

# 1 - Modelos y unidades de diseño en VHDL

El modelo de un sistema varía según el estado del diseño y el uso a que se destina el modelo. Inicialmente, el modelo de un sistema es un conjunto de especificaciones generales de diversa índole. A partir de esas especificaciones generales, por una estrategia de diseño *top-down*, el sistema se particiona en una serie de niveles sucesivos, siguiendo una estructura jerárquica o árbol que representa al sistema. El proceso posterior de integración del sistema consistirá en el ensamble o interconexión de los subsistemas siguiendo una estrategia *bottom-up*. Normalmente, el desarrollo de modelos de un sistema moderno consiste en una mezcla de las dos estrategias: descomponer el sistema en bloques en un proceso *top-down* e integrar en los subsistemas modelos de nuevo desarrollo junto con otros disponibles como resultado de desarrollos previos.

En cualquier caso, al plantearse el desarrollo del modelo de un subsistema, a cualquier nivel dentro de la jerarquía del sistema, se consideran dos aspectos básicos:

- *Estructura*, definida por *componentes, puertos y señales*.
  - Componentes son elementos - puerta, chip, módulo, etc. - cuya complejidad estará relacionada al nivel del subsistema dentro de la estructura del sistema.
  - Puertos son las interfaces por los que el elemento se interconecta a los puertos de otros elementos para formar la estructura del nivel jerárquico que lo contiene.
  - Señales son los nombres asignados a las rutas que interconectan componentes a través de sus puertos, relacionando estructuras y comportamientos de los modelos.
- *Comportamiento* o funcionalidad del subsistema, definida como relación temporal entre respuestas y estímulos que las provocan, es decir, la función de transferencia y los retardos asociados, si los hay.

En VHDL, un sistema digital se representa como una *entidad de diseño* en la que se aplican los conceptos anteriores al modelarla. Para esto se la considera definida con dos *unidades de diseño* o descripciones interrelacionadas: la declaración de entidad o ENTITY y la descripción funcional o ARCHITECTURE. Estos son los dos pilares básicos sobre los que se construyen todas las descripciones en VHDL.

## **1.1 DECLARACIÓN DE ENTIDAD**

Asigna un nombre identificador al elemento y describe su interfaz como un conjunto de señales a las que se les asigna nombre, modo - entrada, salida, bidireccional - y tipo.

En su formato más simple la declaración de ENTITY es como sigue :

```
ENTITY identificador IS
    PORT ( lista_de_señales : modo tipo )
END identificador;
```

Por ejemplo, la declaración de ENTITY para unas puertas AND y OR podría ser :

```
ENTITY and2 IS  
PORT ( e1,e2 : IN BIT;sal: OUT BIT);  
END and2 ;
```

```
ENTITY or2 IS  
PORT ( e1,e2 : IN BIT; sal: OUT BIT);  
END or2 ;
```

Los tipos se refieren a la estructura o características de las señales que se definen en la entidad. En el ejemplo son BIT. En VHDL, el usuario puede definir sus propios tipos.

Nótese que, aparte de la diferencia de nombres de las entidades, las interfaces definidas tras las cláusulas PORT son iguales, es decir, no se hace referencia alguna a la función o comportamiento de los elementos.

Los puertos pueden ser de cuatro modos :

IN : Para puertos de entrada a la entidad de diseño.

OUT : Para puertos de salida de la entidad de diseño.

INOUT : Para puertos bidireccionales.

BUFFER : Para puertos con *driver* único y conectables a nodos simples.

Los puertos modo IN pueden ser leídos dentro de la entidad que los contiene, es decir, puede tomarse su valor para aplicarlo y obtener resultados, pero no pueden ser escritos internamente, ya que ello supondría alterar el valor externo con el que acceden las señales a la entidad de diseño, como resultado del valor que les asignaron sus *drivers*.

