UNIVERSIDAD POLITÉCNICA DE MADRID

DEPARTAMENTO DE SISTEMAS ELECTRÓNICOS Y DE CONTROL

Introducción al lenguaje VHDL

(Versión preliminar)

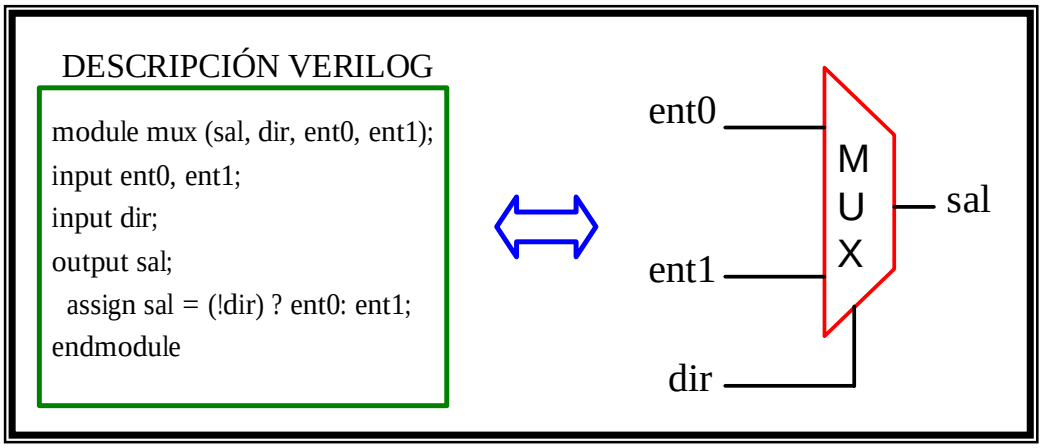
Miguel Angel Freire Rubio

INDICE

INTRODUCCIÓN

1.- Lenguajes de Descripción Hardware

Los lenguajes de descripción hardware (**HDLs**, **Hardware Description Languages**) vienen utilizándose desde los años 70 en los ciclos de diseño de sistemas digitales asistidos por herramientas de CAD electrónico. Al principio surgieron una serie de lenguajes que no llegaron a alcanzar un éxito que permitiera su consolidación en el campo industrial o académico. En los años 80 aparecen los lenguajes **Verilog** y **VHDL** que, aprovechando la disponibilidad de herramientas hardware y software cada vez más potentes y asequibles y los adelantos en las tecnologías de fabricación de circuitos integrados, logran imponerse como herramientas imprescindibles en el desarrollo de nuevos sistemas. En la actualidad ambos lenguajes están normalizados y comparten una posición hegemónica que está arrinconando –y terminará, probablemente, en poco tiempo eliminando del mercado– al resto de lenguajes que de un modo u otro todavía son soportados por algunas herramientas de CAD.



Estos lenguajes son sintácticamente similares a los de programación de alto nivel –Verilog tiene una sintaxis similar al C y VHDL a ADA– y se diferencian de éstos en que su semántica está orientada al modelado del hardware. Su capacidad para permitir distintos enfoques en el modelado de los circuitos y su independencia de la tecnología y metodología de diseño permiten extender su uso a los distintos ciclos de diseño que puedan utilizarse. Por ello, para los profesionales relacionados de alguna manera con el diseño o mantenimiento de sistemas digitales resulta hoy en día imprescindible su conocimiento.

2.- El lenguaje VHDL

Los estudios para la creación del lenguaje VHDL (VHSIC HDL) comenzaron en el año 1981, bajo la cobertura de un programa para el desarrollo de Circuitos Integrados de Muy Alta Velocidad (VHSIC), del Departamento de Defensa de los Estados Unidos. En 1983 las compañías Intermetrics, IBM y Texas Instruments obtuvieron la concesión de un proyecto para la realización del

lenguaje y de un conjunto de herramientas auxiliares para su aplicación. Finalmente, en el año 1987, el lenguaje VHDL se convierte en la norma IEEE-1076 –como todas las normas IEEE, se somete a revisión periódica, por lo que en 1993 sufrió algunas leves modificaciones–.

3.- Características del lenguaje

El lenguaje VHDL fue creado con el propósito de especificar y documentar circuitos y sistemas digitales utilizando un lenguaje formal. En la práctica se ha convertido, en un gran número de entornos de CAD, en el HDL de referencia para realizar modelos sintetizables automáticamente. Las principales características del lenguaje VHDL se explican en los siguientes puntos:

- **Descripción textual normalizada:** El lenguaje VHDL es un lenguaje de descripción que especifica los circuitos electrónicos en un formato adecuado para ser interpretado tanto por máquinas como por personas. Se trata además de un lenguaje formal, es decir, no resulta ambiguo a la hora de expresar el comportamiento o representar la estructura de un circuito. Está, como ya se ha dicho, normalizado, o sea, existe un único modelo para el lenguaje, cuya utilización está abierta a cualquier grupo que quiera desarrollar herramientas basadas en dicho modelo, garantizando su compatibilidad con cualquier otra herramienta que respete las indicaciones especificadas en la norma oficial. Es, por último, un lenguaje ejecutable, lo que permite que la descripción textual del hardware se materialice en una representación del mismo utilizable por herramientas auxiliares tales como simuladores y sintetizadores lógicos, compiladores de silicio, simuladores de tiempo, de cobertura de fallos, herramientas de diseño físico, etc.

- **Amplio rango de capacidad descriptiva:** El lenguaje VHDL posibilita la descripción del hardware con distintos niveles de abstracción, pudiendo adaptarse a distintos propósitos y utilizarse en las sucesivas fases que se dan en el desarrollo de los diseños. Además es un lenguaje adaptable a distintas metodologías de diseño y es independiente de la tecnología, lo que permite, en el primer caso, cubrir el tipo de necesidades de los distintos géneros de instituciones, compañías y organizaciones relacionadas con el mundo de la electrónica digital; y, en el segundo, facilita la actualización y adaptación de los diseños a los avances de la tecnología en cada momento.

- **Otras ventajas:** Además de las ventajas ya reseñadas también es destacable la capacidad del lenguaje para el manejo de proyectos de grandes dimensiones, las garantías que comporta su uso cuando, durante el ciclo de mantenimiento del proyecto, hay que sustituir componentes o realizar modificaciones en los circuitos, y el hecho de que, para muchas organizaciones contratantes, sea parte indispensable de la documentación de los sistemas.

4.- Ejercicio

A lo largo de los capítulos que componen este texto se van a realizar distintos ejercicios con la versión de evaluación del simulador **Veribest**. Se trata de una herramienta moderna (soporta la versión del lenguaje de 1993), eficiente y fácil de manejar. Es, además, el simulador escogido por **Actel,** un importante fabricante de dispositivos lógicos programables, para su entorno de diseño con lógica programable, **Actel DeskTOP** –un entorno, por otra parte, magnífico para empezar a utilizar herramientas VHDL, ya que cuenta también con una muy buena herramienta de síntesis, **Synplicity**, pero que lamentablemente, no dispone de una versión de evaluación– por lo que su aprendizaje puede resultar útil a diseñadores que vayan a trabajar con esta tecnología. La versión de evaluación pone limitaciones (poco importantes para ejercicios de baja o mediana complejidad) al tamaño del código que se desea simular y a la duración de las simulaciones. A continuación se describe el proceso de instalación del software.

Para instalar el programa necesita unos 30 Mbytes de espacio libre en su disco duro y un lector de CDs.

1. Introduzca el CD en el lector.

2. Ejecute **Setup.exe** en el directorio **VHDL_Simulator**.

3. Acepte todas las opciones que aparecen.

4. Espere a que se complete la instalación.

5. Si desea desinstalar el programa, utilice la utilidad que se suministra en el CD de instalación.

El programa será ejecutable desde la barra de programas de Windows. Además del simulador se habrá instalado un tutorial interactivo que puede servirle para aprender a manejar la herramienta.

LIBRERÍAS Y UNIDADES DE DISEÑO

0.- Resumen del Capítulo

Conceptos Teóricos:

- *Conceptos básicos: Estructura del Lenguaje.*
- *Modelado del Hardware. Conceptos Básicos.*
- *Unidades de Diseño VHDL.*
- *Función de las distintas unidades.*
- *Unidades Primarias y Secundarias.*
- *Compilación de unidades de diseño.*
- *Librerías VHDL. La librería WORK.*
- *Reglas de Visibilidad.*
- *Nombres en VHDL.*

Prácticas sobre el simulador Veribest:

- *Procedimientos básicos para la realización de simulaciones con un simulador VHDL.*
- *Gestión de Librerías VHDL*.

Apéndice:

- *Consideraciones prácticas sobre librerías y unidades de diseño en entornos comerciales VHDL.*

En este capítulo se presenta la estructura general del lenguaje VHDL, las librerías y las unidades de diseño; además se muestra el código de un modelo VHDL sencillo (el de una puerta **and**) que sirve para que el lector tenga un primer contacto con una descripción y un ***test-bench*** VHDL; para facilitar su comprensión se avanzan algunos detalles sintácticos y semánticos de las Declaraciones de Entidad y Cuerpos de Arquitectura VHDL, que se tratarán detalladamente en capítulos posteriores.

Los ejercicios que se realizan persiguen dos objetivos: por un lado, presentar el conjunto de procedimientos que hay que seguir para realizar la simulación de una especificación VHDL en el entorno **VeriBest**, ya que resultan básicos para la realización de la mayor parte de los ejercicios que se realizan y proponen en este texto, y, por otro, facilitar la asimilación de los conceptos teóricos presentados.

PARTE PRIMERA. INTRODUCCIÓN AL MODELADO VHDL.

En este apartado se presentan la Declaración de Entidad y el Cuerpo de Arquitectura: las unidades básicas con que se describe un dispositivo hardware. También se explica qué es un test-bench y se realiza un ejercicio sencillo de simulación.

1.- Caracterización de circuitos

Para entender en qué consiste el modelado lógico de un circuito, hay que apreciar los dos aspectos que lo caracterizan:

1. **Un interfaz externo:** una puerta **and** de dos entradas, por ejemplo, tiene tres terminales, dos entradas y una salida, y se diferencia de una puerta **and** de tres entradas, en dos cosas: el número de terminales y el nombre –el nombre de una es **and** *de dos entradas*, el de la otra **and** *de tres entradas*–.

2. **Un algoritmo de procesamiento de la información:** cada dispositivo digital realiza una determinada operación que le permite obtener ciertos niveles lógicos en sus terminales de salida a partir de los aplicados en sus entradas: una puerta **xor** pone un uno cuando sus dos entradas son distintas, una puerta **nor** cuando todas sus entradas tienen un cero.

En los planos de un circuito la representación del interfaz de un dispositivo se realiza gráficamente mediante el dibujo de un determinado símbolo: son característicos y conocidos por todos los de las puertas lógicas o los ***flip-flops***; los bloques funcionales se representan normalmente mediante rectángulos con algún texto distintivo asociado (sumador, decodificador, etc.) y líneas y etiquetas para la especificación de los terminales. Con este sistema parece, a veces, que el símbolo aporta alguna información sobre el funcionamiento: si vemos el símbolo de una puerta **and** sabemos cuál es la función lógica que realiza, pero este conocimiento es aprendido y procede de la asociación de una determinada tabla de verdad, la de la función lógica **and**, con la forma del símbolo que representa el interfaz de la puerta **and**.

La función lógica que realiza un circuito o dispositivo identificable mediante un símbolo se describe (en catálogos de fabricantes, libros de texto o documentaciones de proyectos) mediante tablas de verdad y cronogramas y, cuando resulta conveniente, mediante anotaciones adicionales en lenguaje corriente.

El modelo de un dispositivo o circuito digital se completa añadiendo al modelo lógico información referente a su comportamiento dinámico (indicando tiempos de retardo entre entradas y salidas, tiempos de ***set-up***, etc.), y sus características eléctricas (corrientes, niveles de tensión, carga, disipación de potencia) y físicas (encapsulado, dimensiones).

Los lenguajes de especificación hardware como VHDL están orientados al modelado de las características lógicas y dinámicas del hardware. El modelado de un dispositivo “simple” –uno que no está compuesto por dos o más

conectados entre sí; los dispositivos “complejos” pueden describirse mediante la conexión de otros– con un lenguaje de descripción hardware consiste en la descripción de su interfaz y su comportamiento –incluyendo características dinámicas– haciendo uso de las construcciones de que disponga el lenguaje y respetando las reglas que se impongan.

2.- La Declaración de Entidad y el Cuerpo de Arquitectura

La realización del modelo hardware de un dispositivo en VHDL consiste en la elaboración de dos unidades de código VHDL: una Declaración de Entidad y un Cuerpo de Arquitectura.

La Declaración de Entidad es la unidad de diseño VHDL que sirve para especificar el interfaz de los dispositivos. Cumple, por tanto, funciones equivalentes a las de los símbolos en las representaciones gráficas.

El Cuerpo de Arquitectura es la unidad de diseño VHDL que sirve para especificar el funcionamiento de un dispositivo identificado por una determinada Declaración de Entidad, por lo que se puede considerar el equivalente a las tablas de verdad o a los cronogramas.

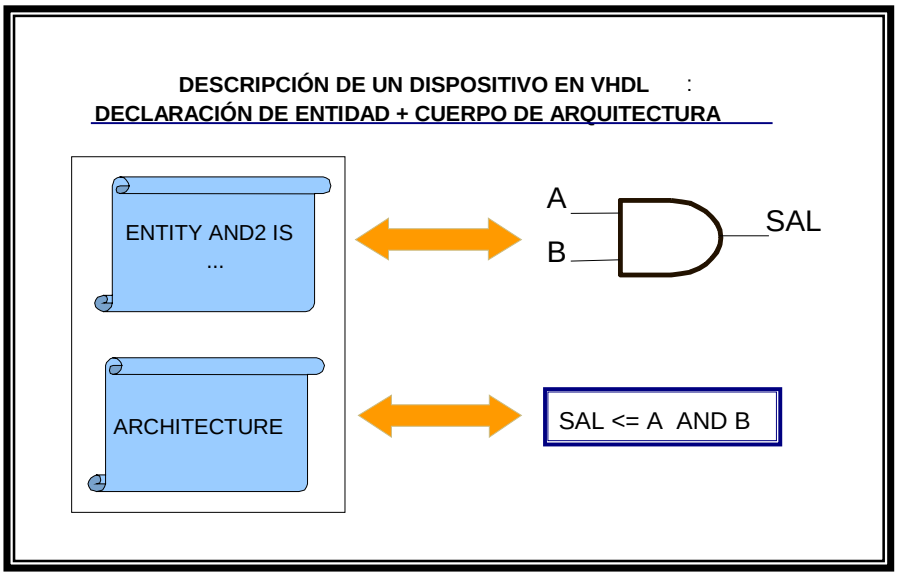


Figura I.1

En la figura I.2 se muestra el código correspondiente al modelo de una puerta **and** de dos entradas.

En la Declaración de Entidad se define el nombre del dispositivo y sus puertos; en el Cuerpo de Arquitectura su funcionamiento, en este caso mediante una sentencia que utiliza un operador lógico.

PUERTA AND DE DOS ENTRADAS

```
ENTITY and2 IS
PORT(
        ent1, ent2      : IN BIT;
        sal             : OUT BIT
    );
END ENTITY;

ARCHITECTURE rtl OF and2 IS
BEGIN
    sal <= ent1AND ent2;
END rtl;
```

Figura I.2

La construcción del modelo de un dispositivo en un entorno VHDL finaliza, en principio, en el momento en que las unidades VHDL que lo describen quedan almacenadas en una librería VHDL. Para ello hay que editar las unidades y compilarlas. En el proceso de compilación se comprueba que no se incumplen una serie de reglas sintácticas y semánticas.

3.- Sintaxis Básica de la Declaración de Entidad

La sintaxis básica de la Declaración de Entidad es la siguiente:

```
ENTITY {nombre del dispositivo}  IS
PORT(
    {lista de puertos de entrada}       : IN      {tipo de dato};
    {lista de puertos bidireccionales} : INOUT   {tipo de dato};
    {lista de puertos de salida}        : OUT     {tipo de dato};
    {lista de puertos de salida}        : BUFFER {tipo de dato});
END ENTITY;
```

Figura I.3

Obviamente, el identificador que sigue a la palabra clave **ENTITY** es el nombre del dispositivo. Los puertos del dispositivo se especifican, entre paréntesis, en la lista de interfaz de la declaración de entidad, que es el campo señalado por la palabra clave **PORT**. Para cada puerto hay que declarar:

1. Su nombre: Los nombres de los puertos son etiquetas definibles por el usuario. No pueden utilizarse palabras reservadas del lenguaje.

2. Su dirección: La dirección del puerto se determina a partir de las características del terminal del dispositivo que se desea modelar: los pines de entrada se definen como de tipo **IN**, los de salida como **OUT** y los bidireccionales como **INOUT**.

3. El tipo de datos que maneja el puerto: En los modelos VHDL hay que definir el tipo de datos de los objetos. La elección del tipo de datos es muy importante, pues determina el conjunto de valores que puede tomar el objeto declarado, así como las operaciones (lógicas, aritméticas y de conversión de tipos) que se le pueden aplicar, cuestión fundamental a la hora de describir el funcionamiento de los dispositivos.

En el ejemplo de la puerta **and** se utiliza el tipo **BIT**. En VHDL este tipo consta de dos valores, los caracteres **'0'** y **'1'**, sobre los que se definen las operaciones lógicas básicas y las de comparación.

4.- Cuerpos de Arquitectura. Nociones Básicas

La sintaxis básica de un Cuerpo de Arquitectura es la siguiente:

```
ARCHITECTURE {nombre_de_arquitectura} OF {nombre_de entidad} IS

        {zona de declaración}

        BEGIN

        {zona de descripción}

        END {nombre de arquitectura};
```

Figura I.4

Las dos etiquetas que forman parte de la cabecera nombran a la propia Arquitectura y a la Declaración de Entidad a la que esta asociada. La primera vuelve a aparecer en la línea que cierra el Cuerpo de Arquitectura.

Por lo demás, se distinguen dos áreas para la inclusión de código con dos propósitos distintos:

1. La comprendida entre la cabecera y la palabra **BEGIN** está destinada a la declaración de objetos que se precisen para realizar la descripción del funcionamiento del dispositivo.

2. Entre la palabra **BEGIN** y **END {arquitectura}** se describe el funcionamiento.

En el ejemplo de la puerta **and** no se declara ningún objeto y la descripción del funcionamiento se realiza utilizando una sentencia concurrente de asignación

de valor a señal.

5.- Simulación del modelo VHDL

La compilación exitosa de la Declaración de Entidad y la Arquitectura VHDL garantiza que se cumplen una serie de reglas sintácticas y semánticas, pero no que el hardware se haya modelado correctamente. Para probar el modelo es preciso simularlo.

Una simulación VHDL se realiza conectando el modelo en pruebas a un conjunto de estímulos que permiten observar la respuesta del mismo. Para ello hay que construir un ***test-bench*** (banco de test). A continuación se muestra un ***test-bench*** VHDL que permitiría simular el modelo de la puerta **and.**

```
ENTITY test_bench_and2 IS
END test_bench;
ARCHITECTURE  test OF  test_bench_and2 IS
   signal  s_ent1, s_ent2, s_sal: BIT;
BEGIN
  s_ent1 <= '1' after 10 ns;
  s_ent2 <= '1' after 5 ns,
            '0' after 10 ns,
            '1' after 15 ns;
  dut: entity work.and2(rtl)
    port map (ent1=>s_ent1, ent2=>s_ent2, sal=>s_sal);
END  test  ;
```

Figura I.5

Como puede verse, consta, al igual que el modelo de un dispositivo, de una Declaración de Entidad y un Cuerpo de Arquitectura, pero con las siguientes peculiaridades:

1. La Declaración de Entidad no tiene puertos.

2. En el Cuerpo de Arquitectura no se describe el funcionamiento de un dispositivo, sino que se conecta el dispositivo a probar a un conjunto de señales (estímulos y salidas).

El conjunto de valores que toman los estímulos a lo largo del tiempo constituyen los **vectores de test** que se aplican al modelo en pruebas.

Un simulador VHDL es capaz de ejecutar ***test-benches*** disponiendo, normalmente, de un conjunto de utilidades que facilitan la depuración de los modelos y la revisión de resultados.

6.- Ejercicio I.1:

1. Invoque la herramienta en la barra de programas de Windows.

2. Seleccione, en el menú principal, la opción **Workspace -> New...**

3. Rellene la ventana que aparece tal y como se muestra en la figura e1 y pulse el botón **Create**. El campo **Workspace Path** es orientativo, debe corresponderse con un directorio de su disco duro destinado a la realización de los ejemplos de este texto.

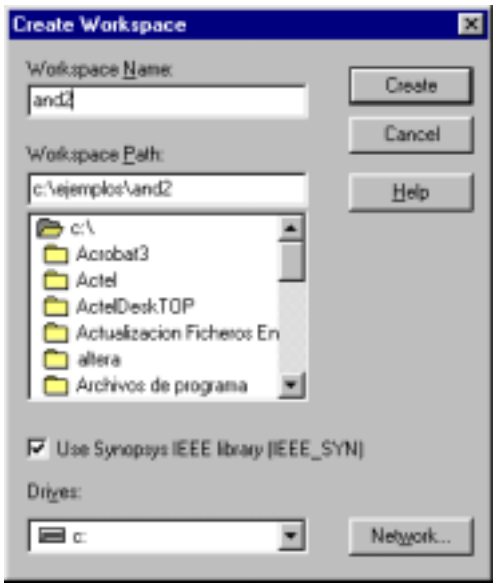


Figura e1

En primer lugar se van a editar los ficheros que contienen el modelo VHDL de la puerta y el ***test-bench***.

4. Seleccione, en el menú principal, la opción **File -> New...** En la ventana que aparece indique que se trata de un fichero con fuentes VHDL y pulse **OK**.

5. En la ventana de edición que aparece, escriba el texto correspondiente a la Declaración de Entidad y Cuerpo de Arquitectura de la puerta **and**. Cuando haya completado esta operación active la opción **File -> Save**, en el menú principal. En la ventana que aparece, indique como nombre del fichero **and2.vhd** y pulse **Guardar**.

6. Cierre la ventana de edición utilizando los botones de control de Windows.

7. Repita las operaciones anteriores para editar el ***test-bench***. Si lo desea puede intentar cerrar la ventana antes de salvar. Observe que, en este caso, se le recuerda que no ha salvado el fichero. Sálvelo como **test_and2.vhd**.

A continuación hay que indicar que se desea que los dos ficheros editados se incorporen al entorno VHDL.

8. Pulse el botón **+** (figura e2) de la ventana que representa el Espacio de Trabajo.

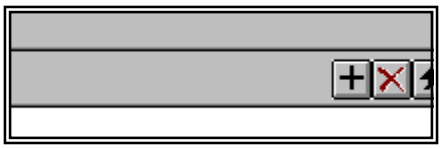


Figura e2

9. En la ventana que aparece seleccione el fichero **and2.vhd** y pulse **Abrir**.

10. Repita la operación para el fichero que contiene el ***test-bench***.

Observe que el nombre de los dos ficheros se incorpora a la ventana del espacio de trabajo (figura e3).

Figura e3

A continuación se van a compilar y simular ambos ficheros, pero antes, es necesario configurar el entorno.

11. Active la opción **Workspace -> Settings** en el menú principal.

12. La ventana que aparece dispone de diversas *carpetas*:

- En **Compile**, active **debug** y desactive **VHDL '87**.
- En **Simulate**, active **Trace On**. En el campo **Entity** escriba el nombre de la Declaración de Entidad del **Test-Bench**: **test_bench_and2** y en el campo **Arch** el de la Arquitectura: **test**.
- El resto de opciones de las carpetas, déjelas como estén.
- Por último, pulse **Aceptar**.

A continuación se ordena la compilación de los ficheros. Esto supone la comprobación de una serie de reglas, de modo que si hay errores se notifican y, si no los hay, se almacenan los modelos en la librería de trabajo.

13. Active la opción **Compile All** del menú **Workspace**.

Si ha introducido el texto VHDL tal y como se le ha indicado, se compilará correctamente la descripción de la puerta **and2** y aparecerán errores en el ***test-bench*** (Figura e4).

