

## 4. Subprogramas

Los subprogramas son secuencias de declaraciones y sentencias con los que se realizan operaciones o algoritmos que se aplican en zonas diferentes de una descripción o que se repiten a lo largo de ella. Para evitar la repetición de código, al igual que en otros lenguajes, el VHDL permite invocar a las subrutinas desde sentencias o expresiones colocadas en la zona del programa donde se necesitan los resultados que suministran los subprogramas. Existen dos tipos de subprogramas : Funciones y Procedimientos.

En los subprogramas cabe distinguir una declaración y un cuerpo, lo que equivale a declarar una interfaz y describir el algoritmo que le corresponde. Posteriormente, desde algún punto de la descripción se invoca o llama al subprograma de acuerdo con la interfaz o estructura de su declaración. Mientras la llamada a funciones es parte de una expresión VHDL, la llamada a procedimientos es una sentencia en sí. Tanto unas como otras se pueden considerar concurrentes o secuenciales, según el área desde la que se llaman.

### 4.1 Funciones

Las funciones tienen siempre un cuerpo donde se define la operación que realizan a partir de los parámetros de entrada para devolver un único valor como resultado. Aunque no es preceptivo declarar las funciones, es conveniente hacerlo porque simplifica la generación de código y se evitan las restricciones que conlleva la ausencia de declaración. Hay dos situaciones en las que la declaración de la función es necesaria :

- Cuando sea recursiva, es decir, cuando pueda autollamarse.
- Si el cuerpo de la función está después de la expresión donde se llama a la función.

Las declaraciones deben hacerse en áreas declarativas con el formato :

```
FUNCTION nombre ( lista formal de parámetros ) RETURN tipo_de_respuesta;
```

La lista formal de parámetros, en su forma más completa, tiene el formato :

```
(objeto nombre : modo tipo )
```

- |          |   |
|----------|---|
| objeto : | Puede ser CONSTANT o SIGNAL. Por defecto es CONSTANT.<br>No puede ser VARIABLE.   |
| nombre : | Puede ser una lista de nombres, separados por ',' si son del mismo tipo.<br>Si los parámetros son de distinto tipo estarán separados por ';' .  |
| modo :   | Solo puede ser modo IN. Por esto se omite, ya que está implícito.   |
| tipo :   | Para cada objeto debe indicarse su tipo. El tipo_de_respuesta puede ser distinto del de los parámetros, caso típico en funciones de conversión. |

Algunos ejemplos de declaraciones son :

```
FUNCTION binario_a_entero ( dato : BIT_VECTOR ) RETURN INTEGER ;  
FUNCTION entero_a_bin ( entero: INTEGER; bits : POSITIVE ) RETURN BIT_VECTOR;  
FUNCTION incremento ( dato : BIT_VECTOR ( N DOWNT0 0 ) ) RETURN BIT_VECTOR;
```

El cuerpo de la función define lo que la función hace y consiste en una relación de sentencias secuenciales. No pueden incluirse sentencias concurrentes.

El formato del cuerpo de las funciones es :

```
FUNCTION nombre_de_función ( lista de parámetros ) RETURN tipo_de_respuesta IS  
    -- sentencias declarativas locales de la función  
BEGIN  
    -- sentencias secuenciales  
    RETURN expresión ;      -- puede haber varias de este formato  
END nombre_de_función ;
```

En las sentencias declarativas se insertan objetos locales alterables, ya que los parámetros de entrada son inalterables por ser de modo IN . Estos objetos, por ser locales, no son utilizables fuera del cuerpo de la función donde han sido declarados.

Aunque pueden existir varias sentencias con la cláusula RETURN y una expresión adjunta con el valor que la función devuelve, solo una de ellas se ejecutará y al hacerlo terminará la ejecución de la función. Este tipo de repeticiones suele ir asociado a condiciones autoexcluyentes que, al cumplirse una de ellas, bien elimina la consideración de las otras o por el orden secuencial se considera prioritaria y es la que determina el resultado que devuelve la función.

