

5. Sentencias Concurrentes

Son las sentencias que se usan directamente dentro del cuerpo de las arquitecturas y, a diferencia de lo habitual en los lenguajes de programación, estas sentencias no se ejecutan en el orden en que están escritas sino que solo se ejecutan cuando cambia alguna de las señales de las que dependen los resultados que se computan en las sentencias. Otra característica es que todas las sentencias se ejecutan una vez al principio de la simulación. Algunas de las sentencias que se ven en esta sección pueden ser concurrentes o secuenciales, dependiendo del área en la que estén. Entre éstas están las ASSERT, vistas en la sección de secuenciales, las asignaciones simples de valor a señales y las llamadas concurrentes a procedimientos.

5.1 Sentencias de COLOCACIÓN DE COMPONENTES

En VHDL, los componentes son elementos que se declaran como una caja negra y de la que solo se describe su interfaz. El formato de declaración de componentes es:

```
COMPONENT nombre_de_componente      -- no incluye la palabra reservada IS
    PORT (local1, local2,... localn : modo y tipo);
END COMPONENT;
```

Tanto en los componentes como en las entidades, los puertos *locales* se asocian a *señales físicas* y, aunque similares en formato, hay diferencias importantes entre ambos:

- Una entidad es una unidad de diseño primaria, compilable directamente.
- Un componente es como una plantilla que, en alguna sentencia posterior a la de su referencia, se mapeará con una entidad. De ahí la semejanza de sus declaraciones.

La declaración del componente puede ocurrir dentro del cuerpo de la arquitectura que referencia al componente, pero también puede hacerse en un empaquetamiento. Es posible colocar un componente que posteriormente se asociará a una entidad, al *especificar la configuración*. De no existir esta alternativa, cualquier referencia a una entidad de nivel jerárquico inferior condicionaría la compilación de la arquitectura a la de dicha entidad, que sería una restricción incómoda y que forzaría a un excesivo orden en la compilación de unidades y descripciones.

Las sentencias de colocación de componentes los “insertan” en las descripciones, lo que equivale a hacer una lista de los componentes que constituyen los circuitos a describir. El formato simple de estas sentencias es :

```
etiqueta_obligatoria : nombre_de_componente
    [ GENERIC MAP ( [nombre_genérico => ] expresión
                    {, [nombre_genérico => ] expresión } ) ]
    PORT MAP ( [nombre_de_puerto => ] expresión
               {, [nombre_de_puerto => ] expresión } );
```

La etiqueta es obligatoria, ya que al configurar el componente se hará por la etiqueta.

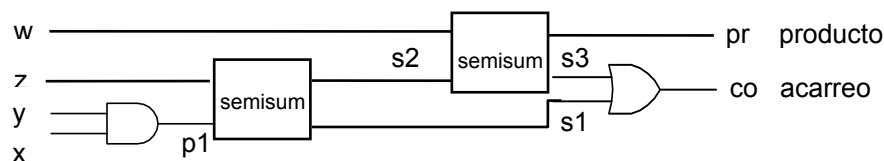
La lista de asociación de genéricos es opcional, como se indica en el formato, y las expresiones allí mencionadas hacen referencia a valores del tipo apropiado a la constante o parámetro a que se refiere el parámetro genérico.

La lista de asociación de puertos, lógicamente, es obligatoria ya que hace el mapeo del componente a que se refiere la sentencia. La lista relaciona señales *locales* del componente, citadas en su declaración, con las *reales* de la arquitectura donde se utiliza y a las que se conecta el componente. Puede utilizarse opcionalmente la asociación nominal indicada en el formato o la posicional con el orden fijado por la declaración del componente. Las expresiones que se indican en el formato de la sentencia referencian a una señal.

En los apartados anteriores se ha visto que visualizando biblioteca y empaquetamientos se puede acortar el tamaño de las descripciones. En consecuencia, conviene definir como componentes a ciertos subsistemas desarrollados a partir de otros de nivel inferior, pero susceptibles de ser utilizados como componentes en otros de nivel superior. Por ejemplo, partiendo de puertas básicas se ha visto que pueden realizarse descripciones de subsistemas de nivel más alto, como comparadores, semisumadores, sumadores, etc. El proceso de diseño *bottom up* puede continuar, pero en cada nuevo nivel que se vaya creando, se utilizarán los subsistemas que se han desarrollado en el nivel inferior como simples componentes, supuesto que éstos están compilados y declarados en un empaquetamiento. Por ejemplo, partiendo de los semisumadores de la sección 1.2, puede declararse un componente como :

```
COMPONENT semisum PORT (a,b : IN BIT; ca, su: OUT BIT);    END COMPONENT;
```

y utilizarlo para modelar un multiplicador de un bit, cuya estructura y descripción son :



```
USE WORK.modelos.ALL;          -- Permite disponer de las declaraciones de modelos
ENTITY multunbit IS
    PORT (x,y,z,w : IN BIT; co, pr : OUT BIT);
END multunbit;
```

```
ARCHITECTURE estructural OF multunbit IS
    -- Aquí ya no es necesario declarar los componentes declarados en el paquete
    FOR all:semisum USE ENTITY WORK.semisum;
    FOR all:and2    USE ENTITY WORK.and2;
    FOR all:or2     USE ENTITY WORK.or2;
    SIGNAL p1, s1, s2, s3 : BIT;          -- Señales internas a mapear
    BEGIN
        C1: and2      PORT MAP (x,y, p1);          -- sentencias de colocación de
        C2: semisum    PORT MAP (p1,z,s1,s2);      -- componentes declarados en
        C3: or2        PORT MAP (s1,s3,co);        -- el empaquetamiento de los
        C4: semisum    PORT MAP (s2,w,s3,pr);      -- modelos, visualizado con USE
    END estructural;
```

y a su vez, el multiplicador de 1 bit puede ser declarado como un componente y añadido al empaquetamiento de modelos, donde figuran todos los que hasta ahora se han utilizado:

```
COMPONENT MULTUNBI PORT (x,y,z,w : IN BIT; co, pr: OUT BIT); END COMPONENT;
```

En el Apéndice C se tiene el empaquetamiento “modelos”, desarrollado para utilizarlo en los ejemplos que se exponen en los apartados que siguen.

5.2 Sentencias BLOCK

Las sentencias BLOCK permiten hacer agrupamientos de sentencias concurrentes en los que se pueden establecer una región declarativa y otra activa o ejecutable. Así, igual que un sistema, una arquitectura puede partitionarse en bloques según criterios de funcionalidad, jerarquía, etc. La arquitectura puede considerarse un bloque que engloba a otros de nivel inferior, lo que equivale a establecer una jerarquía o anidamiento de bloques.

