetes que vamos a utilizar, lo que nos indicará que tipos de puertos y operadores podemos utilizar. Siempre ha de aparecer la definición de las bibliotecas y paquetes antes de la definición de la *entity*.

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

La biblioteca ieee y estos tres paquetes asociados (más adelante se explicará su significado) aparecen por defecto al generar un módulo VHDL en Xilinx ISE

2.1 Entity

Una entidad es la abstracción de un circuito, ya sea desde un complejo sistema electrónico o una simple puerta lógica. La entidad únicamente describe la forma externa del circuito, en ella se enumeran las entradas y las salidas del diseño. Una entidad es análoga a un símbolo esquemático en los diagramas electrónicos, el cual describe las conexiones del dispositivo hacia el resto del diseño.

- Define externamente al circuito o subcircuito.
- Nombre y número de puertos, tipos de datos de entrada y salida.
- Tienes toda la información necesaria para conectar tu circuito a otros circuitos.

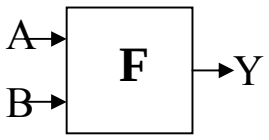
```
entity nombre is
     generic (cte1: tipo := valor1; cte2: tipo:= valor 2; …);
     port (entrada1, entrada2, … : in tipo;
           salida1, salida2, …: out tipo;
           puertoi : modo tipo);
end nombre;
```

Los puertos pueden ser de entrada **in**, salida **out**, entrada-salida **inout** o **buffer**. Los puertos de entrada sólo se pueden leer y no se puede modificar su valor internamente en la descripción del comportamiento del circuito (*architecture*), sobre los puertos de salida sólo

se puede escribir pero nunca tomar decisiones dependiendo de su valor (esto implica una lectura). Si es **estrictamente necesario** escribir sobre un puerto a la vez que se tiene que tener en cuenta su valor el tipo sería *inout* o *buffer*.

Además, en la *entity* se pueden definir unos valores genéricos (**generic**) que se utilizarán para declarar propiedades y constantes del circuito, independientemente de cual sea la arquitectura. A nivel de simulación utilizaremos *generic* para definir retardos de señales y ciclos de reloj, estas definiciones no serán tenidas en cuenta a nivel de síntesis. También se puede utilizar *generic* para introducir una constante que será utilizada posteriormente en la *architecture*, utilizaremos esa constante para hacer nuestro circuito más general. Por ejemplo, podemos definir el comportamiento de un banco de registros teniendo en cuenta que puede tener cualquier número de registros, fijando el número de registros particular que queremos simular e implementar a través de una constante del *generic*. Esto implica que en toda la parte de nuestro código (el que vamos a escribir dentro de *architecture*) donde haga falta el número de registros utilizaremos el nombre de la constante definida en *generic*, de manera análoga a como se haría en cualquier lenguaje de programación convencional. La sentencia *generic* no es necesaria, en caso de que no vayamos a utilizarla puede desaparecer de la *entity*.

A continuación se presenta un ejemplo de descripción externa del circuito (*entity*). Para el ejemplo sabemos que el circuito presentará dos entradas de tamaño N bits y una salida de tamaño un bit, particularizamos la entidad para N igual a 8. Como hemos advertido anteriormente, aunque la función de *generic* es permitirnos generar un código más general, una vez que definimos el circuito, tenemos que particularizarlo, por lo que siempre debe darse un valor a las constantes del campo *generic*.




```
entity F is
     generic (N: natural :=8);
     port (A, B: in bit_vector(N-1 downto 0);
           Y: out bit);
end F;
```

2.2 Architecture

Los pares de entidades y arquitecturas se utilizan para representar la descripción completa de un diseño. Una arquitectura describe el funcionamiento de la entidad a la que hace referencia, es decir, dentro de architecture tendremos que describir el funcionamiento de la entidad a la que está asociada utilizando las sentencias y expresiones propias de VHDL.

- Define internamente el circuito.
- Señales internas, funciones, procedimientos, constantes …
- La descripción de la arquitectura puede ser estructural o por comportamiento.

```
architecture arch_name of entity_name is
-- declaraciones de la arquitectura:
-- tipos
-- señales
-- componentes

begin
-- código de descripción
-- instrucciones concurrentes
-- ecuaciones booleanes
-- componentes
     process (lista de sensibilidad)
     begin
     -- código de descripción
     end process;
end arch_name;
```

El código VHDL propiamente dicho se escribe dentro de *architecture*. Cada *architecture* va asociada a una *entity* y se indica en la primera sentencia. A continuación, y antes de *begin* se definen todas las variables (señales) internas que vas a necesitar para describir el comportamiento de nuestro circuito, se definen los tipos particulares que necesitamos utilizar y los componentes, otros circuitos ya definidos y compilados de los cuales conocemos su interfaz en VHDL (su *entity*).

Desde *begin* hasta *end* escribiremos todas las sentencias propias de VHDL, pero no todas pueden utilizarse en cualquier parte del código. Así pues aquellas sentencias de VHDL que tengan definido un valor para cualquier valor de la entrada (y que nosotros denominamos sentencias concurrentes) podrán ir en cualquier parte del código pero fuera de la estructura *process*. Aunque no es el fin de este manual, puede afirmarse que todas las sentencias concurrentes se traducirán en subcircuitos combinacionales. También fuera de la estructura *process*, se instanciarán los componentes, subcircuitos ya definido sutilizados por el circuito actual, indicando cuáles son sus entradas y sus salidas de entre las señales del circuito del que forman parte.

El **process** es una estructura particular de VHDL (que se describe con mucho más detalle más adelante) que se reserva principalmente para contener sentencias que no tengan obligatoriamente que tener definido su valor para todas las entradas (el ejemplo más común es una estructura *if-else* incompleta). Esto obliga a que la estructura process almacene los valores de sus señales y pueda dar lugar (no siempre) a subcircuitos secuenciales. Además, **en simulación** sólo se ejecutan las sentencias internas a esta estructura cuando alguna de las señales de su lista de sensibilidad cambia de valor.

2.3 Identificadores

En VHDL existen tres clases de objetos por defecto:

- **Constant**. Los objetos de esta clase tienen un valor inicial que es asignado de forma previa a la simulación y que no puede ser modificado durante ésta.
    - `constant identificador: tipo:= valor;`

- **`Variable`**. Los objetos de esta clase contienen un único valor que puede ser cambiado durante la simulación con una sentencia de asignación. Las variables generalmente se utilizan como índices, principalmente en instrucciones de bucle, o para tomar valores que permitan modelar componentes. Las variables **NO** representan conexiones o estados de memoria. Pueden ser declaradas antes del *begin* de la *architecture* y/o antes del *begin* del *process*, en su declaración se les puede asignar un valor por defecto.

  - `variable identificador: tipo [:= valor];`

  La asignación de una variable a un valor se hace mediante el operador :=

  ```
  nombre variable := valor o expresión;
  i := 10;
  ```

- **`Signal`**. Las señales representan elementos de memoria o conexiones y sí pueden ser sintetizados, dicho de otra manera, a cada objeto de nuestro código VHDL que sea declarado como *signal* le corresponde un cable o un elemento de memoria (biestable, registro …) en nuestro circuito. Por lo tanto, su comportamiento en simulación será el esperado de ese elemento físico aunque no lo describamos en el código explícitamente. Tienen que ser declaradas antes del *begin* de la *architecture*. Los puertos de una entidad son implícitamente declarados como señales en el momento de la declaración, ya que estos representan conexiones.

  - `signal identificador: tipo;`

  La asignación de una señal a un valor se hace mediante el operador <=

  ```
  nombre señal <= valor o expresión;
  A <= 10;
  ```

CONSEJO

Si en el código VHDL desarrollado sólo se utiliza *constant* y *signal* no se observarán efectos perversos en la simulación (ver apéndice). Además, el código obtenido podrá ser sintetizado en cualquier herramienta. Por eso mismo en este manual a partir de este momento cuando nos refiramos a una señal nos estaremos refiriendo a un objeto definido como *signal* y sólo trabajaremos con objetos definidos como *signal*.

En las tres definiciones anteriores, como en la definición de los puertos de la *entity* es necesario definir el tipo del objeto. VHDL permite utilizar tipos predefinidos, así como otros definidos por el usuario. Los tipos predefinidos más comunes son los siguientes:

bit	sólo admite los valores 0 y 1. Para hacer una asignación a un objeto tipo bit el valor binario tiene que aparecer entre comas simples (‘0’ o ‘1’)
bit_vector (rango**)**	el rango, siempre entre paréntesis, indica el número de bits del vector, éstos sólo pueden estar formados por ceros y unos. Para un vector de N bits el rango será N-1 **downto** 0, donde el bit más a la izquierda es el más significativo y el bit más a la derecha el menos significativo (notación binaria estándar). Para hacer una asignación el valor tiene que aparecer entre comillas (por ejemplo: “1100”)
boolean	sólo admite los valores **true** y **false**
character	cualquier valor ascii
string	cualquier cadena formada por ascii
integer rango	cualquier número entero dentro del rango, aquí el rango no va entre paréntesis, puede expresarse como 0 **to** MAX
natural rango	cualquier número natural dentro del rango
positive rango	cualquier número positivo dentro del rango
real rango	cualquier número real dentro del rango
std_logic	tipo predefinido en el estándar IEEE 1164. Este tipo representa una lógica multivaluada de 9 valores. Además del ‘0’ lógico y el ‘1’ lógico, posee alta impedancia ‘Z’, desconocido ‘X’ ó sin inicializar ‘U’ entre otros. Para hacer una asignación el valor tiene que aparecer entre comas simples (‘0’, ‘1’, ‘X’, …)
std_logic_vector(rango)	representa un vector de elementos *std_logic*, posee las mismas reglas de asignación y definición del rango que el tipo *bit_vector* pero con un mayor número de valores posibles.