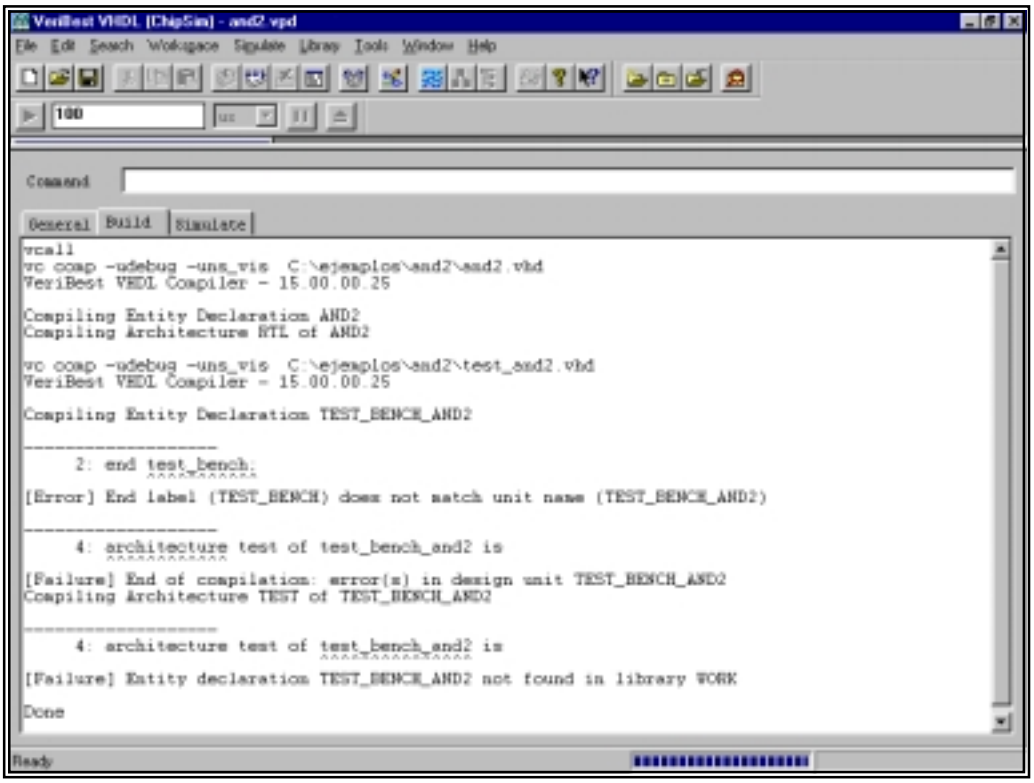


Figura e4

En realidad hay un solo error.

14. Realice una doble pulsación sobre la referencia de error: **2: end test_bench;**

Observe que se abre el fichero que contiene el ***test-bench***. El error consiste en la fórmula de cierre de la Declaración de Entidad. Debería ser: **end entity;**

15. Corrija el error, salve la modificación y cierre el fichero.

Ahora sólo resulta necesario volver a compilar el ***test-bench***.

16. Seleccione en la ventana que representa el Espacio de Trabajo, el fichero **test_and2.vhd** y pulse el icono que representa una pila de papeles apuntada por una flecha vertical.

Ahora el ***test-bench*** debe haberse compilado correctamente.

17. Active ahora la opción **Execute Simulator** en el menú **WorkSpace**. Acepte el mensaje que se le ofrece, indicando que se trata de una versión de evaluación del simulador.

18. Indique, en la parte superior izquierda de la ventana, que desea realizar una simulación que dure **20 ns**. Pulse el botón con la flecha verde (Figura e5).

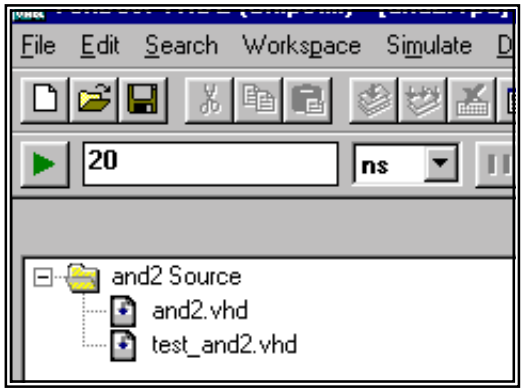


Figura e5

La simulación se completará en unos instantes.

19. Para visualizar los resultados de la simulación hay que invocar la ventana de presentación de señales (Figura e6). Seleccione la opción **New Waveform Window** en el menú **Tools**.

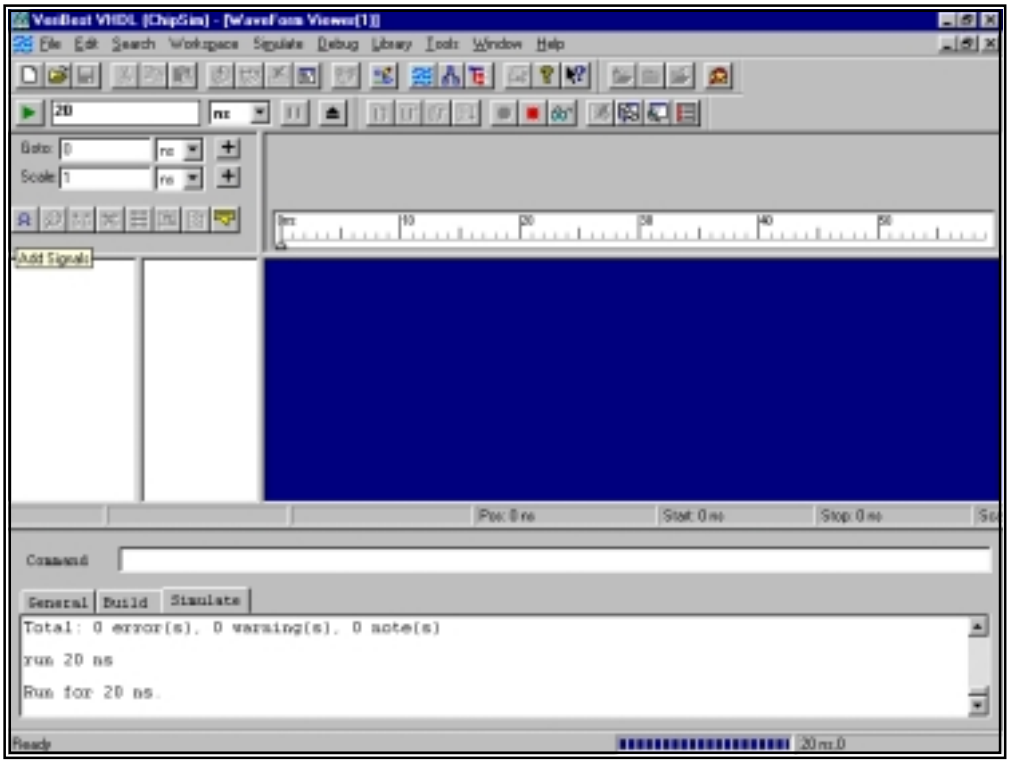


Figura e6

20. Pulse el botón **Add Signals** (el situado más a la izquierda en la botonera de la ventana) para añadir las señales que se desea visualizar.

21. En la ventana que aparece, pulse **Add All** y, a continuación, **Close**.

En la ventana aparecerán los estímulos y la señal de salida conectada al modelo de la puerta **and**. Para revisar cómodamente los resultados puede manejar el factor de **zoom**, cambiando el valor del campo **Scale** (elija, por ejemplo, **400 ps**) y el cursor, que puede mover con el ratón. Observe que al lado de los nombres de las señales aparece el nivel lógico de las mismas en la posición del cursor.

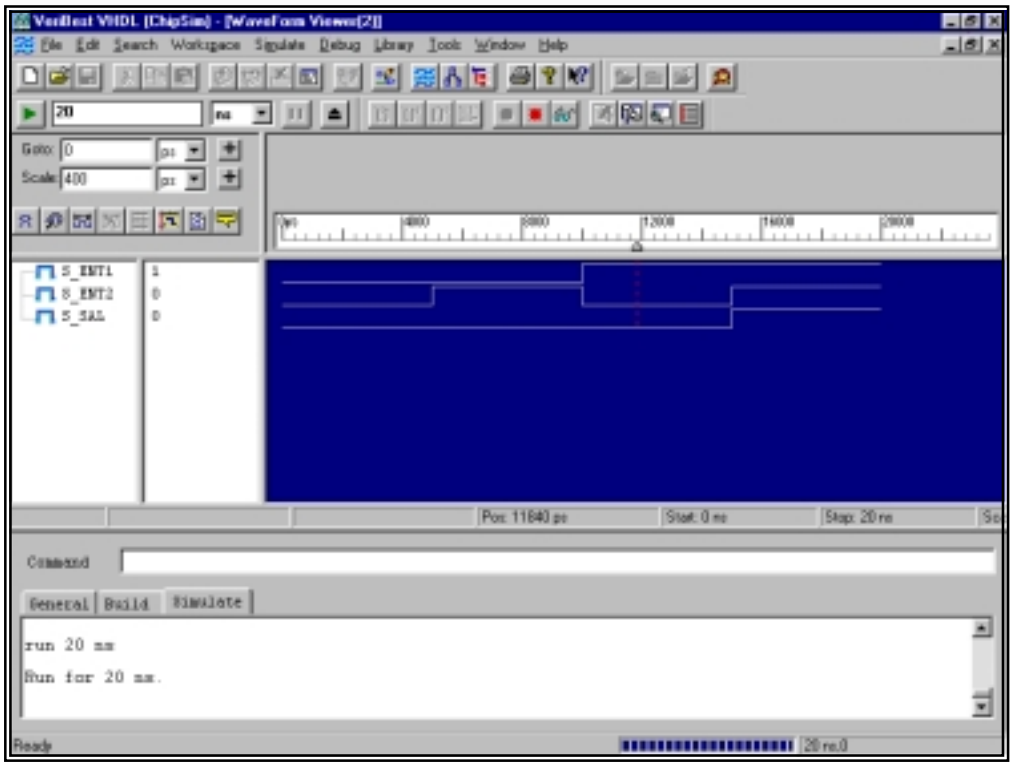


Figura e7

22. Para dar por terminada la simulación, pulse **Ctrl+Q**.

Si lo desea, puede cerrar el simulador mediante los controles de ventana de Windows o en el menú **Files**.

SEGUNDA PARTE. UNIDADES DE DISEÑO Y LIBRERÍAS.

En este apartado se profundiza en algunos de los conceptos básicos del lenguaje VHDL:

- *Unidades de Diseño y Librerías VHDL.*
- *Cláusulas de Visibilidad.*

7.- Unidades de diseño y Librerías VHDL

Un modelo VHDL se compone de un conjunto de **unidades de diseño**. Una unidad de diseño es la mínima sección de código compilable separadamente. Es un concepto muy simple, puede entenderse recurriendo a conocimientos básicos sobre lenguajes de programación: no se puede editar un fichero, escribir únicamente una sentencia **IF** o **CASE** y compilar el fichero, porque una sentencia no es una unidad de código compilable. Podrá compilarse un programa, una función o un conjunto de declaraciones, o cualquier cosa que el lenguaje especifique.

Las unidades de diseño VHDL se construyen combinando construcciones del lenguaje (sentencias, declaraciones, etc). En VHDL existen cinco tipos distintos de unidades de diseño, cada una de las cuales está pensada para el desempeño de una determinada función en el modelado del hardware. Las unidades se clasifican en primarias y secundarias (de acuerdo con las relaciones de dependencia jerárquica que mantienen entre sí: una unidad secundaría está asociada siempre a una unidad primaría), y son:

1.- La Declaración de Entidad: Es la unidad primaria del lenguaje que identifica a los dispositivos, definiendo su interfaz (nombre, terminales de conexión y parámetros de instanciamiento). Puede decirse –simplificando bastante– que desempeña una función equivalente a la del símbolo de un dispositivo en los esquemas de circuitos.

2.- El Cuerpo de Arquitectura: Es una unidad secundaria del lenguaje. Está asociado a una determinada Declaración de Entidad, describiendo el funcionamiento lógico del dispositivo identificado por ésta. Aporta una información equivalente a la de las tablas de verdad o los cronogramas de los circuitos lógicos.

3.- La Declaración de Paquete: Los paquetes desempeñan en VHDL funciones similares a las de las librerías en lenguajes de programación de alto nivel. La Declaración de Paquete es una unidad primaria que contiene la “vista pública” de los paquetes.

4.- El Cuerpo de Paquete: Es una unidad secundaria asociada a una Declaración de Paquete. Se utiliza, si resulta necesario, para definir los elementos declarados en este.

5.- La Declaración de Configuración: Es una unidad primaria que sirve para manejar el emplazamiento de componentes en modelos estructurales.

Las más importantes, porque son las únicas que siempre deben existir en cualquier modelo hardware y exigen un mayor esfuerzo de desarrollo durante las tareas de diseño, son las dos primeras: la Declaración de Entidad y el Cuerpo de Arquitectura. También son muy importantes las unidades con que se construyen los **Paquetes VHDL** (la Declaración y el Cuerpo de Paquete). Los Paquetes VHDL, como ya se ha dicho, realizan una función equivalente a la de las librerías en los lenguajes de programación, es decir, son unidades de código cuyo contenido puede utilizarse desde otras unidades mediante una serie de cláusulas de visibilidad (como *#include* en lenguaje C).

La quinta unidad de código, la Declaración de Configuración, se utiliza rara vez en modelos de baja o media complejidad y su conocimiento no resulta imprescindible para iniciarse en el lenguaje.

Las unidades de diseño VHDL se almacenan en librerías VHDL; dicho de otra manera: una librería VHDL es una colección de unidades de diseño. Una unidad queda almacenada en una librería cuando ha sido compilada correctamente. Las librerías VHDL tienen un *nombre lógico* que permite identificarlas.

Es importante no confundir las librerías VHDL con las de los lenguajes de programación: como ya se ha dicho, el equivalente a estas últimas en VHDL son los paquetes.

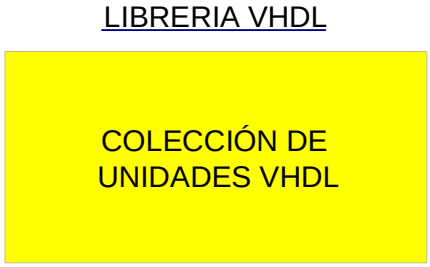


Figura I.6

Las unidades de diseño almacenadas en una misma librería deben cumplir una serie de normas:

1. Una unidad primaria y cualquier unidad secundaria asociada deben almacenarse en la misma librería. Además, primero debe compilarse la unidad primaria (Declaración de Entidad o Declaración de Paquete) y después la secundaria (Cuerpo de Arquitectura o Paquete).

Que sea necesario compilar antes la unidad primaria que cualquier secundaria asociada viene dado porque en ésta última puede utilizarse cualquier objeto visible o declarado en la primaria. La necesidad resulta entonces evidente: la compilación de un Cuerpo de Arquitectura, como el que se muestra en la figura I.7, será incorrecta si, por ejemplo, **res**, **sum1** o **sum2** no son puertos de la Declaración de Entidad **Sum**, o si **sum1** y **sum2** son puertos de salida.

```
ARCHITECTURE rtl OF sum IS
BEGIN
        res <= sum1 + sum2;
END rtl;
```

Figura I.7

La segunda regla se debe a esta misma razón:

2. Si una unidad de diseño tiene visibilidad sobre otra, ésta tiene que compilarse antes que la primera.

Esto afecta fundamentalmente a los Paquetes. Antes de compilar una unidad que utilice un Paquete, la Declaración y el Cuerpo de Paquete deben haberse almacenado en una librería.

La última regla establece algunas condiciones sobre los nombres de las unidades:

3. No pueden existir dos unidades primarias con el mismo nombre en una librería. Sí puede haber –de hecho ocurre con frecuencia– unidades secundarias con el mismo nombre –sólo Cuerpos de Arquitectura, porque los de Paquete no tienen nombre–, la única condición que debe darse en este caso es que estén asociados a distintas unidades primarias.

El hecho de que dos Cuerpos de Arquitectura tengan el mismo nombre no ocasiona ningún problema, puesto que se distinguen por la Declaración de Entidad cuyo comportamiento describen:

```
ARCHITECTURE rtl OF and2 IS
. . .

ARCHITECTURE rtl OF sum IS
. . .
```

Para terminar con las cuestiones relativas a las unidades de diseño y las librerías hay que mencionar tres detalles importantes:

1. Una Declaración de Entidad puede tener asociados varios Cuerpos de Arquitectura.

El lenguaje VHDL admite la posibilidad de que resulte necesario tener que describir de varias maneras el comportamiento de un dispositivo –con el objeto, fundamentalmente, de que se pueda disponer de versiones que lo modelen con distintos grados de abstracción–, por ello una Declaración de Entidad puede tener asociados cualquier número de Cuerpos de Arquitectura.

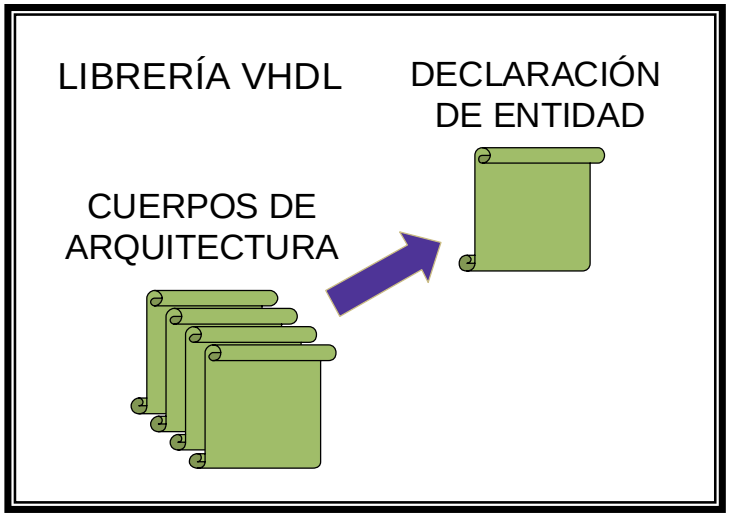


Figura I.8

2. Un Paquete puede estar formado únicamente por una Declaración de Paquete, es decir, el Cuerpo de Paquete puede no existir.

Esto es debido a que los Cuerpos de Paquete suelen contener la especificación de subprogramas –en realidad pueden contener más cosas– cuya declaración aparece en la Declaración de Paquete. Cuando un Paquete no contiene procedimientos o funciones puede que no exista el Cuerpo de Paquete.

3. Como ya se ha indicado anteriormente, cada librería VHDL tiene un nombre lógico. Además, la librería de trabajo que se esté utilizando en un determinado momento puede identificarse mediante la palabra **Work**.

Este último concepto puede entenderse mejor mediante el ejemplo que se va a realizar a continuación, donde, además, se revisarán algunos de los conceptos expuestos en este apartado.

8.- Ejercicio I.2:

1. Invoque la herramienta en la barra de programas de Windows.

Como puede ver, la configuración de arranque es la que tenía cuando la cerró por última vez, cuando realizó el ejercicio de simulación de la puerta **and**.

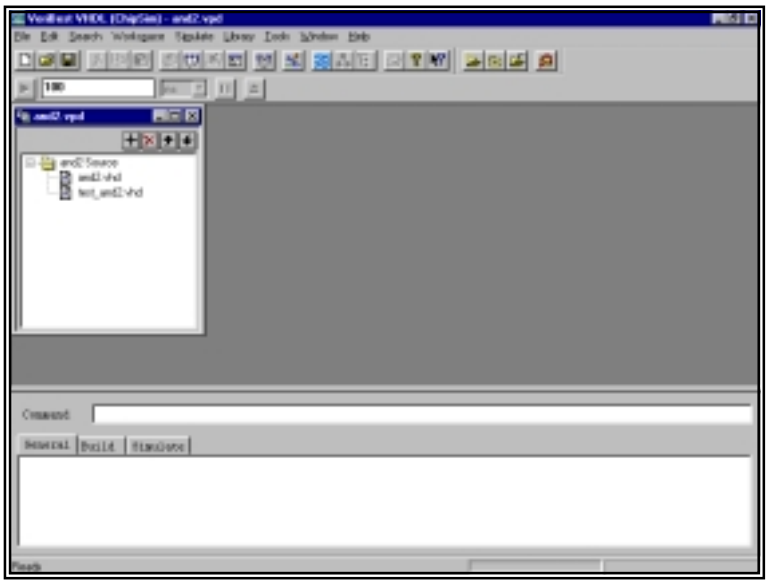


Figura e1

La interfaz del entorno muestra los ficheros de texto con los que está trabajando, pero no las librerías VHDL definidas o las unidades almacenadas en ellas.

2. Para poder verlas hay que activar la opción **Library Browser** en el menú **Library**.

Aparecerá una ventana como la de la figura.

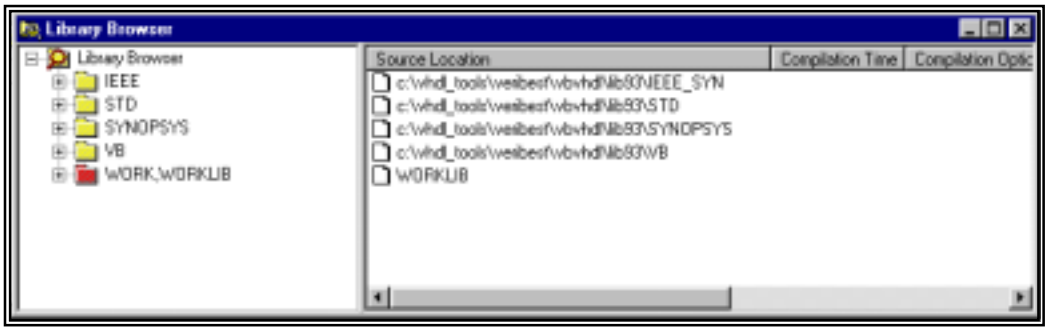


Figura e2

En la parte izquierda de la ventana aparecen las librerías definidas, con su nombre lógico (**IEEE**, **STD**, **SYNOPSYS**, **VB** y **WORK**), en la derecha su ***path*** en la estructura de directorios del ordenador.

3. Para ver las unidades almacenadas en cada librería hay que realizar una doble pulsación con el ratón en el icono que aparece al lado de su nombre. Realice esta operación sobre la librería **WORK** y maximice la ventana.

Observe que se muestran cuatro unidades de diseño: las Declaraciones de

Entidad y los Cuerpos de Arquitectura del modelo de la puerta **and** y del ***test-bench***. La representación gráfica, en este caso, denota las dependencias jerárquicas entre unidades de diseño.

4. Cierre la ventana del **Library Browser**.

5. Utilice los procedimientos descritos en el ejercicio 1 para crear un nuevo fichero con el siguiente contenido:

```
ARCHITECTURE rtl2 OF and2 IS
BEGIN

PROCESS(ent1, ent2)
BEGIN
  IF ent1 = '1' AND ent2 = '1' THEN
    sal <= '1';
  ELSE
    sal <= '0';
  END IF;
END PROCESS

END rtl2;
```

6. Sálvelo, con el nombre **and2_2.vhd**, y añádalo a la ventana del espacio de trabajo, pero no lo compile.

El Cuerpo de Arquitectura no forma parte todavía de la librería de trabajo, porque no se ha compilado. Puede comprobarlo en el **Library Browser**.

7. Compile el fichero **and2_2.vhd** y compruebe nuevamente el contenido de la librería **WORK**.

Como puede ver, la nueva unidad aparece en la librería, ligada, al igual que la arquitectura **rtl**, a la Declaración de Entidad de la puerta **and** (Figura e3).

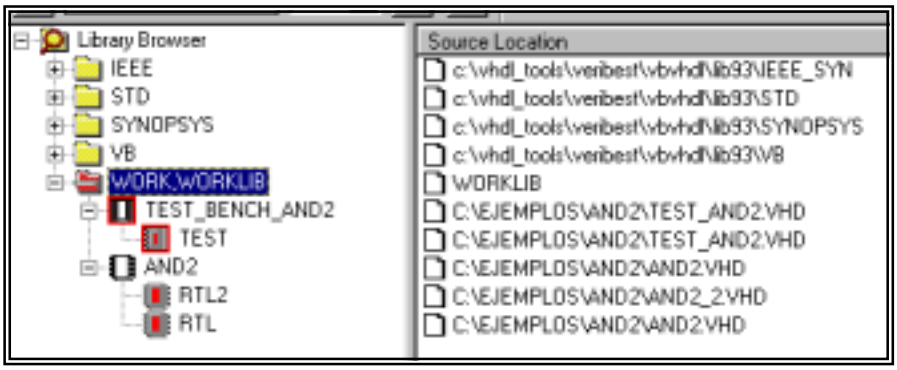


Figura e3

El resto de las librerías que aparecen en el **Library Browser** contienen Paquetes VHDL. En la librería **STD** aparecen los Paquetes **STANDARD** y **TEXTIO**; en ellos no existe el Cuerpo de Paquete; en la **IEEE**, en cambio, sí.