Las funciones se puedan considerar como macrooperadores, pero existen dos tipos de funciones con aplicaciones muy concretas que les dan nombre:

- Funciones de Conversión, utilizadas para conversión de tipos en VHDL.
- Funciones de Resolución , aplicadas para determinar el valor que tendrá una señal a la que varios *drivers* asignan valor simultáneamente.

Dos ejemplos representativos de funciones de conversión son :

```
FUNCTION binario_a_entero ( dato: BIT_VECTOR ) RETURN INTEGER IS  
    VARIABLE resultado : INTEGER := 0;  
BEGIN  
    -- 'RANGE actúa como una función y  
    FOR n IN dato'RANGE LOOP -- devuelve (dato_bajo TO dato_alto)  
        IF dato(n) = '1' THEN  
            resultado := resultado + ( 2** n);  
        END IF;  
    END LOOP;  
    RETURN resultado;  
END binario_a_entero;
```

y la función de conversión complementaria :

```

FUNCTION entero_a_bin ( entero:INTEGER; bits: POSITIVE ) RETURN BIT_VECTOR IS
    VARIABLE binario      : BIT_VECTOR ( bits -1 DOWNT0 0 );
    VARIABLE numero       : INTEGER := 0;
    VARIABLE cociente     : INTEGER := 0;
BEGIN
    numero := entero;
    FOR i IN binario'RANGE LOOP
        cociente := numero / ( 2 ** i );
        numero := numero REM ( 2** i );
        IF ( cociente = 1 ) THEN
            binario ( i ) := '1';
        ELSE
            binario ( i ) := '0';
        END IF;
    END LOOP;
    RETURN binario;
END entero_a_bin;

```

A estas funciones podrían llamarlas expresiones como las siguientes :

```

salida <= binario_a_entero ( objeto de tipo bit_vector )
objeto bit_vector ( bits 1 DOWNT0 0 ) <= entero_a_bin ( objeto INTEGER, bits )

```

en las que se muestra el formato de la llamada a función que, en forma genérica, es :

expresión <= [ expresión ] identificador de función ( valor de parámetros )

El ejemplo de función siguiente, devuelve el binario de entrada incrementado en uno y hace puesta a cero cuando el binario de entrada equivale a 24 :

```

FUNCTION incremento (dato: BIT_VECTOR ( 4 DOWNT0 0 ) ) RETURN BIT_VECTOR IS;
    VARIABLE cuenta : BIT_VECTOR ( 4 DOWNT0 0 );
BEGIN
    cuenta := dato;
    IF cuenta = "11000" THEN
        cuenta = "00000";
    ELSE
        FOR i IN 0 TO 4 LOOP
            IF cuenta(i) = '0' THEN
                cuenta(i) := '1';
                EXIT;
            ELSE
                cuenta(i) := '0';
            END IF;
        END LOOP;
    END IF;
    RETURN cuenta;
END incremento;

```

-- si la cuenta es igual a 24...  
-- cambiarla a cero  
-- y si no, incrementar como se indica  
-- búsqueda de bit a incrementar (\*)  
-- salir del lazo al primer cambio de 0 a 1  
-- que corresponde a incrementar en 1 .  
-- incremento por suma binaria  
'1'+ '1' = '0'  
-- pero falta el acarreo, a hacer en (\*).  
-- salir del IF de reposición a cero  
-- devolver valor de cuenta incrementada.

En el paquete modelos se han incluido las funciones *incrementar* y *decrementar* que serán utilizadas en modelos de contadores :

```
FUNCTION incrementar ( V: BIT_VECTOR ) RETURN BIT_VECTOR IS
    VARIABLE bv : BIT_VECTOR(V'LENGTH-1 DOWNT0 0);
BEGIN
    bv := v;
    FOR i IN 0 TO bv'HIGH LOOP
        IF    bv(i) = '0' THEN    bv(i) := '1';
            EXIT;
        ELSE bv(i) := '0';
        END IF;
    END LOOP;
    RETURN bv;
END incrementar;
```

```
FUNCTION decrementar ( v : BIT_VECTOR ) RETURN BIT_VECTOR IS
    VARIABLE bv : BIT_VECTOR(v'LENGTH-1 DOWNT0 0);
BEGIN
    bv := v;
    FOR i IN 0 TO bv'HIGH LOOP
        IF    bv(i) = '1' THEN    bv(i) := '0';
            EXIT;
        ELSE bv(i) := '1';
        END IF;
    END LOOP;
    RETURN bv;
END decrementar;
```