Para Xilinx ISE todos los puertos de *entity* tienen que ser obligatoriamente de tipo *std_logic* o *std_logic_vector* ya que de esa manera se puede simular un circuito *más real*. Por ejemplo, podría darse el caso de que en el código VHDL todavía no hayamos definido el valor de una señal (ejemplo, valor inicial de un biestable no reseteado), si la señal fuera de tipo bit su valor por defecto sería 0 y si fuera de tipo *std_logic* su valor por defecto sería U (indeterminado) que se acerca más a la realidad. Además, las señales definidas como natural o integer Xilinx ISE las traduce a *std_logic_vector* con el número de bits necesario para su representación completa.

Para poder utilizar el tipo std_logic hay que añadir la librería que lo soporta.

```
use ieee.std_logic_1164.all
```

Para poder utilizar las funciones aritmeticológicas definidas (suma, resta multiplicación)

```
use ieee.std_logic_arith.all
```

Si los vectores están en representación binaria pura

```
use ieee.std_logic_unsigned.all
```

Los vectores están en C_2

```
use ieee.std_logic_signed.all
```

CONSEJO

Podemos escribir todas las asignaciones del código ya sean operaciones sencillas, operaciones aritmeticológicas y comparaciones utilizando *std_logic_vector* y trabajando con ellos como si fueran enteros o naturales gracias a *ieee.std_logic_arith.all* y *ieee.std_logic_unsigned.all*. Definir todas las señales internas como *std_logic* o *std_logic_vector* no complica el código VHDL final y ayuda a su integración en Xilinx.

TIPO ENUMERADO es un tipo de dato con un grupo de posibles valores asignados por el usuario. Los tipos enumerados se utilizan principalmente en el diseño de máquinas de estados.

```
type nombre is (valor1, valor2, …);
```

Suponiendo que hemos definido A como una señal de un tipo enumerado la asignación será: `A <= valori;` donde *valori* debe ser uno de los valores enumerados en la definición del tipo.

Los tipos enumerados se ordenan de acuerdo a sus valores. Los programas de síntesis automáticamente codifican binariamente los valores del tipo enumerado para que estos puedan ser sintetizados. Algunos programas lo hacen mediante una secuencia binaria ascendente, otros buscan cual es la codificación que conviene para tratar de minimizar el circuito o para incrementar la velocidad del mismo una vez que la descripción ha sido sintetizada. También es posible asignar el tipo de codificación mediante directivas propias de la herramienta de síntesis.

TIPOS COMPUESTOS un tipo compuesto es un tipo de dato formado con elementos de otros tipos, existen dos formas de tipos compuestos, arrays y records.

- Un **ARRAY** es un objeto de datos que consiste en una "colección" de elementos del mismo tipo.

```
type nombre is array (rango) of tipo;
```

La asignación de un valor a una posición del array se realiza mediante números enteros (ver ejemplos).

- Un **RECORD** es un objeto de datos que consiste en una "colección" de elementos de distintos tipos.

```
type nombre is record
  elemento1: tipo_de_dato1;
  elemento2: tipo_de_dato2;
end record;
```

La asignación de un valor a un elemento interno de una señal definida de tipo record se realiza mediante un punto (ver ejemplos).

Una vez definido el tipo compuesto (y/o tipo enumerado) y asignado un nombre a éste, se podrá definir cualquier señal como correspondiente a este nuevo tipo definido. La asignación a cualquier señal de un tipo compuesto y enumerado se hará utilizando el operador definido para señales <=.

A continuación se presentan unos ejemplos en los cuales se definen y asignan valores a distintas variables y señales.

```
------------------------------------------------------------
-- Se utilizan dos guiones para introducir comentarios
-- en el código VHDL
------------------------------------------------------------
-- Ejemplos de definiciones y asignaciones
------------------------------------------------------------
constant DATA_WIDTH: integer := 8;
signal CTRL: bit_vector(7 downto 0);
variable SIG1, SIG2: integer range 0 to 15;
------------------------------------------------------------
type color is (rojo, amarillo, azul);
signal BMP: color;
BMP <= rojo;
------------------------------------------------------------
type pal is array (0 to 15) of std_logic_vector (7 downto 0);
signal word: pal;
-- word(integer/natural) <= vector de bits;
word(0) <= "00111110";
word(1) <= "00011010";
…
word(15) <= "11111110";
------------------------------------------------------------
type matrix is array (0 to 15)(7 downto 0) of std_logic;
signal matriz: matrix;
matriz(2)(5)<='1';
------------------------------------------------------------
type conjunto is record
   palabra: std_logic_vector (0 to 15);
   valor: integer range -256 to 256;
end record;
signal dato: conjunto;
dato.valor <= 176;
```

2.4 Operadores

Un operador nos permite construir diferentes tipos de expresiones mediante los cuales podemos calcular datos utilizando diferentes señales. En VHDL existen distintos operadores de asignación con lo que se transfieren y transforman valores de una señal a otra.

+, -, *, /, mod, rem	operaciones aritméticas
+, **-**	cambio de signo
&	concatenación
and, or, nand, nor, xor	operaciones lógicas
:=	asignación de valores a constantes y variables.
<=	asignación de valores a señales.

```
--------------------------------
-- Ejemplos de asignación
--------------------------------

y <= (x and z) or d(0);
y(1)<= x and not z;
y <= x1&x2; -- y="x1x2"
c := 27 + r;

--------------------------------
```

III. Estructura Básica de un Archivo fuente en VHDL

Como hemos visto los modelos VHDL están formados por dos partes: la entidad (*entity*) y la arquitectura (*architecture*); es en esta última donde se escriben las sentencias que describen el comportamiento del circuito, a este modelo de programación en VHDL se le suele denominar *behavioral*.

```
architecture circuito of nombre is
-- señales
begin
-- sentencias concurrentes
process (lista de sensibilidad)
begin
-- sentencias secuenciales
-- sentencias condicionales
end process
end  architecture circuito;
```

Dentro de la arquitectura se encuentra:

i) Tipos y señales intermedias necesarios para la descripción del comportamiento.
ii) Sentencias de asignación que deben realizarse siempre así como sentencias concurrentes.
iii) Uno a varios *process* que tienen en su interior sentencias condicionales y/o asignaciones a señales que dan lugar a hardware secuencial.

3.1 Sentencias Concurrentes

Las sentencias concurrentes son sentencias condicionales que tienen al menos un valor por defecto para cuando no se cumplen ninguna de las condiciones. Aunque podría utilizarse una sentencia común como un if con obligación de else, los desarrolladores de VHDL han preferido utilizar dos sentencias particulares:

WHEN – ELSE

```
señal_a_modificar <=  valor_1 when condición_1 else
                      valor_2 when condición_2 else
                      ...
                      valor_n when condición_n else
                      valor_por defecto;
```

En esta sentencia siempre modificamos el valor de una misma señal, pero las condiciones pueden ser independientes (actuar sobre distintas señales cada una), dónde la colocación de las condiciones indica la preferencia de unas sobre otras, es decir, la

condición 1 tiene preferencia sobre el resto, la condición 2 sobre todas menos la 1 y así sucesivamente.

```
-------------------------------------------
-- Ejemplos when-else
-------------------------------------------
C <=    "00" when A = B else
        "01" when A < B else
        "10";
-------------------------------------------
C <=    "00" when A = B else
        "01" when D = "00" else
        "10";
-------------------------------------------
```

WITH – SELECT – WHEN

```
with señal_condición select
señal_a_modificar <= valor_1 when valor_1_señal_condición,
                     valor_2  when  valor_2_señal_condición,
                     ...
                     valor_n  when  valor_n_señal_condición,
                     valor_por_defecto when others;
```

Esta sentencia es menos general que la anterior. En este caso se modificará el valor de una señal dependiendo de los valores de una señal condición, aparecerán como máximo tantas líneas como valores posibles pueda tener la señal condición.

```
-------------------------------------------
-- Ejemplo with-select
-------------------------------------------
with entrada select
salida <=    "00" when "0001",
             "01" when "0010",
             "10" when "0100",
             "11" when others;
-------------------------------------------
```

Desde un punto de vista de HW estas dos sentencias dan como resultado HW combincional puro, es decir, puertas lógicas, multiplexores, decodificadores …

CONSEJO

Podemos escribir muchas sentencias if-else (siempre que la sentencia tenga else) como cualquiera de las dos sentencias anteriores. El buen programador de VHDL debe acostumbrarse a utilizarlas ya que le quitará de muchos problemas que aparecen con la pareja *process-if-else*.

3.2 Sentencias Condicionales

VHDL permite utilizar otro tipo de sentencias condicionales más parecidas a los lenguajes de programación usados. Todas estas sentencias como se explicará la sección 3.3 tiene que ir **obligatoriamente dentro de un *process*.** Las sentencias condicionales más comunes en VHDL son las siguientes:

IF – THEN – ELSE

```
process (lista de sensibilidad)
begin
if condición then
-- asignaciones
elsif otra_condición then
-- asignaciones
else
-- asignaciones
end if;
end process;

-----------------------------------------------
-- Ejemplo
-----------------------------------------------
process (control, A, B)
begin
if control = "00" then
   resultado <= A + B;
elsif control = "11" then
   resultado <= A – B;
else
   resultado <= A;
end if;
end process;
-----------------------------------------------
```

La sentencia if-else permite cualquier tipo de combinación y encadenamiento, exactamente igual que ocurre en C o PASCAL o cualquier otro lenguaje de programación de alto nivel.