8. Puede revisar cualquier unidad de diseño pulsando el ratón sobre el icono que la representa. Si lo hace, tenga cuidado de no editar inadvertidamente el contenido de los paquetes.

9. Cierre el **Library Browser** .

10. Cree un nuevo espacio de trabajo y llámelo **ejer2**, en la ventana de creación, desactive la opción **Use Synopsys IEEE library (IEEE_SYN)**. (Figura e4).

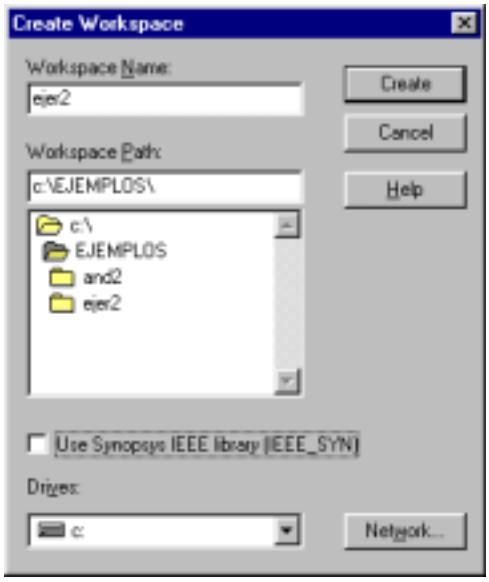


Figura e4

11. Compruebe en el **Library Browser** que la librería **WORK** está vacía. Observe también que no se dispone de las mismas librerías y paquetes que antes.

La librería **WORK** está vacía porque la nueva librería de trabajo lo está. Al crear un nuevo espacio de trabajo en el entorno **VeriBest**, se crea una nueva librería que es la que a partir de ese instante se reconocerá como librería **WORK**.

Si deseamos utilizar las unidades de diseño desarrolladas en un determinado espacio de trabajo desde otro distinto, tenemos que hacer que el primero sea reconocido como una librería VHDL.

12. Active la opción **Add Lib Mapping...** en el menú **Library**.

13. En la ventana que aparece, seleccione el directorio del espacio de trabajo donde se desarrolló el modelo de la puerta **and**. En el campo **Phisical Name:** escriba **WORKLIB**, en **Logical Name**, **LibAnd2**. (Figura e5). Pulse el botón **OK**.

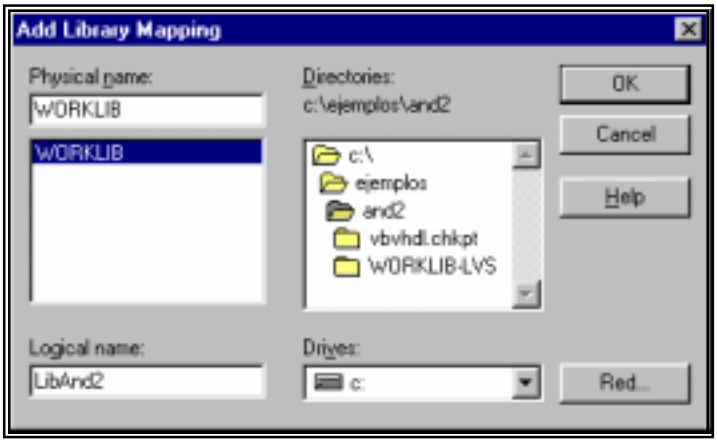


Figura e5

14. Compruebe en el **Library Browser** que se ha añadido una nueva librería que contiene las unidades de diseño desarrolladas hasta ahora. (Figura e6).

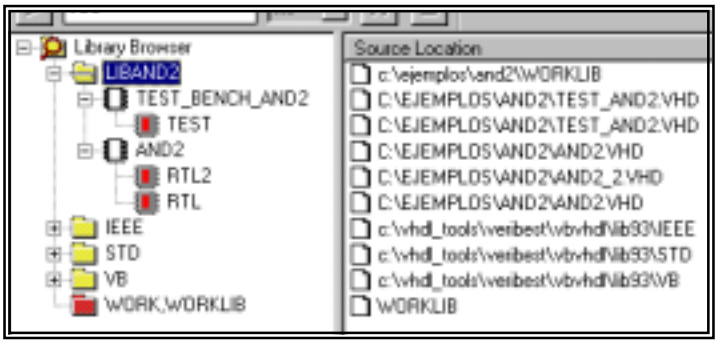


Figura e6

15. Edite e incluya en el espacio de trabajo dos ficheros con los contenidos, el nombre y en el orden que se señala a continuación:

Fichero: **Arch_and2.vhd**

```
ARCHITECTURE rtl OF and2 IS
BEGIN
sal <= '1' WHEN ent1 = '1' AND ent2 = '1'
                ELSE '0';
END rtl;
```

Fichero: **Entity_and2.vhd**

```
ENTITY and2 IS
PORT(
        ent1, ent2: IN BIT;
        sal: OUT BIT);
END ENTITY;
```

16. Active la opción **Compile All**.

Aparece un error que indica que la Declaración de Entidad correspondiente a la arquitectura no forma parte de la librería **WORK**. Esto es así porque todavía no la hemos compilado, ya que los ficheros se compilan en el orden en que aparecen en la ventana del **WorkSpace** cuando se utiliza el comando **Compile All**.

17. Seleccione con el ratón el fichero que contiene la entidad y pulse el botón de la ventana del **Workspace** que contiene una flecha que apunta hacia arriba. Vuelva a ordenar la compilación de los dos ficheros.

Observe que ahora se compilan correctamente.

18. Cierre el entorno **VeriBest**.

9.- Cláusulas de Visibilidad

Los modelos hardware y el contenido de los Paquetes VHDL almacenados en una determinada librería pueden utilizarse en unidades de diseño de otras. Para ello es preciso utilizar cláusulas de visibilidad. La visibilidad sobre las unidades que componen una librería se obtiene mediante la cláusula **Library**. Por ejemplo:

```
LIBRARY libmodelos;

ARCHITECTURE rtl OF modelo1 IS...
```

El código VHDL del ejemplo permite que las unidades de diseño almacenadas en la librería **libmodelos** puedan utilizarse dentro de la arquitectura **rtl** de **modelo1**. Es importante resaltar que está cláusula por si misma no da acceso al contenido de las unidades, sólo permite nombrarlas, lo que en determinadas aplicaciones resulta suficiente.

Las cláusulas de visibilidad se usan muchas veces para poder utilizar los objetos y subprogramas declarados en los Paquetes. Para poder utilizar el contenido de un paquete resulta necesario:

- Obtener visibilidad sobre la librería donde está almacenado.
- Obtener visibilidad sobre los contenidos del propio paquete.

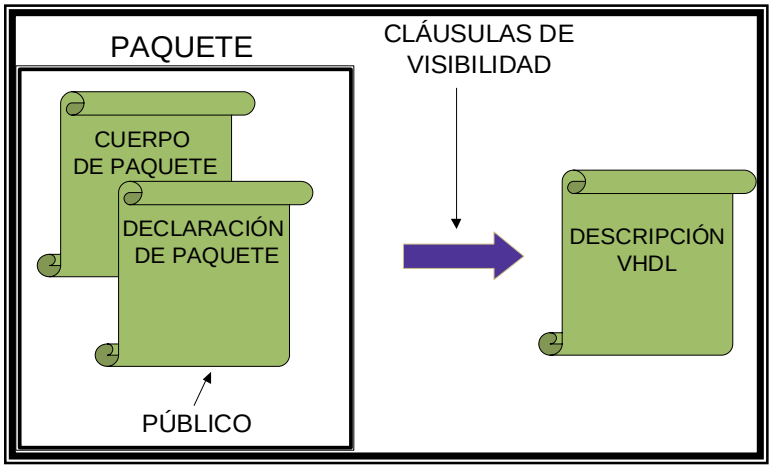


Figura I.9

En el desarrollo de modelos VHDL resulta muy frecuente el uso de los Paquetes de la librería **IEEE**. Dos de los paquetes de esa librería se llaman: **std_logic_1164** y **numeric_std**. El siguiente código:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY modelo1 IS . . .
```

permitiría el uso del contenido del paquete **std_logic_1164** en la Declaración de Entidad **modelo1** y en cualquiera de los Cuerpos de Arquitectura que tenga asociados (debido a las reglas de herencia de visibilidad entre unidades primarias y secundarias). Este otro código:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL, ieee.numeric_std.ALL;

ENTITY modelo1 IS
. . .
```

daría acceso al contenido de los dos Paquetes.

Dentro del Paquete **std_logic_1164** se declara un tipo de datos llamado **std_ulogic**. Si sólo queremos obtener visibilidad sobre ese objeto, podríamos utilizar la fórmula:

```
LIBRARY ieee;
USE ieee.std_logic_1164.std_ulogic;
```

Hay algunas excepciones a las reglas de visibilidad descritas:

1. La librería **Work** siempre es visible.

Esto quiere decir que una cláusula –por lo demás, perfectamente válida– como:

```
LIBRARY WORK;
```

es superflua. Sin embargo, si existiera un Paquete (**packwork**, por ejemplo) en la librería **Work** cuyo contenido se deseára utilizar, si sería necesario utilizar la fórmula:

```
USE WORK.packwork.ALL;
```

2. La librería **STD** y el contenido del paquete **STANDARD** son siempre visibles.

La librería **STD** contiene los paquetes **STANDARD** y **TEXTIO**; en realidad estos Paquetes forman parte del lenguaje, ya que contienen la declaración de los tipos de datos predefinidos y las operaciones de entrada-salida. La visibilidad sobre el primero es automática; para utilizar el paquete TEXTIO hay que hacer uso de la cláusula **USE**.

10.- Ejercicio I.3

1. Invoque el entorno **VeriBest** desde la barra de programas de Windows.

2. Active el espacio de trabajo **and2** seleccionándolo en la lista que aparece en el menú **Workspace** (donde siempre aparecen los últimos espacios creados).

3. Active el **Library Browser**.

Observe que ya no aparece la librería **LibAnd2**. Esto es debido a que la definición de una librería, en el entorno **VeriBest** sólo es válida para el espacio de trabajo donde se definió.

4. Defina el espacio de trabajo **ejer2** como librería **libejer2**, repitiendo la operación realizada en el ejercicio 2.

5. Edite el fichero **test_and2.vhd**, realizando una doble pulsación con el ratón sobre el icono de la ventana del **Workspace** y modifíquelo, de modo que su nuevo contenido sea el siguiente:

```
LIBRARY libejer2;

ENTITY test_bench_and2 IS
END ENTITY;

ARCHITECTURE test OF test_bench_and2 IS
  SIGNAL s_ent1, s_ent2, s_sal: BIT;
BEGIN

 s_ent1 <= '1' AFTER 10 NS;

 s_ent2 <= '1' AFTER 5 NS,
        '0' AFTER 10 NS,
        '1' AFTER 15 NS;

 DUT: ENTITY libejer2.and2(rtl)
PORT MAP( ent1 => s_ent1, ent2 => s_ent2, sal => s_sal);

END test;
```

Con los cambios realizados se ha dado visibilidad a la Declaración de Entidad **test_bench_and2** sobre la librería **libejer2**; la visibilidad es heredada por el Cuerpo de Arquitectura, donde se emplaza el modelo de puerta **and** de esa librería.

6. Salve el fichero y compílelo.

La compilación debe tener éxito.

7. Vuelva a editar el fichero y cambie la posición de la cláusula **LIBRARY**, de modo que quede delante de la cabecera del Cuerpo de Arquitectura del ***test-bench***.

8. Salve el fichero y vuelva a compilarlo para comprobar que el proceso se realiza correctamente.

Esto es así, evidentemente, porque sólo se precisa la visibilidad en el Cuerpo de Arquitectura, donde se utiliza la librería para formar el nombre que identifica el dispositivo bajo prueba.

11.- Nombres VHDL

Para finalizar con los temas que se tratan en este capítulo, se va a comentar algo sobre los nombres de objetos en VHDL. En el ***test-bench*** de los tres ejercicios realizados se nombra al dispositivo bajo prueba mediante su nombre jerárquico:

WORK.and2(rtl)

libejer2.and2(rtl)

Esta es la manera en que se construyen los nombres en VHDL y permite, por ejemplo, que se pueda utilizar el mismo nombre para dos unidades de diseño situadas en distintas librerías sin que exista ambigüedad a la hora de identificarlos.

Respecto a este ejemplo concreto resulta interesante observar una fórmula que aparece en algunas construcciones VHDL para identificar modelos de dispositivos:

Nombre_de_Entidad (Arquitectura)

Sirve para resolver la ambigüedad que puede crear la existencia de más de un Cuerpo de Arquitectura asociado a una Declaración de Entidad.

Los nombres jerárquicos completos en VHDL se construyen encadenando ámbitos de declaración separados por puntos.

Por ejemplo, el puerto **sal** de la Declaración de Entidad de la puerta **and** de la librería de trabajo se nombraría:

WORK.and2.sal

El uso del nombre jerárquico es obligatorio cuando exista ambigüedad y opcional en cualquier otro caso. Por ejemplo, suponga que una entidad dispone de visibilidad sobre dos Paquetes: **paq1** y **paq2**, almacenados en las librerías **lib1** y **lib2** respectivamente y que en cada uno existe un objeto –de cierta clase–, dándose la casualidad de que se llamen igual, por ejemplo **X**. Si el objeto se referencia con su nombre simple, el compilador no puede determinar cuál de los dos se desea utilizar y reporta un error, así que resulta obligatorio nombrarlo

como **lib1.paq1.X** o **lib2.paq2.X**.

12.- Estructura Básica del Lenguaje VHDL

Las unidades de diseño VHDL se realizan componiendo código con las construcciones que se definen en el lenguaje: cabeceras de unidades, declaraciones de objetos, sentencias, etc. Todas ellas están definidas en la norma **IEEE-1176**. Los diseñadores del lenguaje tomaron la decisión de que los tipos de datos predefinidos (**INTEGER, REAL,** etc.) y las operaciones de entrada-salida sobre ficheros se declararán en Paquetes VHDL: los Paquetes **STANDARD** y **TEXTIO** de la librería **STD**. Debido a que la visibilidad sobre estos paquetes es automática, puede trabajarse con ellos obviando está curiosa circunstancia.

Cualquier entorno VHDL standard actual incorpora, además, los paquetes de la librería **IEEE**. Estos Paquetes se crearon para dar soporte a las necesidades descriptivas que surgen en los modelos VHDL orientados a la síntesis lógica automática. En ellos se definen nuevos tipos de datos, operaciones lógicas y aritméticas para esos tipos y funciones de conversión entre tipos. Existe un problema con estos Paquetes: hay dos versiones, la normalizada por **IEEE** y el estándar *de facto* realizado por **Synopsys**. La mayor parte de los entornos utiliza el segundo, en algunos, como **VERIBEST**, el usuario puede elegir entre ambos. Ambas versiones comparten un Paquete (el llamado **ieee.std_logic_1164**), el resto son distintos. En la figura I.10 se muestran las dos configuraciones básicas que se pueden dar.

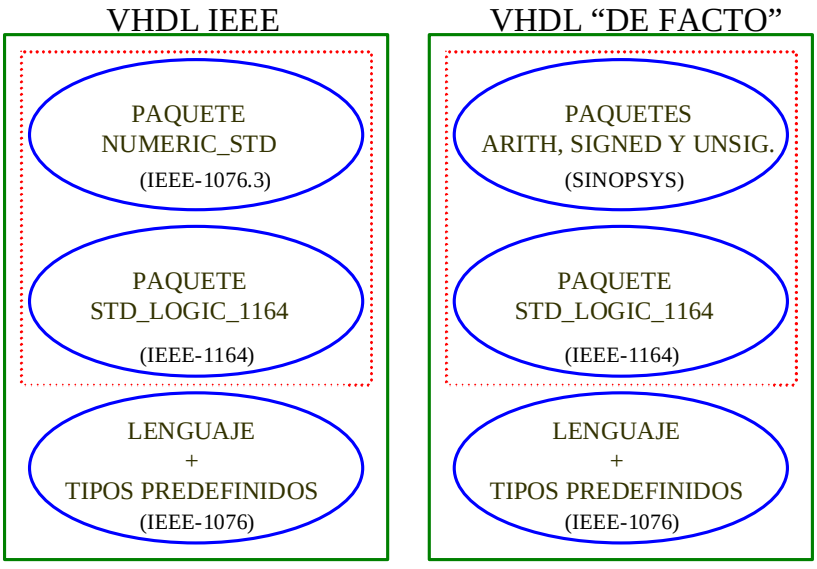


Figura I.10

Hoy por hoy, es preferible elegir la versión de **Synopsys**, porque es la única que soportan la mayoría de los simuladores y sintetizadores VHDL.

Además, algunos entornos proporcionan Paquetes desarrollados por ellos mismos. Su uso puede acarrear problemas de portabilidad, por lo que debe, en la medida de lo posible evitarse.

13.- Apéndice

La norma del lenguaje VHDL no especifica detalles sobre el modo en que deben implementarse las librerías en los entornos VHDL. No se especifican las estructuras de almacenamiento, ni las utilidades que se deben ofrecer al usuario. Por este motivo resulta a veces engorroso transportar código entre entornos.

En muchos entornos se utiliza un directorio del sistema de ficheros del ordenador para almacenar una librería (**VeriBest**, por ejemplo), en otros se utilizan estructuras jerárquicas (**Leapfrog**); a veces los entornos disponen de comandos de importación y exportación de unidades de diseño que facilitan las operaciones y permiten, por ejemplo, indicar la librería donde se desea almacenarlas.

Cada entorno dispone también de distintas utilidades para la creación de librerías, asignación y cambio de nombres lógicos, establecimiento de la librería **WORK**, etc.

Con los ficheros que contienen unidades de diseño también pueden surgir problemas. Hay herramientas VHDL que permiten que en un fichero vaya cualquier número de unidades de diseño (**VeriBest)**, otros que sólo permiten que haya una (**Leapfrog**), algunos establecen reglas que relacionan el nombre de los ficheros y el de las unidades que contienen; la mayoría exige que la extensión de los ficheros sea **vhd**. Además, casi ningún entorno proporciona utilidades para la gestión de versiones.

A continuación se da una serie de consejos que pueden ayudarle a evitar problemas debidos a estas cuestiones:

1. Ponga siempre la extensión **vhd** a los ficheros que contienen código VHDL.

2. No incluya en un fichero más de una unidad primaria; si es posible, cree un fichero para cada unidad de diseño.

3. Nunca mezcle en un mismo fichero unidades de distintas librerías.

4. Ponga a cada fichero que contenga una unidad primaria el nombre de ésta.

5. Utilice nombres de ficheros válidos para los principales sistemas operativos (**UNIX**, **DOS**, **Windows**).

6. Mantenga una copia de cada unidad de diseño de una librería en un directorio fuera de la estructura de librerías del entorno que utilice; es decir, mantenga una gestión de librerías particular donde cada librería debe estar en un directorio de su sistema de ficheros.

OBJETOS Y TIPOS DE DATOS

0.- Resumen del Capítulo

Conceptos Teóricos:

- *Objetos VHDL: Señales, Variables y Constantes.*
- *Puertos de la Declaración de Entidad.*
- *Tipos de Datos Predefinidos.*
- *Tipos y Subtipos de Usuario.*
- *Atributos de los Tipos de Datos.*
- *Paquetes VHDL*

Prácticas sobre el simulador VeriBest:

- *Asignación de valores a señales.*
- *Tipos de datos.*

Apéndice:

- *Paquetes de la librería IEEE.*

En este capítulo se presentan los objetos del lenguaje VHDL, haciendo un especial énfasis en las señales y se profundiza en el estudio de las características de los puertos de la Declaración de Entidad. A continuación se revisan los tipos predefinidos del lenguaje, el mecanismo que proporciona para la definición de tipos y subtipos de usuario y, por último, los tipos definidos en los paquetes de la librería IEEE, el concepto de tipos con función de resolución y la sintaxis de los Paquetes VHDL.

1.- Señales, variables y constantes.

El lenguaje VHDL dispone, al igual que los lenguajes de programación, de **variables** y **constantes** que pueden utilizarse en la realización de descripciones hardware y ***test-benches***. Además, proporciona una nueva clase de objetos: las **señales**.

Todos los objetos VHDL deben ser declarados (el lenguaje establece en qué regiones puede o no declararse cada clase de objeto); en la declaración ha de indicarse su clase (constante, variable o señal), su tipo de datos y, si es preciso –en las constantes– o se desea, su valor inicial.

La manera más sencilla de empezar a entender la naturaleza de las señales VHDL es como objetos que modelan nodos lógicos (lo estrictamente correcto sería decir que los nodos lógicos se modelan con señales). Debido a este propósito, las señales, que por lo demás son muy parecidas a las variables, disponen de un mecanismo de actualización de valor especial. Este mecanismo da lugar a que cuando se simula un modelo VHDL, inmediatamente después de la ejecución de una sentencia de asignación de valor a señal, el valor de la misma no cambia.

Por ejemplo, si **A** es una señal y **B** una variable, cuyos valores son **'0'** y **'1'**, respectivamente, entonces, tras ejecutar las sentencias:

```
A <= B after 5 ns;      -- <= es el símbolo de asignación a señal

B := A;                 -- := es el símbolo de asignación a variable

                        -- los guiones son el símbolo de comentario
```

A y **B** valdrán **'0'**.

2.- Declaración de objetos

La formula de declaración de constantes, variables y señales es:

***Tipo_de_Objeto Nombre_de_Objeto* : *Tipo_de_datos* := *Valor_Inicial*;**

Por ejemplo:

```
SIGNAL nodo1 : BIT := '1';

VARIABLE var1 : BIT;

CONSTANT pi : REAL := 3.14159;
```

La inicialización es optativa en el caso de señales y variables y obligatoria en el de las constantes. El campo que señala el tipo de objeto es optativo cuando el lenguaje define una clase obligatoria o por defecto. Por ejemplo, los puertos de la Declaración de Entidad son, por definición, señales, así que cuando se declaran en la lista de interfaz su nombre no se antecede con la palabra reservada **SIGNAL**; los parámetros de entrada a un subprograma son, por defecto, constantes, de modo que si en la lista de parámetros de un subprograma el nombre de un valor de entrada no está precedido por las palabras **SIGNAL** o **VARIABLE**, se considerará que es una constante –aunque si se desea se puede poner la palabra **CONSTANT**–.

3.- Asignaciones de valor a señal

La consecuencia de una asignación de valor a señal es que **se proyecta la transacción** de un valor a la señal para un determinado instante del **tiempo de simulación**. Esto, dicho así, no resulta fácil de entender. La clave de la cuestión es la siguiente: las asignaciones de valor a señal, como la del ejemplo de la página anterior, forman parte de las descripciones con que se modela el hardware, y su capacidad de modelado se materializa cuando se efectúa una simulación –en un simulador VHDL o en la imaginación del diseñador–. Al realizarse la simulación, el código del modelo se ejecuta en determinados instantes dentro del **tiempo de simulación**; una asignación de valor a señal que se ejecute en un instante **T**, se proyectará para un instante posterior **T+t**, en que se efectuará.

Si el código del ejemplo mencionado se ejecuta en el instante **T = 10 ns**, por ejemplo, **A** mantendrá el valor **‘0’** hasta **T = 15 ns**, en que pasará a valer uno.

El proceso descrito se va a revisar en el siguiente ejercicio.

4.- Ejercicio II.1:

1. Arranque el simulador **VeriBest** en la barra de programas de **Windows**.

2. Deshaga los cambios realizados en el ejercicio I.3, para repetir la simulación del ejercicio I.1, y modifique la sentencia de asignación a señal del modelo de la puerta **and** añadiendo un retardo de **2 ns**:

   sal <= ent1 and ent2 after 2 ns;

3. Vuelva a compilar todos los ficheros del espacio de trabajo y active la opción **Execute Simulator** en el menú **Workspace**.