### 4.1.1 FUNCIONES DE RESOLUCIÓN

Cuando en un nodo concurren varias señales y no todas con igual valor, la asignación está indeterminada. En VHDL esta situación se asocia a una señal que tiene asignación múltiple, por varios *drivers* o procesos. Un ejemplo podría ser el de un circuito con lógica cableada y la siguiente descripción imposible, que no sería aceptada por el compilador:

```
USE WORK.modelos.ALL           -- para usar tipos  cuad.

ENTITY nodoand IS
    PORT ( a,b,c :IN cuad ; z :OUT cuad);
END nodoand;

ARCHITECTURE imposible OF nodoand IS
    SIGNAL mezcla : cuad;
BEGIN
    mezcla <= a;                -- asignaciones concurrentes al mismo nodo
    mezcla <= b;
    mezcla <= c;
    z <= mezcla;
END imposible;
```

Para modelar un circuito en que se tenga esa situación es necesario resolver el conflicto entre *drivers* por medio de una función de resolución que determine el valor a asignar a la señal. La función de resolución, que tiene como entrada un array abierto formado por los valores de los *drivers*, actúa automáticamente al ocurrir un evento en cualquiera de ellos, devolviendo el valor resuelto que estará determinado por la función en base a la fuerza de las señales *drivers*.

Si, por ejemplo, el tipo de las señales que vamos a utilizar en un diseño fuera

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

a partir de él se podría declarar el tipo `cuad_2` array de dos dimensiones

```
TYPE cuad_2 IS ARRAY ( cuad,cuad ) OF cuad;
```

La función de resolución concreta, es decir, la fuerza relativa de las señales que acceden al nodo, es definida o elegida por el diseñador. Desde el punto de vista de simulación, en VHDL se podrán definir varias funciones para cada caso, sin embargo, si el objetivo final del modelo es sintetizar el diseño con alguna herramienta CAD comercial, es probable que el conjunto de funciones soportadas esté limitado a los dos o tres casos más habituales en hardware: triestados, and-cableada y, tal vez, or-cableada.

Una forma de definir la función de resolución se basa en describir la relación de fuerza entre los elementos de dos *drivers* por medio de una tabla bidimensional, cuya aplicación se puede extender a casos de más de dos *drivers* por aplicación sucesiva de las mismas tablas basándose en que éstas son asociativas y conmutativas, es decir, para el ejemplo anterior con tres *drivers* a , b y c del tipo `cuad` definido arriba, deberá cumplirse que

$$a \text{ AND } b \text{ AND } c = (a \text{ AND } b) \text{ AND } c = (a \text{ AND } c) \text{ AND } b = (b \text{ AND } c) \text{ AND } a$$

ya que de otra forma, habría que considerar un posible resultado diferente en función del orden en la aplicación de *drivers*, lo que ni corresponde a situaciones reales ni tampoco se especifica en VHDL al considerar arrays abiertos.

La figura adjunta muestra un ejemplo de tabla bidimensional para dos *drivers* a y b, en los que el elemento fuerte es el '0', lo que permite asociarla a una función AND-cableada.

La tabla puede ser descrita por la constante adjunta a ella :

a b	0	1	Z	X
0	0	0	0	0
1	0	1	Z	X
Z	0	Z	1	X
X	0	X	X	X

```
CONSTANT tabla_and : cuad_2 := ( ('0','0','0','0'),
                                   ('0','1','Z','X'),
                                   ('0','Z','1','X'),
                                   ('0','X','X','X'));
```

y a partir de la misma puede obtenerse la función “AND” sobrecargada como se indica :

```
FUNCTION "AND" (a,b: cuad ) RETURN cuad IS;
  CONSTANT tabla_and : cuad_2 := ( ('0','0','0','0'),
                                     ('0','1','Z','X'),
                                     ('0','Z','1','X'),
                                     ('0','X','X','X'));

  BEGIN
    RETURN tabla_and (a,b);
  END "AND";
```

Dado que el tipo cuad es declarado por el usuario, no hay definido para él ningún operador y, en consecuencia, todas las operaciones o funciones a realizar con él deberán ser definidas previamente a su uso, ya sea por inclusión en un empaquetamiento al que se tiene acceso por empleo de la cláusula USE, o por declaración y definición en la región declarativa del bloque donde se vaya a usar la función.

