

## 2- Características generales del VHDL

En los apartados que se exponen a continuación se revisan las características generales del lenguaje VHDL en referencia a su léxico, clases, formato y tipos de datos, así como aquellos conceptos significativos del lenguaje por su orientación para modelar *hardware*.

### 2.1 LÉXICO DE FICHEROS VHDL

Una descripción VHDL puede constar de uno o varios ficheros. Cada fichero debe estar constituido por el siguiente conjunto restringido de elementos del código ASCII-7:

Mayúsculas	: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Minúsculas	: a b c d e f g h i j k l m n o p q r s t u v w x y z
Números	: 0 1 2 3 4 5 6 7 8 9
Caracteres especiales	: " # & ' ( ) * + , - . / : ; < = > _ ! \$ % @ ? [ \ ] ^ ` { } ~
Códigos de formato	: Espacio (20), LF(0A), CR(0D), FF(0C), HT(09), VT(0B)

Con esos elementos se pueden confeccionar secuencias de caracteres que forman elementos fundamentales que no pueden fraccionarse en otros. En VHDL no se diferencia entre mayúsculas y minúsculas. Los ficheros se componen de esos elementos fundamentales denominados *elementos léxicos*, de los que existen varios tipos :

#### 2.1.1 SEPARADORES

Se permite un número variable de separadores entre *elementos léxicos* adyacentes, antes del primero y después del último en una línea de texto. Son los códigos de formato.

#### 2.1.2 DELIMITADORES

Caracteres que se usan para separar *elementos léxicos* y que tienen algún significado específico en VHDL . Son los caracteres :

& ' ( ) \* + , - . / : ; < = > |

por ejemplo, todas las sentencias en VHDL terminan con el carácter delimitador ; .

Cuando se habla de delimitadores compuestos se hace referencia a secuencias de delimitadores con un significado especial en VHDL, por ejemplo :

=>    \*\*    :=    /=    >=    <=    <>    --

#### 2.1.3 COMENTARIOS

Son *elementos léxicos* precedidos del delimitador -- y que finalizan con la línea donde están. Los comentarios son el último *elementos léxico* de una línea. Pueden seguir a una sentencia VHDL existente en la línea o ser el único *elemento léxico* en la línea.

código VHDL    -- comentarios precedidos de doble guión

#### **2.1.4 IDENTIFICADORES**

Son *palabras reservadas* en VHDL o bien los nombres que el diseñador asigna a los objetos que utiliza en sus descripciones, por ejemplo, para nombrar constantes, variables, señales, entidades, arquitecturas, bloques, subprogramas, etc.

Los *identificadores* son secuencias de caracteres que empiezan por una letra, pueden incluir letras, números y caracteres de subrayado aislados, es decir, no puede haber dos \_\_ seguidos, sino solo uno\_.

En el código VHDL no se diferencian letras mayúsculas y minúsculas, salvo que estén entre comillas, en cuyo caso son elementos diferentes de cadenas literales. Debe evitarse el uso de palabras acentuadas, incluso en los comentarios.

#### **2.1.5 PALABRAS RESERVADAS**

Son elementos léxicos con algún significado especial en las expresiones del lenguaje, por lo que no pueden ser utilizados fuera del contexto para el que están reservados.

En los ejemplos de código VHDL de este documento se presentan en MAYÚSCULAS. Son las siguientes :

<b>abs</b>	<b>else</b>	<b>linkage</b>	<b>procedure</b>	<b>then</b>
<b>access</b>	<b>elsif</b>	<b>literal*</b>	<b>process</b>	<b>to</b>
<b>after</b>	<b>end</b>	<b>loop</b>	<b>pure*</b>	<b>transport</b>
<b>alias</b>	<b>entity</b>			<b>type</b>
<b>all</b>	<b>exit</b>	<b>map</b>	<b>range</b>	
<b>and</b>		<b>mod</b>	<b>record</b>	<b>unaffected*</b>
<b>architecture</b>	<b>file</b>		<b>register</b>	<b>units</b>
<b>array</b>	<b>for</b>	<b>nand</b>	<b>reject*</b>	<b>until</b>
<b>assert</b>	<b>function</b>	<b>new</b>	<b>rem</b>	<b>use</b>
<b>attribute</b>		<b>next</b>	<b>report</b>	
	<b>generate</b>	<b>nor</b>	<b>return</b>	<b>variable</b>
<b>begin</b>	<b>generic</b>	<b>not</b>	<b>rol*</b>	
<b>block</b>	<b>group*</b>	<b>null</b>	<b>ror*</b>	<b>wait</b>
<b>body</b>	<b>guarded</b>			<b>when</b>
<b>buffer</b>		<b>of</b>	<b>select</b>	<b>while</b>
<b>bus</b>	<b>if</b>	<b>on</b>	<b>severity</b>	<b>with</b>
	<b>impure*</b>	<b>open</b>	<b>signal</b>	
<b>case</b>	<b>in</b>	<b>or</b>	<b>shared*</b>	<b>xnor*</b>
<b>component</b>	<b>inertial*</b>	<b>others</b>	<b>sla*</b>	<b>xor</b>
<b>configuration</b>	<b>inout</b>	<b>out</b>	<b>sll*</b>	
<b>constant</b>	<b>is</b>		<b>sra*</b>	
		<b>package</b>	<b>srl*</b>	
<b>disconnect</b>	<b>label</b>	<b>port</b>	<b>subtype</b>	
<b>downto</b>	<b>library</b>	<b>postponed*</b>		

Se han marcado con un asterisco, que no forma parte de la palabra, aquellas que han sido añadidas después de la revisión VHDL-1993.

### 2.1.6 CARACTERES LITERALES

Están formados por un solo caracter entre dos delimitadores apóstrofo (‘ ’).

‘X’ ‘x’ ‘%’ ‘”’ ‘Z’ ‘z’ ‘;’

Nótese que al ir entre apóstrofes, aquí sí se distingue entre ‘X’ y ‘x’.

### 2.1.7 CADENAS LITERALES

Similar al anterior, con secuencias de caracteres entre delimitadores comillas (“ ”).