4. Visualice el fichero **and2.vhd**, haciendo una doble pulsación sobre el icono del fichero en la ventana del espacio de trabajo.

5. Sitúe el cursor de edición, con el ratón, sobre la línea correspondiente a la sentencia de asignación a señal y active la opción **Debug -> Insert/Remove a breackpoint** para fijar un punto de ruptura (figura e1).

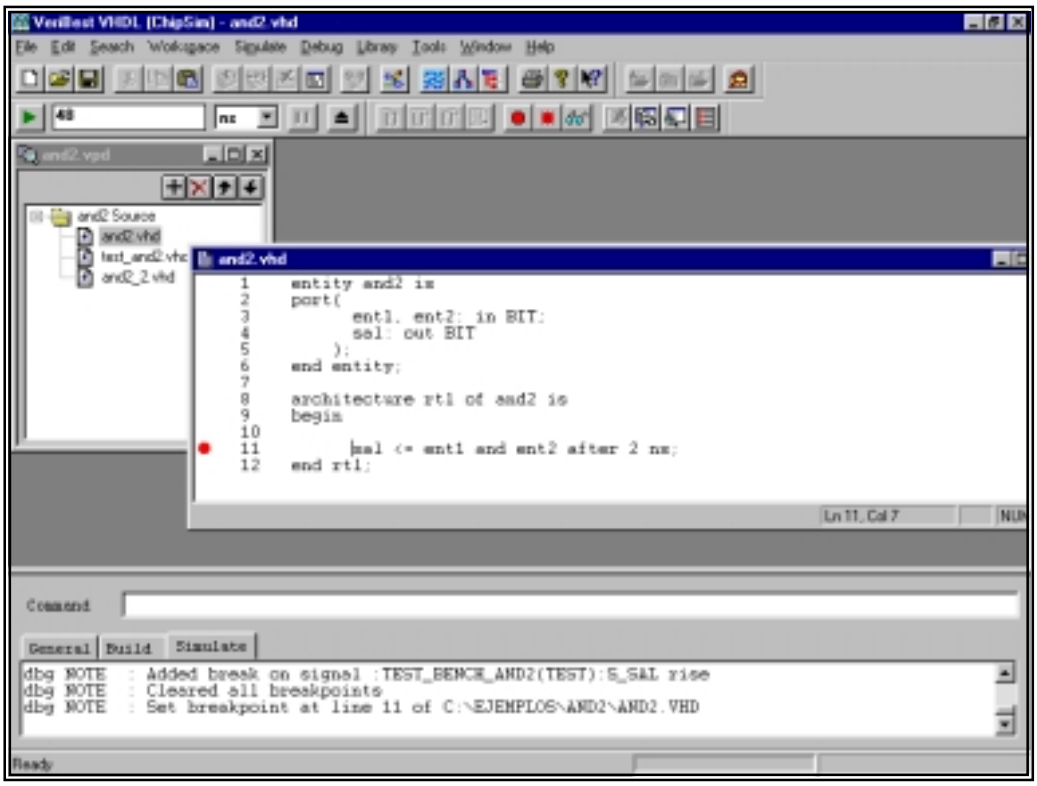


Figura e1

La sentencia que modela el comportamiento de la puerta se activa cada vez que hay un **evento**, un cambio de valor, en las señales a las que es **sensible**, **ent1** o **ent2**. Estas, por su parte, son manejadas por los estímulos a los

que está conectada la puerta en el ***test-bench***: **s_ent1** y **s_ent2**. Las sentencias que proyectan asignaciones de valor para estos estímulos son:

```
s_ent1 <= '1' after 10 ns;
```

y

```
s_ent2 <= '1' after 5 ns,
      '0' after 10 ns,
      '1' after 15 ns;
```

que se ejecutan en el tiempo de simulación **0 ns** y proyectan **transacciones** sobre las señales en los tiempos de simulación **5**, **10** y **15 ns**.

Por tanto, habrá eventos en las señales, y deberá ejecutarse la sentencia que modela la puerta **and**, en esos mismos instantes.

6. Visualice en una ventana de formas de onda las señales **s_ent1**, **s_ent2** y **s_sal**.

7. Active la opción **Run Forever** en el menú **Simulate**.

Observe que la simulación se para en el **tiempo de simulación** 5 ns, antes de ejecutarse la sentencia de asignación de valor a señal. A continuación se va a ejecutar la simulación **paso a paso**.

8. Pulse el botón de simulación **paso a paso**, (el de la izquierda en la figura e2).

Figura e2

Observe que la simulación avanza hasta la siguiente sentencia que se ejecuta, otra vez la de asignación de valor a señal (la única que existe en el modelo que se simula), en el tiempo **10 ns**.

9. Vuelva a pulsar el botón de simulación **paso a paso** para que se ejecute la sentencia en el tiempo **10 ns**.

La simulación avanza hasta **15 ns**. Ahora, **ent1** y **ent2** valen **'1'** y, por tanto, se proyectará la asignación del valor **'1'** sobre **sal** en **2 ns** (para el tiempo de simulación **17 ns**) cuando se ejecute la sentencia. Antes de hacerlo, se va a poner un punto de ruptura condicionado a la ocurrencia de un flanco de subida de la señal **s_sal**.

10. Active la opción **Debug -> Break on signal...** En la ventana que aparece (figura e3), seleccione, con el ratón, la señal **s_sal**, en la ventana **edge condition**, seleccione **rising**, pulse **Set Break** y, por último, **Close**.

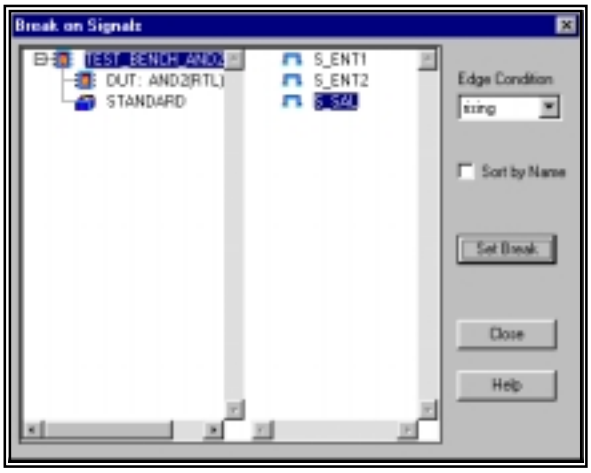


Figura e3

11. Pulse el botón de ejecución **paso a paso**.

La simulación avanza hasta **17 ns** donde se dibuja la transición de la salida.

12. Ordene que la simulación avance otros **8 ns** y maximice la ventana de presentación de formas de onda. Cambie el factor de escala para obtener una vista como la de la figura e4.

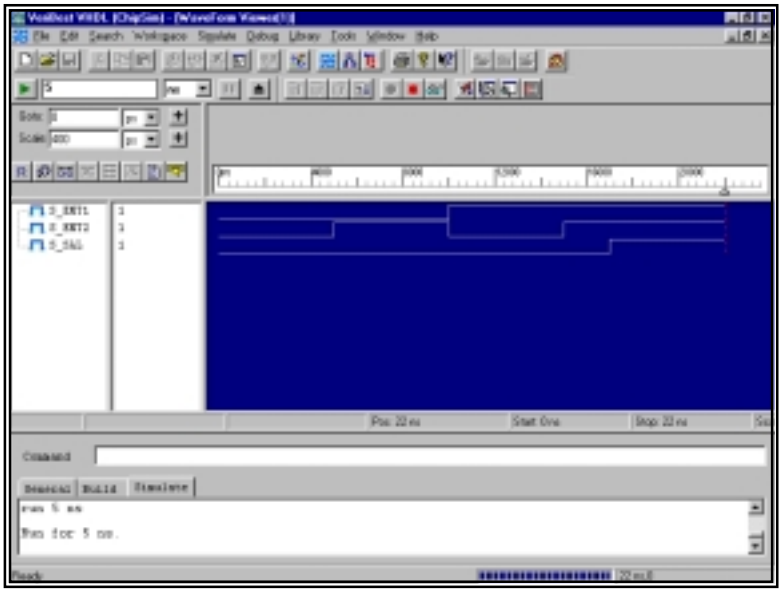


Figura e4

Puede medir el retardo de la salida activando el icono **Add Time Cursor** en la ventana de formas de onda y situando los cursores en los flancos apropiados.

13. Finalice la simulación y cierre **VeriBest**.

5.- Puertos de la Declaración de Entidad y Tipos de Datos

Los puertos de la Declaración de Entidad sirven para modelar los terminales de los dispositivos; pues bien, los puertos son **señales direccionales**. El problema de la dirección de los puertos tiene que ver con el hecho de que siempre van a utilizarse en la descripción del funcionamiento del hardware en el Cuerpo de Arquitectura y entonces deberán respetarse las siguientes normas:

- los puertos de entrada sólo pueden leerse,
- en los puertos de salida sólo puede escribirse,
- en los puertos bidireccionales se puede leer o escribir.

En este contexto, leer ha de entenderse como situarse en la parte derecha de una sentencia de asignación, o formar parte de una condición que sea preciso evaluar para tomar una decisión, y escribir situarse en la parte izquierda de una sentencia de asignación; por ejemplo, en una sentencia de asignación como :

Sal <= NOT Ent;

Si **Sal** y **Ent** son puertos de una Declaración de Entidad, **Sal** deberá ser de salida y **Ent** de entrada para que la sentencia sea correcta en VHDL. En esta otra:

IF ent = '1' THEN...

Si **ent** es un puerto, deberá ser de entrada.

En la mayor parte de las ocasiones, asignando la "dirección hardware" al puerto las normas anteriores permiten modelar el comportamiento con naturalidad; pero en ocasiones puede interesarnos poder leer el valor de un puerto de salida; por ejemplo, es habitual utilizar una sentencia como la siguiente formando parte del modelo de un contador:

Cont <= Cont + 1;

Para resolver estas situaciones el lenguaje provee la dirección de tipo **BUFFER**. **BUFFER** es la dirección que debe tener un puerto de salida VHDL cuyo valor se precisa leer al describir el funcionamiento del dispositivo. Un puerto de tipo **BUFFER** es, desde el punto de vista hardware, de salida y, en cuanto a sus propiedades de lectura y escritura, en el sentido descrito anteriormente, bidireccional. En ningún caso puede utilizarse para modelar puertos hardware bidireccionales pues el lenguaje prohibe que este tipo de puertos tengan más de un ***driver***, lo que impide, como se verá más adelante, que sirvan para este propósito.

Si cuando se declara un puerto de salida no se sabe qué alternativa escoger (**OUT** o **BUFFER**), puede esperarse a la realización del Cuerpo de Arquitectura para tomar la decisión.

6.- Tipos de Datos

Como se adelantó en el capítulo anterior, además de la dirección –y el nombre–, hay que definir el tipo de datos del puerto –esto puede verse también como una consecuencia de que los puertos sean señales–. El problema de la elección del tipo de datos es más complicado. Para decidir el tipo de datos que se debe utilizar hay que evaluar, fundamentalmente, dos factores:

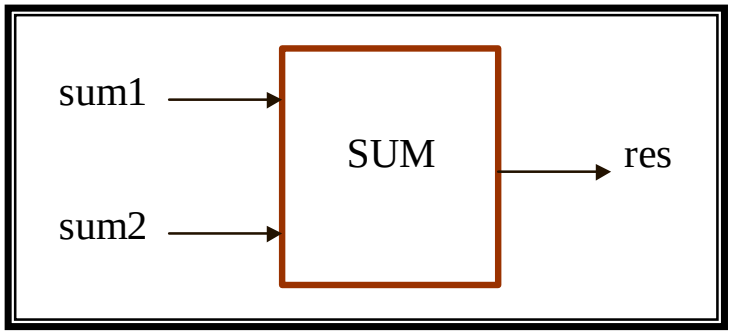
1. El nivel de abstracción del modelo.

2. Las características del dispositivo que se modela.

Además, se debe tener en cuenta que VHDL es un lenguaje con reglas estrictas de asignación de valores: a un objeto sólo se le pueden asignar valores del tipo sobre el que está definido o el valor de un objeto de su mismo tipo.

El lenguaje predefine una serie de tipos (en el paquete **STD** de la librería **STANDARD**) y permite que el usuario defina los que pueda necesitar. Además, se dispone de los tipos definidos en los paquetes de la librería **IEEE** que se pueden utilizar mediante la inclusión de las cláusulas de visibilidad oportunas.

Los tipos de datos se diferencian entre sí por el conjunto de valores que se definen en cada tipo y por las operaciones que se pueden realizar con ellos. Por ejemplo, en VHDL está definido el tipo **INTEGER** igual que en los lenguajes de programación de alto nivel, por lo que se puede utilizar en la declaración de un puerto. Considerando las características del tipo **INTEGER**, puede parecer adecuado para su uso en modelos donde sea preciso modelar operaciones aritméticas con enteros. Así resulta sencillo construir el modelo de un circuito sumador:




```
ENTITY sum IS
PORT(
        sum1, sum2: IN   INTEGER;
        res         : OUT   INTEGER
    );
END sum;

ARCHITECTURE rtl OF sum IS
BEGIN
        res <= sum1 + sum2;
END rtl;
```

El anterior es un modelo abstracto de un circuito sumador, y presentaría problemas si se hubiera construido para realizar una síntesis lógica automática o fuera preciso acceder a determinados bits de los puertos de entrada o salida –en el primer caso porque el sintetizador probablemente generaría un circuito sumador de tantos bits como utilice para representar enteros y, en el segundo, porque un valor de tipo **INTEGER** no se puede descomponer en bits–.

Por otro lado, el lenguaje VHDL tiene definidos dos tipos de datos para el modelado de señales digitales: el tipo **BIT** y el tipo **BIT_VECTOR**; el primero para bits y el segundo para buses (el tipo **BIT_VECTOR** es un *array* de objetos de tipo **BIT**). Utilizando este tipo de datos podría pensarse en modificar la Declaración de Entidad del sumador:

```
ENTITY sum IS
PORT(
  sum1, sum2 : IN   BIT_VECTOR(3 DOWNTO 0);
  res : OUT   BIT_VECTOR(3 DOWNTO 0)
 );
END sum;
```

El problema que surge aquí es que no está definida la operación suma para objetos de tipo **BIT_VECTOR**, de modo que al compilar el cuerpo de arquitectura aparecería un error en la sentencia:

```
res <= sum1 + sum2;
```

Este ejemplo demuestra las consecuencias que puede tener una elección inadecuada del tipo de datos. Antes de proponer alguna receta que facilite la toma de decisiones, se van a presentar los tipos de datos predefinidos.

7.- Tipos de datos predefinidos

Los **tipos predefinidos** –todos ellos declarados en los paquetes **STD** y **TEXTIO**– son los siguientes:

1. **Enumerados:**
   CHARACTER: Formado por los 128 caracteres **ASCII**.
   BOOLEAN: Definido sobre los valores (**FALSE**, **TRUE**).
   BIT: Formado por los caracteres **'0'** y **'1'**
   SEVERITY LEVEL: Formado por los valores **(NOTE, WARNING, ERROR, FAILURE)**.

2. **Escalares:**
   INTEGER y **REAL**: Con un rango que depende de cada herramienta.

3. ***Arrays*:**
   STRING: Definido como un *array* de caracteres.
   BIT_VECTOR: Definido como un *array* de **BIT.**

4. **Físicos:**
   TIME: Definido para especificar unidades de tiempo.

5. **Tipos para operaciones de entrada/salida sobre ficheros:**
   LINE: Puntero a **STRING**.
   TEXT: Fichero de **STRINGS**
   SIDE: Enumerado sobre los valores **(RIGHT, LEFT)**

También existen subtipos predefinidos por restricción de rango:

NATURAL: desde **0** al máximo valor **INTEGER**.
POSITIVE: desde **1** al máximo valor **INTEGER**.
WIDTH: desde **0** al máximo valor **INTEGER**.

Algunos de los tipos de datos anteriores son similares a los de los lenguajes de programación. Las operaciones disponibles para ellos, también. Los tipos **BIT** y **BIT_VECTOR**, **SEVERITY_LEVEL** y **TIME** se proveen específicamente para el modelado hardware.

Los tipos **BIT** y **BIT_VECTOR** están orientados al modelado de niveles lógicos. Una de sus principales características, la inexistencia de operaciones aritméticas predefinidas, ya ha sido expuesta; las consecuencias que acarrea: su inadecuación para el modelado de procesamiento aritmético, también. Igualmente limita su uso el hecho de no tener definidos valores **metalógicos** que representen estados como el indeterminado o el de alta impedancia.

Hay que señalar que en la versión normalizada de las librerías de **IEEE** –en la de **Synopsys** no– existe un paquete, denominado **numeric_bit**, donde se definen operaciones aritméticas para estos tipos. Pero es una utilidad que apenas se utiliza, en primer lugar, por la escasa –por tardía– difusión de la versión normalizada de la librería **IEEE** y, en segundo lugar, porque no resuelve el problema de la indefinición de valores metalógicos y, como pronto se verá, hay otros tipos disponibles que sí disponen de ellos.

Ejemplos:

```
SIGNAL nodo1, nodo2, nodo3: BIT;

SIGNAL bus_datos:BIT_VECTOR(0 TO 15);

CONSTANT dir : BIT_VECTOR(31 DOWNTO 0) := X"C3A9_0000";

VARIABLE val_ini: BIT_VECTOR(63 DOWNTO 0) := (OTHERS => '0');
```

En estos ejemplos pueden observarse algunos detalles sintácticos de interés:

1. En una sentencia de declaración pueden declararse varios objetos si son de la misma clase y comparten el tipo de datos.

2. La fórmula de especificación del rango de un *array* es:

(*limite_izquierdo TO/DOWNTO limite_derecho*)

si se utiliza la palabra **DOWNTO**, para definir un rango descendente, el limite izquierdo debe ser mayor o igual que el derecho; si se utiliza **TO**, para definir un rango ascendente, al revés. De acuerdo con esto, los siguientes rangos:

(0 TO 7)
(1 DOWNTO 0)

son válidos, mientras que estos otros, no:

(7 TO 0)
(0 DOWNTO 1)

3. La señal **bus_datos** es de tipo **BIT_VECTOR**, es decir, es un *array* de valores de tipo **BIT**; por tanto es un **STRING** –un *array* de caracteres–, puesto que el tipo **BIT** se define por enumeración de los caracteres cero (**'0'**) y uno (**'1'**). Observe como se especifica el valor de un **STRING**, entre comillas, mientras que en los caracteres se utilizan apóstrofes.

4. En la inicialización de la constante **dir** se utiliza la base hexadecimal para evitar una larga sucesión de ceros y unos. La letra que antecede a la ristra de números especifica la base de numeración (**O** para octal, **X** para hexadecimal, **B** para binario; sin prefijo se entiende que es binario); además pueden intercalarse "subrayados" para facilitar la legibilidad.

5. La fórmula **(OTHERS => '0')** inicializa todos los valores del *array* a **'0'**. Su uso es frecuente cuando se trabaja con vectores largos.

El modo de nombrar los *arrays* declarados –o a una parte de ellos– puede aprenderse en los siguientes ejemplos:

```
val_ini(31 DOWNTO 0) := dir; -- Se asigna dir a una parte de val_ini

bus_datos(3) <= nodo1;          -- Se asigna al elemento 3 nodo1

bus_datos (0 TO 2) <= nodo3 & nodo2 & nodo1;
```

En el último ejemplo, se utiliza el operador de encadenamiento, **&**, de modo que al elemento **0** del *array* se le asigna el valor de **nodo3**, al **1** el de **nodo2** y al **2** el de **nodo1** (obsérvese que **bus_datos** está declarado con rango ascendente; si tuviera rango descendente, **nodo1** se asignaría al elemento **0** y **nodo3** al **2**).

El tipo **TIME** forma parte de una clase denominada **tipos físicos**. Estos tipos se definen mediante una unidad básica, un conjunto de unidades secundarias y una restricción de rango. Por ejemplo, la declaración del tipo **TIME** en el paquete **STANDARD** es:

```
-- predefined type TIME:
```

```
TYPE TIME IS RANGE
        UNITS
                FS;
                PS = 1000 FS;
                NS = 1000 PS;
                US = 1000 NS;
                MS = 1000 US;
                SEC = 1000 MS;
                MIN = 60 SEC;
                HR = 60 MIN;
        END UNITS;
```

Valores válidos de este tipo son, por ejemplo:

25 NS, **2.7 SEC, 5E3 PS, 2 HR**

que corresponden a 25 nanosegundos, 2'7 segundos, 5000 picosegundos y 2 horas.

Una variable o una constante –ya que no tiene sentido definir una señal con este tipo– de tipo **TIME** se declaran así:

```
VARIABLE retardo: TIME;

CONSTANT dia: TIME:= 24 HR;
```

Los retardos que se especifican en las asignaciones de valor a señal deben ser valores u objetos de tipo **TIME**:

```
Bus_datos(0 TO 2) <= "010" AFTER retardo;

Bus_datos(3) <= '0' AFTER 3 MS;
```

Con los valores y objetos de tipo **TIME** pueden utilizarse operaciones aritméticas y de comparación. Se reporta un error si la evaluación del resultado de una operación es un tiempo (retardo) negativo.

El tipo **SEVERITY_LEVEL** se utiliza en sentencias de comprobación de condiciones, para avisar sobre la gravedad de determinadas situaciones. Por ejemplo, para avisar del incumplimiento de un tiempo de ***set-up*** de un ***flip-flop*** o abortar una simulación cuando se ha producido una situación que da lugar a una alerta de nivel **ERROR** o **FAILURE.** Este tipo de datos no dispone de operaciones.

8.- Operadores predefinidos

En la siguiente tabla se muestran los símbolos de las operaciones predefinidas.

TIPO DE OPERACIÓN	SIMBOLO	FUNCION
Aritméticas de dos operandos	**+** **-** ***** **/** **mod** **rem** ******	**suma** **resta** **producto** **división** **módulo** **resto** **potencia**
Aritméticas de un operando	**+** **-** **abs**	**Incremento** **Decremento** **Valor absoluto**
Relacionales	**=** **/=** **<** **>** **<=** **=>**	**Igual a** **distinto a** **menor que** **mayor que** **menor o igual que** **mayor o igual que**
Lógicas de dos operandos	**and** **or** **nand** **nor** **xor** **xnor**	**And lógico** **or lógico** **nand lógico** **nor lógico** **or exclusiva** **xor negada**
Lógicas de un operando	**not**	**Complementación**
Encadenamiento	**&**	**Encadenamiento**

También se definen operaciones de desplazamiento y rotación para *arrays* de valores **BOOLEAN** y **BIT**. No aparecen en la tabla por su escasa utilidad. Cada tipo predefinido, excepto **SEVERITY_LEVEL**, tiene definida alguna de estas operaciones.

El lenguaje permite la definición de operadores por el usuario y la sobrecarga de nombres, es decir, permite que dos operadores compartan el mismo nombre si se diferencian en el tipo de datos con que operan.

9.- Tipos y Subtipos definidos por el usuario

El lenguaje permite al usuario la definición de **tipos** y **subtipos** de datos. La definición puede hacerse:

- restringiendo el rango de un tipo escalar o enumerado,
- definiendo un tipo enumerado y
- definiendo ***arrays*** o ***records***.

Los subtipos se definen a partir de un tipo mediante una restricción de rango. La diferencia fundamental entre los dos es que la comprobación de las reglas para los tipos se realiza en la compilación del código, mientras que para los subtipos se realiza en tiempo de ejecución (durante la simulación).

Los siguientes ejemplos, sacados del paquete **std_logic_1164** de la librería **IEEE**, muestran sentencias de definición de tipos de datos.

```
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
                   );
```

En este ejemplo se define, por enumeración, un tipo para el modelado de nodos lógicos. Contiene valores que representan estados metalógicos, como **'U'** (no inicializado), **'X'** y **'W'** (valor indeterminado) o **'-'** (valor indiferente).

El tipo **std_ulogic_vector**, se define como un *array* de valores del tipo **std_ulogic**:

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

La expresión **NATURAL RANGE <>**, sirve para indicar que la única restricción del rango del *array* es que sus límites sean valores del tipo **NATURAL**.