Los puertos modo INOUT , al ser bidireccionales, pueden ser escritos o leídos. Es el caso típico de los puertos conectados a un bus, o los puertos de datos de una memoria RAM.

Los puertos modo OUT tienen una restricción similar a los modo IN : El valor de un puerto modo OUT puede ser escrito internamente, es decir, se le puede asignar un valor por medio de unos *drivers*, que será el valor que sale del puerto de la entidad de diseño y, por tanto, el que la entidad aporta al nodo o red a la que el puerto esté conectado. Sin embargo, los puertos modo OUT no pueden ser leídos internamente dentro de la entidad en la que están declarados. Existen dos razones para esto : la primera es que no habría diferencia entre un puerto modo OUT y otro modo INOUT y la segunda razón, relacionada a lo que se denomina *valor efectivo o resuelto* de una señal. Este concepto distingue entre el valor individual que un puerto lleva al nodo donde se conecta y el valor presente en ese nodo como consecuencia de los diferentes puertos que acceden a él. Según la fuerza de las señales y el valor de las mismas, resultará un *valor efectivo* de la señal presente en el nodo, que será el valor real que puede ser leído. Así, el valor individual presente en una puerta de modo OUT solo puede garantizar su inalterabilidad si al nodo al que se conecta no llega ninguna otra señal. Esta restricción es la que define a los puertos de modo BUFFER.

Los puertos de modo BUFFER son puertos de salida cuyo valor está determinado por un único *driver* y que no pueden aplicarse a nodos a los que confluya más de una señal. Como consecuencia de esta restricción, los puertos BUFFER pueden ser leídos dentro de la entidad en que se les asigna valor. Sin embargo, la restricción hace que el empleo de este tipo de puertos sea poco frecuente, por lo que, si se desea leer el valor de puertos de modo OUT y no se desea declararlos de modo BUFFER, se dispone de dos alternativas :

1ª Declarar una señal interna en la región declarativa de la arquitectura. Esto equivale a intercalar una señal entre los *drivers* y el puerto. La señal interna, a la que se le asigna el valor que determinen los *drivers*, sí puede ser leída y su valor podrá ser asignado al puerto OUT con un retardo infinitesimal *delta* que implicará un ciclo de simulación adicional que no afecta al comportamiento del modelo VHDL.

2ª Cuando los *drivers* de la señal interna están dentro de una región de sentencias secuenciales, en lugar de declarar una señal interna se declara una variable local, del mismo tipo que la señal del *driver*, cuyos valores se asignan sin retardo alguno. El valor asignado a estas variables puede ser leído y asignado al puerto modo OUT.

Como se verá en las secciones que siguen, la declaración de estas señales o variables internas es una práctica habitual en el desarrollo de modelos VHDL.

## **1.2 DESCRIPCIÓN DE ARQUITECTURA**

Las arquitecturas son *unidades de diseño secundarias* referenciadas en VHDL con la palabra reservada ARCHITECTURE que se definen para una ENTITY, o *unidad de diseño primaria* y describen el comportamiento, la funcionalidad o la estructura de la entidad. Nótese que un elemento con una interfaz determinada puede tener distintas arquitecturas dentro de una misma funcionalidad, lo que equivale a diferentes formas de realizar la función, utilizando distintas *configuraciones* o interconexiones de elementos o, tal vez, describiendo el funcionamiento de la entidad directamente, como un elemento primitivo.

Si imaginamos la descripción inicial de un elemento, en la fase de especificación de un diseño, la descripción de funcionamiento es casi siempre algorítmica, por lo que cabe pensar que posibles arquitecturas algorítmicas de las puertas and2 y or2, podrían ser :

```
ARCHITECTURE comport OF and2 IS
BEGIN
```

```
    PROCESS ( e1,e2)
```

```
    BEGIN
```

```
        IF (e1 = '1') AND ( e2 = '1') THEN
```

```
            sal <= '1';
```

```
        ELSE      sal <= '1';
```

```
        END IF;
```

```
    END PROCESS;
```

```
END comport;
```

```
ARCHITECTURE comport OF or2 IS
```

```
BEGIN
```

```
    PROCESS (e1,e2)
```

```
    BEGIN
```

```
        IF (e1 = '0') AND ( e2 = '0') THEN
```

```
            sal <= '0';
```

```
        ELSE      sal <= '0';
```

```
        END IF;
```

```
    END PROCESS;
```

```
END comport;
```