CONSEJO

Las sentencias if-else, excepto en los casos que se explicarán en la sección 5, deberían poseer un else. Además, conviene, como se explicará en 3.3, que en las asignaciones de cada rama otorguemos valor siempre a las mismas señales, haga falta estrictamente o no.

CASE – WHEN

```
process (lista de sensibilidad)
begin
case señal_condición is
     when valor_condición_1 =>
     -- asignaciones
     …
     when valor_condición_n =>
     -- asignaciones
     when others =>
     -- asignaciones
end case;
end process;
```

Dentro de las asignaciones pueden parecer también sentencias *if-else*. Es necesario que aparezca en la estructura *when others*, pero no es necesario que tenga asignaciones, se puede dejar en blanco.

```
-----------------------------------------------
-- Ejemplo
-----------------------------------------------
process (control, A, B)
begin
case control is
when "00" =>
   resultado <= A+B;
when "11" =>
   resultado <= A-B;
when others =>
   resultado <= A;
end case;
end process;
-----------------------------------------------
```

Igual que en los lenguajes software, existen distintos tipos de bucles:

FOR – LOOP

```
process (lista de sensibilidad)
begin
for loop_var in range loop
   -- asignaciones
end loop;
end process;
```

Para el for *range* puede ser *0 to N* o *N downto 0*.

```
-----------------------------------------------
-- Ejemplo
-----------------------------------------------
process (A)
begin
for i in 0 to 7 loop
   B(i+1) <= A(i);
end loop;
end process;
-----------------------------------------------
```

WHILE – LOOP

```
process (lista de sensibilidad)
begin
while condición loop
   -- asignaciones
end loop;
end process;

-----------------------------------------------
-- Ejemplo
-----------------------------------------------
process (A)
variable i: natural := 0;
begin
while i < 7 loop
   B(i+1) <= A(i);
   i := i+1;
end loop;
end process;
-----------------------------------------------
```

El bucle tipo **for** está soportado si el rango del índice es estático (*0 to N* ó *N downto 0*, donde N posee siempre el mismo valor) y el cuerpo no contiene sentencias *wait*.
Los bucles de tipo **while** en general no están soportados

3.3 Sentencia process

VHDL presenta una estructura particular denominada *process* que define los límites de un dominio que se ejecutará (simulará) si y sólo si alguna de las señales de su lista de sensibilidad se ha modificado en el anterior paso de simulación.

Un *process* tiene la siguiente estructura:

```
process (lista_de_sensibilidad)
-- asignacion de variables
-- opcional no recomendable
begin
-- Sentenicas condicionales
-- Asignaciones
end process;
```

La sentencia *process* es una de las más utilizadas en programación con VHDL ya que tanto las sentencias condicionales como la descripción de HW secuencial se realiza dentro de él. Pero a la vez es, para aquellos que se acercan por primera vez a la simulación y síntesis con VHDL, el principal problema para un correcto diseño. Por eso a continuación se van a enumerar una serie de normas relacionadas directamente con las propiedades de la sentencia *process*, que serán de obligado cumplimiento para que el código generado simule y sintetice de manera correcta.

Propiedad I: En una estructura *process* sólo se ejecutan las instrucciones internas en el instante 0 de simulación y cuando varía alguna de las señales de su lista de sensibilidad.

Problema: El resultado de la simulación del circuito puede ser inesperada debido al *efecto maligno* de la lista de sensibilidad.

Solución: En la lista de sensibilidad han de incluirse al menos todas las señales que se lean dentro del *process* (señal_escritura <= señal_lectura).

```
-------------------------
-- Efecto de la lista
-- de sensibilidad
-------------------------
process (A)
begin
     if B='1' then
     C <= A;
     end if;
end process;
-------------------------
```

t	0 ns	5 ns	10 ns
A	0	0	1
B	0	1	1
C	U	U	1

En el ejemplo no se asigna a C un valor hasta el instante 10 ns aunque B cambió en el instante 5 ns, esto es debido a que no se entra dentro del process hasta que A no varía (instante 10 ns). Sin embargo, a nivel HW esperaríamos que C tomase el valor de A en el mismo instante en el que B cambia a 1 (en 5 ns). Siguiendo la norma explicada en i el código debería ser el siguiente:

```
process (A, B)
begin
     if B='1' then
     C <= A;
     end if;
end process;
```

t	0 ns	5 ns	10 ns
A	0	0	1
B	0	1	1
C	U	0	1

Propiedad II: Las asignaciones a señales que se realizan dentro de un *process* tienen memoria.

Problema: Si en un paso de simulación se entra dentro del *process* y debido a las sentencias internas se modifica el valor de la señal C, y en otro paso de simulación posterior se entra dentro del *process* pero no se modifica C, la señal C conservará el valor asignado con anterioridad. El resultado de la simulación del circuito puede ser inesperado debido al *efecto maligno* de la memoria del *process*.

Solución: Siempre que se escriba una sentencia condicional es obligatorio asegurar el valor que deben tener todas las señales en cada rama del árbol condicional. Además, excepto si la definición del diseño nos lo prohíbe (ver capítulo 5), toda condición debe tener su rama else.

```
--------------------------
-- Condicional completo
--------------------------
   process (a, b)
   begin
        if a = b then
            c <= a or b;
        elsif a<b then
            c <= b;
        else
            c <= "00";
        end if;
end process;
-------------------------
```

Caso 1			
T	**0 ns**	**5 ns**	**10 ns**
A	01	11	11
B	10	10	11
C	10	00	11
Caso 2			
A	01	11	11
B	11	10	11
C	11	00	11

```
--------------------------
-- Condicional incompleto
--------------------------
process (a, b)
begin
     if a = b then
         c <= a or b;
     elsif a<b then
         c <= b;
     end if;
end process;
-------------------------
```

Caso 1			
T	**0 ns**	**5 ns**	**10 ns**
A	01	11	11
B	10	10	11
C	10	10	11
Caso 2			
A	01	11	11
B	11	10	11
C	11	11	11

En el primer código existe una rama else, lo que implica que en caso de que no se cumpla alguna de las dos condiciones existe un valor por defecto para C (en este caso en 5 ns). Sin embargo, en el segundo código la rama else ha desaparecido, eso hace que en el instante 5 ns para los mismos valores de A y de B se obtenga un resultado de C distinto (comparar resultados de caso 1 y caso 2 en el segundo código). Eso es debido a que los *process* tienen memoria y como la relación entre A y B que se da en 5ns no está

contemplada en el segundo código se guarda el valor obtenido con anterioridad. Obsérvese que en el ejemplo del condicional completo, el caso 1 y el caso 2 en 5ns tienen los mismos valores de A y B y se obtiene el mismo valor de C.

```
--------------------------
-- Condicional incompleto?
--------------------------
   process (a, b)
   begin
        if a = b then
            c <= a or b;
        elsif a<b then
            d <= b;
        else
            c <= "00";
            d <= "11";
        end if;
end process;
-------------------------
```

Caso 3			
t	**0 ns**	**5 ns**	**10 ns**
A	01	10	11
B	10	10	10
C	UU	10	00
D	10	10	11
Caso 4			
A	01	10	11
B	00	10	10
C	00	10	00
D	11	11	11

Se puede comprobar que este código obtiene dos resultados diferentes para las mismas entradas en el instante 5 ns (caso 3 y el caso 4), aunque tiene rama else. Es importante asegurarse siempre, no sólo que tiene rama else, sino también que tienen valores todas las señales de asignación en todas las ramas. En el caso del ejemplo la rama if asigna un valor a C pero no a D y la rama *elsif* asigna un valor a D pero no a C.

Propiedad III: Dentro de un *process* todas las instrucciones se ejecutan en paralelo, igual que ocurre con las instrucciones que se encuentran fuera de los *process*. Sin embargo, si dentro del *process* se asigna valor a una señal en dos sitios diferentes el resultado será aquel de la última asignación, exactamente igual que en los lenguajes de programación comunes.

Solución: Siempre comprobar que no estamos asignando el valor a una señal en dos sitios diferentes del *process* (si puede hacerse en dos ramas diferentes del mismo *if*).

```
process (a, b)
begin
     c <= "00";
     if a = b then
         c <= a or b;
     elsif a<b then
         c <= b;
     end if;
end process;
```

t	**0 ns**	**5 ns**	**10 ns**
A	01	11	11
B	10	10	11
C	10	00	11

En el ejemplo en el instante 0 ns y 10 ns se asigna dos veces un valor a C. A la salida del *process* nos quedamos con el último valor asignado, el de la rama *if* y el de la rama *elsif* respectivamente. En el instante 5 ns, sólo hay una asignación "00", el valor a mostrar es el de dicha asignación.

Propiedad IV: Los *process* se ejecutan en paralelo entre sí.
Problema: Existirá con bastante probabilidad un código con dos o más *process*, en esos casos éstos se ejecutan en paralelo como ocurre con el resto de las sentencias. Si dos *process* se están ejecutando en paralelo no se puede modificar la misma señal en ambos *process*, porque en ese caso no se podría saber cuál es el valor real de la señal (¿el obtenido en el *process* 1 ó el obtenido en el *process* 2?).
Solución: Siempre hay que comprobar que no se modifica la misma señal en dos *process* diferentes, en caso de que esto ocurra habrá que intentar fusionar los dos *process*.