“esto es una cadena”

Si la cadena excede una línea, deberán fraccionarse y concatenarse las fracciones por el delimitador &.

### 2.1.8 CADENAS BIT LITERALES

Son *elementos léxicos* formados por cadenas de dígitos de base hexadecimal (X), binaria (B), u octal (O), delimitados por comillas. Se pueden usar caracteres de subrayado aislados para separar cadenas largas de bits y mejorar la legibilidad del dato. Independientemente de la base utilizada para expresar la cadena, VHDL lo interpreta siempre como el valor de la cadena de bits, por ejemplo, X”A” se interpreta como “1010”.

La interpretación de las cadenas binarias queda al criterio del diseñador: “1010” puede interpretarse como (+10) o como (-6), en función del contexto del dato. Dos ejemplos de cadenas bits equivalentes son:

B”11110000”    B”1111\_0000”    X”F0”  
B”100011001”    B”100\_011\_001”    O”431”

### 2.1.9 LITERALES NUMÉRICOS

*Elementos léxicos* con un valor numérico asociado. Pueden ser enteros o reales.

La base de numeración es decimal o especificada con número decimal, entre 2 y 16.

- Los reales tienen punto (.). La coma (,) no está permitida.
- Con notación exponencial se utiliza E o e. El exponente es siempre en base 10.
- Solo se permiten exponentes negativos en números reales.
- El primer carácter de un número real debe ser un número decimal.
- Se puede utilizar un carácter de subrayado para mejorar legibilidad.
- No se permiten espacios entre caracteres.
- Los números significativos que siguen a la base especificada, van entre caracteres #.
- En base hexadecimal, los números superiores a 9 se expresan con A~F o a~f.

Ejemplos correctos:

7    2E5    007    2e1    262\_144  
3.1416    03.1416    0.31416E1    0.31416e+1    31.416 E-1  
255    0#255#    16#fF#    2#1111\_1111#

Ejemplos incorrectos:

1,000    .5    3E-2    5E -1    0.5 e0    16# fF #    2# 1111 1111 #

## **2.2 CLASES DE OBJETOS : DECLARACIÓN E INICIALIZACIÓN**

En VHDL, un *objeto* es un contenedor o portador en el que se pueden almacenar valores de cierto *tipo* .

Todos los objetos tienen un *tipo* que determina la clase de valor que puede asignarse al objeto. Al declararlos, se les asigna nombre y tipo. El nombre debe cumplir las reglas mencionadas para entidades y los tipos deben haber sido declarados previamente, salvo que sean predefinidos . Hay tres clases de objetos en VHDL :

### **2.2.1 CONSTANTES**

Tienen un valor fijo, asignado al compilarlas, no alterable durante la simulación. Deben ser declaradas con el siguiente formato :

```
CONSTANT nombre_de_constante : TIPO [: = valor inicial];
```

Ejemplos :

```
CONSTANT voltaje : REAL := 5.0;  
CONSTANT pulso : TIME := 100 NS;  
CONSTANT tres: BIT_VECTOR := "0011";
```

### **2.2.2 VARIABLES**

Son objetos normalmente utilizados como portadores temporales de valor alterable. Solo son declarables en áreas *secuenciales*, como en procesos y subprogramas. Deben declararse con el siguiente formato :

```
VARIABLE nombre1,.. nombre_n : TIPO [RESTRICCIONES][:= VALOR INICIAL];
```

Ejemplos :

```
VARIABLE temporal : INTEGER RANGE 0 TO 10 := 5;  
VARIABLE frecuencia, ganancia : REAL RANGE 1.0 TO 10.0;  
VARIABLE octeto: BIT_VECTOR (0 TO 7) := X"FF";
```

Las variables, a diferencia de las señales, no pueden pasar valores entre subprogramas.

### **2.2.3 SEÑALES**

Son portadores del valor de un parámetro común a varias unidades de diseño, utilizándose en descripciones para comunicar cambios del parámetro, normalmente asociados a un tiempo de simulación. Su relación con el hardware es evidente y, a diferencia de las variables, solo pueden ser declaradas en áreas *concurrentes*.

Su formato de declaración y algunos ejemplos son como sigue:

```
SIGNAL nombre : tipo [ restricciones ][:= valor inicial];  
SIGNAL clk : BIT := '0';  
SIGNAL databus : BIT_VECTOR( 7 DOWNT0 0) := B"0000_1111" ;  
SIGNAL dos_bytes : octales(0 TO 1) := (B"0000_1111", B"1111_1111") ;
```

Las señales no deben declararse dentro de Procesos, pero se usan dentro de ellos y como objetos para pasarles valores. Estos aspectos se verán al estudiar los Procesos .

## **2.3 ASIGNACIÓN DE VALORES**

Son expresiones utilizadas para cambiar el valor que tienen las variables y las señales.

### **2.3.1 ASIGNACIÓN A VARIABLES**

Las variables sustituyen *inmediatamente* el valor que tienen con el que se les asigna usando el delimitador compuesto (**:=**) :

```
octeto := b"1111_1110";
frecuencia := 2.0;
temporal := temporal + 1;
```

### **2.3.2 ASIGNACIÓN A SEÑALES**

La asignación *simple* de un valor a una señal indica el nuevo valor que será asignado en un instante futuro. Nótese que, a diferencia de la asignación *inmediata* que ocurre en variables, en las señales hay un retardo que, si no se especifica un valor determinado, tiene una duración infinitesimal denominada *retardo delta*.

El delimitador empleado para asignar valor a señales es (**<=**) :

```
clk <= '0';    clk <= '1' after 5 ns;    databus <= x"0f" after 10 ns;
```

El valor asignado a la señal debe ser del tipo con el que la señal fue declarada.

Dada la importancia del concepto *señal* en cualquier circuito a modelar, las sentencias de asignación de valor a señales son elementos clave y con importantes matices que se verán al estudiar con más detalle este tipo de sentencias.

## **2.4 TIPOS DE OBJETOS**

Un *tipo* es un conjunto de valores al que se le asigna un nombre que identifica al tipo.