A cualquier nivel, un elemento o subsistema se deberá modelar según el uso a que se vaya a destinar el modelo, por ejemplo, para simulación, para documentar sus especificaciones y uso posterior, para síntesis, etc. Así, aún sin conocer la estructura de las descripciones VHDL, podemos comparar tres arquitecturas diferentes de un mismo circuito digital sencillo, un sumador de bits, cuya declaración de entidad es :

```
ENTITY sumador IS
```

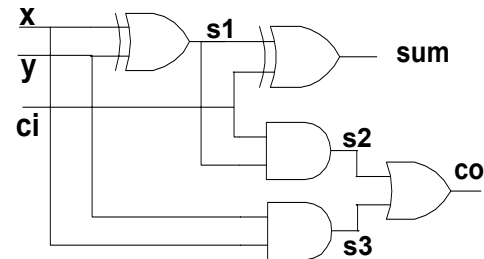
```
    PORT ( x,y,ci : IN BIT; sum, co : OUT BIT);
```

```
END sumador;
```

La primera arquitectura hace una descripción *funcional*, indicando el flujo de datos:

ARCHITECTURE funcional OF sumador IS

```
SIGNAL s1,s2,s3 : BIT;  
BEGIN  
  s1  <= x XOR y;  
  sum <= s1 XOR ci AFTER 10 ns;  
  s2  <= s1 AND ci;  
  s3  <= x AND y;  
  co  <= s2 OR s3 AFTER 12 ns;  
END funcional;
```



Hay una correspondencia directa entre la descripción VHDL y las puertas con las que se realiza el sumador, según el circuito adjunto. Como se verá con detalle más adelante, se declaran las señales internas s1,s2,s3 indicando su tipo -BIT-, en una zona reservada a declarar objetos *locales* a la arquitectura e inmediatamente se describe la funcionalidad de la entidad, expresando por medio del delimitador <= utilizado para asignar valor a señales, la relación entre señales de salida, sum y co, con las señales internas s1,s2,s3 que, a su vez, están relacionadas a las de entrada x, y, ci utilizándose las funciones XOR, AND y OR.

La segunda arquitectura que se muestra hace una descripción *estructural* :

```
USE WORK.modelos.ALL; -- acceso a biblioteca WORK  
ARCHITECTURE estructural OF sumador IS -- y paquete "modelos".  
  
  FOR ALL: xr2 USE ENTITY WORK.xr2; -- especificaciones de configuración  
  FOR ALL: and2 USE ENTITY WORK.and2; -- de primitivos o componentes  
  FOR ALL: or2 USE ENTITY WORK.or2; -- contenidos en paquete "modelos"  
  
  SIGNAL s1,s2,s3 : BIT; -- declaración de señales internas  
BEGIN  
  P1: xr2 PORT MAP (x,y,s1); -- instantiation statements  
  P2: xr2 PORT MAP (s1,ci,sum); -- sentencias de colocación o  
  P3: and2 PORT MAP (s1,ci,s2); -- sentencias de mapeo posicional  
  P4: and2 PORT MAP (x,y,s3);  
  P5: or2 PORT MAP (s2,s3,co);  
END estructural;
```

Es la que describe con más detalle la estructura del sumador, ya que indica a nivel de puertas una realización concreta del circuito :