El formato genérico de una arquitectura particionada en bloques podría ser :

```
ARCHITECTURE particionada OF sistema IS
    región declarativa de la arquitectura
BEGIN
    región ejecutable de la arquitectura, con sentencias concurrentes
    etiqueta_de_bloque_1 : BLOCK
        área declarativa de bloque_1
        BEGIN
            área activa de bloque_1, con sentencias concurrentes
        END BLOCK [etiqueta_de_bloque_1];
    etiqueta_de_bloque_2 : BLOCK
        área declarativa de bloque_2
        BEGIN
            área activa de bloque_2, con sentencias concurrentes
        END BLOCK [etiqueta_de_bloque_2];
END particionada;
```

y el formato más detallado de los bloques es:

```
etiqueta_de_bloque :    BLOCK [ (expresión de GUARDA) ]
                        cabecera del bloque
                        área declarativa de bloque_1
                        BEGIN
                            área activa de bloque_1 de sentencias concurrentes
                        END BLOCK [etiqueta_de_bloque];
```

en donde :

```
cabecera del bloque [cláusula de genéricos [ GENERIC MAP ( lista de asociación) ;] ]
                   [cláusula de puertos      [ PORT MAP ( lista de asociación) ;] ]
```

Un aspecto importante a considerar en esta descripción es la visibilidad de puertos y objetos. Todos ellos son visibles para cualquier bloque interno que se establezca dentro de

la arquitectura de sistema, es decir, las señales de tipo IN pueden ser leídas y a las de tipo OUT se les puede asignar valores. Por el contrario, las señales de la región declarativa de la arquitectura son señales internas de ésta y no son utilizables fuera de ella, pero sí por los bloques internos en que se particione la arquitectura.

Al igual que las señales internas de la arquitectura, las señales internas de bloques no son utilizables fuera de ellos. Cualquier bloque de nivel inferior puede utilizar las señales del entorno o arquitectura que lo engloba, pero los bloques de nivel alto no pueden hacer uso de las señales internas de bloques de igual o inferior nivel. Este hecho limita el uso de los bloques para estructurar diseños.

Un ejemplo del uso de bloques para particionar arquitecturas podría ser :

```
ENTITY procesador IS
    PORT (reloj :    IN BIT;
          direc :   OUT BIT_VECTOR ( 31 DOWNT0 0 );
          datos : INOUT BIT_VECTOR ( 15 DOWNT0 0 );
          controles : OUT control_word;
          ready :    IN BIT );
END procesador;

ARCHITECTURE bloques OF procesador IS

    TYPE bus_datos IS ARRAY ( 15 DOWNT0 0 );
    SIGNAL interna : bus_datos;

BEGIN
    unidad_control : BLOCK
        PORT (      clk : IN BIT;
                  bus_control : OUT control_word;
                  bus_ready : IN BIT;
                  control : OUT bus_datos);

        PORT MAP ( clk => reloj,  control => interna,
                  bus_ready => ready,  bus_control => controles);
        -- otras declaraciones del bloque unidad_control
        BEGIN
            -- sentencias concurrentes del bloque unidad_control
        END BLOCK unidad_control ;

    databus_1 : BLOCK
        PORT (      dir :    OUT BIT_VECTOR (31 DOWNT0 0);
                  dabus_1 : INOUT BIT_VECTOR (15 DOWNT0 0);
                  control :   OUT bus_datos );
        PORT MAP( dir => direc , control => interna , dabus_1 => datos );
        -- otras declaraciones del bloque databus_1
        BEGIN
            -- sentencias concurrentes del bloque databus_1
        END BLOCK databus_1;

END bloques;
```

La descomposición en bloques o módulos de una arquitectura permite el diseño aislado de cada uno de los bloques, siempre con el condicionamiento de mantener las

interfaces que se han definido en modos y tipos. Sin embargo, a la hora de estructurar diseños existen otras posibilidades, tal vez más fáciles de usar que los bloques con sus anidamientos y la posterior configuración.

En las descripciones *estructurales*, se puede definir un componente del tamaño o complejidad que se desee, equivalente a un bloque, combinado con la posibilidad de tenerlo definido dentro de un empaquetamiento y evitando su inclusión en la descripción general. Es importante tener presente las ventajas de utilizar empaquetamientos.

También es posible recurrir al uso de procedimientos concurrentes, almacenables en empaquetamientos, pensando no solo en el desarrollo, sino también en la depuración de los diseños. Por último, hay que tener presente la capacidad descriptiva de los procesos. A veces puede ser conveniente utilizar procesos en lugar de bloques.

5.2.1 BLOQUES GUARDADOS

Es una funcionalidad de las sentencias BLOCK, consistente en la posibilidad de utilizar el valor de una expresión booleana que permite habilitar o inhibir alguno o todos los *drivers* del bloque. El ejemplo que sigue muestra este tipo de bloques:

```
ENTITY biestable IS
    port ( e1,e2, permiso : IN BIT; s1,s2 : OUT BIT);
END biestable;

ARCHITECTURE guardadora OF biestable IS
BEGIN
    blk: BLOCK ( permiso = '1' )
        BEGIN
            s1 <= GUARDED e1 after 10 ns;
            s2 <= e2 after 15 ns;
        END BLOCK blk;
END guardadora;
```

La expresión entre paréntesis que sigue a BLOCK se denomina *expresión de guarda*. Cuando la condición que en ella se expresa sea FALSE, las señales que dentro del bloque estén precedidas de la cláusula GUARDED están bloqueadas y no existe asignación de señal. Las señales no precedidas de la cláusula GUARDED no son afectadas por la expresión de guarda y evolucionan normalmente, según indique su correspondiente *driver*. Nótese que los bloques guardados permiten un tipo especial de asignación de valor a señales, que se conoce como *asignación guardada*. La sentencia con la cláusula GUARDED se ejecuta cuando:

- La expresión de guarda es cierta y hay un cambio en el *driver*.
- Hay un cambio en la expresión de guarda de falso a cierto.
- Si hay un cambio de cierto a falso, la señal mantiene el valor previo .

Existe una señal implícita en este tipo de bloques guardados, a la que se denomina señal GUARD, que no puede ser alterada en sentencia de forma explícita, pero que puede utilizarse como parte de una expresión.

5.3 Sentencias PROCESS

Es una sentencia concurrente que permite modelar el funcionamiento y el retardo de un dispositivo digital y que, al igual que los subprogramas, solo admite sentencias secuenciales en su región activa. Es la sentencia utilizada en VHDL para descripción algorítmica y tiene el formato siguiente :

```
[etiqueta :] PROCESS [ ( lista de sensibilidad ) ]  
    -- región declarativa  
    BEGIN  
    -- región activa . Solamente sentencias secuenciales  
    END PROCESS [etiqueta :]
```

El área declarativa admite declaración de tipos, constantes, variables y subprogramas locales al proceso, pero *nunca* de señales.

La [(*lista de sensibilidad*)] indica las señales cuyo cambio de valor inicia la ejecución del proceso, siendo suficiente que haya un evento en cualquiera de ellas. Como indica el formato, la lista de señales es optativa, pero en caso de no figurar, es necesario que haya dentro de la región activa del proceso alguna sentencia WAIT, para evitar que el proceso esté ejecutándose continuamente. La opción WAIT equivalente a la lista de sensibilidad es la WAIT ON señales, ya que suspende la espera cuando hay un evento en cualquiera de las señales que siguen a WAIT ON. Dado que cuando se inicia una simulación todos los procesos se ejecutan una vez, la sentencia WAIT ON debe ser la última de las sentencias secuenciales, a fin de permitir la ejecución de todas las que le preceden en la región secuencial del proceso. Para evitar conflictos en la activación de un proceso, la lista de sensibilidad es incompatible con sentencias WAIT dentro del proceso o en los posibles subprogramas invocados desde la región secuencial.

Dentro de la región secuencial solo se permiten sentencias secuenciales. La ejecución de éstas tiene lugar en el orden en que figuran en la descripción, por lo que, como en otros lenguajes algorítmicos, el orden de las sentencias es significativo, a diferencia de lo que ocurre en regiones activas concurrentes. Todas las sentencias se ejecutan en un tiempo cero.