El *tipo* de un *objeto* especifica los valores que puede tener y limita las operaciones que pueden realizarse con sus datos a aquellas definidas para el tipo y su rango de valores. En VHDL todos los datos que se utilizan tienen un tipo que es necesario especificar al declarar el objeto que lleva el valor. La compatibilidad entre tipos al asignar valores es un requisito en VHDL.

El empaquetamiento STANDARD, en la biblioteca STD, define tipos básicos como BIT ó INTEGER, pero el usuario de VHDL puede definir tipos y operadores específicos para la aplicación en uso, existiendo cuatro clases de tipos definibles en VHDL :

#### **1. tipos escalares**

Son tipos con valores simples y aislados.

Existen tres tipos escalares: ENUMERADOS, NUMÉRICOS y FÍSICOS.

#### **2. tipos compuestos**

Tienen valores compuestos por conjuntos de valores. Son los ARRAYS y RECORDS.

#### **3. tipos fichero**

Son tipos especiales utilizados para definir objetos relacionados con la escritura y lectura de ficheros o archivos de datos contenidos en el sistema donde se hace la simulación del modelo. Estos ficheros, cuyo formato puede ser especial o de texto, pueden utilizarse como entradas para simulación del modelo o como ficheros en los que se archivan las salidas del mismo. Por su relación al empaquetamiento TextIO, se verán en detalle al estudiar este paquete en una sección posterior.

#### **4. tipos acceso**

Se usan en diseños avanzados. Su estudio no se considera en este volumen.

#### **2.4.1 TIPOS ENUMERADOS**

Su declaración consiste en la enumeración simple de los valores que pueden tener. El empaquetamiento STANDARD contiene la declaración de los cuatro tipos de enumeración básicos de VHDL: boolean, bit, character y severity\_level..

Para declaraciones a nivel *flujo de datos*, donde se describen buses y sus señales de control, es necesario manejar señales que no pueden ser descritas con lógica binaria, por lo que se podría definir un tipo escalar que usara cuatro valores lógicos :

```
TYPE cuad IS ('0','1','Z','X');
```

donde '0' es el valor por defecto, 'Z' = 'alta impedancia' y 'X' = 'valor indefinido'.

Para poder utilizar el tipo cuad en las descripciones se puede :

- Declararlo en la parte declarativa de la arquitectura.
- Incluirlo en un empaquetamiento, al que se llamaría con USE para habilitarlo.

La 2ª opción es la más empleada normalmente en diseños.

#### **2.4.2 TIPOS NUMÉRICOS**

Denominación asignada a los tipos INTEGER y REAL del empaquetamiento STANDARD mostrado en Apéndice A. Los rangos máximos están definidos en el empaquetamiento, pero el usuario puede definir o limitar los rangos de estos dos tipos en sus descripciones, de forma similar a como lo hacen los subtipos NATURAL y POSITIVE también definidos en el mismo empaquetamiento STANDARD, por ejemplo :

```
TYPE centenas IS RANGE 0 TO 100;    TYPE probabilidad IS RANGE 0.0 TO 1.0
```

#### **2.4.3 TIPOS FÍSICOS**

Se pueden definir y declarar tipos relacionados con magnitudes físicas que son de uso frecuente en descripciones de hardware como resistencias, capacidades, frecuencia, potencia, etc. Un ejemplo se tiene en el tipo TIME, declarado en el empaquetamiento STANDARD por su utilidad en VHDL para modelar retardos. Para disponer de tipos físicos en modelos relacionados con resistencias y condensadores se podrían definir los tipos :

<pre> TYPE resistencia IS RANGE 0 TO 1E16 units   mo;      -- miliohms (unidad)   ohms = 1000 mo;   kohms = 1000 ohms;   mohms = 1000 kohms END units; </pre>	<pre> TYPE capacidad IS RANGE 0 TO 1E16 units   ffr;      -- femto faradios (unidad)   pfr = 1000 ffr;   nfr = 1000 pfr;   ufr = 1000 nfr; END units; </pre>
---	--

#### **2.4.4 ARRAYS**

Son tipos compuestos por elementos homogéneos, con igual subtipo. Un ejemplo se tiene en los tipos `STRING` y `BIT_VECTOR` del `PACKAGE.STD`. `STRING` es un array de caracteres y `BIT_VECTOR` es un array de bits. Los arrays se puede considerar y definir como tipos indexables y multidimensionales, pudiendo dejarse sin definir sus dimensiones o rangos. El formato y algunos ejemplos son como sigue :

```

TYPE nombre IS ARRAY (rangos o dimensiones) OF tipo de los elementos;
TYPE cuad_nibble IS ARRAY ( 3 downto 0 ) OF cuad;
TYPE cuad_byte IS ARRAY ( 7 downto 0 ) OF cuad;
TYPE cuad_word IS ARRAY (15 downto 0 ) OF cuad;
TYPE cuad_4por8 IS ARRAY ( 3 downto 0, 0 to 7) OF cuad;
TYPE cuad_2por4 IS ARRAY (1 downto 0, 0 to 3) OF cuad;

```

Nótese que los elementos del array pueden indexarse de dos formas, en las que se especifica el índice dentro del rango de ( izquierda a derecha ).

El array puede tener dimensiones abiertas, útil para describir diseños genéricos :

```

TYPE bit_vector IS ARRAY (natural range <>) OF bit;      --(ver PACKAGE.STD).

```

El delimitador compuesto `<>` indica que es un array con rango abierto, cuyos límites vendrán especificados posteriormente en la aplicación que use el tipo así declarado.

Pueden declararse tipos enumerados en relación a los índices de un array, por ejemplo:

```

TYPE orden IS ( uno, dos, tres, cuatro, cinco);
TYPE orden_de_array IS ARRAY ( orden RANGE uno TO cinco) OF INTEGER;

```

que nos indica que los valores almacenados de “uno” a “cinco” son enteros, que el primer elemento del array tiene por nombre “uno” y que con él nos podemos referir a ese elemento, sea para asignarle un valor o para leerlo después.