Por último, se define el subtipo **X01** a partir de una restricción de rango –los subtipos siempre se definen así– aplicada al tipo **std_ulogic**.

```
SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';
```

En este ejemplo aparece una palabra, **resolved** (es el nombre de una función), que indica que es un tipo que tiene asociada una función de resolución. Este concepto se explicará en el apartado siguiente.

10.- Tipos de Datos para el modelado de buses

En los circuitos digitales hay nodos lógicos cuyo nivel puede ser fijado por la salida (en colector o drenador abierto o con control de estado de alta

impedancia) de más de un dispositivo. Para modelar en VHDL este tipo de nodos es preciso utilizar señales cuyo tipo tenga asociada una función de resolución.

Ninguno de los tipos predefinidos del lenguaje es de esta clase –esta es la tercera característica que penaliza el uso del tipo **BIT**–, por lo que han de ser necesariamente definidos por el usuario a partir de otro predefinido o definido por el usuario. Un ejemplo de esto último se ha visto con el tipo **X01**. Para definir un tipo con función de resolución:

- tiene que existir o haber sido declarado previamente el tipo **básico** con el que se forma,
- tiene que haberse declarado una función de resolución y, por último,
- hay que declarar el tipo **resolved**.

Por ejemplo, la secuencia de declaraciones necesarias para la correcta definición del tipo **X01** es:

Definición del tipo básico:

```
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
                   );
```

Declaración de la función de resolución:

```
FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic;
```

Declaración del tipo **resolved**:

```
SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';
```

La palabra **resolved** que aparece en la última declaración es el nombre de la función de resolución. Es el que se suele utilizar siempre para este tipo de funciones por lo “legible” que resulta luego la declaración del tipo.

En el paquete **std_logic_1164** se definen dos tipos con función de resolución muy importantes, puesto que son los que se usan habitualmente en lugar de los predefinidos **BIT** y **BIT_VECTOR**:

```
SUBTYPE std_logic IS resolved std_ulogic;

TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
```

Estos tipos son los más frecuentemente utilizados en el modelado del hardware orientado a la síntesis lógica. Son tipos de datos creados para el modelado de nodos lógicos que:

- tienen definidos valores para la representación de niveles metalógicos,
- pueden modelar buses por ser tipos **resolved** y
- tienen definidas operaciones aritméticas en los paquetes **std_logic_signed**, **std_logic_unsigned** y **std_logic_arith** (en la versión de **Synopsys** de la librería **IEEE**) y en el paquete **numeric_std** (en la versión de **IEEE**).

Como ya se sabe, para poder utilizar estos tipos hay que obtener visibilidad sobre el paquete **std_logic_1164**, donde están declarados; además, si se desea que representen números binarios sin signo (binario natural), hay que utilizar el paquete **std_logic_unsigned**; si lo que se necesita es que modelen números con signo (en complemento a dos), entonces se debe usar el paquete **std_logic_signed**. En estos paquetes se definen:

- Operaciones aritméticas: suma, resta, producto y valor absoluto (ésta sólo existe en el **std_logic_signed**).
- Operaciones relacionales y de desplazamiento: igual, distinto, menor, mayor, etc.
- Funciones de conversión con el tipo **INTEGER**.

Si se desea mezclar objetos en binario natural y en complemento a dos, debe utilizarse el paquete **std_logic_arith**. En este paquete se definen los tipos **signed** (complemento a dos) y **unsigned** (binario natural):

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;

type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Para estos tipos se dispone de las mismas operaciones que para los otros, disponiendo además de operadores “mixtos”; por ejemplo, se puede sumar un valor **signed** con otro **unsigned**.

En valores de estos tipos, el bit más significativo es siempre el correspondiente al limite izquierdo en la declaración del *array*; es decir, si se declara, por ejemplo:

```
SIGNAL numero: std_logic_vector (3 DOWNTO 0);
```

numero(3) es el bit de mayor peso y **numero(0)** el de menor peso. En cambio, si se declara:

```
SIGNAL numero: std_logic_vector(0 TO 3);
```

numero(0) es el bit de mayor peso y **numero(3)** el de menor peso.

Para finalizar, decir solamente que en la versión normalizada por **IEEE**, el paquete **numeric_std** coincide en contenidos, básicamente, con el paquete **std_logic_arith**. Si se desea realizar descripciones compatibles entre ambas

versiones de la librería, lo adecuado es utilizar siempre los tipos **signed** y **unsigned**. En el apéndice A de este capítulo se muestran las declaraciones de los paquetes de esta librería.

11.- Atributos de los Tipos de Datos.

Algunos elementos del lenguaje tienen definidos **atributos**. Los **atributos** de los **tipos de datos** permiten identificar un valor a partir de una característica del mismo dentro de la declaración del tipo. La expresión con que se utilizan es:

nombre'identificador_de_atributo

Algunos de los atributos predefinidos para los tipos de datos son:

'Left: El valor correspondiente al límite izquierdo del tipo –el tipo debe ser escalar–; por ejemplo, el primero en un tipo enumerado:

BIT'Left es **'0'**,
std_logic'Left es **'U'**
POSITIVE'Left es **1**

El valor de este atributo es muy importante porque al declarar un objeto, si no se inicializa explícitamente, su valor inicial será el que tenga el atributo **Left** en su tipo:

```
SIGNAL x: std_logic_vector(3 DOWNTO 0); -- el valor inicial es "UUUU"
```

'Right: Es el valor correspondiente al límite derecho. Ejemplos:

BIT'Right es **'1'**,
std_logic'Right es **'-'**

'Low y **'High**: Valores correspondientes a los límites inferior y superior del tipo. Por ejemplo:

```
TYPE cero_a_siete IS INTEGER RANGE 0 TO 7;
TYPE siete_a_cero IS INTEGER RANGE 7 DOWNTO 0;
```

cero_a_siete'Low es **0**
cero_a_siete'High es **7**
siete_a_cero'Low es **0**
siete_a_cero'High es **7**

observe, para diferenciarlos de los atributos **'Left** y **'Right**, que:

cero_a_siete'Left es **0**
siete_a_cero'Left es **7**

12.- Declaraciones de Paquete

La sintaxis de la **Declaración de Paquete** es:

PACKAGE nombre_de_paquete IS

Zona de Declaración

END PACKAGE;

En la zona de declaración pueden declararse tipos y subtipos de datos, funciones y procedimientos y otros elementos del lenguaje.

13.- Ejercicio II.2

1. Cree un nuevo **Workspace**, en su directorio de ejemplos, y llámelo **ejer_paq**.

2. Edite un fichero, con el código que se muestra a continuación, y sálvelo con el nombre **mi_paq.vhd**.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE mi_paq IS

SUBTYPE ENTERO_3_8 IS INTEGER range 3 TO 8;

TYPE tipo_byte IS ARRAY (7 downto 0) OF std_logic;

END PACKAGE;
```

3. Añádalo al **Workspace**.

4. Repita la operación 2 con el siguiente código, que corresponde a un ***test-bench*** "vacío"; llame al fichero **test_vacio.vhd**.

```
USE WORK.mi_pack.ALL;

ENTITY test_vacio IS
END ENTITY;

ARCHITECTURE test OF test_vacio IS
  SIGNAL s_int: INTEGER := 3;
  SIGNAL s_right, s_left: INTEGER;
  SIGNAL s_int38, s_low, s_high: ENTERO_3_8;
  SIGNAL byte1: tipo_byte;

BEGIN
byte1(7 DOWNTO 0) <= "ZZZZZZZZ" AFTER 25 NS,
                     X"55" AFTER 50 NS,
```

```
                    "ZZZZZZZZ" AFTER 75 NS,
                    X"FF" AFTER 100 NS;

    byte1(7 DOWNTO 0) <= X"00" AFTER 25 NS,
                    X"FF" AFTER 50 NS,
                    "LLLLLLLL" AFTER 75 NS;

    s_int <= s_int + 1  AFTER 50 NS;
    s_int38 <= s_int;

    s_right <= INTEGER'RIGHT;
    s_left <= INTEGER'LEFT;

    s_low <= entero_3_8'LOW;
    s_high <= entero_3_8'HIGH;

    END TEST;
```

El objeto que se persigue con este ***test-bench*** es verificar algunas de las propiedades de los tipos de dato explicadas en este capítulo.

5. Antes de ordenar la compilación de las unidades anteriores, configure el **Workspace** (**Settings...**), activando las opciones **Debug** y **Trace On**, e indicando el nuevo nombre del ***test-bench*** que se desea simular.

6. Compile los ficheros, ordene la simulación y abra una ventana de visualización de formas de onda.

7. Ordene que se simule durante **200 ns**.

8. Seleccione la visualización de la señal **byte1** en la ventana de visualización de formas de onda.

   El aspecto de la señal se muestra en la figura e1.

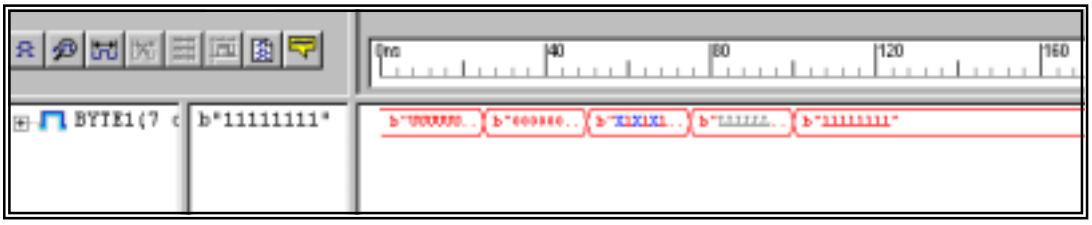


Figura e1

Esta señal es de tipo **resolved**, puesto que el tipo **tipo_byte** es un array de valores **std_logic**. En el ***test_bench*** hay dos sentencias que le asignan valor (la señal tiene dos ***drivers***) y es la función de resolución asociada al tipo **std_logic** la que calcula el valor que se asigna a la señal en cada instante. El valor inicial es **"UUUUUUUU"** puesto que no se indica otra cosa en la declaración de la señal y **'U'** es el valor correspondiente al atributo **LEFT** del tipo **std_logic**. Las sentencias que "manejan" el valor de la señal proyectan para el tiempo **25 ns**

los valores **'Z'** y **'0'** por lo que la función de resolución determina que el valor efectivo sea **'0'**; para el tiempo **50 ns** una proyecta **X"55"** y la otra **X"FF"**, por lo que hay bits en las que ambas fijan un **'1'** y otros en los que hay conflicto, para estos últimos la función calcula el valor **'X'**; para el tiempo **75 ns** una de las sentencias fija el valor **'L'**, que representa un **'0'** "débil" (**'H'** es un **'1'** débil y **'W'** una indeterminación débil) –cuando un valor débil entra en conflicto con uno fuerte, se asigna el valor del fuerte (el valor más débil es **'Z'**)– y la otra **'Z'**, por lo que se asigna el valor **'L**'. Por último, al cambiar la **'Z'** por **'1'** este valor se impone.

9. Añada ahora las señales **s_left** y **s_right**.

El valor de estas señales es el límite izquierdo y derecho del tipo **INTEGER**. Puesto que el simulador que está utilizando representa los enteros con 32 bits, el límite derecho es $\mathbf{2^{32}-1}$ y el izquierdo $\mathbf{-2^{32}}$.

10. Visualice las señales **s_low** y **s_high**.

Los valores que toman corresponden al menor y mayor, respectivamente, del tipo **entero_3_8**.

11. Visualice la señal **s_int38**.

Esta señal es del subtipo **entero_3_8**, compuesto por los valores enteros entre el 3 y el 8. A esta señal se le está asignando el valor de otra de tipo **INTEGER**, que se incrementa cada **50 ns**. La asignación es válida mientras esta última tenga un valor que pertenezca al subtipo **entero_3_8**. De acuerdo con la norma del lenguaje, la comprobación de asignaciones a objetos definidos sobre un subtipo se hace durante las simulaciones –ya que no puede saberse, antes de aplicar unos estímulos concretos, si van a ser correctas o no– y debe reportarse un error si se pretende asignar un valor fuera de rango.

12. Ordene que la simulación continúe durante otros **200 ns**.

El resultado obtenido es inesperado (figura e2). Se asignan valores a la señal fuera de rango y la simulación continúa sin problemas. Esto es un error de implementación de la norma; es normal, casi todas las herramientas tienen algún fallo de este estilo.

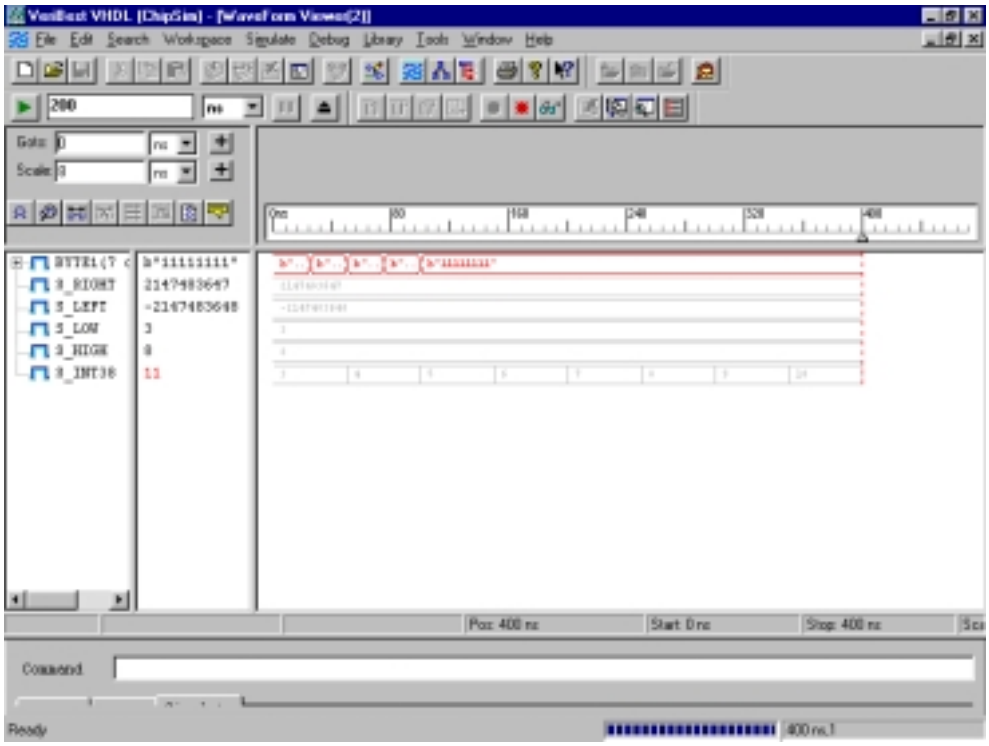


Figura e2

13. Finalice la simulación y cierre la herramienta

14.- Apéndice: Declaraciones de Paquete de las Librerías IEEE y STD

1. Declaración del Paquete STD_LOGIC_1164.

```
PACKAGE std_logic_1164 IS

    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    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
                       );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with the resolution function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;


    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

    ------------------------------------------------------------------
    -- common subtypes
    ------------------------------------------------------------------
    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')

    ------------------------------------------------------------------
    -- overloaded logical operators
    ------------------------------------------------------------------

    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

    ------------------------------------------------------------------
    -- vectorized overloaded logical operators
    ------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
```

```
    FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "xnor" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "xnor" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;

    ----------------------------------------------------------------------
    -- conversion functions
    ----------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic;       xmap : BIT := '0') RETURN BIT;
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;

    FUNCTION To_StdULogic       ( b : BIT             ) RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR      ) RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR      ) RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector  ) RETURN std_ulogic_vector;

    ----------------------------------------------------------------------
    -- strength strippers and type convertors
    ----------------------------------------------------------------------

    FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( b : BIT               ) RETURN  X01;

    FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;

    FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;

    ----------------------------------------------------------------------
    -- edge detection
    ----------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

    ----------------------------------------------------------------------
    -- object contains an unknown
    ----------------------------------------------------------------------
    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;

END std_logic_1164;
```

2. Declaración del Paquete STD_LOGIC_UNSIGNED.

```
package STD_LOGIC_UNSIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end STD_LOGIC_UNSIGNED;
```

3. Declaración del Paquete STD_LOGIC_SIGNED.

```
package STD_LOGIC_SIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "ABS"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;


    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end STD_LOGIC_SIGNED;
```

4. Declaración del Paquete STD_LOGIC_ARITH.

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

package std_logic_arith is

   type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
   type SIGNED is array (NATURAL range <>) of STD_LOGIC;
   subtype SMALL_INT is INTEGER range 0 to 1;

   function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
   function "+"(L: SIGNED; R: SIGNED) return SIGNED;
   function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
   function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
   function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
   function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
   function "+"(L: SIGNED; R: INTEGER) return SIGNED;
   function "+"(L: INTEGER; R: SIGNED) return SIGNED;
   function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
   function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
   function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
   function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

   function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
   function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
   function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
   function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
   function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

   function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
   function "-"(L: SIGNED; R: SIGNED) return SIGNED;
   function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
   function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
   function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
   function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
   function "-"(L: SIGNED; R: INTEGER) return SIGNED;
   function "-"(L: INTEGER; R: SIGNED) return SIGNED;
   function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
   function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
   function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
   function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

   function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
   function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
   function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
   function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
   function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

   function "+"(L: UNSIGNED) return UNSIGNED;
   function "+"(L: SIGNED) return SIGNED;
   function "-"(L: SIGNED) return SIGNED;
   function "ABS"(L: SIGNED) return SIGNED;

   function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
   function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
   function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
   function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;

   function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
   function "*"(L: SIGNED; R: SIGNED) return SIGNED;
```

```
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;

function "*"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;

function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;

function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;

function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;

function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;

function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;

function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;

function CONV_INTEGER(ARG: INTEGER) return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED) return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return UNSIGNED;
```

```
    function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return SIGNED;

    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                                                        return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER)
                                                        return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER)
                                                        return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER)
                                                        return STD_LOGIC_VECTOR;
    -- zero extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

    -- sign extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end Std_logic_arith;
```

5. Declaración de los Paquetes de la Librería STD.

```
PACKAGE STANDARD IS

        -- predefined enumeration types:

        TYPE BOOLEAN IS (FALSE,TRUE);

        TYPE BIT IS ('0', '1');

        TYPE CHARACTER IS (
                NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
                BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
                DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
                CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
                ' ', '!', '"', '#', '$', '%', '&', ''',
                '(', ')', '*', '+', ',', '-', '.', '/',
                '0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', ':', ';', '<', '=', '>', '?',
                '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                'X', 'Y', 'Z', '[', '\', ']', '^', '_',
                '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
                'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
                'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                'x', 'y', 'z', '{', '|', '}', '~', DEL,
                C128, C129, C130, C131, C132, C133, C134, C135,
                C136, C137, C138, C139, C140, C141, C142, C143,
                C144, C145, C146, C147, C148, C149, C150, C151,
                C152, C153, C154, C155, C156, C157, C158, C159,
                ' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
                '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
                '°', '±', '²', '³', '´', 'µ', '¶', '·',
                '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
                'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
                'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
                'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
                'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
                'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
                'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
                'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
                'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');

        TYPE SEVERITY_LEVEL IS (NOTE, WARNING, ERROR, FAILURE);

        -- predefined numeric types:

        TYPE INTEGER IS RANGE -2147483648 TO 2147483647;

        TYPE REAL IS RANGE -1.0E38 TO 1.0E38;

  -- predefined type TIME:

        TYPE TIME IS RANGE - 2**62 -2**62 TO 2**62 - 1 + 2**62
                UNITS
                        FS;
                        PS = 1000 FS;
                        NS = 1000 PS;
                        US = 1000 NS;
                        MS = 1000 US;
                        SEC = 1000 MS;
                        MIN = 60 SEC;
                        HR = 60 MIN;
                END UNITS;

        SUBTYPE DELAY_LENGTH IS TIME RANGE 0 FS TO TIME'HIGH;

        -- function that returns the current simulation time:

        FUNCTION NOW RETURN DELAY_LENGTH;

        -- predefined numeric subtypes:

        SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;
```

```
        SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;

        -- predefined array types:

        TYPE STRING IS ARRAY (POSITIVE RANGE <>) OF CHARACTER;

        TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;

        TYPE FILE_OPEN_KIND IS (READ_MODE, WRITE_MODE, APPEND_MODE);

        TYPE FILE_OPEN_STATUS IS (OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR);

        ATTRIBUTE FOREIGN: STRING;

END STANDARD;


package TEXTIO is

   -- Types definitions for Text I/O

   type LINE is access string;

   type TEXT is file of string;

   type SIDE is (right, left);

   subtype WIDTH is natural;

   -- Standard Text Files

   file input : TEXT open READ_MODE is "STD_INPUT";
   file output : TEXT open WRITE_MODE          is "STD_OUTPUT";

   -- Input Routines for Standard Types

   procedure READLINE(file f: TEXT; L: inout LINE);

   procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out bit);

   procedure READ(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out bit_vector);

   procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out BOOLEAN);

   procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out character);

   procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out integer);

   procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out real);

   procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out string);

   procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
   procedure READ(L:inout LINE; VALUE: out time);

   -- Output Routines for Standard Types

   procedure WRITELINE(file f : TEXT; L : inout LINE);

   procedure WRITE(L : inout LINE; VALUE : in bit;
                   JUSTIFIED: in SIDE := right;
                   FIELD: in WIDTH := 0);

   procedure WRITE(L : inout LINE; VALUE : in bit_vector;
                   JUSTIFIED: in SIDE := right;
                   FIELD: in WIDTH := 0);
```

```
    procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in character;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in integer;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in real;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0;
                    DIGITS: in NATURAL := 0);

    procedure WRITE(L : inout LINE; VALUE : in string;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0);

    procedure WRITE(L : inout LINE; VALUE : in time;
                    JUSTIFIED: in SIDE := right;
                    FIELD: in WIDTH := 0;
                    UNIT: in TIME := ns);

end TEXTIO;
```

DESCRIPCIÓN DEL FUNCIONAMIENTO

0.- Resumen del Capítulo

Conceptos Teóricos:

- *Estilos de descripción*
- *Procesos*
- *Descripciones Estructurales*
- *Modelo de Simulación*
- *Sentencias secuenciales y Sentencias concurrentes.*

Prácticas sobre el simulador VeriBest:

- *Simulación de modelos con varios procesos.*
- *Simulación de modelos estructurales.*

Apéndices:

- *Modelo de Simulación de una jerarquía de diseño.*
- *Sentencias secuenciales y concurrentes.*

En este capítulo se explican los distintos estilos con que se puede modelar un circuito digital y se presentan los procesos y componentes VHDL. Se muestran los elementos imprescindibles para empezar a realizar modelos del hardware y se enseñan las claves necesarias para comprender cómo la simulación de un modelo VHDL emula el funcionamiento del hardware real.

Los ejercicios que se realizan persiguen dos objetivos: por un lado, verificar los conceptos relativos al modelo de simulación del lenguaje, por otro, comprender la interacción de los procesos para representar el funcionamiento de circuitos complejos.

1.- Estilos de Descripción

Los Cuerpos de Arquitectura sirven para describir el funcionamiento de los dispositivos hardware. La descripción puede hacerse, fundamentalmente, de dos formas:

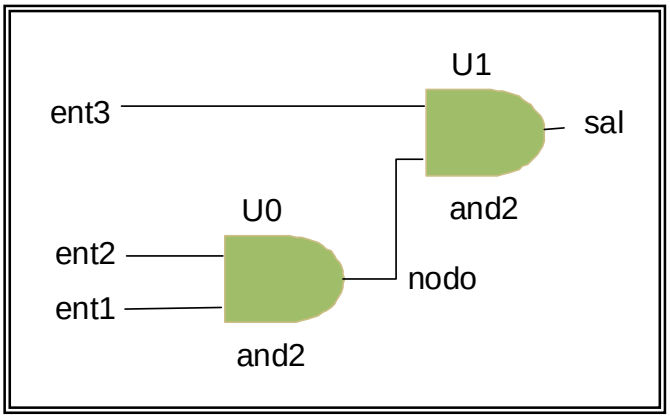
1. Mediante construcciones que representan un algoritmo de procesamiento.