Un proceso siempre está activo, lo que equivale a un subprograma con un bucle iterativo infinito pero, a diferencia de los subprogramas, los procesos no pueden anidarse por lo que si es necesario hacer anidamientos dentro del proceso se invocan subprogramas, ya que estos actúan indistintamente como sentencias concurrentes o secuenciales.

Dado que los procesos son sensibles a señales y que éstas son el objeto de comunicación entre los procesos, pueden existir sentencias internas a un proceso del tipo

señal_a <= expresión

que, aunque tiene el mismo formato que la asignación concurrente, es una sentencia de asignación secuencial. Las asignaciones condicionales y selectivas de señal son sentencias exclusivamente concurrentes por lo que no podrán utilizarse dentro de los procesos.

Si hay varias asignaciones secuenciales a una misma señal dentro de un proceso, la única asignación efectiva será la última y lo será cuando el proceso haya terminado, ejecutándose todas las sentencias secuenciales que lo componen.

5.3.1 RETARDO Y CONCURRENCIA EN EL ENTORNO DE PROCESS

El retardo de propagación es un concepto fundamental para modelar sistemas digitales, donde las señales fluyen en paralelo. Si los modelos deben representar circuitos físicos, los puertos se asocian a señales y las arquitecturas describen comportamientos de entidades, todos los objetos declarados para puertos y arquitecturas deberán ser señales.

En VHDL el retardo de propagación se asocia a señales. Por el contrario, las variables están asociadas al desarrollo de algoritmos y no tienen retardo asociado por lo que su empleo está limitado a procesos y subprogramas, donde no tiene sentido declarar señales, aunque sea posible asignarles valor como resultado del algoritmo.

Dado que las sentencias dentro de los cuerpos secuenciales se ejecutan en el orden en que están escritas, el retardo asociado a las señales que sean asignadas durante un proceso, dependerá tanto del orden de las sentencias como del tipo de éstas. Por ejemplo :

```
ENTITY promediador IS
    PORT ( w,x,y,z : IN INTEGER; P : OUT INTEGER);
END promediador;
ARCHITECTURE inmediata OF promediador IS
BEGIN
    PROCESS ( w,x,y,z)
        VARIABLE P1, P2 , PT: INTEGER;
    BEGIN
        P1 := w + x;
        P2 := y + z;
        PT := P1 + P2;
        P <= PT;
    END PROCESS;
END inmediata;
```

Es evidente que el orden en la ejecución de sentencias es significativo en el resultado de P: al menos P1 y P2 deben ejecutarse antes que PT, y ésta antes de asignar el valor a P. Por el contrario, si la arquitectura fuese :

```
ARCHITECTURE indirecta OF promediador IS
    SIGNAL P1,P2 : INTEGER;
BEGIN
    PROCESS ( w,x,y,z)
    BEGIN
        P1 <= w + x    AFTER 20 ns;
        P2 <= y + z    AFTER 20 ns;
        P <= P1 + P2   AFTER 30 ns;
    END PROCESS;
END indirecta;
```

al hacerse la asignación a las señales P1 y P2 con un retardo de 20 ns, da igual el orden con que se ejecuten : El proceso se repite indefinidamente, activándose cada vez que hay un evento en alguna de las señales de la lista de sensibilidad.

Cuando en lugar de tener retardos especificados con tiempo finito y un proceso se tiene:

```
ARCHITECTURE sin_retardo OF promediador IS
    SIGNAL P1,P2 : INTEGER;
BEGIN
    P1 <= w + x;
    P2 <= y + z;
    P  <= P1 + P2;
END sin_retardo;
```

existe la tendencia a seguir considerando un orden en la ejecución, e incluso a concluir que los valores que resultan de ejecutar P1 en primer lugar, y P2 a continuación, serán los argumentos con que se ejecute inmediatamente después la 3ª sentencia que asigna el valor a P. Sin embargo, si consideramos cada sentencia como un proceso, por ejemplo:

P1 <= w + x ;	equivale a	PROCESS (w,x);
		BEGIN
		P1 <= w + x;
		END PROCESS;

entonces la estructura se ve como tres procesos concurrentes, cada uno con *retardo delta* por defecto, y cada proceso activado por los cambios de las señales a las que es sensible. Esto significa que el orden es indiferente y que cada vez que haya un cambio en w, x, y o z, habrá otro cambio en el valor de P, que se manifestará con un retardo de 2*delta, uno debido al retardo por defecto en P1 o P2 y el otro debido a P. El retardo total será n*delta, donde n indica el número de procesos que se hayan ejecutado, cada uno aportando un delta, pero considerando que cada ciclo de simulación que se avance o cada delta que se añada no aumenta el tiempo de simulación, lo que es de hecho una definición del retardo delta.

La conclusión que se obtiene es que las asignaciones concurrentes de señales sin especificar un tiempo de simulación o retardo finito, debido al retardo delta implícito, se comportan de forma similar a aquellas en las que se especifican retardos. De ahí resulta que cualquier asignación a señales supone un retardo, es decir,

S <= valor ; o S <= valor AFTER expresión de tiempo;

no actualizan a S de manera inmediata y, en consecuencia, cualquier otra sentencia que dependa de S y se ejecute en el curso del ciclo de simulación actual, utilizará el antiguo valor de S, no el que está adquiriendo ahora que estará disponible a partir del siguiente ciclo de simulación. En el caso de la sentencia con cláusula AFTER, la consecuencia es clara y dice que el resultado no se asigna hasta que hayan transcurrido n unidades de tiempo, lo que supone avanzar el tiempo de simulación en ese lapso, y que todas las sentencias que se ejecuten con S durante ese lapso, tomarán su antiguo valor. Por un razonamiento similar, cuando el retardo es delta, el nuevo valor estará disponible *después de delta* y podremos considerar tantos deltas como se quiera sin que el tiempo de simulación avance.

Los ejemplos de arquitecturas que siguen muestran las diferencias en retardo que se obtienen dependiendo de la descripción realizada. Para ello se utiliza un simple inversor, cuya entidad se declara inicialmente y una variable o señal que ayuden a ver las diferencias entre las distintas arquitecturas de la misma entidad :


```

ENTITY inv_retard IS
    PORT ( e : IN BIT; sal : OUT BIT);
END inv_retard;

ARCHITECTURE un_nano OF inv_retard IS
BEGIN
    PROCESS ( e)
        VARIABLE v : BIT;
    BEGIN
        v := e; -- asignación inmediata
        sal <= NOT v AFTER 1 ns; -- retardo un nanoseg.
    END PROCESS;
END un_nano;

ARCHITECTURE otro_nano OF inv_retard IS
    SIGNAL v : BIT;
BEGIN
    v <= e; -- retardo un delta
    sal <= NOT v AFTER 1 ns; -- retardo un nanoseg.
END otro_nano;

ARCHITECTURE dos_nanos OF inv_retard IS
    SIGNAL v : BIT;
BEGIN
    v <= e AFTER 1 ns; -- retardo un nanoseg.
    sal <= NOT v AFTER 1 ns; -- retardo un nanoseg.
END dos_nanos;

ARCHITECTURE doble_e OF inv_retard IS
    SIGNAL v : BIT;
BEGIN
    PROCESS( e )
    BEGIN
        v <= e; -- retardo un delta.
        sal <= NOT v; -- retardo un delta, pero después de que cambie e.
    END PROCESS; -- por segunda vez, ya que el proceso se activa con e.
END doble_e; -- Retardo indefinido. Condicionado a retardo de 2ª e.

ARCHITECTURE doble_delta OF inv_retard IS
    SIGNAL v : BIT;
BEGIN
    PROCESS ( e,v ) -- lista de sensibilidad aumentada a señal v
    BEGIN
        v <= e; -- retardo un delta.
        sal <= NOT v; -- retardo un delta, pero después de que cambie v.
    END PROCESS; -- ya que el proceso se activa con e y con v .
END doble_delta; -- retardo total dos deltas.