Las tablas y constantes asociadas, es decir, los contenidos de las tablas, se establecen por el usuario y su asociación a la AND-cableada es voluntaria y caprichosa, como un ejemplo de sobrecarga de la función AND predefinida. Se pueden definir otras funciones cuyas tablas representen otras funciones lógicas más complejas.

De forma similar a la función “AND”, se puede definir una función cuya tabla bidimensional es similar a la anterior, pero donde el elemento de mayor fuerza será ahora el ‘1’ cuad y que, por esta fuerza del elemento ‘1’, se asocia a la función “OR” :

a b	0	1	Z	X
0	0	1	1	X
1	1	1	1	1
Z	1	1	1	1
X	X	1	1	X

```
CONSTANT tabla_or : cuad_2 := ( ('0','1','1','X'),
                                 ('1','1','1','1'),
                                 ('1','1','1','1'),
                                 ('X','1','1','X'));
```

a partir de la cual puede obtenerse la función “OR”, también sobrecargada, como :

```
FUNCTION "OR" (a,b: cuad ) RETURN cuad IS;
  CONSTANT tabla_or : cuad_2 := ( ('0','1','1','X'),
                                   ('1','1','1','1'),
                                   ('1','1','1','1'),
                                   ('X','1','1','X'));

  BEGIN
    RETURN tabla_or (a,b);
  END "OR";
```

Una vez definida la función de resolución, existen al menos dos posibles métodos para obtener el valor de la señal resuelta obtenido por la función :

## 1º Declaración de señal resuelta

Consiste en especificar la señal resuelta y la función de resolución asociada a los *drivers* en la región declarativa del bloque donde se utiliza la señal. El formato sería :

```
SIGNAL nombre_de_resuelta : nombre_funcion_resolución tipo_resuelto
```

así, para la entidad *nodoand* descrita anteriormente, pendiente de una arquitectura posible, se podrían declarar los siguientes tipos

```
TYPE cuad_vector IS ARRAY ( NATURAL RANGE <> ) OF cuad;    -- declaración
FUNCTION andcableada ( cables: cuad_vector ) RETURN cuad;    -- declaración
```

y la siguiente función de resolución :

```
FUNCTION andcableada ( cables: cuad_vector ) RETURN cuad IS    -- definición
    VARIABLE union : cuad := '1';
BEGIN
    FOR i IN cables' RANGE LOOP
        union := union AND cables(i);
    END LOOP;
    RETURN union;
END andcableada;
```

- El parámetro de la función de resolución, al que se ha denominado *cables*, es un array cuyos elementos tienen el mismo tipo que el resultado que devuelve la función, siendo los elementos del array los valores de las señales *drivers*.

- Nótese que al definir el array *cuad\_vector* de rango abierto, se permite definir a la función sin especificar el número de *drivers* - *cables*- que acceden al nodo, y que la función obtiene por el atributo 'RANGE'.

- La función AND, al estar entre objetos de tipo *cuad*, buscará la función "AND" sobrecargada definida para este tipo.

- El bucle LOOP, se repite para el número de *cables* o *drivers* que devuelva el atributo 'RANGE, y en cada paso del bucle se realiza la función "AND" de un nuevo cable con el resultado resuelto de todos los anteriores.

- La variable *union* se inicializa a '1', ya que como se ve en la tabla\_AND, este elemento no altera el valor que aporte el primer cable, por ser '1' el valor más débil.