- Obsérvese que declara las mismas señales internas que se declararon en la arquitectura funcional - s1,s2,s3.
- Es evidente que este tipo de arquitectura requiere un diseño previo de *hardware*, para establecer las interconexiones entre señales de entidad sumador - x, y, ci, sum, co -, las señales internas o locales - s1,s2,s3 - y las interfaces de las entidades primitivas que se utilizan como componentes disponibles, sobre las que aquí se hace un mapeo posicional de todas las señales por medio de las sentencias con las cláusulas PORT MAP. Se pueden hacer dos tipos de asociaciones :

1. *Asociación posicional* : Solo incluye los nombres de las señales *reales* del modelo o arquitectura, en orden y separados por comas. Los puertos *locales* se asocian a los *reales* relacionando el orden en la declaración de componente con el orden *posicional* :

P1 : xr2 PORT MAP( x,y,s1); -- Orden de señales fijado por componente

2. *Asociación nominal*: Relaciona explícitamente señales *locales* y *reales* por medio del delimitador =>, con el formato *local* => *real* , por ejemplo :

P5 : or2 PORT MAP ( sal => co, e1 =>s2, e2 => s3);

Al ser explícita, puede alterarse el orden de las señales en las declaraciones.

- Utiliza elementos *primitivos*, acudiendo para ello a entidades - xr2, or2, and2- que están disponibles en otras *unidades de diseño* denominadas empaquetamientos o paquetes y en VHDL con la palabra reservada PACKAGE. Para poder acceder al paquete y biblioteca que contiene los modelos de los primitivos, como se verá a continuación, se deben hacer visibles por medio de la cláusula USE que precede a la descripción de la arquitectura.
- Este tipo de arquitecturas permite optimizar la *configuración* de un diseño y elegir la arquitectura más idónea de cada entidad que interviene en el diseño *bottom-up*.

Finalmente, se muestra una tercera arquitectura con descripción *algorítmica* .

Aunque su significado puede intuirse, requiere un conocimiento de las sentencias secuenciales que la integran y que se verán en detalle en una sección posterior. Nótese que la descripción detalla la arquitectura sin referencia a constitución o estructura *hardware*.

```

ARCHITECTURE comportamiento OF sumador IS
BEGIN
sumbit:PROCESS ( x,y,ci)                                -- sentencia concurrente PROCESS
    VARIABLE var : BIT_VECTOR (1 TO 3);
    VARIABLE ind : INTEGER RANGE 0 TO 3;                 -- declaración de variables
    VARIABLE suma, carry : BIT;                          -- locales a PROCESS
    BEGIN
        ind := 0;                                         -- inicialización de variables locales
        var := x & y & ci;
        FOR i IN 1 TO 3 LOOP                             -- sentencia secuencial LOOP FOR
            IF (var(i)) = '1' THEN ind := (ind+1);        -- uso de variables locales
            END IF;
        END LOOP;
        CASE ind IS                                       -- sentencia secuencial CASE
            WHEN 0 => suma := '0';carry := '0';
            WHEN 1 => suma := '1';carry := '0';          -- uso de variables locales
            WHEN 2 => suma := '0';carry := '1';
            WHEN 3 => suma := '1';carry := '1';
        END CASE;
        sum <= suma AFTER 10 NS;
        co <= carry AFTER 12 NS;
    END PROCESS sumbit;                                  -- cierre de sentencia PROCESS
END comportamiento;
    
```

### **1.3 BIBLIOTECAS**

Después de describir un modelo, la tarea inmediata es realizar un análisis en el que se compruebe que el código escrito es correcto, para lo que se verifica normalmente:

- Léxico: Solo puede tener caracteres permitidos: No puede haber Ñ o ñ , por ejemplo.
- Sintaxis: Por ejemplo, si se ha utilizado BEGIN o IF, debe existir un END o END IF.
- Semántica : Se buscan inconsistencias en la descripción, por ejemplo, a un objeto de tipo BIT, puede asignarsele '1' o '0', pero no 1 ni 0, ya que estos son tipo INTEGER.