```

y ésta última equivale a la siguiente :

```

ARCHITECTURE dos_deltas OF inv_retard IS
    SIGNAL v : BIT;
BEGIN
    v <= e; -- retardo un delta
    sal <= NOT v; -- retardo un delta
END dos_deltas;

```

5.4 Sentencias de ASIGNACIÓN DE VALOR A SEÑALES

Estas sentencias, manteniendo su formato, pueden ser concurrentes o secuenciales, dependiendo de que se encuentran en un área de sentencias concurrentes de una arquitectura o dentro de un área secuencial, un proceso o subprograma.

La ejecución de la sentencia ocurre siempre que hay un cambio o evento en la expresión que define el valor que se asigna a la señal, por ejemplo :

```
[etiqueta_ de _reloj :] reloj <= impulsos AND permiso;
```

la señal *reloj* se activará una vez al comienzo de la simulación del modelo y posteriormente siempre que haya un cambio en las señales *impulsos* y *permiso*, por ser la señal *reloj* sensible a las señales *impulsos* y *permiso* o, expresado en otra forma habitual en VHDL, porque éstas constituyen su *lista de sensibilidad*.

Siempre que un cambio en las señales *drivers*, escritas a la derecha del delimitador compuesto *<=*, provoca un cambio en el valor de la señal destino, a la izquierda del delimitador, se tiene un evento en la señal a la que se asigna valor. Sin embargo, pueden ocurrir cambios en los *drivers* que no provoquen eventos en la señal destino, por ejemplo, si la señal *permiso* fuese '0', los cambios de '0' a '1' o de '1' a '0' en la señal *impulsos* no ocasionarían eventos en la señal *reloj*. En estos casos, en la literatura americana, se dice que en la señal destino, en lugar de eventos, hay *transactions*.

5.4.1 RETARDOS INERCIALES Y DE TRANSPORTE

La ejecución de la sentencia, es decir, el evento o asignación del nuevo valor a la señal destino se hace con un retardo infinitesimal denominado *retardo delta*, a menos que se especifique un retardo concreto, por ejemplo:

```
[etiqueta_ de _reloj :] reloj <= impulsos AND permiso AFTER 30 ns;
```

con el que se modela un retardo de 30 ns en la propagación del efecto de las señales *drivers* en la señal destino. Lo anterior nos permite modelar una sencilla puerta NAND como :

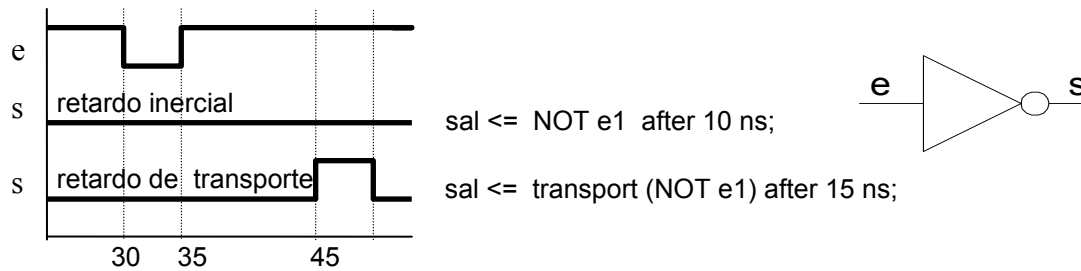
```
ENTITY nand2 IS
  PORT ( e1,e2 : IN BIT; sal: OUT BIT);
END nand2;
ARCHITECTURE lenta OF nand2 IS
BEGIN
  sal <= e1 NAND e2 AFTER 200 NS;
END lenta;
```

en la que, además de los conceptos anteriores, está implícito un retardo *inercial* por defecto, que implica que cualquier cambio en las señales *drivers* cuya duración sea menor que el retardo de propagación - 200 ns - no se transmitirá a la señal destino. Es una forma de modelar por defecto la inercia al cambio de estado en un circuito o la pérdida de impulsos estrechos de ruido o de especificar mínimas anchuras de impulso a las que responde un circuito lento. Si se desea expresar un retardo no inercial, hay que utilizar la palabra reservada *TRANSPORT*, con la que se modelan retardos en la propagación

independientes de la duración de los impulsos que se propagan, por ejemplo :

```
sal <= TRANSPORT ( e1 NAND e2 ) AFTER 200 ns;
```

Si se especifica *retardo de transporte* de forma explícita, se modela un dispositivo que transmite los cambios siempre. La figura adjunta muestra estos retardos para un inversor con un retardo de propagación de 10 ns.



5.4.2 ASIGNACIONES CONDICIONALES

Su formato es

```
[etiqueta :] nombre <= [transport] onda1 WHEN condición1 ELSE
onda2 WHEN condición2 ELSE
....
ondaN WHEN condiciónN ELSE
ondaX;
```

Tan pronto una de las condiciones se cumple, es TRUE, la sentencia se ejecuta y termina. Si no se cumple ninguna de las condiciones, se asigna el valor por defecto - ondaX -, como muestra el ejemplo para un inversor triestado, con especificación de parámetros genéricos :

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY inv_tristado IS
    GENERIC ( tplt : TIME:= 10 NS; tphl: TIME := 12 NS);
    PORT ( e1,en: IN BIT; sal: OUT std_logic);
END inv_tristado;
ARCHITECTURE tres_salidas OF inv_tristado IS
BEGIN
    sal <= '1' AFTER tplt WHEN ( e1 = '0' AND en = '1' ) ELSE
        '0' AFTER tphl WHEN ( e1 = '1' AND en = '1' ) ELSE
        'Z' AFTER tplt;
END tres_salidas;
```

La sentencia de asignación se ejecuta siempre que hay un cambio o evento en cualquiera de las señales a las que es sensible, es decir, en este caso del inversor triestado del ejemplo, siempre que cambie el valor de las señales e1 o en.

Nótese que en el ejemplo los valores '1', '0' y 'Z' son fijos, pero podrían ser *ondas* especificadas como una relación de valores en tiempos definidos en cada caso, por ejemplo :

```
ENTITY generador_serie IS
    PORT(ent: IN INTEGER RANGE 0 TO 3 ; sal: OUT BIT);
END generador_serie;

ARCHITECTURE reducida OF generador_serie IS
BEGIN
    sal <= '1' AFTER 30 ns, '0' AFTER 40 ns WHEN ent = 1 ELSE
        '1' AFTER 10 ns, '0' AFTER 20 ns WHEN ent = 2 ELSE
        '1' AFTER 30 ns, '0' AFTER 40 ns WHEN ent = 3 ELSE
        '0';
END reducida;
```

En este ejemplo, la condición es de tipo INTEGER y la salida es una forma de onda que cambia según el valor de la señal ent. La sentencia se ejecuta cada vez que cambia el valor de la señal ent o cuando haya un cambio en la señal *driver* de la salida. El caso más general es que las formas de onda descritas en el ejemplo con valores discretos sean expresiones de otras señales de entrada, por ejemplo:

```
ENTITY peque_alu IS
    PORT( a,b, sel1,sel2 : IN BIT; sal: OUT BIT);
END peque_alu;

ARCHITECTURE funcional OF peque_alu IS
    SIGNAL sel : BIT_VECTOR (0 TO 1);
BEGIN
    sel <= sel1 & sel2;
    sal <= a AND b      AFTER 10 ns WHEN sel = "11" ELSE
        a OR b         AFTER 10 ns WHEN sel = "10" ELSE
        a XOR b         AFTER 10 ns WHEN sel = "01" ELSE
        NOT(a XOR b)    AFTER 10 ns;
END funcional;
```

En este ejemplo de una ALU pequeña, la asignación a la señal sal ocurre siempre que cambien a o b , así como cuando cambie cualquiera de las señales de selección sel1 o sel2, además de la asignación que siempre ocurre una vez al comienzo de la simulación.

La asignación condicional es equivalente al proceso siguiente :

```
asig_cond : PROCESS ( a , b , condic )
BEGIN
    IF condic = xx      THEN sig <= expresión_1;
    ELSIF condic = xy   THEN sig <= expresión_2;
    ELSIF condic = yy   THEN sig <= expresión_3;
    ELSE                THEN sig <= expresión_4;
    END IF;
END PROCESS asig_cond;
```

5.4.3 ASIGNACIONES SELECTIVAS

Recuerdan a la sentencia secuencial CASE y tienen el formato siguiente :

```
[etiqueta :] WITH expresión SELECT
    nombre <= [transport] onda1 WHEN grupo de selección1,
    onda2 WHEN grupo de selección2,
    ....
    ondaN WHEN grupo de selecciónN,
    [ondaX WHEN OTHERS];
```

La sentencia se ejecuta cuando ocurre un evento en cualquiera de las señales de la expresión que determina la selección que se hará entre las alternativas de la sentencia, o en cualquiera de las expresiones que definan las ondas que se seleccionan con la expresión. Véanse los ejemplos siguientes :

```
ENTITY decoder_3_a_8 IS
    PORT ( adr : IN BIT_VECTOR ( 2 DOWNT0 0 );
          sal : OUT BIT_VECTOR ( 7 DOWNT0 0 ) );
END decoder_3_a_8;

ARCHITECTURE directa OF decoder_3_a_8 IS
BEGIN
    WITH adr SELECT
    sal <= "00000001" AFTER 15 NS WHEN "000", -- atención a "comas" (,)
    "00000010" AFTER 15 NS WHEN "001",
    "00000100" AFTER 15 NS WHEN "010",
    "00001000" AFTER 15 NS WHEN "011",
    "00010000" AFTER 15 NS WHEN "100",
    "00100000" AFTER 15 NS WHEN "101",
    "01000000" AFTER 15 NS WHEN "110",
    "10000000" AFTER 15 NS WHEN "111"; -- atención: "punto y coma" (;)
END directa;
```

La asignación selectiva de señal equivale igualmente a otro proceso como sigue:

```
asig_sel : PROCESS ( sel1, sel2 , a, b, c )
BEGIN
    CASE ( sel1 & sel2 ) IS
        WHEN "10"      => sig <= a AFTER 10 ns;
        WHEN "01"      => sig <= b AFTER 20 ns;
        WHEN OTHERS => sig <= a AND b AFTER 30 ns;
    END CASE;
END PROCESS asig_sel;
```

El ejemplo siguiente utiliza el paquete STD_LOGIC_1164 y, además muestra la posibilidad de hacer un grupo de selecciones por medio del delimitador | que equivale a "OR". También se incluye al final de la sentencia la cláusula WHEN OTHERS ya que en los grupos de selección anteriores no se han indicado todas las posibles combinaciones de la señal selects que al ser std_logic presenta combinaciones no expresadas explícitamente, pero que todas se concentran en OTHERS.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY mux_cuatro_a_1 IS
    PORT (i0,i1,i2,i3, s0,s1: IN std_logic;  z:OUT std_logic);
END mux_cuatro_a_1;

ARCHITECTURE multivaluada OF mux_cuatro_a_1 IS
    SIGNAL selects : std_logic_vector(0 TO 1);

BEGIN
    selects <= s1&s0;    -- concatenación de "s1" y "s0" en "s1s0".

    WITH selects SELECT
        z <= i0      AFTER 7 ns      WHEN "00",
            i1      AFTER 7 ns      WHEN "01" | "0Z",
            i2      AFTER 7 ns      WHEN "10" | "Z0",
            i3      AFTER 7 ns      WHEN "11" | "ZZ",
            'Z'     AFTER 15 ns     WHEN OTHERS;

END multivaluada;
```

Nótese que las señales $i0 \sim i3$ pueden ser formas de onda o señales internas a la arquitectura dependientes de las señales de entrada a través de expresiones. La sentencia se ejecuta siempre que cambie alguna señal de selección o una entrada de la que dependa el *driver* seleccionado.

5.4.4 ASIGNACIÓN GUARDADA

La asignación condicional puede causar, en ciertos casos, cambios extras en los *drivers* que no corresponden a la realidad del hardware. Por ejemplo, la descripción siguiente :

```
ENTITY flipflop IS
    GENERIC ( retardo1:TIME := 4 ns;
              retardo2 :TIME := 5 ns );
    PORT ( d,c : IN BIT;
          q,qb : OUT BIT);
END flipflop;

ARCHITECTURE condicional OF flipflop IS
    SIGNAL estado : BIT;
BEGIN
    estado <= d WHEN ( c = '1' AND NOT c' STABLE) ELSE estado;
    q <= estado AFTER retardo1;
    qb <= NOT estado AFTER retardo2;
END condicional;
```

La asignación condicional fuerza que cuando haya cambios en 'd' ó 'c' que no son el que interesa, es decir, el flanco ascendente de 'c', habrá *transactions* extras, aunque no eventos, por asignación de "estado" del lado *driver* al " estado" destino, lo que no ocurre en la realidad. Para evitar las *transactions* "ocultas", se usan asignaciones GUARDADAS con el formato :

SINTAXIS

```
etiqueta : BLOCK (condición de guarda )
    BEGIN
        señal destino <= GUARDED asignación;
    END BLOCK etiqueta;
```

Cuando la condición de BLOCK es cierta, TRUE, las señales se asignan con la expresión que sigue a GUARDED. Mientras la condición de BLOCK sea FALSE, la señal destino está totalmente desconectada de la parte *driver*. Véase el siguiente ejemplo :*driver*.

```
ARCHITECTURE guardada OF flipflop IS
BEGIN
    ff: BLOCK (c = '1' AND NOT c'STABLE )
    BEGIN
        q   <= GUARDED      d AFTER retardo1;
        qb  <= GUARDED NOT d AFTER retardo2;
    END BLOCK ff;
END guardada;
```