2. Mediante la interconexión de otros dispositivos.

Por ejemplo, siguiendo la primera metodología, para describir el funcionamiento de una puerta **and** de tres entradas, puede utilizarse una sentencia como:

sal <= ent1 AND ent2 AND ent3;

o, de acuerdo con la segunda opción, también podría utilizarse el circuito de la figura, en el que se emplean puertas **and** de dos entradas.



U0: ENTITY WORK.and2(RTL) PORT MAP(ent1, ent2, nodo);
U1: ENTITY WORK.and2(RTL) PORT MAP(nodo, ent3, sal);

Además, cuando se opta por la descripción del algoritmo de procesamiento pueden emplearse distintos niveles de abstracción.

Los distintos enfoques con que puede abordarse el modelado del funcionamiento de los circuitos se denominan **estilos**. Este concepto no se trata en la norma del lenguaje; procede de su aplicación práctica.

Una descripción en la que el funcionamiento se describe mediante la interconexión de otros dispositivos se denomina **estructural** y es la de menor grado de abstracción; a continuación viene el estilo denominado **RTL** (**register transfer level**), en el que se utilizan construcciones simples (ecuaciones lógicas, operaciones aritméticas básicas), muy cercanas a la lógica descrita, de las que resulta fácil inferir la estructura hardware modelada –son las

descripciones más abstractas que “entienden” las herramientas de síntesis lógica automática–. El mayor grado de abstracción corresponde a las descripciones de **comportamiento** (**behavioural**); en este estilo, el procesamiento realizado por el hardware se describe mediante algoritmos tan complejos como los que pueden realizarse con los lenguajes de programación de alto nivel; suele utilizarse para la evaluación e investigación de arquitecturas de sistemas digitales y, durante el ciclo de diseño para facilitar la simulación de circuitos en fase de desarrollo. En los últimos años han aparecido algunas herramientas de “síntesis arquitectural” que admiten este estilo, con fuertes restricciones, como código de entrada.

Los distintos niveles de abstracción pueden adaptarse a distintos propósitos y utilizarse en las sucesivas fases que se dan en el ciclo de diseño de los circuitos digitales. Por ejemplo, si se está investigando la validez de una arquitectura para la realización de un determinado algoritmo de procesamiento resulta sencillo y rápido modelar el hardware con un estilo de comportamiento. Si se decide diseñar el sistema con la ayuda de un sintetizador lógico, se debe traducir la descripción a un estilo, RTL, inteligible por la herramienta de síntesis, que tendrá condicionado el empleo de determinados elementos del lenguaje. Finalmente, el software que realiza la síntesis lógica puede generar una **netlist** –descripción estructural– en lenguaje VHDL.

Para optimizar la eficacia en el uso del lenguaje debe emplearse siempre el mayor nivel de abstracción posible, ya que el número de líneas que ocupa una descripción –y el tiempo que se tarda en realizarla–, así como las dificultades que presenta la modificación del modelo, crecen cuanto más “cerca” se está del hardware.

En la práctica, la mayor parte del código VHDL que se escribe está orientado a la síntesis lógica y las herramientas de síntesis tradicionales admiten como entrada únicamente descripciones RTL o estructurales (o una mezcla de ambas).

Las descripciones estructurales no tienen misterios: para realizarlas basta con conocer las normas del lenguaje que indican como conectar modelos de dispositivos en un Cuerpo de Arquitectura; la tarea de realizar una descripción estructural puede resultar tediosa pero es sencilla, además en muchos entornos de CAD es posible generar automáticamente este tipo de descripciones a partir de esquemas gráficos.

Las descripciones RTL para síntesis son más complicadas porque obligan a seguir un conjunto de reglas de construcción, dependientes en alguna medida de la herramienta de síntesis que se vaya a usar. A pesar de ello existe un conjunto de patrones de codificación (cómo y cuáles construcciones utilizar) que resultan válidos para la inmensa mayoría de las herramientas de síntesis y permiten esbozar unas reglas básicas de codificación. Las de comportamiento se realizan utilizando las mismas construcciones del lenguaje que las RTL, pero sin otras restricciones que no sean las sintácticas y semánticas del lenguaje.

Por último hay que decir que en la práctica los modelos VHDL combinan los diferentes estilos: basta con que el hardware sea moderadamente complejo para que existan elementos que describan aspectos estructurales y siempre habrá dispositivos básicos que no admitan ser descritos a través de la interconexión de otros.

2.- Descripciones RTL y de Comportamiento

Tal y como se adelantó en el primer capítulo, la sintaxis básica de los Cuerpos de Arquitectura es:

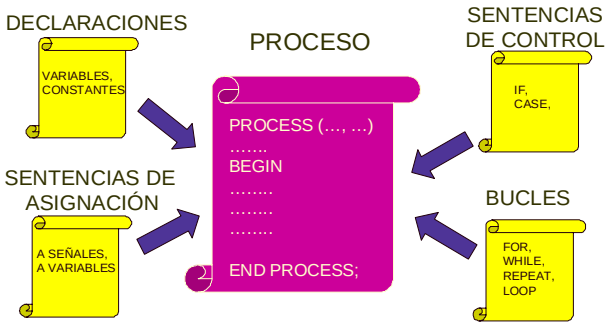
```
ARCHITECTURE {nombre_de_arquitectura} OF {nombre_de entidad} IS

        {zona de declaración}

        BEGIN

        {zona de descripción}

        END {nombre de arquitectura};
```

El nombre de la arquitectura suele coincidir con el del estilo que se emplea en la descripción, de ahí que frecuentemente se encuentren etiquetas como **estructural**, **RTL** o **behavioural**.

El aspecto de las zonas de declaración y descripción varía bastante según el estilo que se emplee. Si se trata de descripciones RTL o de comportamiento, en la zona de declaración es frecuente encontrar declaraciones de señales, constantes y, a veces, tipos de datos mientras que el elemento descriptivo básico es una construcción VHDL llamada **proceso**.

3.- Procesos

Un **proceso** es un bloque de código que representa un algoritmo de procesamiento. El algoritmo que se codifica dentro del proceso puede construirse con sentencias similares a las de los lenguajes de programación de alto nivel y sentencias de asignación de valores a variables y señales. La interpretación del algoritmo es similar a la que se realiza en el caso de los lenguajes de programación, con la salvedad del funcionamiento del mecanismo de asignación de valores a señales.



La sintaxis básica de los procesos es la siguiente:

```
PROCESS (lista de señales)
{zona de declaración}
BEGIN

{algoritmo de procesamiento}

END PROCESS;
```

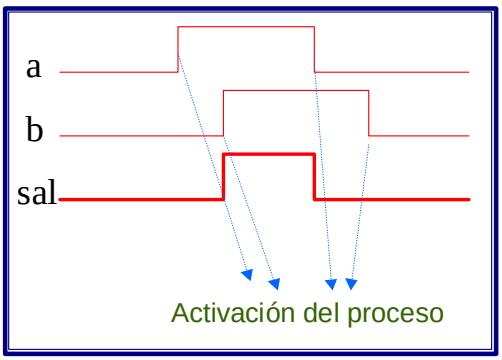
La lista de señales que aparece en la cabecera de la construcción es, aunque muy frecuentemente utilizada, opcional. Si no existe, dentro del conjunto de sentencias que definen el algoritmo de procesamiento deberá intercalarse obligatoriamente al menos una sentencia **WAIT** (una sentencia de suspensión de ejecución). El lenguaje prohibe que en un proceso pueda haber sentencias **WAIT** y lista de señales al mismo tiempo.

Esto es debido a que los procesos son **sensibles** a eventos –cambios de valor– de las señales que aparecen en la lista o, alternativamente, de las que aparezcan en las sentencias **WAIT**. Que un proceso sea sensible a eventos de un conjunto de señales quiere decir que las sentencias del proceso se ejecutan cuando estos eventos se producen.

Los eventos de las señales ocurren en coordenadas de tiempo dentro del **tiempo de simulación** y los procesos se ejecutan, por tanto, en esos instantes determinados, un número de veces que dependerá de los eventos que sucedan.

Por ejemplo, el siguiente código corresponde a un proceso que modela una puerta **and** de dos entradas:

```
PROCESS(a, b)
      BEGIN
      IF a = '1' AND b = '1' THEN
        sal <= '1';
      ELSE
        sal <= '0';
      END IF;
END PROCESS;
```



El proceso es sensible a las señales **a** y **b**. Esto quiere decir que al simular un modelo que contenga este proceso y someterlo a estímulos tal y como se muestra en la figura de la derecha, el proceso se ejecutará cuatro veces, en cada una de ellas la sentencia **IF** se evaluará y, teniendo en cuenta los valores de la condición, se **proyectará** una asignación de valor **'1'** o **'0'** para la salida. Cada activación de un proceso con lista de señales supone la

ejecución secuencial de las sentencias que lo componen. Cuando se llega a la última sentencia el proceso se **suspende** hasta que se produzca un nuevo evento en alguna de las señales de la lista.

Volviendo a la sintaxis de los procesos; en la zona de declaración suelen encontrarse declaraciones de variables y constantes. El ámbito de los objetos declarados en ella se limita a la zona de descripción del proceso, es decir, los objetos ahí declarados no serán visibles fuera del proceso.

En la zona de descripción se combinan sentencias de ejecución secuencial, como las de los lenguajes de programación, para construir algoritmos que representan la operación lógica modelada por el proceso. Normalmente –esto que sigue no es cierto en los procesos **pasivos**– el algoritmo de procesamiento dará lugar a la proyección de asignaciones de valor a señal.

El sentido de este mecanismo dentro del modelado del hardware se pone claramente de manifiesto, por ejemplo, en el caso de un proceso que describa un circuito combinacional: el proceso que modele el circuito debe ser sensible a señales que representan sus entradas y el algoritmo codificado en el proceso debe calcular las combinaciones de salida y asignar los valores apropiados a las señales que modelan las salidas. De acuerdo con esto, el modelo mantiene estables los valores de las señales de salida mientras las entradas no cambien, como sucede en un circuito real, y las modifica, cuando resulta necesario, si hay eventos en las entradas.

Recapitulando ideas:

1. El proceso se ejecuta cuando hay algún cambio de valor en alguna de las señales a las que es sensible.

2. Como consecuencia de la ejecución del proceso se proyectan asignaciones de valor sobre señales.

4.- Sentencias WAIT

La descripción de las sentencias secuenciales que pueden incluirse en los procesos se incluye en el apéndice B de este capítulo. Las únicas que se van a tratar aquí, por su peculiaridad, son las sentencias **WAIT**.

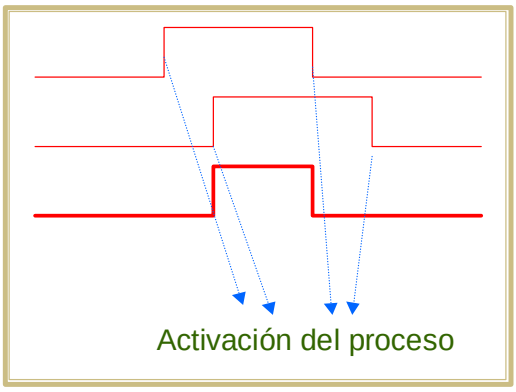
Las sentencias **WAIT** permiten suspender la ejecución de los procesos. En un proceso que no tenga lista de señales en la cabecera puede haber cualquier número de sentencias **WAIT**. Existen distintos formatos para esta sentencia:

El primer formato obliga a que la ejecución quede suspendida hasta que se produzca un evento en alguna de las señales que se indican en la propia sentencia. Su fórmula es:

WAIT ON lista_de_señales;

En la figura se presenta un proceso que modela una puerta **and** de dos entradas haciendo uso de este formato de sentencia **WAIT**. Es casi equivalente al anterior, que utilizaba una lista de señales –lo sería del todo si la sentencia **WAIT** fuera la última del proceso en vez de la primera; el porqué se explica al final de este apartado–; de hecho, la lista de sensibilidad es una forma sintáctica resumida de otro proceso con el mismo algoritmo de procesamiento y cuya última sentencia fuera una **WAIT ON** seguida de la lista de señales.

```
PROCESS
      BEGIN
      WAIT ON a, b;
      IF a = '1' AND b = '1' THEN
        sal <= '1';
      ELSE
        sal <= '0';
      END IF;
END PROCESS;
```



El proceso se activaría cada vez que hubiera un evento en **a** o **b** y discurriría secuencialmente hasta que volviera a alcanzar la sentencia **WAIT**, donde se quedaría esperando un nuevo evento en las mismas señales. En un caso más general el proceso discurre desde una sentencia **WAIT** hasta la siguiente, donde se suspende en espera de alguna condición.

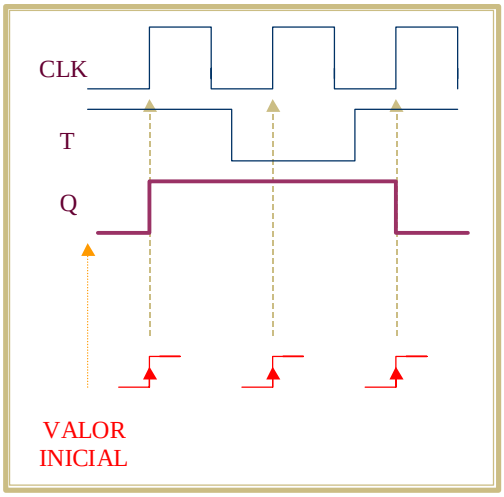
La siguiente fórmula corresponde a la sentencia de suspensión condicional.

WAIT UNTIL condición_lógica;

Esta sentencia suspende la ejecución del proceso hasta que se verifique la condición lógica indicada. El proceso que la utilice será sensible a eventos de las señales que formen parte de la condición. En la figura se muestra un proceso que la utiliza.

El proceso modela el funcionamiento de un ***flip-flop*** **T**. En la sentencia **WAIT** se utiliza la señal **clk**; en concreto se indica que la condición de activación del proceso es que haya un evento en la señal **clk** y que el valor de **clk** sea uno. Sobre esta condición hay que apuntar un par de detalles interesantes.

```
ENTITY tff IS
PORT(clk: IN BIT;
        t: IN BIT;
        q: OUT BIT);
END reloj;

ARCHITECTURE rtl OF fft IS
BEGIN
PROCESS
BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  IF t = '1' THEN q <= NOT q;
  END IF,
END PROCESS;
END rtl;
```



clk'EVENT toma el valor TRUE cuando hay un evento en clk

1. Se utiliza el atributo **'EVENT**. Se trata de un atributo de señal, que se evalúa a un valor booleano **TRUE** en los instantes del tiempo de simulación en que hay un evento en la señal a la que se aplica; en otro caso es **FALSE**. Los atributos de señal que se usan más frecuentemente en las descripciones se presentan en el apéndice B de este capítulo.

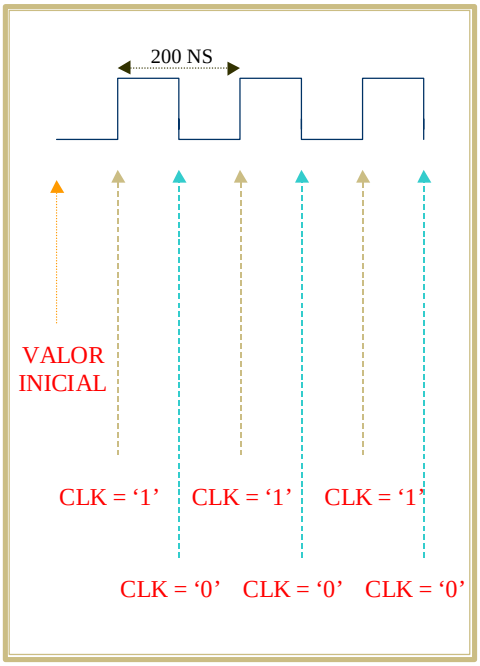
2. La comprobación de que hay un evento en la señal **clk** es redundante. La propia sentencia es sensible a los eventos en la señal **clk**, es decir, la condición lógica sólo se evalúa en los instantes del tiempo de simulación en que hay eventos en **clk**. Se incluye en la condición porque lo que ésta representa es la ocurrencia de un flanco de subida en **clk**; en definitiva, el algoritmo codificado en el proceso describe el funcionamiento de un ***flip-flop*** **T** disparado por flanco positivo. Cuando hay un flanco positivo, si la entrada **T** está a **'1'** se proyecta la inversión de la salida **Q**, si vale **'0'**, no se hace nada (**Q** no cambia).

Otro formato de la sentencia es:

WAIT FOR retardo;

El **retardo** es un valor o un objeto de tipo **TIME**; esta fórmula suele utilizarse únicamente en ***test-benches*** y descripciones de comportamiento. Es útil para modelar relojes, como en el siguiente ejemplo.

```
ENTITY reloj IS
PORT(clk: BUFFER BIT);
END reloj;
ARCHITECTURE simple OF reloj IS
BEGIN
PROCESS
BEGIN
    WAIT FOR 100 ns;
    clk <= NOT clk;
END PROCESS;
END simple;
```



La sentencia suspende el procesamiento del proceso hasta que transcurra el tiempo especificado.

El último formato corresponde a la suspensión definitiva:

WAIT;

Si se incluye esta sentencia en un proceso y se llega a ejecutar, el proceso no volverá a activarse bajo ningún concepto.

Para terminar con el tema, un par de cuestiones de interés:

1. En un modelo, como el del reloj del último ejemplo, ¿cuándo se ejecuta por primera vez la sentencia **WAIT**? La respuesta la da el modelo de simulación del lenguaje: en el tiempo de simulación **0**, porque se indica que en ese instante inicial un simulador VHDL debe dar a todos los objetos su valor inicial y ejecutar todos los procesos hasta que se suspendan, cuando alcancen una sentencia **WAIT** o, si tienen lista de señales, cuando se llegue a la última sentencia del proceso –por este motivo no es equivalente un proceso con lista de señales a otro semejante que contenga una sentencia **WAIT ON** al principio del proceso, y sí lo es si la tiene al final; en la inicialización el primero se suspende en la sentencia **WAIT** antes de ejecutar el algoritmo, mientras que el segundo ejecuta las sentencias que contiene y luego se suspende–.

2. ¿Qué ocurre si un proceso sin lista de señales no incluye una sentencia **WAIT**? Esto podría dar lugar a un error. En algunos simuladores se ha comprobado que se admite y, en consecuencia, ocurre que la simulación no pasa del instante **0**, porque en el procedimiento de inicialización descrito se ejecutan indefinida y cíclicamente las sentencias del proceso.

5.- Modelado del paralelismo hardware

Los dispositivos digitales que forman parte de un circuito funcionan en paralelo. En cada instante de tiempo cualquiera de ellos mantiene una actividad. Habrá intervalos de tiempo en los que sus salidas, o el estado interno, permanecen estables, bien porque no cambian las señales de entrada, o bien porque los cambios que se producen no tienen repercusión en las señales de salida, e instantes en los que sí se producen cambios. Conectando dispositivos se consigue coordinar las operaciones para conseguir una determinada funcionalidad (la de un dispositivo “complejo” o un circuito) a partir de la independencia funcional de cada uno.

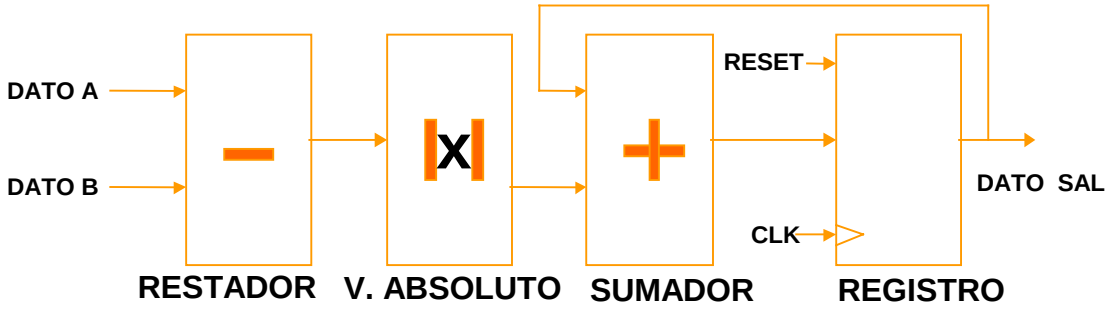
Los procesos VHDL sirven para modelar el funcionamiento de dispositivos. Para poder emular el funcionamiento paralelo del hardware es preciso que su ejecución sea concurrente –paralela–. Un proceso que no se ejecuta en un determinado periodo de tiempo modela el funcionamiento de un dispositivo durante los intervalos de tiempo en que sus entradas no cambian; un proceso que se ejecuta en un instante del tiempo de simulación –como ya se sabe, cuando hay un evento en alguna de las señales a las que es sensible– representa la operación del hardware que puede dar lugar, o no, a conmutaciones en los niveles lógicos de salida o en el estado interno del dispositivo cuando cambian los niveles lógicos de entrada. Los procesos, en definitiva, son los elementos que facilitan el modelado del comportamiento de dispositivos.

Conectando procesos mediante señales se puede modelar circuitos. La conexión puede realizarse dentro de un Cuerpo de Arquitectura, puesto que en éste puede haber cualquier número de procesos y, desde luego, pueden declararse señales que modelen nodos lógicos de interconexión. La interacción del hardware, que es debida a que salidas de un determinado dispositivo están conectadas a entradas de otro, se reproduce en el modelo haciendo que un proceso sea sensible a señales a las que asigna valor otro.

En el ejercicio que se va a realizar a continuación se simulará el modelo de un circuito descrito mediante cuatro procesos. Se utilizarán también varias de las construcciones explicadas en este capítulo. Además, se realiza una ejecución paso a paso de parte de la simulación para observar la interacción de los procesos; una lectura del Apéndice A de este capítulo puede facilitar la comprensión del ejercicio.

6.- Ejercicio III.1

En el siguiente ejercicio se va a realizar y simular el modelo de un circuito que acumula el valor absoluto de la diferencia de dos números, codificados en binario natural, de 4 bits. La arquitectura hardware descrita se muestra en la figura:



El modelado se va a realizar con cuatro procesos, uno por bloque funcional, conectados mediante señales declaradas en el Cuerpo de Arquitectura.

Las señales internas que se declaran son:

```
SIGNAL nodo_int1: SIGNED(4 DOWNTO 0);
SIGNAL nodo_int2: UNSIGNED(3 DOWNTO 0);
SIGNAL nodo_int3: UNSIGNED(7 DOWNTO 0);
```

El código del proceso correspondiente al **restador** es el siguiente:

```
proc_diff:
  PROCESS
   VARIABLE num1, num2: SIGNED(4 DOWNTO 0);
   BEGIN
num1 := CONV_SIGNED(dato_a, 5);
num2 := CONV_SIGNED(dato_b, 5);
nodo_int1 <= num1 - num2 AFTER 3 NS;
WAIT ON dato_a, dato_b;

  END PROCESS;
```

En la cabecera del proceso se ha incluido una etiqueta, que es opcional, para identificarlo. Puede servir como ayuda para distinguirlo en utilidades de las herramientas de simulación y síntesis. Se utilizan también funciones de conversión entre tipos para obtener en la salida (codificada en complemento a dos, **nodo_int1** de tipo **signed**), la diferencia entre los dos valores de entrada (codificados en binario natural, **unsigned**). En estas funciones se pasan como parámetros el valor a convertir y el número de bits, **5**, que se desea que tenga el valor convertido. Como puede observarse, el proceso es sensible a todas las entradas del bloque. Esto es imprescindible para modelar correctamente cualquier circuito combinacional; si no fuera así y se produjera un cambio en

algún valor de entrada que afecta a la salida del bloque, el proceso no se ejecutaría y no podría, por tanto, volver a calcularse el valor de la salida.

La “salida” de este bloque, **nodo_int1**, es la entrada al bloque que calcula el valor absoluto de este número. Este valor absoluto se almacena en la señal **nodo_int2**, **unsigned** de cuatro bits. El código del proceso que realiza esta operación es:

```
proc_calc_abs:
  PROCESS(nodo_int1)
   BEGIN
     nodo_int2 <= CONV_UNSIGNED(ABS(nodo_int1), 4) after 2 ns;

   END PROCESS;
```

El proceso es sensible a **nodo_int1**, señal a la que asigna valor el proceso del circuito **restador**. Para calcular el valor absoluto se hace uso de la operación **ABS**, disponible para el tipo **signed**, y el resultado se convierte a un valor **unsigned** de cuatro bits para poder asignarlo a **nodo_int2**.