Una vez que los arrays están declarados, o llamado el empaquetamiento en que se hayan incluido, se puede asignar valores a los elementos de los arrays o utilizar estos elementos para asignar valores a objetos de tipo compatible, por ejemplo:

```

TYPE dispositivos IS (ram, micro, fpga, resistor, condensador, bobina);
TYPE array_activo IS ARRAY ( dispositivos RANGE ram TO fpga) OF INTEGER;
TYPE activos_pasivos IS ARRAY ( dispositivos RANGE <>) OF INTEGER;

```

permiten asignar valores a señales declaradas como :

```
SIGNAL pasivos : activos_pasivos (range resistor to bobina);  
SIGNAL databus : cuad_byte := "ZZZZZZZZ";
```

"ZZZZZZZZ" es el valor inicial forzado en la declaración, en lugar del valor "00000000" que tendría si no hubiese inicialización, ya que el valor por defecto del tipo `cuad` es '0'.

En los ejemplos que siguen pueden verse posibles formas de asignación de valores a señales relacionadas con los tipos array declarados anteriormente:

```
SIGNAL s1 :cuad;  
                                s1 <= s16 (3);  
SIGNAL s4 : cuad_nibble;  
                                s4 <= s16 (2) & s16(3) & s16(4) & s16(5);  
SIGNAL s8 : cuad_byte;  
                                s8 <= s16 ( 15 downto 8 );  
SIGNAL s16 : cuad_word;  
                                s16 (7 downto 0) <= s4 & s8( 3 downto 0 );
```

Además de la indexación por enteros, si la declaración del array especifica un tipo o tipos como indicación del rango discreto de ese array, la indexación está definida por una lista o tabla cuyos elementos estan indexados por filas y columnas ordenadas igual que los elementos de los tipos que describen los rangos discretos del nuevo tipo, al igual que los elementos  $a_{ij}$  de una matriz de  $i$  filas y  $j$  columnas:

```
TYPE cuad_2dim IS ARRAY ( cuad,cuad ) OF cuad;
```

este TYPE estará definido por una tabla de (4 x 4), siendo el orden de filas y columnas el definido para el tipo `cuad`, es decir, ('0','1','Z','X'), ('0','1','Z','X').

La inicialización de los valores de una señal del tipo array multidimensional se efectúa dentro de paréntesis anidados, separados por comas, en el mismo orden o dirección en que están declarados los elementos del array.

```
SIGNAL s_2_4 : cuad_2por4 := (('0','1','1','Z'), ('Z','0','0','0'));
```

En relación con los arrays, se utilizan a veces dos elementos que permiten reducir el código o mejorar su interpretación. Son los **agregados** y los **alias**.

Los **agregados** pueden considerarse arrays de literales, ya que con ellos puede especificarse un array y los valores de sus elementos. Su formato es :

```
identificador_de_tipo' ( [selección =>] expresión  
                        {, [selección =>] expresión } )
```

donde

- El `identificador_de_tipo` se refiere a cualquier tipo array de rango definido.
- La selección opcional se refiere al índice del elemento del array, una secuencia de los índices o la cláusula `others`.
- La expresión es el valor de los elementos del array. Algunos ejemplos son :



```

TYPE cinco_datos IS BIT_VECTOR ( 1 TO 5 );
SIGNAL dato      :      cinco_datos;
VARIABLE x, y    :      BIT;
dato <= cinco_datos'( '0', '1', x AND y, x OR y, '0');
dato(1) <= '0'; dato(2) <= '1'; dato(3) <= x AND y; dato(4) <= x OR y ;
cinco_datos'( '1', '0','0','0','1');      cinco_datos'( 2 TO 4 => '0', others => '1');
cinco_datos'( 3 =>'0', 2 ='0', 4 => '0', others => '1' );

```

Un *alias* es un identificador o nombre para una fracción de un array o como nombre alternativo del array. Se utilizan cuando se quieren distinguir campos o partes de un array o cuando éste se referencia indistintamente con dos identificadores, por ejemplo :

```

SIGNAL mensaje : BIT_VECTOR ( 15 DOWNT0 0 );
ALIAS cabeza   : BIT_VECTOR ( 2 DOWNT0 0 ) IS mensaje (15 DOWNT0 13);
ALIAS cuerpo   : BIT_VECTOR ( 7 DOWNT0 0 ) IS mensaje (12 DOWNT0 5);
ALIAS cola     : BIT_VECTOR ( 2 DOWNT0 0 ) IS mensaje ( 4 DOWNT0 2);
ALIAS test     : BIT_VECTOR ( 1 DOWNT0 0 ) IS mensaje ( 1 DOWNT0 0);

```

Algunas herramientas de síntesis no soportan los arrays multidimensionales, por lo que se hace necesario descomponerlos en arrays unidimensionales, por ejemplo :

```

CONSTANT memdim : INTEGER := 3;
TYPE memdato IS ARRAY ( 0 TO memdim, 7 DOWNT0 0 ) OF BIT;

```

el array bidimensional `memdato` puede sustituirse por dos arrays unidimensionales como

```

TYPE palabras   IS ARRAY ( 7 DOWNT0 0 ) OF BIT;
TYPE memdato    IS ARRAY ( 0 TO memdim ) OF palabras;

```

Con lo visto hasta aquí, y suponiendo que se tienen declarados los tipos anteriores apropiadamente en el paquete “modelos” sería posible modelar una sencilla memoria ROM como sigue :

```

USE WORK. modelos. ALL;
ENTITY memrom IS
    PORT (      direc : IN INTEGER ;
            datos : OUT palabras);
END memrom;

ARCHITECTURE minima OF memrom IS
    CONSTANT romdatos : memdato := (      ('0', '0', '0', '0', '0', '1', '1', '1'),
            ('1', '1', '1', '1', '1', '0', '0', '0'),
            ('0', '0', '0', '1', '1', '0', '0', '0'),
            ('1', '1', '1', '0', '0', '1', '1', '1') );

BEGIN
    datos <= romdatos( direc ) AFTER 25 ns;
END minima;

```

## 2.4.5 RECORDS

Son tipos compuestos por elementos de tipos diferentes o heterogéneos. Al igual que en los arrays, los elementos de los records forman conjuntos ordenados. Un ejemplo es un record para especificar posibles colores de una imagen con coordenadas limitadas:

```
TYPE color_tipo IS ( rojo, verde, azul, cian, amarillo, magenta);
TYPE datos_de_pixel IS RECORD
    abscisa : integer range 1 to 640;      -- declaración de campo
    ordenada : integer range 1 to 480;    -- declaración de campo
    color : color_tipo;                  -- declaración de campo
END RECORD;
```

Un posible *agregado* para un pixel de la imagen podría ser: (320, 218, cian).