BLOCK es una sentencia concurrente que permite anidamientos. Por ejemplo, en el caso de que se tuviese una entrada ENABLE para permitir disparo del FF sólo cuando ENABLE = '1' y haya flanco ascendente del clock, la descripción del FF podría ser así:

```
ENTITY enab_ff IS
    GENERIC ( retardo1:TIME := 4 ns;
              retardo2 :TIME := 5 ns );
    PORT ( d,c,en : IN BIT; q,qb : OUT BIT);
END enab_ff;

ARCHITECTURE guardante OF enab_ff IS
BEGIN
    flanco: BLOCK (c = '1' AND NOT c'STABLE )           -- condición GUARD
    BEGIN
        enable: BLOCK ( en = '1' AND GUARD )           -- condición de guarda
        BEGIN
            q   <= GUARDED      d AFTER retardo1;
            qb  <= GUARDED NOT d AFTER retardo2;
        END BLOCK enable;
    END BLOCK flanco;
END guardante;
```

En las expresiones anteriores la palabra GUARD, dentro del BLOCK enable: interno del anidamiento, hace que la expresión :

(en = '1' AND GUARD)

sea equivalente a

(en = '1') AND (c= '1' AND NOT c' STABLE)

es decir, la expresión GUARD se utiliza como señal implícita a partir de la que se establecen condiciones, pero que sólo toma el valor que resulta de la expresión que la genera.

5.2.5 ATRIBUTOS DE SEÑALES

Los atributos de señales permiten un uso más eficiente del lenguaje al disponer de codificaciones adicionales que simplifican la descripción de aspectos del hardware tales como la detección de flancos y sincronismos. Se utilizan para encontrar valores de señales tales como eventos, flancos, tiempos transcurridos a partir de un instante de simulación, etc.

Hay cinco atributos que se comportan como funciones, devolviendo valores :

señal'EVENT	Devuelve TRUE, si hay un evento en el tiempo <i>delta</i> actual.
señal'ACTIVE	Devuelve TRUE, si hay un cambio en un <i>driver</i> en el tiempo <i>delta</i> actual.
señal'LAST_EVENT	Devuelve el tiempo transcurrido desde el último evento en la señal.
señal'LAST_VALUE	Devuelve el valor del tipo de la señal antes del último evento.
señal'LAST_ACTIVE	Devuelve el tiempo transcurrido desde el último cambio en un <i>driver</i> .

Al hacer referencia a un “cambio en un *driver* “ se quiere indicar que puede haber cambio en la señal o no, a diferencia de cuando se refiere a “evento“ que significa que hay cambio efectivo en el valor de la señal. La diferencia entre ambos conceptos se debe a que, dependiendo del valor de la señal en un instante, un cambio en un *driver* puede ordenar que la señal adquiera un valor que ya tiene en cuyo caso no hay cambio en el valor de la señal. Por contra, si el “cambio en el *driver*“ supone que la señal cambie de valor, entonces, además de “cambio de *driver* “ hay “evento”.

El “cambio en un *driver*“, en la literatura americana se denomina *transaction*.

Nótese que los atributos **'ACTIVE** y **'LAST_ACTIVE** están relacionados a *transactions*, mientras los **EVENT** y **LAST_EVENT** lo están a eventos.

Un ejemplo sencillo de **'EVENT** es para describir flancos de señales reloj :

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY d_flip_flop IS
    PORT ( d,clk : IN std_logic; q : OUT std_logic);
END d_flip_flop;

ARCHITECTURE flanco_reloj OF d_flip_flop IS
BEGIN
    PROCESS(clk)
    BEGIN
        IF ( clk = '1') AND ( clk'EVENT ) THEN -- condición de flanco de subida
            q <= d;
        END IF;
    END PROCESS;
END flanco_reloj;
```

Dado que la señal clk es de tipo std_logic en este ejemplo, si se considera la posible transición de 'X' a '1' en la señal clk como no admisible, se podría modificar la sentencia

reforzando la condición de flanco ascendente con el atributo 'LAST_VALUE' como sigue

```
IF ( clk = '1') AND ( clk'EVENT ) AND ( clk' LAST_VALUE = '0' ) THEN
    q <= d;
END IF;
```

que asegura completamente la transición de '0' a '1' en la señal clk

El atributo '**LAST_EVENT**' se utiliza para determinar el tiempo transcurrido después de una transición - no *transaction* - o evento. Una aplicación frecuente es comprobar si se han sobrepasado tiempos de *set up* o *hold*, así como la duración o separación entre impulsos.

Un ejemplo aplicable a la entidad d_flip_flop del último ejemplo podría ser :

```
ARCHITECTURE test_setup OF d_flip_flop IS
    setup : TIME := 5 ns;
BEGIN
    PROCESS(clk)
    BEGIN
        IF ( clk = '1') AND ( clk'EVENT ) THEN -- condición de flanco de subida

            ASSERT ( d'LAST_EVENT >= setup ) -- tiempo entre flancos de d y clk
            REPORT "el tiempo transcurrido desde el último cambio en d hasta el
                    evento ocurrido en clk es inferior al tiempo de setup"
            SEVERITY ERROR;

            q <= d;
        END IF;
    END PROCESS;
END test_setup;
```

Existen otros cuatro atributos de señal que suministran nuevas señales a partir de la señal a que se refieren, lo que permite usarlas igual que aquellas de las que se derivan, aportando información no disponible directamente en las señales originales. Estos atributos son los que siguen:

- señal'DELAYED [(tiempo)]** Proporciona una señal del mismo tipo que la referencia, con un retraso respecto a la referencia indicado en el campo opcional.
- señal'STABLE [(tiempo)]** Proporciona una señal **booleana** que es TRUE si en el tiempo que se indica en el campo opcional no hubo un evento en la señal referida.
- señal'QUIET [(tiempo)]** Proporciona una señal **booleana** que es TRUE si en el tiempo que se indica no hubo "cambio en *drivers*" de la señal base.
- señal'TRANSACTION** Proporciona una señal de tipo BIT que cambia de valor con cada TRANSACTION o "cambio en *drivers*".

Un ejemplo con el atributo **'STABLE** podría ser :

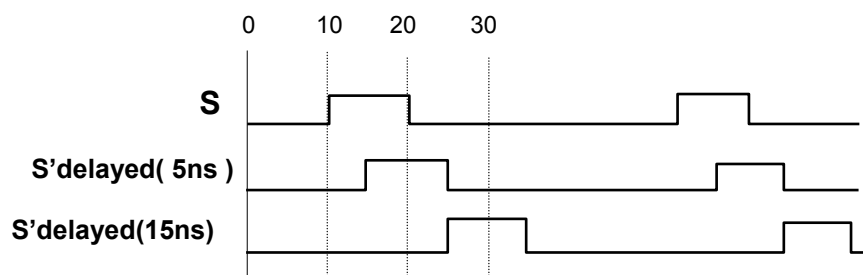
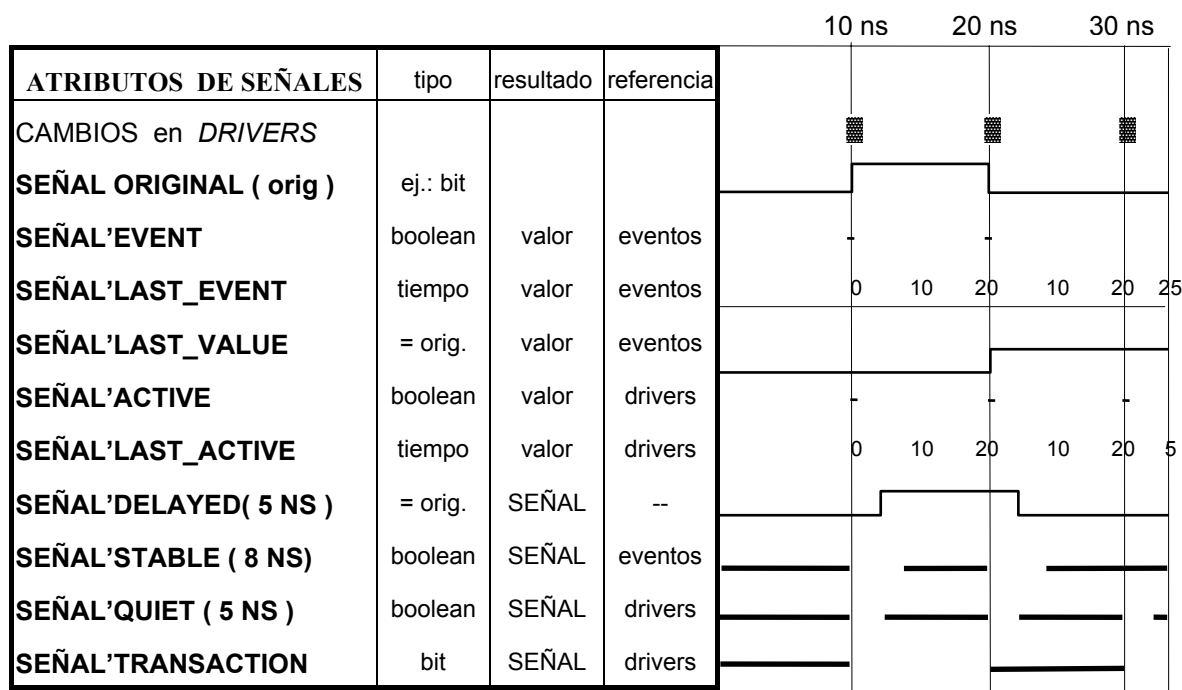
```