Propiedad V: Los valores de las señales que se modifican internamente en los *process* no se actualizan hasta que no se ha ejecutado el *process* completo.
Problema: Como resultado de esta regla, si no se ha escrito correctamente la lista de sensibilidad podría ocurrir lo que se observa en el ejemplo "actualización 1", C va retardada con respecto a B, problema que se soluciona escribiendo correctamente la lista de sensibilidad.
Solución: **Si la asignación problemática se realiza dentro de una estructura *process* + sentencias que dan como lugar HW secuencial (ver capitulo 5) la solución pasa por sacar alguna asignación del *process* y llevarla a otro *process* nuevo independiente.**

```
-------------------------
-- Actualizacion 1
-------------------------
process (A)
begin
     C <= A;
     B <= C;
end process;
```

t	0 ns	5 ns	10 ns
A	0	1	0
B	0	0	1
C	0	1	0

```
-------------------------
-- Sensibilidad arreglada
-------------------------
process (A, C)
begin
     C <= A;
     B <= C;
end process;
```

t	0 ns	5 ns	10 ns
A	0	1	0
B	0	1	0
C	0	1	0

3.4 Descripción Estructural

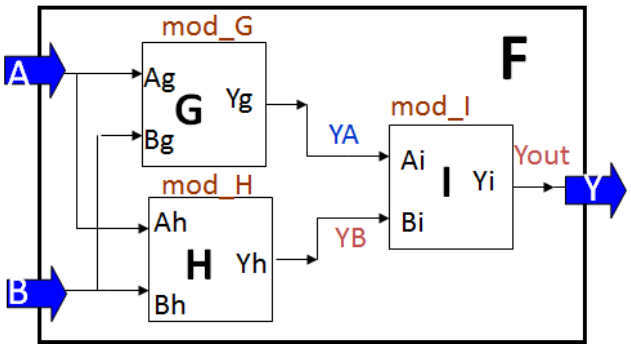
Esta descripción utiliza para la creación de la arquitectura de la entidad entidades descritas y compiladas previamente, de esta manera en VHDL podemos aprovechar diseños ya realizados, o realizar diseños sabiendo que se utilizarán en otros más complicados. Así se ahorra trabajo al diseñador-programador.

Se declaran los componentes que se van a utilizar y después, mediante los nombres de los nodos, se realizan las conexiones entre los puertos. Las descripciones estructurales son útiles cuando se trata de diseños jerárquicos *botton-up*.

```
architecture circuito of nombre is
   component subcircuito
        port (...);
   end component;
   -- señales
begin
   chip_i: subcurcuito port map (...);
   -- Se puede combinar con la descripción
   -- behavioral (por comportamiento)
end circuito;
```

Se pueden utilizar tantos componentes como necesite el diseño, estos componentes son entidades anteriormente declaradas o compiladas, en el ejemplo, *subcircuito* debe ser el nombre de una entidad que se ha declarado anteriormente en el código o que se ha compilado como parte de otro fichero VHDL. Se puede utilizar un componente tantas veces como sea necesario (podemos asignar *chip_i* y *chip_j* al mismo componente pero con distintas señales de entrada y salida).

Podemos realizar un diseño estructural puro, mediante la unión de componentes que describen las distintas funcionalidades del circuito, en el que necesitaremos definir las señales intermedias entre las que se encontrará siempre las señales que interconectan las salidas de los módulos con las salidas de la *entity*.



```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity F is
     port (A, B: in std_logic;
          Y: out std_logic);
end F;

architecture estructura of  F is
     component G
          port (Ag, Bg: in std_logic; Yg: out std_logic);
     end component;
     component H
          port (Ah, Bh: in std_logic; Yh: out std_logic);
     end component;
     component I
          port (Ai, Bi: in std_logic; Yi: out std_logic);
     end component;
     signal YA, YB, Yout: std_logic;
begin
     mod_G: G port map (A, B, YA);
     mod_H: H port map (A, B, YB);
     mod_I : I port map (YA; YB; Yout);
     Y<=Yout;
end estructura;
```

IMPORTANTE

Obsérvese en el ejemplo que la señal intermedia Y (interconexión entre salida de *component* y salida de la *entity*) es siempre necesaria, mientras que no son necesarias las señales intermedias entre las entradas de la *entity* y las entradas de los *components*.

Dentro de la descripción estructural de un circuito aparece la sentencia **GENERATE**, ésta se utiliza para poder generar de forma automática un array de interconexiones mediante instrucciones de asignación o componentes. Es decir, *generate*, permiten crear varias copias de un conjunto de interconexiones de manera análoga a la utilización de un *for* en otros lenguajes. La sentencia *for-generate* siempre va fuera de los *process*.

```
for indice in rango generate
   -- rango 0 to N o N downto 0
   -- instrucciones de asignación
   -- components
end generate;
```

El resultado final de la utilización de la sentencia *generate*, tanto en simulación como en síntesis, es el desenrollado del bucle y su ejecución como si fueran N asignaciones diferentes que se realizan en paralelo. A continuación se exponen dos ejemplos diferentes. Las etiquetas para marcar los bucles con *generate* (*gen1* y *gen2*) son opcionales.

```
signal a, b: std_logic_vector(0 to 7)
…
gen1: for i in 0 to 7 generate
   a(i) <= not b(i);
end generate gen1;

--------------------------------

component subcircuito
   port( x: in std_logic; y: out std_logic);
end component comp
…
signal a, b: std_logic_vector(0 to 7)
…
gen2: for i in 0 to 7 generate
   u: subcircuito port map(a(i),b(i));
end generate gen2;
```

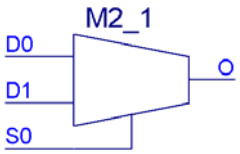
Se pueden encadenar tantos bucles como sea necesario para la correcta descripción del circuito. Además, La sentencia *generate* permite la utilización de la construcción *if* siempre que ésta vaya asociada al índice del *for* y no a ningún otro factor. A continuación se exponen dos ejemplos, en el primero (ejemplo correcto) el bucle se desenrollaría creando un desplazamiento hacia la derecha de *a* respecto *b* menos para el caso a(0) que en este trozo de código no estaría definido, este resultado de implementación sería siempre el mismo independientemente de los valores de las señales de entrada. En el ejemplo incorrecto el hardware que se obtendría depende del valor de la señal b, por lo que variaría dependiendo de los valores que se asignen a b, lo cual es imposible.

```
-------------------------------------
-- Ejemplo correcto
-------------------------------------
signal a, b: std_logic_vector(0 to 7)
…
bucle: for i in 0 to 7 generate
   condicion: if i > 0 generate
        a(i) <= b(i-1);
   end generate condicion;
end generate bucle;

-------------------------------------
```

```
-------------------------------------
-- Ejemplo incorrecto
-------------------------------------
signal a, b: std_logic_vector(0 to 7)
…
bucle: for i in 0 to 7 generate
   condicion: if b(i) = '0' generate
        a(i) <= b(i-1);
   end generate condicion;
end generate bucle;
-------------------------------------
```

3.5 Ejemplos

MULTIPLEXOR 2x1:
Un multiplexor 2x1 se correspondería con el siguiente módulo visto desde fuera:



Por lo que su entidad correspondiente sería:

```
entity mux2 is
    port (D0, D1, S0: in std_logic; O out std_logic);
end mux2;
```

La descripción de lo que hace internamente se puede realizar de varias maneras, así pues un multiplexor es un módulo que hace:

S0	O
0	D0
1	D1

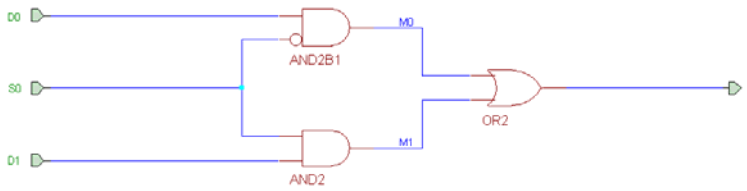
Lo cual podría corresponder perfectamente a la siguiente arquitectura:

```
architecture behavioral1 of mux2 is
begin
     O <= D1 when (S0 = '1') else D0;
end behavioral1;
```

O a esta otra:

```
architecture behavioral2 of mux2 is
begin
multiplexor: process(D0,D1,S0)
     if(S0 = '1') then
          O <= D1;
     else
          O <= D0;
     end if;
end process;
end behavioral2;
```

También podemos saber mediante la realización de la tabla de verdad como está diseñado por dentro un multiplexor:



Lo que daría como resultado la siguiente arquitectura si lo hacemos de manera estructural:

```
architecture structural of mux2 is
-- declaración de componentes
     component AND2
          port (I0,I1: in std_logic; O: out std_logic);
     end component;
     component OR2
          port (I0,I1: in std_logic; O: out std_logic);
     end component;
     component INV
          port (I0: in std_logic; O: out std_logic);
     end component;
-- declaración de señales
     signal S1,S2,S3: std_logic;
begin
     U1: INV port map (S0,S1);
     U2: AND2 port map (D0,S1,S2);
     U3: AND2 port map (S0,D1,S3);
     U4: OR2 port map (S2,S3,O);
end structural;
```

O esta otra si lo hacemos por comportamiento:

```
architecture mixed of mux2 is
     signal S1,S2: std_logic;
begin     S1 <= D0 and not S0;
     S2 <= D1 and S0;
     O <= S1 or S2;
end mixed;
```

IV. Simulación en VHDL

VHDL realiza la simulación siguiendo la técnica de **simulación por eventos discretos** (*Discrete Event Time Model*). Esta es una técnica que permite avanzar el tiempo a intervalos variables, en función de la planificación de ocurrencia de eventos (cambio de valor de alguna señal). Esto significa que no se simula el comportamiento del circuito pico-segundo a pico-segundo, si no desde que ocurre un evento hasta el siguiente, donde puede pasar un pico-segundo o varios segundos. Durante el intervalo de tiempo en el que no se produce ningún evento se mantiene el valor de todas las señales.