El bloque sumador se modela así:

```
proc_suma:
  PROCESS(nodo_int2, dato_sal)
   BEGIN
    nodo_int3 <= dato_sal + nodo_int2 after 3 ns;

  END PROCESS;
```

Suma dos valores **unsigned** de ocho y cuatro bits para calcular el valor a acumular. Los dos operandos son el puerto de salida (**BUFFER**) **dato_sal** y la señal interna **nodo_int2**, a la que asigna valor el proceso que calcula el valor absoluto de la diferencia de las entradas.

El cuarto proceso modela un registro de ocho bits con **reset** asíncrono:

```
proc_reg:
  PROCESS(reset, clk)
   BEGIN
IF reset = '1' THEN
  dato_sal <= (OTHERS => '0');

ELSIF clk'EVENT AND clk = '1' THEN
  dato_sal <= nodo_int3 AFTER 2 NS;

END IF;

  END PROCESS;
```

Sobre este código hay varios detalles que resulta interesante señalar:

1. El proceso modela un circuito síncrono con **reset** asíncrono. En un circuito síncrono las salidas sólo pueden cambiar cuando ocurra un flanco activo del

reloj o si se activa una señal asíncrona (en el caso de que la tenga); por tanto, un proceso que modele este tipo de circuitos sólo deberá activarse en eventos de la señal de reloj y de las señales asíncronas. Por ello el proceso es sensible únicamente a las señales **clk** y **reset** (puertos de entrada del circuito). Si no tuviera **reset**, o éste fuera síncrono, sólo sería sensible a la señal de reloj.

2. La detección del flanco se hace combinando una lista de sensibilidad con una sentencia **IF**. La prioridad del funcionamiento asíncrono sobre el síncrono se establece por la precedencia de la comprobación del estado de la señal de **reset** sobre la ocurrencia de un flanco activo. Por motivos similares a los aducidos cuando se explico la sentencia **WAIT UNTIL clk'EVENT and clk = '1'**, la inclusión de **clk'EVENT** en la detección del flanco es superflua. El código de un proceso equivalente utilizando una sentencia **WAIT** sería el siguiente:

```
proc_reg:
  PROCESS
   BEGIN
IF reset = '1' THEN
  dato_sal <= (OTHERS => '0');

ELSIF clk'EVENT AND clk = '1' THEN
  dato_sal <= nodo_int3 AFTER 2 NS;

END IF;
WAIT UNTIL (clk'EVENT AND clk = '1') or reset'EVENT;

  END PROCESS;
```

   Resulta interesante observar que aquí resulta imprescindible la condición **clk'EVENT** para que el modelo sea correcto, ya que si no fuera así y la señal de **reset** pasará de **'1'** a **'0'** con el reloj a **'1'**, el modelo, sin que se dé un flanco positivo, asignaría a la salida del registro el valor de **nodo_int3**. Cambiando ligeramente la sentencia **WAIT**, **clk'EVENT** volvería a ser redundante:

```
WAIT UNTIL (clk'EVENT AND clk = '1') or reset = '1';
```

De esto último debe obtenerse una conclusión importante: es conveniente utilizar fórmulas claras y probadas para el modelado (**clk'EVENT AND clk = '1'**, por ejemplo), aunque no sean perfectas o resulten rutinarias, para evitar problemas que muchas veces causan verdaderos quebraderos de cabeza.

El código completo que describe el circuito se muestra a continuación; como se ve, el nombre de la arquitectura es **rtl**. Puede ir haciéndose una idea del grado de abstracción de este estilo comparando el nivel descriptivo con el hardware que se modela.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.STD_LOGIC_ARITH.ALL;
```

```
ENTITY acum4b IS
PORT(
        reset, clk: IN STD_LOGIC;
        dato_a, dato_b: IN UNSIGNED(3 DOWNTO 0);
        dato_sal: BUFFER UNSIGNED(7 DOWNTO 0)
   );
END ENTITY;

ARCHITECTURE RTL OF ACUM4B IS
  SIGNAL nodo_int1: SIGNED(4 DOWNTO 0);
  SIGNAL nodo_int2: UNSIGNED(3 DOWNTO 0);
  SIGNAL nodo_int3: UNSIGNED(7 DOWNTO 0);
BEGIN

proc_diff:
  PROCESS
   VARIABLE num1, num2: SIGNED(4 DOWNTO 0);
   BEGIN
        num1 := CONV_SIGNED(dato_a, 5);
        num2 := CONV_SIGNED(dato_b, 5);
        nodo_int1 <= num1 - num2 AFTER 1 NS;
        WAIT ON dato_a, dato_b;

  END PROCESS;

proc_calc_abs:
  PROCESS(nodo_int1)
   BEGIN
        nodo_int2 <= CONV_UNSIGNED(ABS(nodo_int1), 4) AFTER 1 NS;

   END PROCESS;

proc_suma:
  PROCESS(nodo_int2, dato_sal)
   BEGIN
    nodo_int3 <= dato_sal + nodo_int2 AFTER 1 NS;

  END PROCESS;

proc_reg:
  PROCESS(reset, clk)
   BEGIN
        IF reset = '1' THEN
          dato_sal <= (OTHERS => '0') AFTER 1 NS;

        ELSIF clk'EVENT AND clk = '1' THEN
          dato_sal <= nodo_int3 AFTER 1 NS;

        END IF;
```

```
  END PROCESS;
END RTL;
```

1. Cree un nuevo **Workspace** en su directorio de trabajo, configurado para utilizar las librerías de **Synopsys**. Póngale el nombre que prefiera.

2. Edite el fichero anterior, sálvelo con el nombre **acum4b**.**vhd** y añádalo al espacio de trabajo.

   Para simular el modelo del circuito se va a utilizar este **test-bench**:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.STD_LOGIC_ARITH.ALL;

ENTITY test_acum4b IS
END ENTITY;

ARCHITECTURE test OF test_acum4b IS
 SIGNAL s_reset, s_clk: STD_LOGIC;
 SIGNAL s_dato_a, s_dato_b: UNSIGNED(3 DOWNTO 0);
 SIGNAL s_dato_sal: UNSIGNED(7 DOWNTO 0);

BEGIN

 PROCESS
  BEGIN
   s_clk <= '1';
   WAIT FOR 10 NS;
   s_clk <= '0';
   WAIT FOR 10 NS;

 END PROCESS;

 PROCESS
  VARIABLE I, J: INTEGER RANGE 0 TO 15;
  BEGIN
   s_dato_a <= (OTHERS => '0') AFTER 5 NS;
   s_dato_b <= (OTHERS => '0') AFTER 5 NS;

   WAIT UNTIL S_CLK'EVENT AND S_CLK = '0';
   s_reset <= '1' AFTER 5 NS;

   WAIT FOR 60 NS;
   s_reset <= '0' AFTER 5 NS;

   WAIT UNTIL s_clk'EVENT AND s_clk = '0';
   FOR I IN 0 TO 15 LOOP
    s_dato_a <= CONV_UNSIGNED(I, 4) AFTER 1 NS;

    FOR J IN 0 TO 15 LOOP
     s_dato_b <= CONV_UNSIGNED(J, 4) AFTER 1 NS;
     WAIT UNTIL s_clk'EVENT AND s_clk = '0';

    END LOOP;

   END LOOP;

   WAIT;

 END PROCESS;

 DUT: ENTITY WORK.acum4b(rtl)
     PORT MAP(reset => s_reset, clk => s_clk,
```

```
                dato_a => s_dato_a, dato_b => s_dato_b,
                dato_sal => s_dato_sal);

        END test;
```

Para manejar los estímulos se utilizan dos procesos: uno para generar la señal de reloj, similar al visto anteriormente en un ejemplo de este capítulo, y otro para dar valores a los dos buses de entrada. En el código de este último, se utilizan sentencias **WAIT** para temporizar la ocurrencia de eventos y bucles **FOR** anidados para barrer todas las combinaciones posibles de valores de entrada con un código compacto. La última sentencia **WAIT** evita que las sentencias del proceso se ejecuten cíclicamente de manera indefinida.

3. Edite el fichero anterior, sálvelo con el nombre **test_acum**.**vhd** y añádalo al espacio de trabajo.

4. Compile los ficheros tras configurar el entorno con las opciones **debug**, **Trace on** e indicar las unidades del ***test-bench*** que se desea simular.

5. Arranque la simulación, abra una ventana de presentación de formas de onda y visualice todos los estímulos del ***test-bench*** (figura e1).

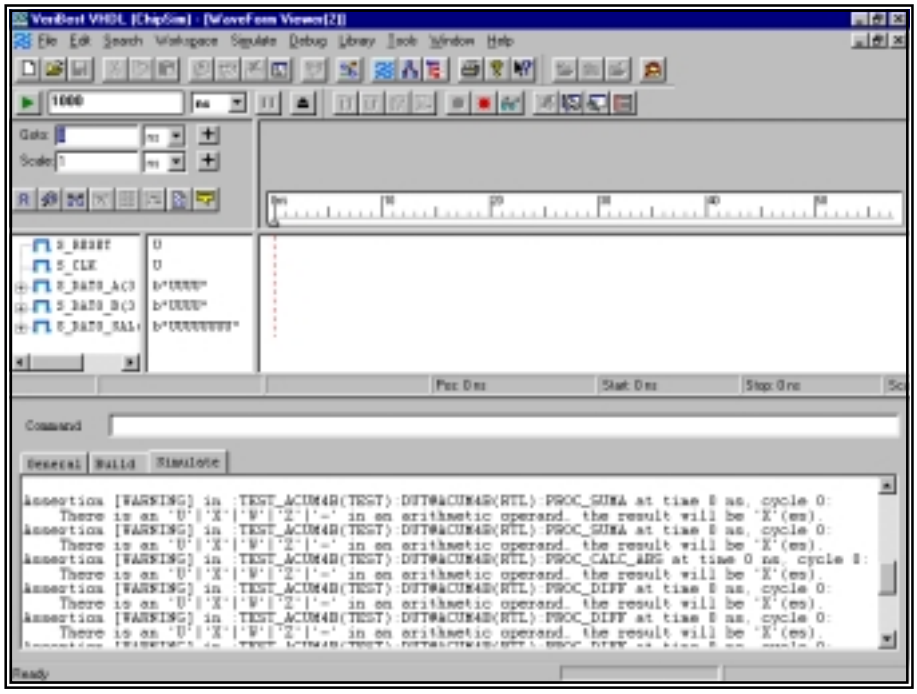


Figura e1

Si observa el contenido de la ventana de información del desarrollo de las simulaciones, distinguirá una serie de mensajes que comienzan con la fórmula Assertion [WARNING]... Aparecen porque cuando usted arranca el simulador –lo que sigue es específico de esta herramienta– se ejecuta la fase de inicialización de la simulación en el tiempo de simulación 0. En este proceso se aplican las reglas del lenguaje para la inicialización de objetos y se ejecutan todos los procesos del modelo que se simula hasta que se suspende su ejecución (por que se alcanza una sentencia WAIT o se llega

a la última línea del proceso en los que tienen lista de señales). Puesto que no se ha inicializado ningún estímulo, los procesos se ejecutan con valores por defecto ('U' puesto que se utilizan tipos derivados del **std_ulogic**) para las señales. Entre las instrucciones que se ejecutan hay operaciones aritméticas definidas en el paquete **std_logic_arith**, que contienen, en su definición en el Cuerpo de Paquete, código que advierte de anomalías en los operandos que se les pasan (como, en este caso, que alguno de los operandos contenga valores metalógicos sin sentido para la operación).

6. Para seguir mejor la simulación es conveniente que los valores de los buses se presenten en base hexadecimal. Seleccione con el ratón la señal **s_dato_a** sobre la ventana de formas de onda y pulse el botón derecho para desplegar un menú de **pop-up**. Seleccione la opción **Hexadecimal** en **Bus display** (figura e2).

Figura e2

7. Repita la operación para el resto de los buses.

8. Ordene que la simulación avance **3 microsegundos**.

   Puede revisar los resultados para verificar que el modelo funciona correctamente.

9. Abra, en la ventana del **Workspace** el fichero que contiene el ***test-bench*** y sitúe un punto de ruptura, seleccionando con el ratón la línea que asigna valor a la señal **s_dato_b**, dentro del bucle **FOR**, y ejecutando **Debug -> Insert/Remove a breakpoint**.

10. Abra también el fichero con el modelo del circuito; cambie el tamaño y la ubicación de las ventanas en pantalla para obtener una vista como la que se muestra en la figura e3.

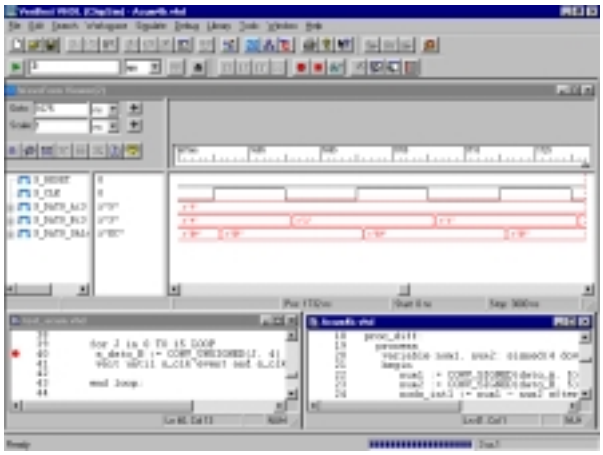


Figura e3

11. Añada los nodos internos del modelo del circuito a la ventana de presentación de formas de onda realizando una pulsación sobre el símbolo (figura e4) con la etiqueta **DUT:ACUM4B(RTL)**. Seleccione las señales internas y pulse **Add**; después cierre la ventana.

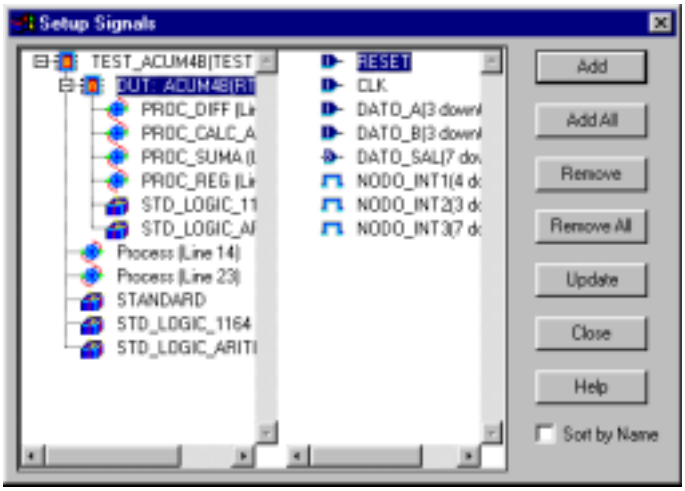


Figura e4

12. Intercale las señales entre **s_dato_b** y **s_dato_sal**. Cambie la representación de valores a hexadecimal.

13. Ordene que continúe la simulación con **Simulate -> Run Forever**.

    La simulación se interrumpe en el punto de ruptura. La sentencia de asignación de valor sobre **s_dato_b** se ejecuta cuando se verifica la ocurrencia de un flanco de bajada; por tanto, estamos en un instante del tiempo de simulación en que ha habido un evento en la señal de reloj, **s_clk**. Para ver la sucesión de acciones que ocurren en el simulador se va a realizar una ejecución paso a paso. La sentencia pendiente de ejecución se señala con una flecha verde en el margen del texto. Observe que el tiempo de simulación es **3010 ns**.

14. Ordene que se avance un paso (en adelante, esta acción se resumirá con **PaP)**. Se proyecta la asignación de valor con un retardo de **1 ns** (para **3011 ns**) y se avanza hasta la siguiente sentencia (**WAIT**); es la última que se ejecutará en este tiempo de simulación en este proceso.

15. **PaP**. La siguiente sentencia es la primera del proceso **proc_reg**, ya que este proceso es sensible a la señal **clk**, que está conectada en el ***test-bench*** a **s_clk** (en la que ha habido un evento).

16. Con un **PaP** podrá comprobar la ejecución del proceso: ni **reset** vale **'1'**, ni hay un flanco positivo.

17. **Pap**. El tiempo de simulación avanza.

    Esto es debido a que se ejecutaron los dos procesos sensibles a la señal de reloj en el tiempo **3010 ns**. En el que modela el registro no sucedió nada relevante; en el de manejo de estímulos se proyectó una asignación para **T = 3011 ns,** tiempo en el que nos encontramos ahora. Como la asignación sobre **s_dato_b** se efectúa en este instante, se produce un evento en la señal. La flecha de ejecución apunta a la última sentencia – por tener lista de señales - del único proceso, **proc_diff**, que es sensible a **dato_b**.

18. Haga cuatro **PaP** para ejecutar las sentencias del proceso.

    Por motivos similares a los ya apuntados, el tiempo de simulación avanza hasta 3012 ns.

19. Dos **PaP** y estará en **T = 3013 ns** ejecutando **proc_suma**.

    Observe que en el modelo no hay ningún proceso sensible a la señal a la que se asigna valor en este proceso (nodo_int3 es la entrada de datos del registro), por lo que la cadena de comunicación entre procesos "combinacionales" finaliza.

20. **PaP**.

    Se avanza hasta **T = 3020 ns** porque el siguiente proceso que se activa es el que genera la señal de reloj –este proceso había asignado el valor **'0'** en **3010 ns** y quedó desactivado **10 ns** por la sentencia **WAIT FOR...**–.

21. **PaP**. Se llega a la sentencia que proyecta la asignación del valor **'1'** al reloj.

22. **PaP**. Se llega a la sentencia **WAIT FOR...**.

    Cuando se ejecute la sentencia WAIT se podría creer que ha terminado la ejecución de procesos en este instante de tiempo, pero no es exactamente así. Hay que tener en cuenta que se ha ejecutado una sentencia como:

    ```
    s_clk <= '1';
    ```

En esta sentencia se asigna valor a una señal sin especificar retardo, lo que implica que se debe actualizar el valor en el mismo valor del tiempo de simulación en el que se realiza la proyección de asignación de valor. Un simulador VHDL trata está situación del siguiente modo:

- sin avanzar el tiempo de simulación, actualiza el valor de la señal **s_clk** y, a continuación,

- ejecuta los procesos que sean sensibles a eventos de **s_clk**.

23. **PaP**. Observe que **s_clk** pasa a valer uno y se pasa a ejecutar el proceso **proc_reg**.

24. Con dos **PaP** puede verificar la ejecución: se detecta el flanco positivo y se asigna el valor de entrada a la salida del registro.

25. **Pap**. Se avanza a **T = 3021 ns** donde se realiza la asignación proyectada sobre **dato_sal**. Al existir un evento en esta señal, a la que es sensible el proceso **proc_suma**, a continuación se apunta a la sentencia de este proceso.

26. Dos **PaP** y se avanza hasta **T = 3030 ns** donde se ejecuta el proceso de asignación del valor **'0'** a **s_clk** y, si se ejecutan tres **PaP** más, se volverá a una situación similar a la del punto 14 del ejercicio.

27. Si lo desea puede trazar de nuevo los pasos que se ejecutan repitiendo las instrucciones 14 a 27.

28. Para finalizar la ejecución paso a paso, ejecute el comando **Debug -> Clear All Breakpoints** y a continuación, **Debug -> Continue**. Puede ordenar que la simulación continúe o abortarla y cerrar el simulador.

7.- Descripciones Estructurales

El circuito utilizado en el ejemplo anterior está compuesto por cuatro bloques funcionales modelados mediante cuatro procesos. La estructura del circuito se modela conectando los procesos mediante señales dentro de un Cuerpo de Arquitectura. Este aspecto del modelo anterior corresponde a una descripción **estructural**, así que, en realidad, el modelo VHDL se ha realizado con una combinación de estilos: **RTL** para modelar en cada proceso la funcionalidad de un módulo y **estructural** para modelar la topología del circuito.

No es éste, desde luego, un procedimiento que se pueda generalizar para realizar descripciones **estructurales**; suponga que debe modelar un circuito compuesto por miles de bloques, muchos de ellos de la misma clase (cien puertas **and** iguales, veinticinco sumadores de cuatro bits, etc.). Repetir un gran número de veces el código del mismo proceso tiene muchos inconvenientes:

- Dificulta las tareas de edición y modificación: el código VHDL tendría muchas más líneas de las que parecen necesarias; la modificación del modelo de un componente obliga a cambiar tantos procesos como veces se haya utilizado; no se puede cambiar el estilo de descripción de un componente haciendo uso de la facilidad que proporciona la disponibilidad de distintos Cuerpos de Arquitectura para la descripción de un dispositivo.

- Dificulta la interpretación del modelo: debido a lo complejo que puede llegar a ser el código.

- Impide la generación de jerarquías en la estructura: el modelo de interconexión presentado es plano, ya que dentro de un proceso no hay posibilidad de describir una nueva estructura.

- Complica el procesamiento en las herramientas de CAD: el simulador distinguirá cien procesos distintos y el ejecutable de simulación será mucho más complicado que si, por ejemplo, un mismo proceso se utiliza cien veces.

En conclusión, la técnica empleada en el ejercicio anterior sólo debe utilizarse cuando se está realizando una descripción sencilla, es decir, de pocas líneas de código y donde no se repita un proceso varias veces. En cualquier otro caso, lo indicado es utilizar las construcciones proporcionadas por el lenguaje para la realización de descripciones **estructurales**.

Siguiendo con el ejemplo anterior, se intuye que el circuito podría describirse si se dispone de modelos VHDL de los cuatro bloques: un restador, un sumador, un registro y un módulo que calcule el valor absoluto de un número en complemento a dos. Resultaría sencillo elaborarlos encapsulando cada proceso en un Cuerpo de Arquitectura y realizando las Declaraciones de

Entidad de cada uno de ellos. El código VHDL de los cuatro módulos del circuito sería entonces:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY calc_abs IS
PORT(
        ent: IN SIGNED(4 DOWNTO 0);
        sal: OUT UNSIGNED(3 DOWNTO 0)
        );
END ENTITY;

ARCHITECTURE RTL OF CALC_ABS IS
BEGIN

proc_calc_abs:
 PROCESS( ent)
  BEGIN
    sal <= CONV_UNSIGNED(ABS(  ent), 4) AFTER 1 NS;

  END PROCESS;

END rtl;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY SUMADOR IS
  PORT(
          a: IN UNSIGNED(3 DOWNTO 0);
          b: IN UNSIGNED(7 DOWNTO 0);
          sal: OUT UNSIGNED(7 DOWNTO 0)
          );
END ENTITY;

ARCHITECTURE  rtl  OF sumador IS
BEGIN
proc_suma:
  PROCESS(a, b)
   BEGIN
    sal <= a + b AFTER 1 NS;

  END PROCESS;

END rtl;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY restador IS
PORT(
        a,b: IN UNSIGNED(3 DOWNTO 0);
        sal: OUT SIGNED(4 DOWNTO 0)
        );
END ENTITY;

ARCHITECTURE  rtl  OF restador IS
BEGIN

proc_diff:
 PROCESS
  VARIABLE num1, num2: SIGNED(4 DOWNTO 0);
  BEGIN
     num1 := CONV_SIGNED(a, 5);
     num2 := CONV_SIGNED(b, 5);
     sal <= num1 - num2 AFTER 1 NS;
     WAIT ON a, b;

  END PROCESS;

END rtl;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY reg IS
  PORT(
         reset , clk: IN STD_LOGIC;
         dato_ ent: IN UNSIGNED(7 DOWNTO 0);
         dato_ sal: BUFFER UNSIGNED(7 DOWNTO 0)
          );
END ENTITY;

ARCHITECTURE  rtl  OF reg IS
BEGIN
proc_reg:
  PROCESS( reset, clk)
   BEGIN
     IF reset = '1' THEN
        dato_sal <= (OTHERS => '0') AFTER 1 NS;

     ELSIF  clk'EVENT  AND clk = '1' THEN
        dato_sal <= dato_ ent  AFTER 1 NS;

     END IF;

  END PROCESS;
END RTL;
```

Para realizar el modelo estructural es necesario poder emplazar **instancias** de estos dispositivos en un Cuerpo de Arquitectura del acumulador. El lenguaje VHDL permite que esto pueda hacerse de varios modos. El método más sencillo (sintáctica y conceptualmente), aunque también el menos "potente" –en cuanto a las facilidades de manejo de la estructura–, consiste en apuntar directamente la Declaración de Entidad y el Cuerpo de Arquitectura que se desea emplazar en la propia instancia. Es el que se ha aplicado en los ***test-bench*** de los ejercicios realizados hasta ahora. En este caso, la sintaxis de una sentencia de emplazamiento de una instancia es:

nombre_de_la_instancia: ENTITY nombre_de_la_Entidad (Arquitectura)
PORT MAP (lista_de_conexión);

El **nombre_de_la_instancia** es una etiqueta de usuario que identifica cada sentencia de emplazamiento. El **nombre_de_la_Entidad** apunta a la Declaración de Entidad del dispositivo que se desea utilizar; debe incluir la librería, aunque sea la de trabajo, donde se encuentra almacenada. A continuación se indica, entre paréntesis, qué arquitectura se desea emplear, para resolver la ambigüedad que podría existir si hay declaradas más de una; es opcional si sólo existe una. Por último, se indica, en la **lista_de_conexión**, a qué señales se conectan los puertos del dispositivo.

Aplicando este sistema de emplazamiento, el acumulador podría describirse de la siguiente manera:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.STD_LOGIC_ARITH.ALL;

ENTITY acum4b IS
PORT(
        reset, clk: IN STD_LOGIC;
        dato_a, dato_b: IN UNSIGNED(3 DOWNTO 0);
        dato_sal: BUFFER UNSIGNED(7 DOWNTO 0)
   );
END ENTITY;

ARCHITECTURE estructural OF acum4b IS
  SIGNAL nodo_int1: SIGNED(4 DOWNTO 0);
  SIGNAL nodo_int2: UNSIGNED(3 DOWNTO 0);
  SIGNAL nodo_int3: UNSIGNED(7 DOWNTO 0);
BEGIN


U0: ENTITY WORK.restador(rtl)  PORT MAP (dato_a, dato_b, nodo_int1);
U1: ENTITY WORK.calc_abs(rtl)  PORT MAP (nodo_int1, nodo_int2);
U2: ENTITY WORK.sumador(rtl)  PORT MAP (nodo_int2, dato_sal,
nodo_int3);
U3: ENTITY WORK.reg(rtl)          PORT MAP (reset, clk, nodo_int3,
dato_sal);

END estructural;
```

La conexión de instancias reproduce la topología del circuito. Dentro del lenguaje sirve para conectar **procesos**; en realidad, el modelo que se describe es el mismo que el del último ejercicio. Cualquier descripción estructural sirve, en última instancia, para conectar procesos que modelan dispositivos en Cuerpos de Arquitectura. El modelo de comportamiento global –los procesos del modelo y su interacción– puede reconstruirse a partir de los procesos que modelan cada dispositivo empleado y de la conexión de las instancias.