Así, podríamos definir una arquitectura posible:

```
USE WORK. paquete_de_usuario.ALL ;    --acceso a tipos y función andcableada
ARCHITECTURE posible OF nodoand IS
    SIGNAL mezcla : andcableada cuad; -- declaración de señal resuelta
BEGIN
    mezcla <= a;    -- asignación concurrente de señales al mismo nodo.
    mezcla <= b;
    mezcla <= c;
    z <= mezcla;
END posible;
```

La declaración de SIGNAL llama a la función "andcableada" cada vez que ocurre un cambio en alguna de las señales a, b, c, *drivers* de la señal resuelta "mezcla". La forma en que se asocia el valor de los *drivers* con el parámetro del array de la función es una concatenación de todos los drivers. Dicha concatenación se realiza con la asignación múltiple de valores a la señal cuyo nombre figura en la declaración de la señal a resolver junto con el de la función de resolución para hacerlo.

### 2º Declaración de un subtipo resuelto

Es similar al método anterior y consiste en los siguientes pasos :

- Declarar el tipo base, el tipo array abierto correspondiente , la función de resolución y un subtipo con el nombre de la señal que se desea resuelta junto a la función para hacerlo, por ejemplo :

```
TYPE cuad IS ( '0','1','Z','X');
TYPE cuad_vector IS ARRAY ( NATURAL RANGE <> ) OF cuad;
FUNCTION andcableada ( cables: cuad_vector ) RETURN cuad;
SUBTYPE mezcla IS andcableada cuad;
```

- Declarar la señal resuelta, siendo su tipo el subtipo resuelto.

#### 4.1.2 TIPOS STD\_LOGIC . EMPAQUETAMIENTO IEEE 1164

Un ejemplo importante del método anterior son los tipos resueltos std\_logic del empaquetamiento std\_logic\_1164, cuya declaración y definición parcial son como sigue :

```
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);

TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
SUBTYPE std_logic IS resolved std_ulogic;
```

y una vez definida la función de resolución en la forma que se ve en el PACKAGE BODY, por medio de la tabla resolution\_table , se pasa a definir los tipos y subtipos siguientes:

```
-- *** industry standard logic type ***
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_logic ;
SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1' ;      -- ( 'X','0','1' )
SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z' ;      -- ( 'X','0','1','Z' )
SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1' ;      -- ( 'U','X','0','1' )
SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z' ;      -- ( 'U','X','0','1','Z' )
```



que permiten, utilizando tipos estandarizados por el paquete STD\_LOGIC\_1164, modelar o describir hardware con varias posibilidades de valores lógicos.

```

PACKAGE BODY std_logic_1164 IS
-----
-- local types
-----
TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

CONSTANT resolution_table : stdlogic_table := (
-----
--   | U   X   0   1   Z   W   L   H   -   | |
-----
      ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), -- | U |
      ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- | X |
      ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), -- | 0 |
      ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), -- | 1 |
      ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), -- | Z |
      ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), -- | W |
      ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), -- | L |
      ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), -- | H |
      ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X')); -- | - |

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
  VARIABLE result : std_ulogic := 'Z'; -- weakest state default
BEGIN
  -- the test for a single driver is essential otherwise the
  -- loop would return 'X' for a single driver of '-' and that
  -- would conflict with the value of a single driver unresolved
  -- signal.
  IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;

```

Dado que estos tipos son resueltos ya, no se les puede aplicar una función de resolución .

Además de las declaraciones del tipo `std_ulogic` y los subtipos derivados, de los que quizá el más importante sea el `std_logic`, el paquete define las funciones lógicas AND, NAND, OR, NOR, XOR y NOT para los tipos declarados, ya que de otra forma no podrían aplicarse esas funciones lógicas, predefinidas solo para los tipos BIT, es decir, hace *sobrecarga* de dichos operadores. También se definen en este paquete *funciones de conversión* entre tipos de los paquetes STANDARD Y STD\_LOGIC\_1164 :

```

bit y std_ulogic
bit_vector y std_ulogic_vector o std_logic_vector
std_logic_vector y std_ulogic_vector