4.1 Fases de la simulación

La simulación consta de tres fases:

- Fase 0: la simulación comienza en la fase de inicialización donde a las señales se les asignan unos valores iniciales y se pone el tiempo a cero. La asignación se hace rellenando una lista de eventos para el instante t = 0.
- Fase 1: todas las transiciones planificadas para ese tiempo son ejecutadas. Es decir, se ejecuta el código ordenadamente teniendo en cuenta cuales son las señales que se han modificado, cumpliendo las normas de ejecución explicadas para los *process*.
- Fase 2: Las señales que se han modificado como consecuencia de las transiciones planificadas en el instante t se escriben en la lista de eventos planificándose para el instante t + δ. Donde δ es un instante infinitesimal.

Se repiten la fase 1 y 2 hasta que no existen más transiciones. Además en los instantes entre eventos se mantienen los valores de las señales.

A continuación para ilustrar como se realiza la simulación se describirán 3 ejemplos.

El primer y el segundo ejemplo simularemos asignaciones fuera del *process*, donde A tiene valor 0 en el instante 0 ns y valor 1 en el instante 5 ns.

```
-- asignaciones concurrentes
     B <= A;
     C <= B;
```

t (ns)	0	0 + δ	No hay más cambios	5	5 + δ	No hay más cambios
A	0	0		1	1	
B	U	0		0	1	
C	U	0		0	1	

```
-- asignaciones concurrentes
     C <= B;
     B <= A;
```

t	0	0 + δ	0 + 2δ	No hay más cambios	5	5 + δ	5 + 2δ	No hay más cambios
A	0	0	0		1	1	1	
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	

En el tercer ejemplo se muestran dos asignaciones dentro de un *process* con lista de sensibilidad.

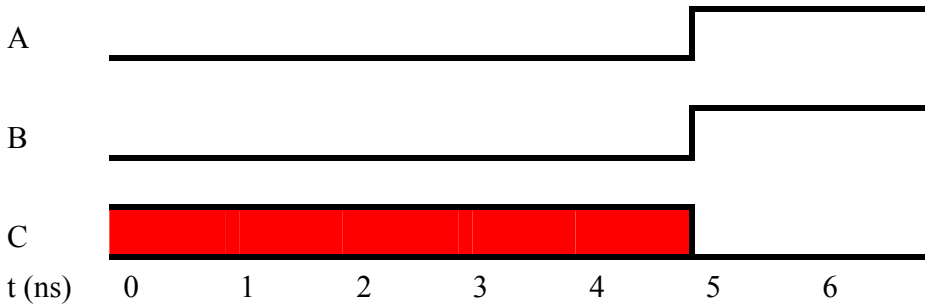
```
process (A)
begin
     B <= A;
     C <= B;
end process;
```

t	0	0 + δ	No hay más cambios	5	5 + δ	No hay más cambios
A	0	0		1	1	
B	U	0		0	1	
C	U	U		U	0	

En el instante 0 ns se entra dentro del *process* y se asigna el valor de A a B y el de B (valor que tenía cuando se entró al *process*) a C y se escriben los cambios para el siguiente paso de simulación. En este siguiente paso de simulación 0 + δ no ha cambiado A por lo que no se entra al *process* y se mantienen los valores anteriores. No se vuelve a realizar la simulación hasta el instante 5 ns que es cuando vuelve a cambiar una señal, en este caso A. Como cambia A se vuelve a entrar al *process* y se asigna el valor de A a B y el de B (valor que tenía cuando se entró al *process*) a C y se escriben los cambios para el siguiente paso de simulación. En este siguiente paso de simulación 5 + δ no ha cambiado A por lo que no se entra al *process* y se mantienen los valores anteriores. En la siguiente figura se puede observar lo que veríamos gráficamente en cualquier simulador de VHDL. Nótese que δ es infinitesimal y por lo tanto no es visible en simulación.

Resultados para los dos primeros ejemplos

Resultados para el ejemplo con process



4.2 Sentencias de simulación

VHDL presenta una sentencia específica, **WAIT**, que detiene la ejecución del código hasta que se cumpla una condición. La sentencia *wait* debe aparecer obligatoriamente si el *process* no tiene lista de sensibilidad. Además en muchos tutoriales se utiliza para generar HW secuencial de manera análoga a como se va a explicar en el siguiente capítulo con la sentencia *if*.

A continuación se describen las expresiones comúnmente utilizadas con la sentencia wait:

`wait on lista_de_señales;`	No se ejecutan las instrucciones posteriores hasta que no se modifique (a cualquier valor) alguna de las señales de la lista.
`wait for tiempo;`	No se ejecutan las instrucciones posteriores hasta que no pase el tiempo indicado desde que se llegó a la instrucción de *wait*.
`wait until condicion;`	No se ejecutan las instrucciones posteriores hasta que no se cumpla la condición.

Utilizaremos el siguiente código para ilustrar el comportamiento de la sentencia *wait*:

```
process
begin
     B <= A;
     wait until A = '1';
     C <= B;
end process
```

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

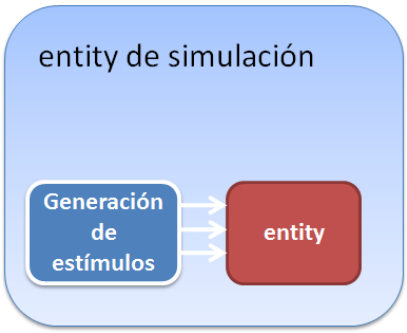
En el ejemplo el valor de C no se puede actualizar hasta que A no valga 1, cosa que no sucede hasta el instante 5ns. Como resultado en un *process* con *wait* se ejecutan siempre todas las instrucciones anteriores al *wait* y sólo se ejecutan **todas las instrucciones** posteriores al *wait* cuando se cumpla la condición.

4.3 Plantillas para simulación en VHDL

Muchas herramientas de simulación y síntesis con VHDL tienen un asistente gráfico para crear los estímulos (valores de las entradas) necesarios para comprobar que nuestro diseño funciona correctamente. Sin embargo, a nivel profesional son los mismos diseñadores los que crean un fichero con los estímulos que quieren probar para comprobar que su diseño funciona correctamente.

Independientemente de que tenga un asistente gráfico o no, lo que termina haciendo la herramienta es creando un fichero vhdl de simulación con lo siguiente (ver figura):

- Se crea una *entity* para la simulación sin puertos de entrada ni puertos de salida.
- Se instancia un *component* que se corresponde con la *entity* del diseño a examinar.
- Se generan dentro de un *process* y con la ayuda de la sentencia *wait* los valores de las señales de entrada y los tiempos en los cuales se van a modificar.



A continuación describimos cada una de las partes del fichero de simulación.
El fichero de simulación tiene obligatoriamente las siguientes bibliotecas:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use STD.TEXTIO.ALL;
```

Además se crea la *entity* simulación sin señales de entrada ni de salida:

```
entity simulacion is
end simulacion;
```

A continuación se instancia la arquitectura, que debe tener como componente la *entity* a estudio, se definen como señales intermedias las entradas y salidas de la *entity* a

estudio. Y se escribe al menos un *process*, donde se describa como cambian los valores de las señales a lo largo del tiempo. Nótese que sólo hay que forzar el cambio de los valores de las señales intermedias que se corresponden con entradas de la *entity* a estudio, ya que el valor de las salidas será el resultado de la simulación.

```
architecture testbench_arch of simulacion is

     component circuito
     port (entrada: in std_logic; …
          salida: out std_logic);
     end component;

     -- señales intermedias, mismos nombres y tipo que los
     -- de circuito
     signal entrada: std_logic := '0';
     …
     signal salida: std_logic;
     -- las señales out no se inicializan

begin

     UUT : circuito port map (entrada, …, salida);

     process
     begin

          wait for 200 ns;
          entrada <= '1';
          …
          ----------------------------------------
          wait for 100 ns;    -- Total: 300 ns
          entrada <= '0';
          …
          ----------------------------------------
          wait for T ns;      -- Total: 300 + T ns
          entrada <= '1';
          …
          ----------------------------------------
          wait for …
          …
          ----------------------------------------
          wait for 100 ns;

     end process;

end testbench_arch;
```

La función del primer *wait* que aparece en el código (`wait for 200 ns;`) es mantener los valores iniciales durante 200 ns. Ya que no se ejecutara ninguna instrucción posterior, asignaciones a *entrada* y *wait*, hasta que no se cumpla la condición del primer *wait*. Una vez han pasado 200 ns se ejecutan las asignaciones posteriores hasta encontrarnos con el siguiente wait (`wait for 300 ns;`). No se ejecutarán las instrucciones posteriores a ese *wait* hasta que no hayan pasado 100 ns mas, es decir, hasta que el tiempo total no sea 300 ns, y así sucesivamente. El último *wait* nos garantiza que los últimos valores asignados no se modifican hasta pasados 100 ns más.

El siguiente código y su figura asociada ilustra el resultado de aplicar las reglas anteriormente escritas sobre la señal A (inicializada a 0):

```
process
begin

    wait for 5 ns;
    A <= '1';
    ----------------------------------------
    wait for 5 ns;      -- Total: 10 ns
    A <= '0';
    ----------------------------------------
    wait for 10 ns;     -- Total: 20 ns
    A <= '1';
    ----------------------------------------
    wait for 5 ns       -- Total: 25 ns
    A <= '0';
    ----------------------------------------
    wait for 10 ns;

end process;
```

Dentro del fichero de simulación existe también un *process* particular para definir el reloj de una manera rápida y sencilla.