Cada campo del RECORD define un área para contener datos que pueden ser leídos o escritos con la restricción que imponga el tipo previsto para el campo. Así, en un entorno de aplicación del ejemplo anterior se podrían tener las sentencias siguientes que ilustran sobre el formato de punto (.) *nombre de record . nombre de campo* para referirse a un campo :

```
VARIABLE pixel : datos_de_pixel;
VARIABLE horiz, verti : integer;
VARIABLE croma : color_tipo;

BEGIN  ----- separación entre áreas declarativa y activa

    horiz := pixel.abscisa;      -- lecturas de campos y asignación
    verti := pixel.ordenada;    -- de valor a variables
    croma := pixel.color;
    pixel.abscisa := 240;        -- escritura de campos con valor de variables
    pixel.ordenada := pixel.abscisa;
    pixel := (100, verti, rojo); -- asignación de valor con un agregado .
```

Los tipos record, al permitir agrupar en sus campos elementos de tipos diferentes, tienen una gran capacidad descriptiva para modelar diseños complejos. Un campo de record puede ser de tipo compuesto, es decir, un array de dimensiones definidas u otro record como se muestra en el ejemplo siguiente :

```
TYPE palabra IS ARRAY (0 TO 3) OF cuad;
TYPE mensaje IS ARRAY (31 DOWNT0 0) OF palabra;

TYPE direcciones IS RECORD
    origen : integer;
    clave : integer;
END RECORD;

TYPE paquetes IS RECORD
    dir : direcciones;
    datos : mensaje;
    datotest : entero;
END RECORD;
```

## **2.5 SUBTIPOS**

Son subconjuntos de los valores de un tipo predefinido, al que se llama tipo base, del cual se obtienen fijando alguna restricción. Los distintos subtipos obtenidos de una misma base son totalmente compatibles entre sí y con su base, lo que permite usar con ellos todas las funciones definidas para el tipo base.

Todos los tipos son subtipos de ellos mismos, por lo que *subtipo* se usa a veces para referirse a todos los tipos y subtipos declarados de una misma base.

Los subtipos se declaran de forma similar a los tipos, indicando después de la palabra clave SUBTYPE el nombre que se le asigna, seguido de IS y del nombre del tipo base del que se derivan, por ejemplo :

```
SUBTYPE cuarteto IS bit_vector ( 3 DOWNT0 0);
SUBTYPE decadas IS integer RANGE 0 TO 9;
```

otros ejemplos se tienen en el empaquetamiento STANDARD :

```
TYPE INTEGER IS RANGE -2147483648 TO 2147483647;
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;
```

Existen tres razones principales para el uso de subtipos :

- Limitar el número de “casos posibles” en ciertas sentencias, como en las de asignación selectiva de valor a señales o en la sentencia secuencial CASE.
- Facilitar la creación de *funciones de resolución* para la asignación de valor a una señal con múltiples *drivers*. Este tema se revisa como caso especial de funciones.
- Ser innecesaria la conversión de tipos al hacer asignaciones entre objetos con base común, por ejemplo, si tuviésemos definido un tipo como :

```
TYPE semibyte IS ARRAY ( 3 DOWNT0 0 ) OF BIT;
```

para asignar el valor de un objeto de ese tipo a otro del tipo BIT\_VECTOR, sería necesario hacer una conversión para pasar los elementos del tipo "semibyte" al tipo BIT\_VECTOR, lo cual sería innecesario si el objeto *driver* fuese del tipo “cuarteto” definido arriba.

La asignación de valores de un subtipo a objetos de su tipo base es siempre posible, ya que es un subconjunto de la base. Por razón similar, un valor del tipo base puede no ser asignable a un objeto del subtipo, por ejemplo, si se tienen los subtipos :

```
SUBTYPE triple IS cuad RANGE '0' TO 'Z' ;
SUBTYPE binar IS cuad RANGE '0' TO '1' ;
```

los objetos del subtipo "binar" son directamente asignables a objetos "triple" o "cuad" y viceversa, pero cuidando no provocar asignaciones que sobrepasen la dimensión definida para el subtipo destino, ya que se tendría un mensaje de error por asignación fuera de rango.

Nótese, por contra, que el subtipo "binar" no es compatible con el tipo BIT definido en el paquete STANDARD. Aunque sus elementos son los mismos, en "binar" la base es "cuad".

## **2.6 TIPOS STD LOGIC**

Es evidente que para modelar o simular hardware digital, el tipo BIT del paquete STANDARD es insuficiente ya que solo dispone de los valores 0 y 1. Por otra parte, si cada usuario define sus propios tipos y paquetes resultarán incompatibilidades que dificultarán los desarrollos compartidos o reutilizables. Para evitar estos inconvenientes, el IEEE desarrolló un nuevo paquete denominado STD\_LOGIC\_1164, donde se declara el tipo std\_ulogic con nueve valores.

El paquete STD\_LOGIC\_1164 está dentro de una biblioteca denominada IEEE, por lo que, a diferencia de los tipos bit y bit\_vector, que por estar en el paquete y biblioteca STANDARD siempre están visibles, para usar los tipos std\_ulogic es necesario iniciar la descripción con las sentencias:

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

que, al hacer visibles la biblioteca y el paquete, permiten utilizar los tipos allí declarados.