```

También se definen funciones especiales para detección de flancos o de valor 'X'

```
FUNCTION rising_edge ( SIGNAL s : std_ulogic) RETURN BOOLEAN;  
FUNCTION falling_edge ( SIGNAL s : std_ulogic) RETURN BOOLEAN;  
FUNCTION Is_X ( s : std_ulogic_vector ) RETURN BOOLEAN;  
FUNCTION Is_X ( s : std_logic_vector ) RETURN BOOLEAN;  
FUNCTION Is_X ( s : std_ulogic ) RETURN BOOLEAN;
```

que devuelven como resultado el valor booleano obtenido, cierto o falso, al aplicar la función correspondiente en la descripción al uso.

```
FUNCTION rising_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS  
BEGIN  
    RETURN (s'EVENT AND (To_X01(s) = '1') AND (To_X01(s'LAST_VALUE) = '0'));  
END;
```

```
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS  
BEGIN  
    RETURN (s'EVENT AND (To_X01(s) = '0') AND (To_X01(s'LAST_VALUE) = '1'));  
END;
```

```
FUNCTION Is_X ( s : std_logic_vector ) RETURN BOOLEAN IS  
BEGIN  
    FOR i IN s'RANGE LOOP  
        CASE s(i) IS  
            WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;  
            WHEN OTHERS => NULL;  
        END CASE;  
    END LOOP;  
    RETURN FALSE;  
END;
```

```
FUNCTION Is_X ( s : std_ulogic ) RETURN BOOLEAN IS  
BEGIN  
    CASE s IS  
        WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;  
        WHEN OTHERS => NULL;  
    END CASE;  
    RETURN FALSE;  
END;
```

Dado que el paquete `std_logic_1164` es un standard IEEE, donde se contemplan casi todas las funciones y valores deseables en lógica multivalor, es recomendable su uso en diseños, en particular los tipos `std_logic` y `std_logic_vector`, ya que por su compatibilidad estandard son reconocidos por herramientas CAD de prestigio, lo que a su vez permite acceso a nuevos tipos o paquetes que son *de facto standard*, al ser compatibles o reconocidos por otras herramientas CAD. Todo esto hace innecesario el esfuerzo de definir nuevos tipos, así como las correspondientes funciones de manejo de los mismos y

disminuye el riesgo de incompatibilidades con futuros diseños. Es también recomendable asegurarse de que tipos o funciones del paquete `std_logic_1164` están soportadas en la herramienta VHDL que se vaya a usar ya que, por ejemplo, las funciones *rising\_edge* y *falling\_edge* no están en ciertos compiladores.

Un ejemplo de aplicación de los tipos `std_logic` como ejemplo de funciones resueltas es el de dos inversores triestado conectados directamente a un mismo nodo :

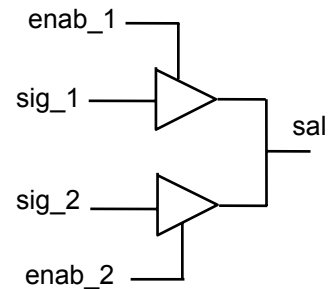
```
LIBRARY IEEE;                                -- Cláusulas obligatorias para visualizar
USE IEEE.std_logic_1164.all;                  -- biblioteca y paquete std_logic_1164

ENTITY triestados IS
    PORT ( enab_1, sig_1, enab_2, sig_2: IN std_logic ; sal : OUT std_logic);
END triestados;
```

una posible arquitectura con descripción de comportamiento podría ser :

```
ARCHITECTURE comportam OF triestados IS
BEGIN
    sig_uno: PROCESS( sig_1, enab_1)
    BEGIN
        sal <= 'Z';
        IF ( enab_1 = '1' ) THEN
            sal <= sig_1 ;
        END IF;
    END PROCESS;

    sig_dos: PROCESS( sig_2, enab_2)
    BEGIN
        sal <= 'Z';
        IF ( enab_2 = '1' ) THEN
            sal <= sig_2 ;
        END IF;
    END PROCESS;
END comportam;
```



y otra posible arquitectura de esa misma entidad, con descripción en estilo *funcional* o de *flujo de datos* puede ser :

```
ARCHITECTURE flujdats OF triestados IS

BEGIN
    sal <= sig_1 WHEN ( enab_1 = '1' ) ELSE 'Z';
    sal <= sig_2 WHEN ( enab_2 = '1' ) ELSE 'Z';
END flujdats;
```

## **4.2 Procedimientos**

Son subprogramas con las siguientes características:

1. Pueden tener más de un valor como respuesta.
2. Pueden tener parámetros de tipo IN, OUT e INOUT.  
Por defecto, si no se especifica el tipo de un parámetro será IN .
3. La clase de los parámetros puede ser constante, señal o variable. Por defecto es constante.
4. Si el modo es OUT o INOUT y no se especifica clase, por defecto es variable.  
El valor de parámetros modo INOUT puede ser modificado por los procedimientos.
5. En los procedimientos no es necesario tener sentencias RETURN para devolver un valor.  
En particular, los parámetros modo OUT no reciben valores de los modelos sino que son evaluados dentro de los procedimientos y el valor obtenido se pasa a los modelos.
6. Los procedimientos se llaman por sentencias aisladas - *procedure call* - mientras que las funciones son parte de expresiones contenidas dentro de sentencias, a las que refuerzan con su capacidad de evaluación externa a la sentencia.
7. Si un objeto en la lista de parámetros es de tipo variable, el procedimiento solo se podrá llamar desde regiones secuenciales, ya que solo en éstas se usan variables.

La declaración de los procedimientos tiene el formato :

PROCEDURE nombre ( lista formal de parámetros con clase, modo y tipo ) ;

que, igual que en las funciones, debe existir si el procedimiento es recursivo o su cuerpo es posterior a la primera sentencia de llamada, que deberá ir precedida de la declaración.

El cuerpo de los procedimientos define su funcionalidad. Igual que el cuerpo de las funciones, consiste en una secuencia de sentencias, que no pueden ser concurrentes, y que describen un algoritmo. Si no está definido en el cuerpo de un paquete visible, deberá estar incluido en regiones declarativas, teniendo presente que, a menos que esté previamente declarado, no podrá ser llamado desde otras áreas a igual nivel dentro de la arquitectura que lo contenga.

Los cuerpos de procedimientos tienen el formato genérico siguiente :

```
PROCEDURE nombre_de_procedimiento ( lista formal de parámetros con modo y tipo ) IS
    región declarativa local del procedimiento
BEGIN
    región activa o ejecutable con sentencias secuenciales
END nombre_de_procedimiento;
```

Normalmente, en modelos que utilicen empaquetamientos de usuario, se declaran tanto las funciones como los procedimientos en la declaración del empaquetamiento y se definen en el cuerpo del mismo. En los demás casos, el cuerpo del subprograma deberá ir definido en la misma región declarativa donde se declara. Veamos algunos ejemplos de procedimientos:

El algoritmo que describe la función `entero_a_bin` vista anteriormente puede obtenerse por un procedimiento como el que indica el ejemplo siguiente :

```
PROCEDURE enter_a_binar ( entero: IN INTEGER; binario : OUT BIT_VECTOR) IS
    VARIABLE temp : INTEGER;
BEGIN
    temp := entero;
    FOR i IN 0 TO (binario'LENGTH -1) LOOP
        IF ( temp MOD 2 = 1 ) THEN
            binario(i) := '1';
        ELSE
            binario(i) := '0';
        END IF;
        temp := temp / 2;
    END LOOP;
END enter_a_binar;
```

En el ejemplo siguiente se muestra un procedimiento cuyo objeto es generar de forma sencilla, con una sentencia que será la llamada al procedimiento, una serie de bit\_vectores que podrán utilizarse, por ejemplo, para constituir una sucesión de estímulos o test-vectores. El procedimiento presenta una estructura genérica, siendo válido para generar vectores de distinta longitud. Previamente se define el tipo:

```
TYPE serie IS ARRAY ( NATURAL RANGE <> ) OF INTEGER ;
```

que al ser un array abierto, permite adaptar el procedimiento para la generación de un número variable de vectores que se indica al llamarlo:

```
PROCEDURE tst_vect ( SIGNAL vectores : OUT BIT_VECTOR ;
                    CONSTANT valores : IN serie;
                    CONSTANT periodo : IN TIME) IS
    VARIABLE longitud : BIT_VECTOR ( vectores'RANGE );
BEGIN
    FOR i IN valores'RANGE LOOP
        enter_a_binar ( valores(i), longitud );
        vectores <= longitud AFTER i* periodo ;
    END LOOP;
END tst_vect;
```