El resultado de estos análisis se almacena en bibliotecas, codificado en un formato apropiado para acceso por otras herramientas y para uso en otras aplicaciones tales como simulación, síntesis, etc. o para realizar nuevos análisis y utilizar los resultados en otras descripciones. Las bibliotecas son el recurso necesario de que se dispone en el entorno de trabajo para poder compartir las unidades de diseño, reutilizar diseños anteriores y llevar a cabo tareas de integración o configuración de subconjuntos, haciéndolos accesibles a los distintos usuarios y permitiendo que los modelos de un cierto nivel jerárquico no necesiten incluir el código de los submodelos, sino que sea suficiente hacer referencia a la biblioteca donde están descritos.

En VHDL existen dos bibliotecas predefinidas : la biblioteca STD y la WORK.

La STD está visible de forma implícita y no es necesario referirse a ella. Contiene los empaquetamientos STANDARD y TEXTIO. La biblioteca WORK es aquella donde se almacenan los resultados de los análisis en curso y, al utilizarla por defecto, no es necesario llamarla. Además, en VHDL se pueden usar las *bibliotecas de recursos* , a las que se accede para referencia de datos pero en las que no se puede escribir de manera directa.

Las bibliotecas son subdirectorios o ficheros de un *host*, cuyo sistema operativo los gestiona con un nombre *físico*. Desde el lado del usuario de VHDL, las bibliotecas se identifican con un nombre *lógico*, con el cual se accede a ellas o se las hace visibles, por medio de sentencias basadas en la cláusula USE, reservada en VHDL a este fin. La relación entre el nombre *físico* y el *lógico* es un aspecto a contemplar desde el sistema operativo del *host*, para lo cual no hay intervención del usuario de VHDL.

Las bibliotecas de recursos se acceden con la palabra reservada LIBRARY, por ejemplo:

```
LIBRARY chips;
```

La cláusula USE puede utilizarse de tres formas, con la notación de punto (.) :

USE LIBRARY. empaquetamiento;	USE chips. puertas;
USE empaquetamiento . objeto;	USE puertas. and2;
USE LIBRARY. empaquetamiento . objeto;	USE chips.puertas.and2;

en las que el nombre *-sufijo-* que sigue al punto (.) debe estar declarado o contenido en el empaquetamiento o biblioteca *-prefijo-* que precede al mismo punto. Como alternativa para evitar referencias largas o listas de objetos, es posible usar la cláusula .ALL; que visualiza todo lo declarado o contenido en su prefijo

## **1.4 DECLARACIONES DE GENÉRICOS Y COMPONENTES**

Cuando se desea disponer de información adicional que, sin estar directamente relacionada con la estructura del circuito, puede afectar al comportamiento de éste o a algún otro aspecto del mismo, en VHDL se hace uso de la declaración de GENÉRICOS a través de la que se puede pasar información genérica que posteriormente se particulariza en cada aplicación o componente donde su uso pueda ser relevante.

El uso típico de los genéricos es para introducir parámetros tales como tiempos de conmutación, por su influencia en posibles “carreras” o funcionamientos anormales, rangos o tamaños de objetos, como aplicación particular para dimensionar puertos, buses o estructuras iterativas donde un módulo se repite varias veces o como parámetros aislados del tipo temperatura, fanout, etc.

Al igual que los puertos, los genéricos se declaran en listas, permitiéndose objetos de clase CONSTANT y modo IN exclusivamente, por lo que no es necesario especificarlos. Sin embargo, sí es necesario especificar el tipo de la constante : TIME, INTEGER, etc. Dado que son de modo IN, pueden ser leídos, pero no asignados o transferidos a otros parámetros de modo OUT o INOUT.