ARCHITECTURE condicional_imperfecta OF d_flip_flop IS
    SIGNAL estado : BIT;
BEGIN
    estado <= d WHEN ( clk = '1' AND NOT clk'STABLE )
        ELSE estado; -- estado sigue a d si la condición es TRUE.
                    Si condición FALSE, estado no cambia.

    q <= estado;
    qb <= NOT estado;
END condicional_imperfecta;
    
```

Como se verá en ejemplos posteriores esta arquitectura ejemplo de uso de **'STABLE** tiene defectos mejorables cambiando la forma de asignar el valor de la señal estado a la señal q.

Las gráficas siguientes resumen los atributos de señales :



De las formas de onda anteriores se pueden exponer como ejemplos de aplicación de atributos las siguientes expresiones :

S'delayed(15 ns)	'1'	-- Resultados o valores que en el instante 30 ns se
S'delayed(5 ns)	'0'	-- obtienen al aplicar los atributos que se indican a
S'STABLE(15 ns)	FALSE	-- la señal S. Observar que se puede aplicar atributos
S'STABLE(5 ns)	TRUE	-- a las señales S'delayed(5 ns) y S'delayed(15 ns).
S'EVENT	FALSE	
S'LAST_EVENT	10 ns	
S'LAST_VALUE	'1'	

5.5 Sentencias ASSERT

Es similar a la sentencia ASSERT secuencial, con la diferencia de que puede tener una etiqueta que no está permitida en la sentencia secuencial y que, al ser concurrente, no puede ir dentro de una sentencia PROCESS, pero equivale a un proceso de tipo *pasivo*, es decir, uno que NO asigna valor a señales, aunque sea sensible a ellas. Al igual que la sentencia secuencial, se emplea para verificar la ocurrencia de condiciones que se desea hacer patentes por medio de un mensaje durante la simulación, por ejemplo, detectar valores fuera de rangos permitidos, violación de márgenes de diseño, de tiempos de guarda tales como *setup*, *hold*, duración de impulsos, etc. Su formato es el siguiente :

```
[etiqueta] : ASSERT expresión booleana
              REPORT "mensaje a emitir si la expresión booleana es FALSE"
              SEVERITY nivel del error;
```

Mientras la sentencia secuencial se ejecuta con el orden impuesto en la descripción del proceso o subprograma que la contiene, la sentencia concurrente se ejecuta una vez al iniciarse la simulación y, posteriormente, como las otras sentencias concurrentes, siempre que haya un evento en alguna de las señales que intervienen en la expresión booleana y a las que el proceso *pasivo* es sensible. Un ejemplo puede ser :

```
negativo :  ASSERT ( (a(0) = '0') AND ( b(0) = '0') )
              REPORT "operando negativo"
              SEVERITY WARNING;
```

que equivale al proceso :

```
negativo :  PROCESS ( (a(0) , b(0) )
              BEGIN
                  ASSERT ( (a(0) = '0') AND ( b(0) = '0') )
                  REPORT "operando negativo"
                  SEVERITY WARNING;
              END PROCESS negativo;
```

Una aplicación particular de esta sentencia es que puede usarse como proceso pasivo dentro de un formato más complejo de la ENTITY donde está permitida una región para sentencias concurrentes, lo que permite realizar comprobaciones aplicables a cualquiera de las arquitecturas que describan la entidad, por ejemplo :

```
ENTITY Srlatch IS
  PORT ( s,r : IN BIT; q,qn : OUT BIT);
BEGIN
  verificar :   ASSERT NOT ( ( S = '1') AND ( R = '1') )      -- sentencia concurrente
                REPORT " S = R = '1' simultáneamente "
                SEVERITY ERROR;
END Srlatch;
```

que verifica la condición prohibida del SR latch, probando si NO es cierta la condición.

5.6 Sentencias GENERATE

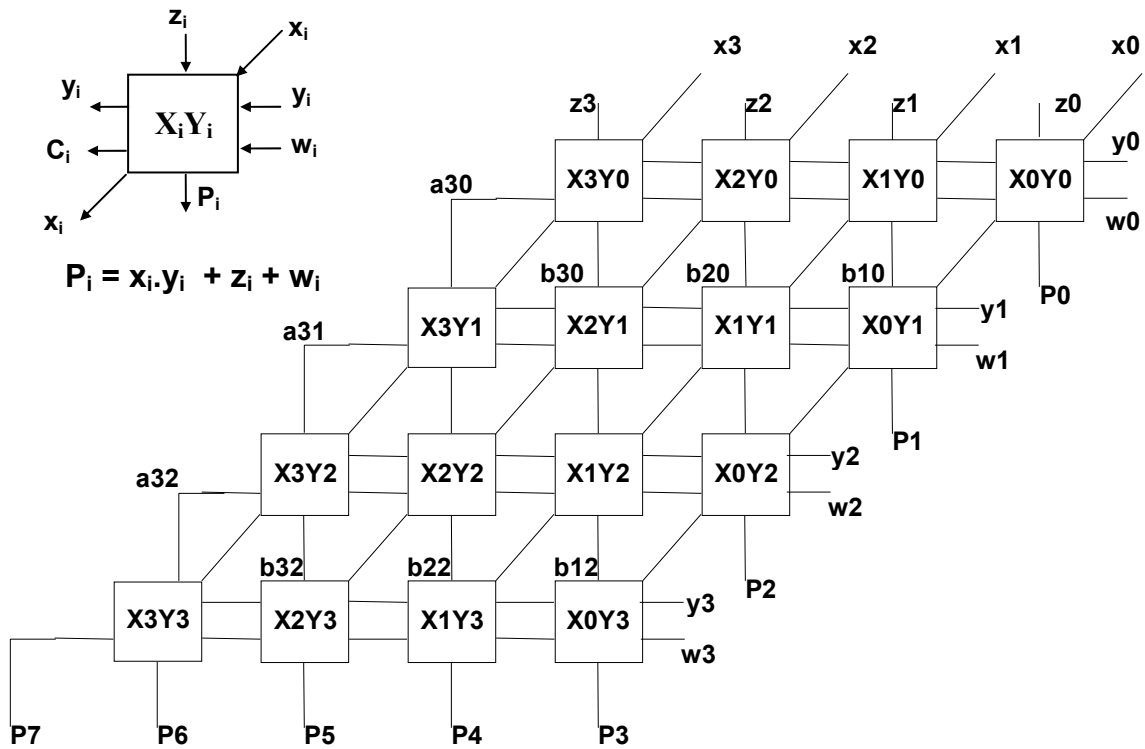
Existen numerosos circuitos digitales compuestos por módulos o células que se repiten para realizar ciertas tareas. Contadores, memorias, registros etc. son subsistemas donde se repite un elemento para formar estructuras expandibles, y en ciertos casos reconfigurables, según la aplicación a que se destinan. En VHDL se dispone de sentencias que permiten realizar descripciones de estos subsistemas de manera eficiente, sin necesidad de repetir las descripciones de los elementos integrantes, simplemente describiendo la forma en que se interconectan. La sentencia GENERATE se utiliza para aplicar iteraciones sobre el elemento o componente que se repite en esas descripciones. El formato básico es el siguiente :