La llamada al procedimiento supone la existencia de la señal `abcd` , declarada como:

```
SIGNAL abcd : BIT_VECTOR ( 7 DOWNT0 0 );
```

que define la longitud de los vectores que deberán generarse, es decir, ( 7 downto 0 ).

y la llamada al procedimiento se realiza con sentencias del formato y ejemplo siguientes :

```
tst_vect ( abcd, lista_de_enteros, expresión de tiempo en unidades )
```

```
tst_vect ( abcd, 01&12&10&09&14&11&07, 1000 ns)
```

que como puede verse coinciden con la estructura de la declaración del procedimiento.

El algoritmo del ejemplo opera como sigue:

- La señal abcd, que hace de parámetro SIGNAL vectores, aporta la dimensión de los vectores a generar, en este caso :

```
SIGNAL abcd : BIT_VECTOR ( 7 DOWNT0 0 );
```

- El parámetro CONSTANT valores , definido del tipo IN serie declarado previamente, es una sucesión de enteros que serán convertidos a binarios por el procedimiento.
- La sucesión de enteros concatenados puede tener cualquier longitud. Por medio del atributo predefinido 'RANGE se determinan las iteraciones del bucle

```
FOR i IN valores'RANGE LOOP
```

que internamente invoca al algoritmo de conversión de enteros a binarios, contenido

en el procedimiento visto como ejemplo anterior

```
enter_a_binar ( valores(i), longitud );
```

- Los parámetros de llamada a este otro procedimiento coinciden con la estructura del mismo, siendo la serie de enteros a convertir y la longitud de los vectores binarios a devolver, proporcionados por el atributo predefinido 'RANGE aplicado a vectores

```
VARIABLE longitud : BIT_VECTOR ( vectores'RANGE );
```

donde el parámetro vectores'RANGE está definido por la declaración

```
SIGNAL abcd : BIT_VECTOR ( 7 DOWNT0 0 );
```

- Una vez que el procedimiento interno devuelve el entero convertido a binario, sigue la ejecución del bucle y se asigna el binario a la señal vectores, parámetro de salida del procedimiento tst\_vect, con retardos múltiplos de índice i y del parámetro

```
CONSTANT periodo : IN TIME
```

que al invocar al procedimiento se le asigna la duración de 1000 ns en el ejemplo.

Como se ve en el ejemplo, las llamadas a procedimientos se hacen adjuntando al nombre o identificador una lista *real* o *actual* de parámetros, que deberá estar asociada a la

lista *formal* de la declaración o definición del procedimiento. Dicha asociación puede ser de tipo posicional o nominal, adaptándose las listas de parámetros en clases, modos y tipos.

Si en lugar de seleccionar un juego reducido de vectores, como se hizo en el ejemplo anterior se desea todas las combinaciones posibles de *n* bits, es decir, un juego de vectores *exhaustivo*, se podría obtener por la función *entero\_a\_bin* vista anteriormente como se muestra en el ejemplo siguiente :

```
USE WORK. modelos. ALL;                -- Funcion de conversión en paquete "modelos"
ENTITY genvect IS
    GENERIC ( bits : POSITIVE := 8 ; periodo : TIME := 1 us );    ---- inicializados
    PORT ( sincro : IN BIT; vectores : OUT BIT_VECTOR( bits-1 DOWNT0 0 ));
END genvect;

ARCHITECTURE exhaustiva OF genvect IS
    -- datos genéricos : 8 bits y 1 microseg.    -- para realizar un ejemplo particular
BEGIN
    PROCESS (sincro)
    BEGIN
        FOR vect IN 0 TO 2**( bits -1) LOOP
            vectores <= entero_a_bin (vect, bits) AFTER vect*periodo;
        END LOOP;
    END PROCESS;
END exhaustiva;
```

Tanto en las llamadas a procedimientos secuenciales como en las concurrentes, debe evitarse la posibilidad de cambiar desde el procedimiento algún objeto que no figure en su lista de parámetros. El resultado puede producir *efectos colaterales* que son difíciles de depurar.