El formato general de los genéricos es :

```
GENERIC( [identificador_de_constante : tipo[:= valor] ]
        { ; [identificador_de_constante : tipo[:= valor] ] } );
```

Por su similitud a los puertos, los genéricos se declaran dentro de un formato más avanzado o completo de las entidades, así como en las *declaraciones de componentes*, como se indica en los ejemplos siguientes:

ENTITY and2 IS	COMPONENT and2
GENERIC(tphl, tplh:TIME; fanout:POSITIVE);	GENERIC(tphl,tplh:TIME;fanout:POSITIVE);
PORT (e1,e2 : IN BIT; sal: OUT BIT);	PORT (e1,e2 : IN BIT; sal: OUT BIT);
END and2;	END COMPONENT;

La referencia a datos genéricos simplemente se anuncia o se describe como existente, pero no se detalla, salvo que se desee indicar que es una información por defecto, susceptible de ser actualizada o sobrescrita para cualquier caso o *colocación* de componente, es decir , que cada *colocación* de componente puede aportar sus valores particulares de genéricos.

La información genérica puede ser utilizada en expresiones, por ejemplo :

```
ARCHITECTURE generica OF and2 IS
  SIGNAL funcion : BIT;
BEGIN
  funcion <= e1 AND e2;
  sal <= funcion AFTER tphl WHEN funcion = '1'
        ELSE
        funcion AFTER tphi ;
END conmutaciones;
```

En la arquitectura *estructural* del sumador, supuesto que el componente and2 está descrito en el empaquetamiento con las expresiones de genéricos descritas arriba, se podría particularizar para las dos puertas and2 en la forma siguiente:

```
FOR ALL : and2 USE ENTITY WORK.and2;           -- paquete "modelos"
P3: and2 GENERIC MAP ( 20 ns, 30 ns, 5)
      PORT MAP (s1,ci,s2);                       -- mapeo posicional
P4: and2 GENERIC MAP ( 10 ns, 20 ns, 8)
      PORT MAP (x,y,s3);
```

pero si en la *declaración del componente* se tuviese :

```
COMPONENT and2
  GENERIC ( tphl :TIME := 15 ns, tphl :TIME := 25 ns; fanout: INTEGER := 6 );
  PORT ( e1,e2 : IN BIT; sal: OUT BIT );
END COMPONENT;
```

se podría hacer mapeo de aquellos componentes en los que se tenga interés y dejar el resto con los valores por defecto que tienen los genéricos en la declaración del componente :

```
P3: and2 PORT MAP (s1,ci,s2);                    -- sentencia de mapeo por defecto
P4: and2 GENERIC MAP( 10 ns, 20 ns, 8) -- sentencia de mapeo particular
      PORT MAP (x,y,s3);
```

Tanto el mapeo de genéricos como el de puertos, en lugar de ser posicional podría ser *nominal*, por ejemplo la entidad :

```
ENTITY rom IS
  GENERIC ( palabra, dimension : POSITIVE );
  PORT ( direccion : IN BIT_VECTOR ( 0 TO dimension-1 );
        datos : OUT BIT_VECTOR ( palabra-1 DOWNT0 0 ));
END rom;
```

podría estar asociada a un componente declarado como :

```
COMPONENT memorom
  GENERIC ( bit_datos, bits_direc : POSITIVE );
  PORT ( dir : IN BIT_VECTOR ( 0 TO dimension-1 );
        dat : OUT BIT_VECTOR ( palabra-1 DOWNT0 0 ));
END memorom;
```

y para un determinado empleo en una posición dentro de una placa, a la que se etiqueta como posición C3, y donde se requiere que tenga 256 palabras de 16 bits, por lo que se mapea *nominalmente* como :

```
C3 : memorom GENERIC MAP ( bit_datos => 16, bits_direc : 8);
      PORT MAP ( dat => progdata, dir => progdir( 0 TO 7 );
```

donde "progdata y progdir( 0 TO 7)" se deben suponer como los nombres de las señales del componente en posición C3.



## **1.5 DECLARACIÓN DE EMPAQUETAMIENTOS**

Cuando se tienen diseños de gran tamaño es conveniente particionar el modelo en varios ficheros o unidades de diseño que puedan analizarse y compilarse separadamente, manteniéndolas accesibles para los diferentes usuarios y diseñadores, para poder compartir los desarrollos y recursos comunes y asegurar la compatibilidad de los desarrollos.