```
etiqueta : { FOR parámetros_o_rango | IF condiciones } GENERATE
  [{bloque de declaraciones }           -- bloque optativo
  BEGIN                               -- bloque optativo
    {sentencias concurrentes } ]       -- bloque optativo
  END GENERATE [ etiqueta ] ;
```

En la mayoría de los casos, las estructuras difieren en los elementos de la periferia, por lo que suele aplicarse una *regla de generación* acorde con la estructura de los elementos básicos y su modo de interconexión en el subsistema donde se repiten. Un ejemplo donde se observa una expansión o generación bidimensional es en un multiplicador binario, basado en el multiplicador de un bit, descrito anteriormente en el apartado 5.1 .

La descripción siguiente se basa en el uso de multiplicadores de 1-bit, declarado como componente en el empaquetamiento “modelos”, por lo que la descripción se inicia con la cláusula USE. En la *regla de generación* se han considerado siete tipos de estructuras en relación a las señales aplicadas y generadas en los módulos multiplicadores de 1-bit. Las etiquetas relativas a filas y columnas colocadas en cada sentencia GENERATE , así como las X_iY_j de las sentencias de mapeo, explican la regla de generación aplicada para determinar la interconexión y asignación de señales a cada módulo en relación a la estructura del multiplicador, cuya estructura para 4 bits se muestra en la página siguiente.

Nótese que las señales internas a y b que aparecen en el esquema del multiplicador se declaran como dos arrays bidimensionales :



```

USE WORK.modelos.ALL;
ENTITY mulnbits IS
    GENERIC (N : POSITIVE);
    PORT( x,y,z,w : IN BIT_VECTOR( N-1 DOWNT0 );
          p : OUT BIT_VECTOR((2*N)-1 DOWNT0 ));
END mulnbits;

ARCHITECTURE generica OF mulnbits IS

    TYPE matriz IS ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> ) OF BIT;
    FOR ALL : multunbi USE ENTITY WORK.multunbit;
    SIGNAL a,b : matriz (N-1 DOWNT0 0, N-1 DOWNT0 0 );    -- señales internas

BEGIN
    X0Y0 :    multunbi PORT MAP ( x(0), y(0), z(0), w(0), a(0,0), p(0));

    fila0 :   FOR i IN 1 TO N-1 GENERATE
    XiY0 :    multunbi PORT MAP( x(i), y(0), z(i), a(i-1,0), a(i,0), b(i,0) );
    END GENERATE;

```

```
columna0 :   FOR j IN 1 TO N-1 GENERATE
X0Yj :      multunbi PORT MAP( x(0), y(j), b(1,j-1), w(j), a(0,j), p(j) );
            END GENERATE;

            filaN :   FOR i IN 1 TO N-2 GENERATE
XiYn :      multunbi PORT MAP(x(i), y(N-1), b(i+1,N-2),a(i-1,N-1), a(i,N-1),p(N-1+i));
            END GENERATE;

columnaN :   FOR j IN 1 TO N-2 GENERATE
XnYj :      multunbi PORT MAP(x(N-1), y(j), a(N-1, j-1), a(N-2, j), a(N-1, j), b(N-1, j));
            END GENERATE;

columna j :   FOR i IN 1 TO N-2 GENERATE
            fila i :   FOR j IN 1 TO N-2 GENERATE
XiYj :      multunbi PORT MAP( x(i), y(j), b(i+1,j-1), a(i-1,j), a(i,j), b(i,j) );
            END GENERATE;
            END GENERATE;

XnYn :      multunbi PORT MAP (x(N-1), y(N-1), a(N-1, N-2),
                           a(N-2, N-1), p((2*N) -1), p((2*N) -2));
END generica;
```

5.7 Sentencias de LLAMADA A PROCEDIMIENTOS

Tiene el mismo formato que las sentencias de llamada secuencial, es decir :

```
identif_proced ( [señales_IN,] [parámetros_OUT,] [parámetros_INOUT] )
```

que por ser una sentencia concurrente equivale a un proceso cuya lista de sensibilidad será la de las señales que actúan como parámetros de entrada al procedimiento, o sea :

```
PROCESS ( lista_de_señales_IN )
BEGIN
    identif_proced ( [señales_IN,] [parámetros_OUT,] [parámetros_INOUT] )
END PROCESS;
```

que se activará siempre que cambie alguna de las señales a las que es sensible.

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.

Una utilidad particular de este tipo de sentencias es ejecutar concurrentemente *varias* operaciones descritas por el mismo procedimiento. Por ejemplo, comprobar que varios buses en un mismo momento tienen todas sus líneas en estado 'Z'. Podría definirse un procedimiento como sigue :

```
PROCEDURE bustest ( SIGNAL busX : IN cuad_vector;
                   SIGNAL error: OUT BOOLEAN) IS
    VARIABLE NOZ : BOOLEAN := FALSE;
BEGIN
```

```
    FOR i IN busX'RANGE LOOP
        IF busX(i) /= 'Z' THEN
            NOZ := TRUE;
            RETURN;
        END IF;
    END LOOP;
    error <= NOZ;
END;
```

pudiendo llamar al procedimiento con un bloque de sentencias como el siguiente :

```
testbuses: BLOCK
    SIGNAL bus1: cuad_vector( 0 TO 3);
    SIGNAL bus2: cuad_vector( 0 TO 7);
    SIGNAL bus3: cuad_vector( 0 TO 15);

    SIGNAL error1, error2,error3 : OUT BOOLEAN;

    BEGIN
        bustest ( bus1, error1);
        bustest ( bus2, error2);
        bustest ( bus3, error3);
    END BLOCK testbuses;
```