```
process
   begin
       wait for 10 ns;
       CLOCK_LOOP : loop
           clk <= '0';
           wait for tiempo_en_baja ns;
           clk <= '1';
           wait for tiempo_ en_alta ns;
       end loop CLOCK_LOOP;
end process;
```

Este *process*, gracias a la sentencia *loop*, genera una señal que varía constantemente de valor según los tiempos que se impongan. El primer *wait* tiene el mismo efecto que en el ejemplo anterior, mantiene el valor inicial durante 10 ns. El segundo wait indica cuanto tiempo va a estar el reloj en baja. Y el último wait indica cuanto tiempo va a estar el reloj en alta antes de repetir el *loop*. Por lo tanto *periodo = tiempo_en_baja + tiempo_en_ alta*, en caso de que se quiera un reloj simétrico habrá que poner el mismo valor de tiempo a baja que a alta.

V. Descripción de Lógica Secuencial

Una de las propiedades más importantes de un *process* es la capacidad de la estructura para almacenar los valores de las señales que se asignan en su interior si durante el paso de simulación no se entra dentro del *process* o no se realiza ninguna asignación a esa señal. Debido a esta característica se utilizarán los *process* para generar HW secuencial.

IMPORTANTE

Que un *process* genere HW secuencial no implica que las distintas instrucciones internas a un *process* se ejecuten secuencialmente.

5.1 Hardware Secuencial

Para la descripción de biestables y registros utilizaremos *process* en los que la señal de reloj CLK actúe por flanco conjuntamente con un *if* sin rama *else*.

```
if (CLK'event and CLK='1') then …
```

Así pues si se quiere representar un biestable deberíamos añadir el siguiente process:

```
entity Biestable_D is
     port(d, clk: in std_logic; q: out std_logic);
end Biestable_D;

architecture ARCH of Biestable_D is
begin
     process (clk, d)
     begin
          if (clk'event and clk = '1') then q <= d;
          end if;
     end process
end ARCH;
```

Biestable tipo D con reset asíncrono. El proceso se activa cuando hay eventos en las señales de reset o clock como indica su lista de sensibilidad. La primera sentencia del proceso comprueba si la señal de reset está a 1. Si esta señal se ha activado el biestable se pone a cero. Si no se ha activado, el proceso funciona como el proceso descrito anteriormente.

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

architecture ARCH_ASYN of Biestable_rD is
begin
     process (clk, reset, d)
     begin
          if (reset = '1') then q <= '0';
          elsif clk = '1' and clk'event then q <= d;
          end if;
     end process;
end ARCH_ASYN;
```

Biestable tipo D con reset síncrono:

```
architecture ARCH_SYN of Biestable_rD is
begin
     process (clk, reset, d)
     begin
          if clk = '1' and clk'event then q <= d;
               if (reset = '1') then q <= '0';
               end if;
          end if;
     end process;
end ARCH_SYN;
```

Para conseguir crear el HW secuencial esperado hay que cumplir una serie de reglas:

- Una sentencia *if* que tenga por condición una especificación de flanco no puede tener rama *else,* en caso contrario la rama *else* debería realizarse en todo momento menos en el preciso instante en el que el reloj cambia.
- En sentencias *if-then-elsif* la especificación de flanco sólo podrá ser la condición del último *elsif* (que no podrá tener rama *else*).
- Una sentencia *if* que tenga por condición una especificación de flanco puede tener internamente sentencias *if-else* encadenadas.
- En un process solo puede existir una única especificación de flanco, en caso contrario se estaría especificando HW secuencial sensible a varios relojes.

El hecho de incluir una sentencia con especificación de flanco hace que todas las instrucciones que se escriban dentro de ese *if* formaran hardware secuencial, pero eso nunca significa que por defecto se ejecutarán de manera secuencial.

Se ilustra la manera de crear HW secuencial con los siguientes tres ejemplos. La diferencia entre el primer y el segundo ejemplo es el orden de asignación de los valores a *b* y *c*, en el primer caso esa asignación se haría como en cualquier lenguaje de programación, en el segundo caso parecería que la asignación es incorrecta. Para comprender los resultados hay que tener en cuenta dos propiedades de los *process* mencionadas con anterioridad:

- Todas las asignaciones dentro de un *process* se hacen en paralelo, por lo que el orden de las asignaciones no altera el resultado final.
- No se actualizan los valores de las señales modificadas internamente en el *process* hasta que no se han terminado de ejecutar todas sus instrucciones internas. En este caso esto se ve agravado por el hecho de que dentro del *if* sólo se entra en el momento exacto en el cual se produce el flanco, lo que se corresponde con un único δ de simulación.

```
-----------------------------------------------
-- Ejemplo 1
-----------------------------------------------
process (clk, a, b, reset)
begin
     if reset = '1' then
          b <= '0';
          c <= '0';
     elsif clk'event and clk ='1' then
          b <= a;
          c <= b;
     end if;
end process;



-----------------------------------------------
-- Ejemplo 2
-----------------------------------------------
process (clk, a, b, reset)
begin
     if reset = '1' then
          b <= '0';
          c <= '0';
     elsif clk'event and clk ='1' then
          c <= b;
          b <= a;
     end if;
end process;
-----------------------------------------------
```

flanco clk	0 ns	5 ns	10 ns
reset	1	0	0
a	1	1	1
Ejemplo 1			
b	0	1	1
c	0	0	1
Ejemplo 2			
b	0	1	1
c	0	0	1

En el tercer ejemplo *b* y *c* se asignan directamente a *a* y por lo tanto los valores de *b* y *c* se modifican igual y a la vez.

```
-----------------------------------
-- Ejemplo 3
-------------------------------------------
process (clk, a, b, reset)
begin
     if reset = '1' then
          b <= '0';
          c <= '0';
     elsif clk'event and clk ='1' then
          c <= a;
          b <= a;
     end if;
end process;
-------------------------------------------
```

flanco clk	0 ns	5 ns	10 ns
reset	1	0	0
a	1	1	1
b	0	1	1
c	0	1	1

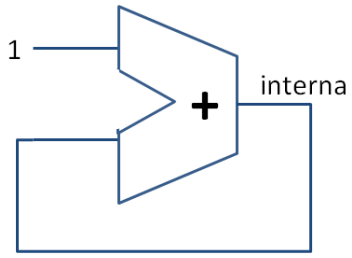
5.2 Contadores

Uno de los circuitos HW que más se utilizan en el diseño de sistemas es el contador. A la vez uno de los códigos VHDL que los diseñadores suelen escribir para describir el funcionamiento de un contador es el que se presenta a continuación. Sin embargo, ese código no funciona como un contador, ¿por qué?

```
entity contador is
     port (reset : in std_logic;
           numero : out std_logic_vector(3 downto 0));
end contador;

architecture circuito of contador is
signal interna: std_logic_vector(3 downto 0);
begin
     process (reset, interna)
     begin
          if (reset = '1')
               interna <= "0000";
          else
               interna <= interna + 1;
          end if;
     end process;
     numero <= interna;
end circuito;
```

En el circuito anterior en la rama *else* aparece `interna <= interna +1;` esa asignación escrita de esa forma jamás va a funcionar en VHDL ni a nivel de simulación, ni a nivel hardware.

t (ns)	0	0 + δ	5	5 + δ	5 + 2δ	5 + 3δ	5 + 4δ
reset	**1**	1	**0**	0	0	0	0
interna	**U**	**0**	0000	**0001**	**0010**	**0011**	**0100**



Un contador es un circuito que genera un número nuevo cada ciclo de reloj, en el código anterior el reloj no aparece en ninguna parte. En VHDL tenemos que escribir expresamente que se quiere un número nuevo cada ciclo de reloj, como muestra el siguiente código:

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

```
architecture circuito of contador is
signal interna: std_logic_vector(3 downto 0);
begin
     process (reset, clk, interna)
     begin
          if (reset = '1')
               interna <= "0000";
          elsif clk'event and clk = '1' then
               interna <= interna + 1;
          end if;
     end process;
     numero <= interna;
end circuito;
```

La señal interna está definida como un *std_logic_vector(3 downto 0*), aplicando las definiciones del estándar del ieee obtenemos que "1111" + 1 = "0000", por lo que el código anterior representa un contador que cuenta de 0 a 15 y vuelta a empezar.

> Para que Xilinx reconozca correctamente cualquier circuito secuencial es necesario que ese circuito presente un *reset*, ya sea asíncrono (como en los ejemplos) o síncrono.
> Se pueden diseñar contadores que cuenten hasta un determinado valor máximo y vuelvan a cero, o se queden ese valor, o que puedan partir de un valor cargado mediante una señal de load. Para describir todas estas funcionalidades se utilizarán sentencias *if-else* internas a la sentencia *elsif clk'event and clk = 1*

A continuación se presenta un contador genérico que cuenta hasta un valor máximo:

```
entity contador is
     generic (maximo: natural := max; N: natural := 8);
     port (reset, clk : in std_logic;
           numero: out std_logic_vector(N-1 downto 0));
end contador;

architecture circuito of contador is
signal interna: std_logic_vector(N-1 downto 0);
begin
     process (reset, clk, interna)
     begin
          if (reset = '1')
               interna <= (others<='0');
     -- Esta sentencia pone todos los bits de interna a cero
          elsif clk'event and clk = '1' then
               if interna < max
                    interna <= interna + 1;
               else
```

```
                    interna <= (others=>'0');
     -- Esta sentencia pone todos los bits de interna a cero
               end if;
          end if;
     end process;
     numero <= interna;
end circuito;
```

Del análisis del código anterior cumpliendo las reglas de simulación del *process*: ¿el contador cuenta hasta max o hasta max-1?