En VHDL existen cinco *unidades de diseño* compilables separadamente, de las que ya se han mencionado dos, ENTITY y ARCHITECTURE. Sin embargo, desde un punto de vista de orden en la compilación, la primera unidad a compilar es la declaración del PACKAGE, empaquetamiento o paquete, referido en el apartado anterior.

La función de la declaración del PACKAGE es contener las declaraciones de elementos que se repiten con cierta frecuencia en otras unidades de diseño, a fin de evitar la repetición de código y reducir así el tamaño de las descripciones o ficheros VHDL. Su estructura es :

```
PACKAGE identificador IS
    declaraciones de tipos, constantes, componentes, subprogramas, etc.
END identificador;
```

Las declaraciones más habituales son las de tipos, subprogramas y componentes.

Para que otras unidades de diseño puedan utilizar las declaraciones contenidas en el empaquetamiento, éste deberá ser analizado y compilado previamente en *bibliotecas*, que serán accesibles o visibles por medio de la cláusula USE, con el formato genérico:

```
USE identificador_de_biblioteca . identificador_de_package . ALL;
```

así, por ejemplo, en la arquitectura *estructural* del sumador la sentencia

```
USE WORK.modelos. ALL;
```

tenía por objeto hacer visibles los componentes xr2, or2 y and2 declarados en el paquete denominado “modelos” dentro de la biblioteca WORK , con el siguiente formato:

```
PACKAGE modelos IS
    COMPONENT and2 PORT (e1,e2 : IN BIT; sal: OUT BIT); END COMPONENT;
    COMPONENT or2 PORT (e1,e2 : IN BIT; sal: OUT BIT); END COMPONENT;
    COMPONENT xr2 PORT (e1,e2 : IN BIT; sal: OUT BIT); END COMPONENT;
END modelos;
```

y evitar la inclusión de los tres componentes en la zona declarativa de la arquitectura.

La cláusula ALL visualiza todos los componentes del paquete, pero también pueden utilizarse los formatos siguientes :

```
USE WORK.modelos. and2;
LIBRARY componentes;      USE componentes.modelos.ALL;
```

La última línea se usaría en caso de tener la biblioteca “componentes” para almacenar el paquete “modelos”. Como se vió, las bibliotecas especiales hay que visualizarlas con USE.

Al ser el VHDL un lenguaje en el que el uso de tipos es una característica que afecta a todos los objetos que contienen un valor, los tipos deberán estar declarados antes de su uso. Por otra parte, la compatibilidad entre modelos se sustenta en que los objetos usados por los distintos ficheros y unidades empleen los mismos tipos. Por esto es casi obligado que los distintos diseñadores implicados en un desarrollo común compartan los paquetes y que estos sean la primera unidad compilada.

El empaquetamiento STANDARD, mostrado en el Apéndice A, está dentro de la biblioteca STD y contiene la declaración de los tipos básicos, tales como BIT, BIT\_VECTOR, CHARACTER, INTEGER y REAL sin los que sería imposible escribir código. Por esto, el empaquetamiento STANDARD está predefinido y visible por la cláusula implícita :

```
USE STD. STANDARD. ALL;
```

Dado que el paquete STANDARD contiene un número muy reducido de tipos, existe otro paquete standard del IEEE: el std\_logic\_1164, contenido en la biblioteca IEEE. Su uso requiere el empleo de USE y, por su importancia, se verá con algún detalle más adelante.

### **1.6 CUERPOS DE EMPAQUETAMIENTOS**

Son la *unidad de diseño secundaria* correspondiente a las declaraciones de paquetes.

Puede existir una declaración de paquete que no requiera la existencia de esta unidad secundaria. Esto ocurre siempre que no haya subprogramas ni constantes diferidas en la declaración.