El tipo base declarado en el paquete std\_logic\_1164 es el tipo std\_ulogic, conocido también como MVL9 - *multi value logic* - y tiene sus nueve valores declarados como :

```
TYPE std_ulogic IS ( 'U'    -- uninitialized
                    'X'    -- forcing unknown
                    '0'    -- forcing 0
                    '1'    -- forcing 1
                    'Z'    -- High Impedance
                    'W'    -- Weak unknown
                    'L'    -- Weak 0
                    'H'    -- Weak 1
                    '-'    -- Don't care );
```

y, al igual que en el paquete STANDARD, inmediatamente después se declara el tipo array

```
TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
```

El valor 'U' , *uninitialized* , valor no inicializado, se emplea para comprobar que la lógica, particularmente la secuencial, ha sido inicializada correctamente y no tiene valores imprevistos. Por defecto, los dispositivos secuenciales estarán inicializados en 'U'. Cuando comienza la simulación deben inicializarse a '0' o '1', pero si por olvido o fallo no sucediese, el simulador avisará, ya que 'U' es el más fuerte y ningún otro le hará cambiar.

El valor 'X' se reserva para conflictos de señales motivados por asignación múltiple de valores diferentes a una misma señal, por lo que se diferencia de 'U' reservado únicamente para inicialización por defecto, en espera del valor impuesto por el diseñador.

El valor '-' se usa exclusivamente en aplicaciones de síntesis. Durante procesos de simulación se asimila a 'Z'.

El sentido de usar los tipos MVL9 o std\_logic\_1164 se verá con más detalle al estudiar las funciones de resolución en la sección 4.

## **2.7 ATRIBUTOS PREDEFINIDOS DE TIPOS Y ARRAY'S**

Los atributos son valores específicos tales como datos, funciones, tipos o rangos asociados con los objetos a que se refieren. Los atributos permiten un uso más eficiente del lenguaje al disponer de codificaciones adicionales que simplifican las descripciones

El formato tipo es :

nombre\_de\_objeto 'nombre\_de\_atributo

### **2.7.1 ATRIBUTOS DE TIPOS**

En los tipos enumerados y en los numéricos, INTEGER O REAL, permiten encontrar el valor de los elementos, su posición, elemento que sigue, elemento que precede, etc. En estos atributos predefinidos se considera que al declarar un tipo enumerado se asocia a cada elemento un índice que indica su posición en la lista, siendo 0 para el primer elemento enumerado e incrementando en 1 para cada elemento sucesivo. Los ejemplos muestran los atributos predefinidos del tipo *dias* que se declara como:

TYPE dias IS ( lun, mar, mie, jue, vie, sab, dom);

<b>BASE</b>	- Proporciona el nombre del tipo	mar'BASE = dias
<b>LEFT</b>	- Da el valor del elemento izquierdo del tipo	dias'LEFT = lun
<b>RIGHT</b>	- Da el valor del elemento derecho del tipo	dias'RIGHT = dom
<b>HIGH</b>	- Entrega el valor del elemento con índice más alto	dias'HIGH = dom
<b>LOW</b>	- Entrega el valor del elemento con índice más bajo	dias'LOW = lun
<b>POS</b>	- Suministra el índice del elemento referido	dias'POS(jue) = 3
<b>VAL</b>	- Suministra el valor del elemento con el índice dado	dias'VAL(3) = jue
<b>SUCC</b>	- Indica el valor de la base que sigue al referido	dias'SUCC(jue) = vie
<b>PRED</b>	- Indica el valor de la base que precede al referido	dias'PRED(vie) = jue
<b>LEFTOF</b>	- Valor del elemento a la izquierda del referido	dias'LEFTOF(jue) = mie
<b>RIGHTOF</b>	- Valor del elemento a la derecha del referido	dias'RIGHTOF(lun) = mar

Otros ejemplos se obtienen del paquete STANDARD, a partir de los tipos CHARACTER y BIT:

character'POS(NUL) = 0 ;                      character'POS('0') = 48  
 bit'LEFT = '0'                      bit'RIGHT = '1'                      bit'POS('0') = 0                      bit'VAL(0) = '0'

Nótese que NUL va directamente, mientras que el carácter '0' va entre apóstrofes.

De forma similar, para tipos INTEGER como los *subir* y *bajar* definidos a continuación, se tienen los ejemplos siguientes, en los que deben apreciarse las diferencias que existen :

TYPE subir IS RANGE 0 TO 9;	TYPE bajar IS RANGE 9 DOWNT0 0;
subir'LEFT = 0	bajar'LEFT = 9
subir'RIGHT = 9	bajar'RIGHT = 0
subir'PRED(5) = 4	bajar'PRED(5) = 4
subir'SUCC(5) = 6	bajar'SUCC(5) = 6
subir'LOW = 0	bajar'LOW = 0
subir'LEFTOF(3) = 2	bajar'LEFTOF(3) = 4
subir'RIGHTOF(4) = 5	bajar'RIGHTOF(4) = 3

### 2.7.2 ATRIBUTOS DE ARRAYS

Devuelven valores relativos al rango, longitud o límites del array. Dado que pueden existir arrays multidimensionales, los atributos pueden opcionalmente hacer referencia a una de las dimensiones del array. El formato general es :

nombre\_de\_array ' atributo [(n)]

Existen los siguientes atributos de arrays :

<b>array'LEFT(n)</b>	Devuelve el límite izquierdo del subarray de índice n.
<b>array'RIGHT(n)</b>	Devuelve el límite derecho del subarray de índice n.
<b>array'HIGH(n)</b>	Devuelve el límite alto del subarray de índice n.
<b>array'LOW(n)</b>	Devuelve el límite bajo del subarray de índice n.
<b>array'LENGTH(n)</b>	Da la longitud del subarray de índice n.
<b>array'RANGE(n)</b>	Da el rango del subarray de índice n.
<b>array'REVERSE RANGE(n)</b>	Da el rango inverso del subarray de índice n.