5.3 Ejemplos

REGISTRO DE 8 BITS

```
entity registro_8 is
   port (clk, reset: in std_logic;
        A: in std_logic _vector(7 downto 0);
        B: out std_logic _vector(7 downto 0));
end registro;

architecture arch_reg of registro_8 is
begin
process(clk, reset)
   begin
        if reset='1' then B<="00000000";
        elsif (clk'event and clk='1') then B<=A;
        end if;
   end process;
end arch_reg;
```

REGISTRO DE 8 BITS A PARTIR DE BIESTABLES DE 1 BIT

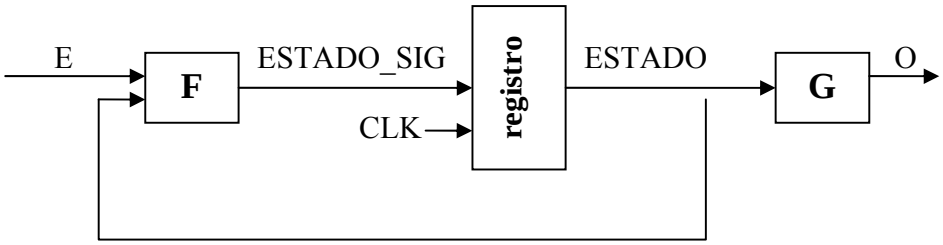
Definimos el registro de 8 bits como la unión de 8 biestables (descripción concurrente):

```
entity biestable is
     port (clk, reset, C: in std_logic;
          D: out std_logic);
end biestable;

architecture arch of biestable is
begin
process(clk, reset)
     begin
```

```
          if reset='1' then D<='0';
          elsif (clk'event and clk='1') then D<=C;
          end if;
     end process;
end arch;

entity registro_8 is
     port (clk, reset: in std_logic;
          A: in std_logic_vector(7 downto 0);
          B: out std_logic_vector(7 downto 0));
end registro_8;

architecture estructural of registro_8 is

component biestable
     port(clk, reset, c: in std_logic; d: out std_logic);
end component biestable;

signal F: bit_vector(7 downto 0);

begin
     gen: for i in 0 to 7 generate
          u: biestable port map(clk, reset, A(i),F(i));
     end generate gen;
     B <= F;
end estructural;
```

VI. Diseño de una Máquina de Estados

VHDL permite realizar descripciones algorítmicas de alto nivel de máquinas de estados. De esta forma, el diseñador se evita tareas como generar la tabla de transición de estados o la obtención de las ecuaciones de excitación basadas en un tipo de biestable.



Una Máquina de Estados Finita (FSM) se puede describir en VHDL de varias formas la que se propone en este capítulo es la forma estándar para cualquier herramienta que trabaje con VHDL.

En primer lugar en la sección de declaraciones de la arquitectura, se define un tipo enumerado en el que se asignan identificadores a cada estado. Suele ser recomendable utilizar identificadores ilustrativos para los estados. La herramienta de síntesis será la encargada de codificar estos estados.

```
type ESTADOS is (sube, baja, parado, …);
```

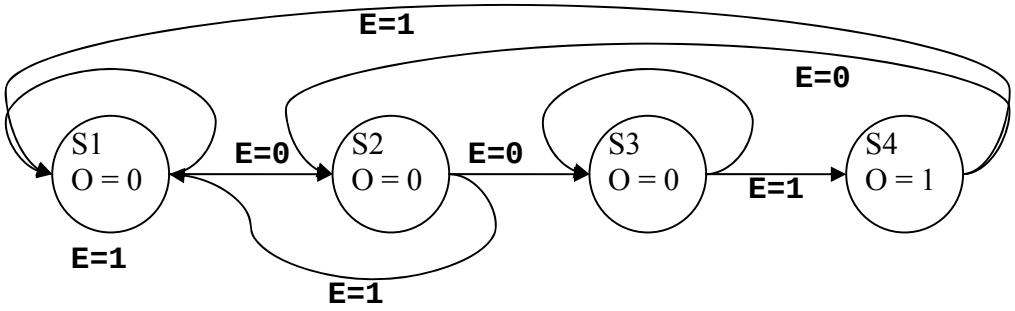
Posteriormente, en el cuerpo de la arquitectura se define la función de transición de estados (F) y la función de salida (G) en un *process* y el registro es decir el cambio de *estado_sig* a *estado* en otro *process*.

Por lo tanto tenemos:

- Un proceso secuencial que modela los biestables de estado; por lo tanto, que actualiza el estado (ESTADO).
- Un proceso combinacional que modela las funciones F y G; por lo tanto deriva el siguiente estado (ESTADO_SIG) y actualiza las salidas (O).

Para ilustrar la descripción de máquinas de estados vamos a pensar en un ejemplo Se trata de diseñar una máquina de estados que active una salida S cuando se detecta la secuencia "001" en una línea de datos E de un bit sincronizada con un reloj. Este detector de secuencia se puede realizar con una máquina de Moore de cuatro estados.

— S1: Esperar el l^{er} Cero de la secuencia.
— S2: Esperar el 2° Cero de la secuencia.
— S3: Esperar el uno de la secuencia.
— S4: Activar la salida.



Para la implementación en VHDL. Primero se define un tipo enumerado, formado por los nombres de los estados y se declaran dos señales de este tipo:

```
type ESTADOS is (S1, S2, S3, S4);
signal ESTADO, ESTADO_SIG: ESTADOS;
```

A continuación creamos un proceso combinacional en el que se determina el siguiente estado (ESTADO_SIG) y la salida S en función del estado actual (ESTADO, E).

El programa quedaría al final de la siguiente manera:

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

entity FSM is
      port(reset, E, clk: in std_logic; O: out std_logic);
end FSM;

architecture ARCH of FSM is

type ESTADOS is (S1, S2, S3,S4);
signal ESTADO, SIG_ESTADO: ESTADOS;
begin

SINCRONO: process(clk,reset)
begin
            if reset ='1' then
                  ESTADO<=S1;
            elsif clk'event and clk='1' then
                  ESTADO<= SIG_ESTADO;
            end if;
end process SINCRONO;
```

```
COMBINACIONAL: process(ESTADO,E)
begin
            case ESTADO is
            when S1 =>
                  O <= '0';
                  if (E='0') then
                        SIG_ESTADO<=S2;
                  else
                        SIG_ESTADO<=S1;
                  end if;
            when S2 =>
                  O <= '0';
                  if (E='0') then
                        SIG_ESTADO<=S3;
                  else
                        SIG_ESTADO<=S1;
                  end if;
            when S3 =>
                  O <= '0';
                  if (E='0') then
                        SIG_ESTADO<=S3;
                  else
                        SIG_ESTADO<=S4;
                  end if;
            when S4 =>
                  O <= '1';
                  if (E='0') then
                        SIG_ESTADO<=S2;
                  else
                        SIG_ESTADO<=S1;
                  end if;
end case;
end process COMBINACIONAL;
end ARCH;
```

F

G

Puede ocurrir que Xilinx no reconozca la máquina de estados, en ese caso borrará muchas de las señales intermedias y agrupará condiciones. Para conseguir que nos reconozca la máquina de estados hay que cumplir dos normas:

- Siempre debe existir un *reset* para incializar la máquina de estados.
- Dentro del case del *process* combinacional siempre se debe dar un valor a *estado_sig* bajo cualquier condición (aunque pueda parecer redundante).

VII. Funciones, Procedimientos y Paquetes

VHDL permite utilizar dos tipos de subprogramas, que ayudan a mejorar la descripción, estructuración y legibilidad de los modelos, estos subprogramas deben definirse dentro de una estructura particular que se denomina paquete.

```
package paquete is
function fnombre (señales_de_entrada) return tipo;
procedure pnombre (señales_de entrada; señales de salida);
end paquete;

package body paquete is
...
end paquete;
```

Dentro de cuerpo del paquete aparecen:

- **Funciones:** Realizan un cálculo puntual, devuelven un valor sin que corra el tiempo.
    - No pueden modificar los parámetros que se les pasan (todos serán in).
    - No pueden modificar señales ni variables declaradas externamente.
    - Siempre devuelven un valor cuyo tipo se especifica en la propia declaración de la función.
    - Se ejecutan en tiempo nulo, por lo que no pueden contener ninguna sentencia *wait*.