Una constante diferida se declara, sin asignarle valor, en la unidad primaria del empaquetamiento y se le asigna en el cuerpo del empaquetamiento. De esta forma, si se quiere hacer simulaciones con distintos valores de esta constante diferida, no será necesario recompilar todas las declaraciones, sino solo el cuerpo del empaquetamiento donde se cambia de valor. Por esto, la declaración del empaquetamiento y su cuerpo son unidades de diseño compilables separadamente.

En VHDL, como se verá más adelante, existen dos tipos de subprogramas : funciones y procedimientos. En caso de que estos sean de uso frecuente por distintos usuarios será, tal vez, conveniente incluirlos en los empaquetamientos, para lo que será necesario declararlos en la unidad primaria y posteriormente definirlos en la unidad secundaria. De forma similar a las constantes diferidas, las distintas unidades de diseño pueden hacer referencia a subprogramas declarados en el empaquetamiento, pero la operación de los subprogramas es susceptible de modificaciones, al igual que las constantes diferidas. En tal caso, la recompilación solo afectaría al cuerpo del paquete y no a la unidad primaria donde está la declaración. El formato de los cuerpos de empaquetamientos es :

```
PACKAGE BODY identificador IS
    asignación de valor a constantes diferidas
    definiciones de subprogramas;
END identificador;
```

El identificador es el mismo usado para la declaración de PACKAGE.

## **1.7 DECLARACIÓN DE CONFIGURACIÓN**

Cuando se desea simular un diseño como el descrito con la arquitectura *estructural* del sumador en el que existan arquitecturas alternativas de sus componentes, en lugar de cambiar la arquitectura de los componentes en cada simulación, lo que se hace es una descripción genérica de la arquitectura, sin que figuren *especificaciones de configuración*. A cambio se construye una unidad de diseño, la declaración de configuración, en la que con un formato anidado se especifican, hasta el nivel jerárquico más bajo que se desee, las entidades de diseño o parejas ENTITY / ARCHITECTURE con las que se desea mapear a cada referencia o colocación de los componentes con los que se desea hacer la simulación. De esta forma, se mantiene la unidad de diseño de la arquitectura que se compila una vez y servirá para todas las simulaciones. A cambio, habrá que recompilar la configuración que resulta para cada alternativa o combinación de componentes que se desee simular.

La CONFIGURATION es una unidad de diseño *primaria*, pero que debe ser analizada y compilada después de haberlo hecho con la arquitectura a la que se configura.

Su formato, para el ejemplo de la arquitectura *estructural* del sumador, podría ser:

```
USE WORK.ALL;
CONFIGURATION ejemplo_de_config OF sumador IS
  FOR estructural
    FOR ALL : xr2
      USE ENTITY WORK. xor2 (lenta);           -- entity (architecture x)
    END FOR;
    FOR ALL: and2
      USE ENTITY WORK. and2 (rapida);          -- entity (architecture y)
    END FOR;
    FOR ordos : or2
      USE ENTITY WORK. or2;                    -- entity solamente y la
                                              -- arquitectura, por defecto
    END FOR;
  END FOR;
END ejemplo_de_config;
```

En caso de que un componente se desee configurar para una arquitectura específica, se hace referencia a la etiqueta particular de esa *colocación*, por ejemplo :

```
FOR P2 : xr2
  USE ENTITY WORK. xor2 ( especial);          -- entity (architecture x)
END FOR;
```

y si después de una etiqueta hasta el resto de ellas, todas las referencias están configuradas para la misma entidad / arquitectura, puede usarse la cláusula OTHERS, por ejemplo :

```
FOR OTHERS : semisum
  USE ENTITY WORK. semisum (lenta);           -- entity (architecture x)
END FOR;
```

Nótese que la descripción está precedida de USE WORK.ALL; con el fin de hacer visibles desde la arquitectura a la que se refiere la configuración, todas las entidades de diseño y declaraciones de los componentes.