Para arrays definidos con rangos ascendentes:

array'LEFT = array'LOW  
array'RIGHT = array'HIGH

Para arrays definidos con rangos descendentes :

array'LEFT = array'HIGH  
array'RIGHT = array'LOW

Aplicados a un array bidimensional como

```
TYPE brillo is BIT_VECTOR( 7 DOWNT0 0 );
TYPE pantalla IS ARRAY ( 1 TO 640, 480 DOWNT0 1 ) OF brillo;
```

se tendría

pantalla'LEFT(1)	1,	pantalla'LEFT(2)	480
pantalla'LOW(1)	1,	pantalla'LOW (2)	1
pantalla'RIGHT(1)	640,	pantalla'RIGHT(2)	1
pantalla'HIGH(1)	640,	pantalla'HIGH(2)	480
pantalla'LENGTH(1)	640,	pantalla'LENGTH(2)	480
pantalla'RANGE(1)	1 TO 640	pantalla'RANGE(2)	480 DOWNT0 1
pantalla'REVERSE RANGE(1)	640 DOWNT0 1	pantalla'REVERSE RANGE(2)	1 TO 480

Un posible uso de los atributos 'LOW y 'HIGH de arrays se ve en el ejemplo siguiente:

```
FUNCTION convertir_a_entero ( dato: BIT_VECTOR ) RETURN INTEGER IS
  VARIABLE resultado : INTEGER := 0;
BEGIN
  FOR n IN dato'LOW TO dato'HIGH LOOP
    IF dato(n) = '1' THEN
      resultado := resultado + ( 2** n);
    END IF;
  END LOOP;
  RETURN resultado;
END convertir_a_entero;
```

## **2.8 OPERADORES Y EXPRESIONES**

Asociados en grupos en los que todos los operadores tienen igual prioridad y ordenados los grupos por relación de precedencia, se tiene la clasificación siguiente:

OPERADORES LÓGICOS	:	<b>AND OR NAND NOR XOR</b>
OPERADORES RELACIONALES	:	<b>= /= &lt; &lt;= &gt; &gt;=</b>
OPERADORES DE ADICIÓN	:	<b>+ - &amp;</b>
OPERADORES DE SIGNO	:	<b>+ -</b>
OPERADORES DE MULTIPLICAR	:	<b>* / mod rem</b>
OPERADORES MISCELÁNEOS	:	<b>** abs NOT</b>

### **2.8.1 OPERADORES LÓGICOS : AND OR NAND NOR XOR**

Definen operaciones sobre variables o señales de tipo BIT o BOOLEANO y se obtienen resultados del mismo tipo que tengan los operandos. Dado que todos los operadores del grupo tienen igual precedencia, según las expresiones que se manejen, puede ser necesario el empleo de paréntesis para precisar la operación a realizar con los operandos, por ejemplo:

a := b AND c OR d AND e      -- expresión ilegal

es una expresión ilegal que puede expresarse correctamente como :

$$a := (b \text{ AND } c) \text{ OR } (d \text{ AND } e)$$

Nótese que los operadores AND y OR, por ser asociativos, permiten escribir indistintamente

$$\begin{array}{ll} a := b \text{ AND } c \text{ AND } d & \text{o bien} \quad a := c \text{ AND } d \text{ AND } b \\ z := b \text{ OR } c \text{ OR } d & \text{o bien} \quad z := c \text{ OR } b \text{ OR } d \end{array}$$

mientras que los operadores NAND y NOR, no asociativos, requieren el empleo del operador NOT con más de dos operandos, por ejemplo, para representar la función NOR de tres entradas, no se puede formular como :

$$z \leq x \text{ NOR } y \text{ NOR } w \quad \text{-- es ilegal}$$

ni tampoco como

$$z \leq (x \text{ NOR } y) \text{ NOR } w \quad \text{-- legal, pero no es la función NOR}$$

debiendo expresarse como

$$z \leq \text{NOT } (x \text{ OR } y \text{ OR } w)$$

que justifica la mayor prioridad de NOT.

### 2.8.2 OPERADORES RELACIONALES : = /= < <= > >=

Deben comparar objetos del mismo tipo y el resultado de las expresiones que contengan operadores de este tipo es siempre tipo booleano: cierto o falso, TRUE o FALSE. Algunos ejemplos de expresiones, en relación con los tipos, pueden ser :

```
tempext := 10;    tempint := 18;
    ( tempext < tempint )      -- resultado es TRUE
signal_A <= 3;    signal_B <= 5;
    ( signal_B <= signal_A )  -- resultado es FALSE
```

En este último ejemplo puede verse el doble significado del delimitador compuesto <= , cuyo sentido correcto es deducido por el contexto de la expresión donde figure.

### 2.8.3 OPERADORES DE ADICIÓN : + - &

+ y - son operadores aritméticos, para operandos de igual tipo: entero, real o físico.

El resultado es del mismo tipo que los operandos.

El operador de concatenación, &, enlaza dos o más elementos del mismo tipo en una única unidad o elemento nuevo. Se utiliza especialmente en arrays unidimensionales de igual tipo base, obteniéndose un nuevo array resultado cuya longitud es la suma de las longitudes de los arrays operandos. Los ejemplos que siguen muestran los resultados de concatenar :



```

signal_A <= '0';          signal_B <= '1';
signal_A & signal_B      tiene por valor "01"
VARIABLE marca           : STRING ( 1 TO 4 );           := "opel";
VARIABLE modelo          : STRING ( 5 DOWNT0 1);        := "corsa";
VARIABLE color           : STRING ( 1 TO 3 );           := "roj";
VARIABLE coche           : STRING ( 1 TO 14 );
coche := marca & ' ' & modelo & ' ' & color; -- coche tiene valor "opel corsa roj"

```

Nótese en este ejemplo el uso de apóstrofo y dobles comillas, ya que el espacio ' ' es un carácter y los STRING son arrays de caracteres.

#### 2.8.4 OPERADORES DE PRODUCTO : \* / MOD REM

\* y / , multiplicación y división , están definidos para tipos entero y real.

Los objetos de tipo físico, puestos a la izquierda del operador \* o / , pueden ser multiplicados o divididos por enteros o reales y el resultado será del tipo físico. Dos objetos del mismo tipo físico pueden ser divididos y el resultado es de tipo entero. Véanse ejemplos:

```

VARIABLE t1,t2,t3 : TIME := 10 ns;          r1 := v1*r2;          -- r1 vale 50 kohms
VARIABLE r1, r2, r3 : resistencia := 10 kohms; t1 := 5.0*t2;      -- t1 vale 50.0 ns
VARIABLE v1, v2 : entero := 5;              v2 := r1/r2;          -- v2 vale 5
                                              v1 := r1/v2;          -- es ilegal
                                              r1 := r2*r3;          -- es ilegal

```

**mod** y **rem** están definidos solo para tipo INTEGER.

**mod** está definido por la expresión  $A = B*N + (A \text{ mod } B)$

donde  $(A \text{ mod } B)$  tiene el signo de B y un valor absoluto menor que el de B. N es un número entero para el que se debe cumplir la expresión anterior.