```
function identificador(…) return tipo
-- señales, variables
begin
-- cuerpo del programa
-- se puede utilizar cualquier sentencia
-- propia de VHDL
return valor;
end function identificador
```

- **Procedimientos:** Pequeñas descripciones de "circuitos".
    - Posibilidad de intercambio bidireccional con el exterior
    - Posibilidad de incluir una sentencia wait
    - Posibilidad de efectuar asignaciones a señales
    - Se define en la zona de declaraciones de la arquitectura

```
procedure nombre(parámetros)
-- señales, variables
begin
-- cuerpo del procedimiento
end procedure nombre;
```

7.1 Ejemplo

SUMA DE DOS VECTORS DE BITS UTILIZANDO PAQUETES

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

package ope_aritmeticas is

      function vector_to_natural (v:in std_logic_vector) return natural;
      function natural_to_vector (nat : in natural; length : in natural)
                                 return std_logic_vector;
      procedure vector_add ( v1, v2 : in std_logic_vector;
                             v_result : out std_logic_vector);

end ope_aritmeticas;

package body ope_aritmeticas is

   function vector_to_natural (v:in std_logic_vector) return natural is
        variable aux : natural:=0;
        begin
            for i in v'range loop
                  if v(i)='1' then
                        aux := aux + (2**i);
                  end if;
            end loop;
            return aux;
    end vector_to_natural;

    function natural_to_vector (nat : in natural; length : in natural)
                               return std_logic_vector is

        variable v: std_logic_vector(length-1 downto 0);
        variable cociente, aux, i, resto: natural;
        begin
            aux:= nat;
            i:=0;
            while (aux/=0) and (i<length) loop
               cociente := aux/2;
               resto := aux mod 2;
               if resto=0 then v(i):='0';
               else v(i):='1';
               end if;
               i := i+1;
               aux := cociente;
            end loop;
            for j in i to length-1 loop
               v(j):='0';
            end loop;
            return v;
    end natural_to_vector;
```

```
    procedure vector_add ( v1, v2 : in std_logic_vector;
                           v_result : out std_logic_vector) is
      variable suma,long: natural;
            begin
            long:=v1'length;
            suma:= vector_to_natural(v1) + vector_to_natural(v2);
            v_result := natural_to_vector(suma,long);
    end vector_add;
end ope_aritmeticas;

--------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.ope_aritmeticas.all; -- Para poder utilizar el paquete

entity sum is
    port (v1,v2: in std_logic_vector;
          v_result : out std_logic_vector);
end sum;

architecture beh of sum is
begin
   p1: process(v1, v2)
    variable suma: natural;
    begin
      vector_addu(v1,v2,suma);
      v_result<= suma;
     end process p1;
end beh;
```

VIII. Ejemplo: Diseño de una Memoria RAM

Utilizando todos los conceptos descritos hasta el momento podemos diseñar en VHDL el tipo de memoria más utilizado en los problemas de diseño de circuitos y computadores, una memoria RAM de escritura síncrona y lectura asíncrona:

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


-- Memoria RAM de 32 palabras de 8 bits

entity ram is
        port (addr: in std_logic_vector (4 downto 0);
                we, clk: in std_logic;
                data_i: in std_logic_vector(7 downto 0);
                data_o: out std_logic_vector(7 downto 0));
end ram;

architecture archxi of ram is

type ram_table is array (0 to 31) of std_logic_vector(7 downto 0);
signal rammemory: ram_table;

begin
        process(we, clk, addr)
        begin
                if clk'event and clk='1' then
                        if we='1' then
                                rammemory(conv_integer(addr))<=data_i;
                        end if;
                end if;
        end process;
        data_o <= rammemory(conv_integer(addr));
end archxi;
```

La función *conv_integer* viene dentro del paquete *ieee.std_logic_unsigned.all*, dado un número representado como un vector binario puro lo convierte en un número entero. Nótese que en VHDL el índice de acceso al array es un entero.

> Xilinx reconoce el código anterior como una memoria, pero no lo sintetiza en los bloques reservados para tal fin (BLOCK_RAMS). Para que eso ocurra la memoria tiene que ser síncrona, tanto en lectura como en escritura. Es decir, la asignación a *data_o* tiene que estar dentro de *if clk'event and clk='1' then*.

Utilizando todos los conceptos descritos hasta el momento podemos diseñar en VHDL una memoria RAM síncrona genérica con un puerto de lectura y otro de escritura,

una señal de *enable*, otra de *write* y otra de *read*:

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


entity SRAM is
generic(        w:      integer:=4; -- ancho de palabra
                d:      integer:=4; -- nº de palabras
                a:      integer:=2); -- ancho dirección
port(   Clock:          in std_logic;
        Enable:         in std_logic;
        Read:           in std_logic;
        Write:          in std_logic;
        Read_Addr:      in std_logic_vector(a-1 downto 0);
        Write_Addr:     in std_logic_vector(a-1 downto 0);
        Data_in:        in std_logic_vector(w-1 downto 0);
        Data_out:       out std_logic_vector(w-1 downto 0)
);
end SRAM;

architecture behav of SRAM is

-- Utilizamos un array para guardar los valores de la memoria
type ram_type is array (0 to d-1) of std_logic_vector(w-1 downto 0);
signal tmp_ram: ram_type;

begin

    -- Lectura
    process(Clock, Read)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
                if Read='1' then
                    Data_out <= tmp_ram(conv_integer(Read_Addr));
                else
                    Data_out <= (Data_out'range => 'Z');
                    -- Todos los bits de Data_out se ponen a 'Z'
                end if;
            end if;
        end if;
    end process;

    -- Escritura
    process(Clock, Write)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
                if Write='1' then
                    tmp_ram(conv_integer(Write_Addr)) <= Data_in;
                end if;
            end if;
        end if;
    end process;
```

```
end behav;
```

Apéndices

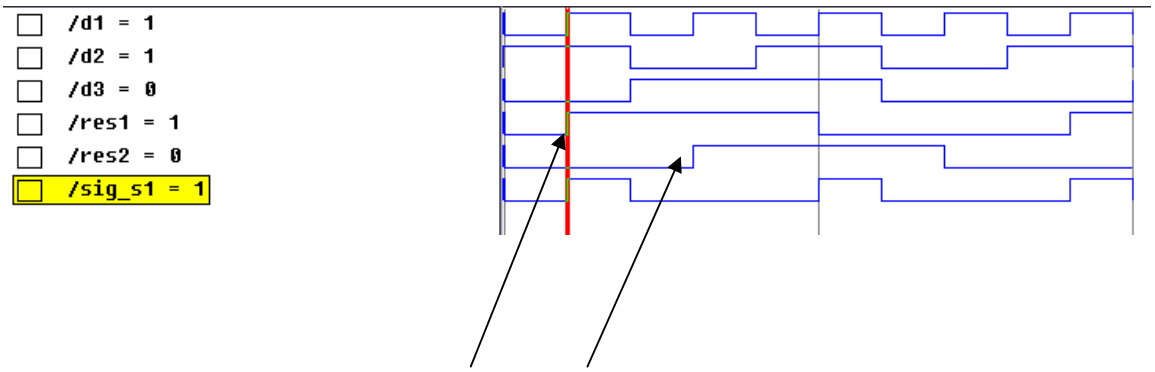
A.1 Discusión utilización de señales frente a utilización de variables

Las señales se utilizan para conectar los componentes del diseño llevando la información del "estado" actual del circuito a simular. Por otro lado las variables son utilizadas dentro de los procesos para computar valores particulares. El siguiente ejemplo muestra la diferencia:

```
entity sig_var is
port( d1, d2, d3:  in std_logic;
      res1, res2:  out std_logic);
end sig_var;

architecture behv of sig_var is

  signal sig_s1: std_logic;

begin

  proc1: process(d1,d2,d3)

    variable var_s1: std_logic;

  begin
      var_s1 := d1 and d2;
      res1 <= var_s1 xor d3;
  end process;

  proc2: process(d1,d2,d3)
  begin
      sig_s1 <= d1 and d2;
      res2 <= sig_s1 xor d3;
  end process;

end behv;
```

Aparentemente los dos procesos deberían ofrecer el mismo resultado pues realizan las mismas operaciones

El resultado de la simulación es el siguiente:



¿Por qué res2 se pone a '1' más tarde que res1?

A.2 Discusión efecto de la lista de sensibilidad en la implementación final de Xilinx

Xilinx cuando implementa un circuito no tiene en cuenta la lista de sensibilidad de los *process*, ya que si la tuviera en cuenta se producirían diseños HW poco estables. Esto implica que si no se escribe correctamente el código VHDL se obtendrán resultados distintos entre simulación e implementación final.
Como ya se ha indicado:

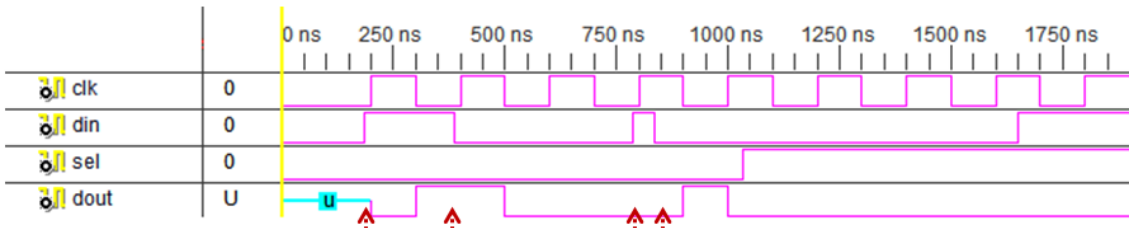
En la lista de sensibilidad de un process deben aparecer todas las señales que son leídas dentro de él

A continuación se presenta un ejemplo para comparar los resultados de la simulación con los resultados reales de la implementación.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity das is
     port(
     din, sel, clk : in std_logic;
     dout: out std_logic);
end das;
```

```
architecture Behavioral of das is
signal A, B, C: std_logic;
begin
     Tipo_C: process(clk)
     begin
          dout <= not C;
     end process Tipo_C;
     Tipo_B: process(clk)
     begin
          if clk'event and clk='1' then
          B <= not din;
          end if;
     end process Tipo_B;
     Tipo_A: process(clk)
     begin
          A <= not din;
     end process Tipo_A;

     C <= A  when (sel = '0') else B;

end Behavioral;
```

Simulación del código



Simulación tras la síntesis e implementación