**rem** está definido por la expresión  $A = (A/B)*B + (A \text{ rem } B)$

donde  $(A \text{ rem } B)$  tiene el signo de A y un valor absoluto menor que el de B.

A	B	A/B	AremB	Amod B
19	4	4	3	3
- 19	4	- 4	- 3	1
19	- 4	- 4	3	- 1
- 19	- 4	4	- 3	- 3

#### 2.8.5 OPERADORES DE SIGNO : + -

Solamente afectan al signo de los operandos numéricos. Por la precedencia de los operadores, no pueden ir en las expresiones inmediatamente detrás de los operadores de multiplicación ni misceláneos, por lo que deberán ir entre paréntesis en tales casos.

```

TYPE temperatura is INTEGER RANGE -50 TO 100;
VARIABLE referencia : temperatura := 3 ;
SIGNAL radiador, congelador : temperatura;
congelador <= - referencia;          -- congelador vale -3
referencia := -radiador;             -- referencia vale 50

```

radiador/ -congelador	-- es ilegal
radiador/(-congelador)	-- es correcto
radiador** -congelador	-- es ilegal
radiador**(-congelador)	-- es correcto

### 2.8.6 OPERADORES MISCELÁNEOS :    \*\*    ABS   NOT

**\*\***    operador de exponenciación.

La base debe ser de tipo entero o real y el exponente de tipo entero.

El exponente solo puede ser negativo con bases de tipo real.

El resultado es del mismo tipo que la base.

**abs**    Operador unario que obtiene el valor absoluto del operando.

**NOT**    Operador lógico NOT, para tipos BIT y booleanos. Algunos ejemplos son :

```
VARIABLE X,Y,Z: INTEGER := 10;
VARIABLE A: REAL := 5.0;
SIGNAL S1,S2,S3 : BIT := '1';

Z := X**2;                -- Z vale 100;
A := 0.5 ** (-1);         -- A vale 2.0
Z := X**(A);              -- es ilegal
X := A** 3;               -- es ilegal
Y := X**(-2 );           -- es ilegal
X := ABS ( Z - ( Y**2) ); -- x vale 90
s1 <= s2 AND NOT s3      -- s1 recibe el valor '0'
```

## **2.9 SOBRECARGA DE TIPOS Y OPERADORES**

En VHDL el concepto de sobrecarga - *overloading* - se refiere a la posibilidad de tener objetos o elementos con más de un significado, por ejemplo, en el paquete STANDARD tenemos ya sobrecarga en los caracteres '0' y '1' que se definen en la segunda declaración del paquete como tipo BIT y en la tercera como tipo CHARACTER.

Igualmente, el usuario puede definir tipos sobrecargados, por ejemplo :

```
TYPE colores_primarios IS ( rojo, verde, azul);
TYPE leds is ( rojo, ambar, amarillo, verde, azul);
```

supone sobrecarga de elementos.

De la misma forma, se puede tener sobrecarga de operadores, por ejemplo, si se tienen las señales w, x, y, z de tipo BIT, se podría escribir la sentencia :

w <= ( x AND y ) OR z;

Si posteriormente las mismas variables se tuviesen de tipo cuad, definido previamente, sería posible mantener la expresión anterior si antes se hubieran definido los operadores AND y OR para los tipos cuad, ya que estos operadores solo están predefinidos para los tipos BIT

y BOOLEAN. La expansión o sobrecarga de los operadores AND y OR, se hace definiendo un nuevo operador, con igual nombre, por medio de una función que devuelve resultados de tipo *cuad* a partir de operandos *cuad*. La función o subprograma hará el efecto de un macrooperador, como se verá con más detalle en la sección 4 sobre subprogramas.

Las situaciones de sobrecarga se resuelven en VHDL por contexto, es decir, si los operandos son, por ejemplo, del tipo *cuad* y hay una expresión donde se llama a un subprograma existente para aplicarles la *función* AND u OR, el VHDL aplicará la *función sobrecargada* para dichos tipos y devolverá un resultado que será acorde con el que se espera que, puede ser de tipo BIT, INTEGER, *cuad* u otro cualquiera definido, siempre que exista la función o subprograma sobrecargado que corresponda al tipo de parámetros de entrada y salida especificados al invocarlo. Es decir, el concepto de sobrecarga se extiende a tipos, operadores y subprogramas.

Cuando se sobrecargan operadores, al declararlos y definirlos, estos van entre comillas, por ejemplo “AND” y “OR”, para diferenciarlos de las funciones. La razón es que los operadores van entre los operandos mientras que los subprogramas se llaman con los parámetros entre paréntesis. En el empaquetamiento “modelos” del Apéndice C se tienen algunos ejemplos de operadores sobrecargados.

A veces, existen expresiones en las que puede existir una ambigüedad que el VHDL no puede resolver por la sobrecarga que tengan los objetos, por ejemplo, si se tuviese la expresión

... IF ( 'Z' < 'X' ) ...

y se estuviese en un contexto donde están visibles los tipos *cuad*, aparte de los CHARACTER, siempre visibles por ser declarados en el paquete STANDARD, se tendría una situación cuyo resultado depende del tipo que se considere para los elementos sobrecargados, por ejemplo:

( 'Z' < 'X' ) es TRUE si los elementos son del tipo *cuad* ( '0', '1', 'Z', 'X' )

( 'Z' < 'X' ) es FALSE si los elementos son del tipo CHARACTER

Para evitar situaciones como la del ejemplo, se deben *cualificar* las expresiones susceptibles de ambigüedad. Esto se hace por *marcado de tipos* o *expresiones cualificadas* que consisten en forzar el tipo de los elementos según el formato de expresión siguiente :

TIPO' (expresión con tipos sobrecargados)

así , la expresión ambigua anterior dejaría de serlo expresándola como :

... IF ( *cuad*'('Z') < *cuad*'('X') )