Observe las ventajas que proporciona la nueva descripción del acumulador:

- Es clara y compacta. En una unidad aparece la topología del circuito. Resultaría muy sencillo dibujar el esquema del circuito a partir de ella.

- Está mejor organizada: Los componentes utilizados están descritos y pueden modificarse individualmente sin afectar al resto de las unidades.

8.- Ejercicio III.2

Este ejercicio pretende, simplemente, constatar que el nuevo modelo del acumulador es, en esencia, equivalente al primero.

1. Cree un nuevo **Workspace**; edite y añada al mismo las unidades del modelo estructural del acumulador.

2. Copie el ***test-bench*** del último ejemplo al directorio del espacio de trabajo actual y añádalo al mismo.

3. Compile los ficheros y arranque la simulación (configure el entorno con las mismas opciones que en el último ejercicio).

4. Repita las operaciones descritas en los pasos 5 al 9 del ejercicio anterior.

5. Añada los nodos internos a la ventana de simulación. Observe que en la ventana que aparece, en la representación de la jerarquía de diseño aparecen las instancias emplazadas en el modelo (figura e1).

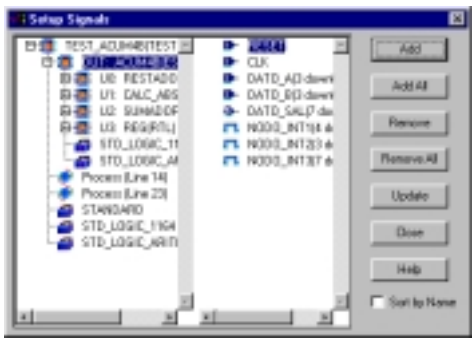


Figura e1

6. Repita los pasos que van del 12 al 23 en el ejercicio anterior. Observe que la secuencia de ejecución de procesos es similar.

7. Ejecute el comando **Window -> Tile Vertically** para distribuir las ventanas tal y como se muestra en la figura.

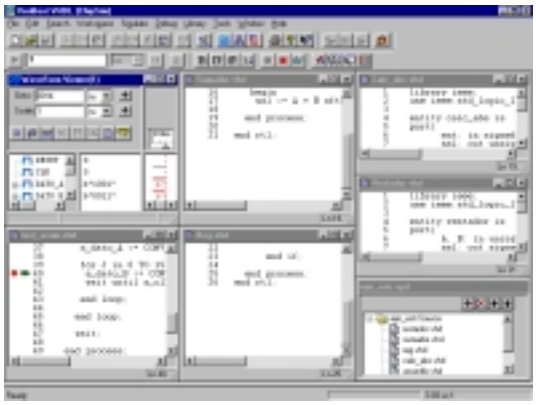


Figura e2

8. Si lo desea, repita nuevamente el procedimiento de simulación paso a paso (con la nueva disposición de las ventanas podrá seguir mejor la ejecución de la simulación).

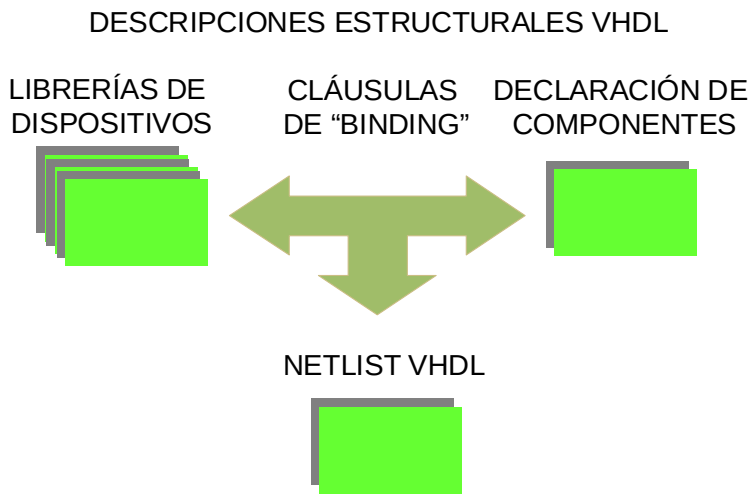
9. Termine la simulación y cierre el simulador.

9.- Componentes y configuraciones

El procedimiento utilizado para construir el modelo estructural del último ejemplo (el emplazamiento directo de las instancias) se añadió en la revisión de la norma del lenguaje en el año 1993 –no lo soportan las herramientas VHDL más antiguas– para facilitar la realización de modelos estructurales de poca complejidad. El mecanismo original es más complicado, pero también mucho más potente; se describe a continuación:

- Para emplazar el modelo de un dispositivo en un Cuerpo de Arquitectura estructural hay que utilizar un **componente**. Los **componentes** son algo así como zócalos para la "inserción" de una Declaración de Entidad y uno de sus Cuerpos de Arquitectura. Antes de emplazar un **componente** hay que declararlo en una zona visible por el Cuerpo de Arquitectura donde se utilice. Cada sentencia de emplazamiento de un **componente** da lugar a una **instancia**.

- Una instancia de un **componente** emplaza la Declaración de Entidad y el Cuerpo de Arquitectura especificado por una cláusula de **configuración**. Hay dos construcciones en el lenguaje que permiten **configurar** instancias de componentes: la **especificación de configuración**, que es una sentencia que aparece en la zona de declaración de los Cuerpos de Arquitectura, y la **Declaración de Configuración**, que, como ya se señaló en el primer capítulo, es una unidad primaria de diseño del lenguaje. El uso de la **especificación de configuración** es apropiado para modelos estructurales sencillos, la **Declaración de Configuración** para modelos complejos.



El principal problema que presenta este procedimiento es que genera modelos con una importante "sobrecarga sintáctica". Por ejemplo, en la versión original del lenguaje, para describir la arquitectura estructural del circuito acumulador –de la manera más sencilla posible– habría que:

- Declarar componentes para cada uno de los dispositivos que se desea utilizar.
- Especificar el dispositivo que se emplaza en cada instancia de componente.
- Colocar las instancias en el Cuerpo de Arquitectura.

La declaración de componente resulta ser redundante –como se verá–, en la mayoría de los casos –en los más sencillos siempre–, con la Declaración de Entidad del dispositivo que se emplaza y la cláusula de configuración prácticamente también. La impresión que causa el uso de este mecanismo en la mayor parte de las aplicaciones es la de haber editado un montón de líneas de código inútiles. Puede comprobarlo comparando el ejemplo que se muestra a continuación para el circuito **acumulador** con el que se realizó anteriormente.

```
ARCHITECTURE estructural_v2 OF acum4b IS

-- declaración de componentes:

COMPONENT restador IS
PORT(
a, b: IN UNSIGNED(3 DOWNTO 0);
sal: OUT SIGNED(4 DOWNTO 0)
   );
END COMPONENT;

COMPONENT calc_abs IS
PORT(
ent: IN SIGNED(4 DOWNTO 0);
sal: OUT UNSIGNED(3 DOWNTO 0)
```

```
   );
END COMPONENT;

COMPONENT sumador IS
PORT(
a: IN UNSIGNED(3 DOWNTO 0);
b: IN UNSIGNED(7 DOWNTO 0);
sal: BUFFER UNSIGNED(7 DOWNTO 0)
   );
END COMPONENT;

COMPONENT reg IS
PORT(
reset, clk: IN STD_LOGIC;
dato_ent: IN UNSIGNED(7 DOWNTO 0);
dato_sal: BUFFER UNSIGNED(7 DOWNTO 0)
   );
END COMPONENT;



-- configuracion de componentes:

 FOR U0: restador USE ENTITY WORK.restador(rtl);
 FOR U1: calc_abs USE ENTITY WORK.calc_abs(rtl);
 FOR U2: sumador  USE ENTITY WORK.sumador(rtl);
 FOR U3: reg      USE ENTITY WORK.reg(rtl);

-- DECLARACIÓN DE SEÑALES INTERNAS:

  SIGNAL nodo_int1: SIGNED(4 DOWNTO 0);
  SIGNAL nodo_int2: UNSIGNED(3 DOWNTO 0);
  SIGNAL nodo_int3: UNSIGNED(7 DOWNTO 0);

BEGIN

-- EMPLAZAMIENTO DE INSTANCIAS

U0: restador PORT MAP (dato_a, dato_b, nodo_int1);
U1: calc_abs PORT MAP (nodo_int1, nodo_int2);
U2: sumador  PORT MAP (nodo_int2, dato_sal, nodo_int3);
U3: reg         PORT MAP (reset, clk, nodo_int3, dato_sal);


END estructural_v2;
```

Las especificaciones de configuración pueden eliminarse si, como en el caso del ejemplo anterior, los componentes que se utilizan para emplazar los modelos de los dispositivos tienen una interfaz idéntica a las Declaraciones de Entidad de éstos.

En la mayor parte de los casos en que resulta necesario realizar una descripción estructural, el emplazamiento directo de los dispositivos es la solución más cómoda y eficiente. Lamentablemente este formato sólo puede utilizarse en herramientas que soporten completamente la revisión de la norma VHDL del año 1993; si esto no sucede, la forma más sencilla de realizar una descripción estructural consiste en:

1. Declarar componentes idénticos a la Declaración de Entidad del dispositivo que van a emplazar –esto facilita, además, las tareas de edición ya que basta con copiar el código de la Declaración de Entidad y cambiar la palabra **ENTITY** por **COMPONENT**–.

2. Emplazar las instancias de los componentes sin hacer uso de especificaciones de configuración.

Para finalizar, pues el tema se tratará con mayor profundidad en capítulos posteriores, se va a explicar la sintaxis básica –faltan algunos detalles importantes– de la Declaración de Componente y las sentencias de emplazamiento de instancias.

El formato más sencillo de una Declaración de Componentes es:

```
COMPONENT nombre IS
    PORT( lista_de_puertos);
END COMPONENT;
```

La lista de puertos se confecciona con reglas similares a la de la Declaración de Entidad.

La sentencia de emplazamiento de una instancia:

```
nombre_de_instancia: nombre_de_componente
PORT MAP( lista_de_conexión);
```

La lista de conexión establece una correspondencia entre puertos de componente y señales. Siguiendo el procedimiento de elaboración de las descripciones estructurales descrito anteriormente, un puerto de componente se corresponde con el (conecta al) puerto de la Declaración de Entidad que se llame igual, tenga la misma dirección y el mismo tipo de datos que él, de modo que la lista de conexión une señales (declaradas en el Cuerpo de Arquitectura estructural o que sean puertos de la Declaración de Entidad a la que está asociado éste) con puertos del dispositivo emplazado. La conexión puede

realizarse por posición o por nombre, tal y como se muestra en la figura.

```
BEGIN

U0: and2
    PORT MAP (a =>ent1,
              b => ent2,
              sal => nodo);
U1: and2
    PORT MAP (a =>nodo,
              b => ent3,
              sal => sal);

END estruct;
```

```
BEGIN

U0: and2
    PORT MAP (ent1, ent2, nodo);
U1: and2
    PORT MAP (nodo, ent3, sal);

END estruct;
```

En la conexión por posición cada señal queda conectada con el puerto que se corresponde en orden de declaración con la posición; en la conexión por nombre con el que se indique explícitamente.

Hay algunas reglas importantes que hay que respetar cuando se realiza la lista de conexión:

1. Un puerto de entrada sólo puede conectarse a una señal cuyo valor pueda leerse. No puede conectarse, por ejemplo, a un puerto de salida de la Declaración de Entidad del dispositivo que se describe –tampoco a uno que fuera de tipo **BUFFER**, aunque su valor sí puede leerse–.

2. Un puerto de salida sólo puede conectarse a una señal cuyo valor pueda escribirse –aquí también están excluidos los puertos de tipo **BUFFER**–.

3. Un puerto de tipo **BUFFER** sólo puede conectarse con una señal declarada en el Cuerpo de Arquitectura o un puerto de tipo **BUFFER** de la Declaración de Entidad del dispositivo que se describe.

Si se desea dejar sin conectar algún puerto, puede indicarse mediante la palabra **OPEN**. Puede ocurrir también que se desee conectar algún puerto a un determinado valor en vez de a una señal. En este caso, si se trabaja con una herramienta que soporte la revisión de la norma del año 1993, puede hacerse en la propia sentencia de emplazamiento, por nombre o por posición, tal y como se muestra:

```
U0: inv PORT MAP('1', sal);
```

Si no es así, hay que recurrir a una señal auxiliar:

```
ent <= '1';
U0: inv PORT MAP(ent, sal);
```

10.- Ejercicio III.3

1. Repita el ejercicio III.2 con la arquitectura **estructural_v2**. Para ello debe editar y compilar la arquitectura e indicar en la sentencia de emplazamiento del ***test-bench*** que es ésta la que se desea utilizar. Compruebe que la descripción estructural sigue siendo válida aunque comente las especificaciones de configuración.

11.- APÉNDICE A: Modelado del funcionamiento del hardware

En este apéndice se explica detalladamente el modelo de tiempo y el modo en que se lleva a cabo una simulación VHDL.

La capacidad descriptiva del lenguaje VHDL –y de cualquier otro HDL– se manifiesta en que los modelos simulados presentan el mismo comportamiento lógico en el discurrir del tiempo que el hardware real sometido a los mismos estímulos lógicos; esto es importante porque pone de manifiesto una característica esencial de los lenguajes de descripción hardware: el motivo por el que un proceso describe un comportamiento hardware y, por ejemplo, un programa en **C** no, es que los procesos se ejecutan en **el tiempo de simulación** –que es una **representación** del tiempo **real** que discurre durante el funcionamiento de un circuito–, mientras que la ejecución de un programa en **C** no se realiza dentro de ninguna clase de modelo de tiempo.

El modelo del tiempo es una característica descriptiva esencial del lenguaje VHDL, se materializa en el concepto de **tiempo de simulación** y en el **modelo de ejecución de las simulaciones** definido en el lenguaje (la forma de ejecutarse los procesos forma parte de este modelo de simulación); para poder interpretar correctamente los modelos descritos en VHDL resulta imprescindible entender el papel desempeñado por este modelo del tiempo.

Los procesos modelan el hardware digital emulando la actividad de éste: en el transcurso del **tiempo de simulación**, para entenderlo basta con entender que:

1. El estado o las salidas de un circuito digital se mantienen estables mientras las entradas del circuito no cambien – esto debe aceptarse en sentido general; evidentemente puede alegarse en contra que hay funciones digitales aestables en las que no es cierto, los procesos pueden también, por supuesto, modelar estas “excepciones” -. Este comportamiento se modela haciendo que los procesos no se ejecuten cuando sus entradas (las señales a las que son sensibles) no sufren eventos.

2. Cuando las entradas de un circuito digital cambian puede haber o no cambios en las salidas o estado del circuito. Este comportamiento del hardware se modela mediante la ejecución del proceso cuando hay eventos en las señales a las que es sensible durante el **tiempo de simulación**.

Lo que queda por aclarar es cuándo y cómo se materializa el **tiempo de simulación**. El **tiempo de simulación** aparece cuando el modelo VHDL se somete a prueba mediante estímulos en un ***test-bench***. Los estímulos suceden en el **tiempo de simulación** porque son asignaciones de valor a señal controlados por retardos o por sentencias **WAIT FOR**. El instante absoluto del tiempo de simulación en que ocurren se deriva de que son asignaciones de valor a señal que ocurren dentro de procesos y todos los procesos de un ***test-bench*** VHDL se ejecutan por primera vez en el origen del tiempo de

simulación (tiempo de simulación 0 fs). Esto último está especificado en el modelo de simulación del lenguaje.

El modelo de simulación del lenguaje especifica que las simulaciones se llevan a cabo de la siguiente manera:

1. En el origen del tiempo de simulación (T = 0) todos los objetos (señales, variables y constantes) del modelo toman su valor inicial de acuerdo con las reglas especificadas en el lenguaje (el valor por defecto o el especificado en su declaración). A continuación se ejecutan todos los procesos del modelo hasta que se suspendan (bien porque alcanzan una sentencia **WAIT**, bien, si tienen lista de sensibilidad, porque llegan a la última del proceso).

En un ***test-bench*** la consecuencia de esta acción es la proyección de asignación de valores a estímulos para un conjunto de valores de tiempo, digamos futuro –ese tiempo futuro puede ser, como se verá más adelante, el mismo en que se realiza la proyección de asignación de valor (si el retardo especificado es 0) –, haciendo que el **tiempo de simulación** avance hasta la primera coordenada de tiempo en que esté proyectada una asignación. Alternativa o simultáneamente puede ocurrir que en algún proceso se alcanzara una sentencia **WAIT FOR**; en este caso también se define un instante de tiempo futuro en que deberá reactivarse algún proceso. Cualquiera de estos dos mecanismos extiende la ejecución de los modelos dentro del tiempo. Obsérvese que ambos mecanismos producen un avance de tiempo hasta un instante futuro en que pueden –o deben en el caso de la sentencia **WAIT FOR**– activarse procesos, puesto que son instantes en que pueden producirse eventos sobre señales.

2. El tiempo de simulación avanza hasta la primera coordenada en que haya proyectada una asignación –o se haya diferido la continuación de la ejecución de un proceso mediante una sentencia **WAIT FOR**– donde se ejecutará un **ciclo de simulación**.

El **ciclo de simulación** es la parte del modelo de simulación que materializa la actualización de señales y ejecución de los procesos dentro del **tiempo de simulación**. Se desarrolla en dos fases:

- **Primera parte del ciclo de simulación**: El **ciclo de simulación** comienza efectuando las asignaciones proyectadas para el **tiempo de simulación** actual.

- **Segunda parte del ciclo de simulación**: Cuando todas las señales han sido actualizadas, se ejecutan **concurrentemente** todos los procesos afectados por la actividad de las señales actualizadas en la primera parte del ciclo de simulación, o cuya ejecución resulto aplazada por una sentencia **WAIT FOR**, hasta que se suspende su ejecución.

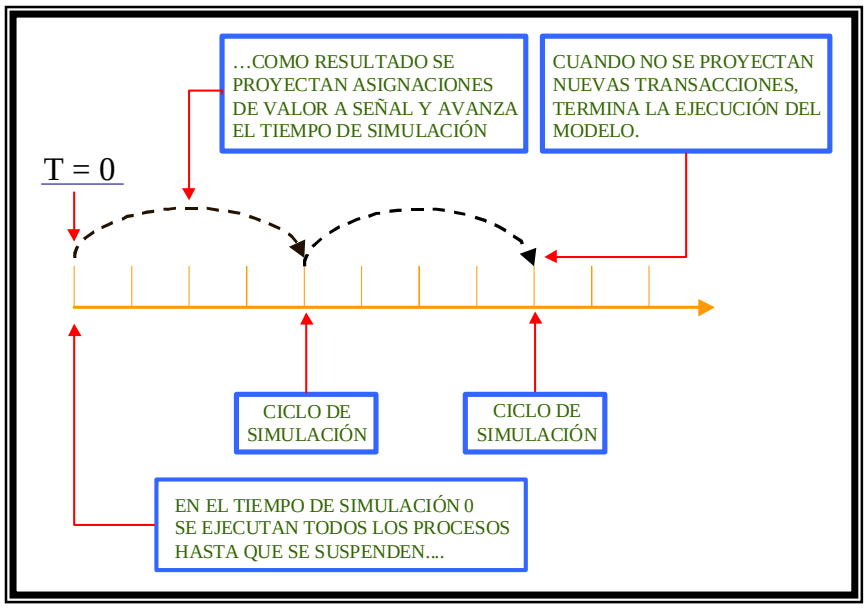
A partir de este momento, el tiempo de simulación avanza por proyecciones de asignación de valor a señal o sentencias **WAIT FOR**

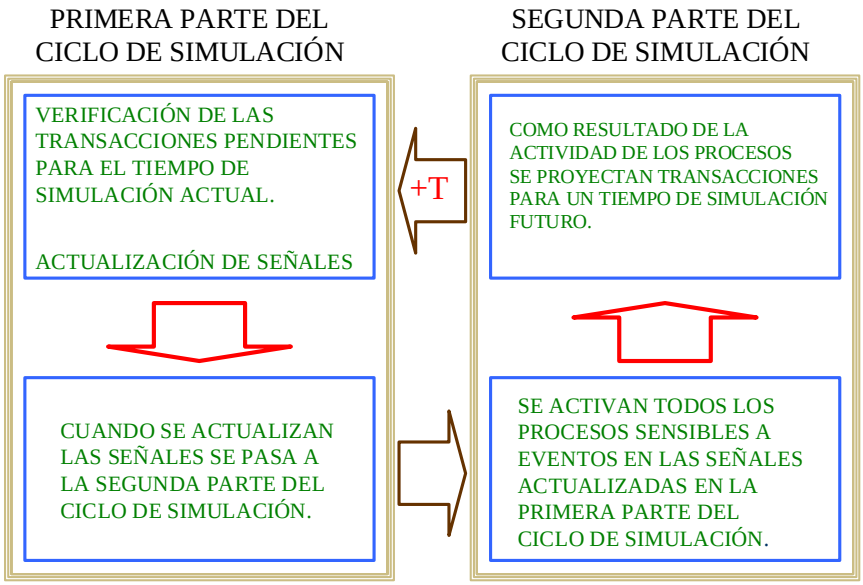
existentes tanto en los procesos del modelo hardware como en los que manejan estímulos en el ***test-bench***.

3. Cuando termina la ejecución del **ciclo de simulación** el tiempo avanza hasta el siguiente instante en que haya proyectada una transacción a una señal (o deba reactivarse un proceso con una sentencia **WAIT FOR**) y volverá a ejecutarse otro **ciclo de simulación**.

4. La simulación finaliza cuando se alcanza un tiempo máximo especificado o cesa el avance del tiempo de simulación.

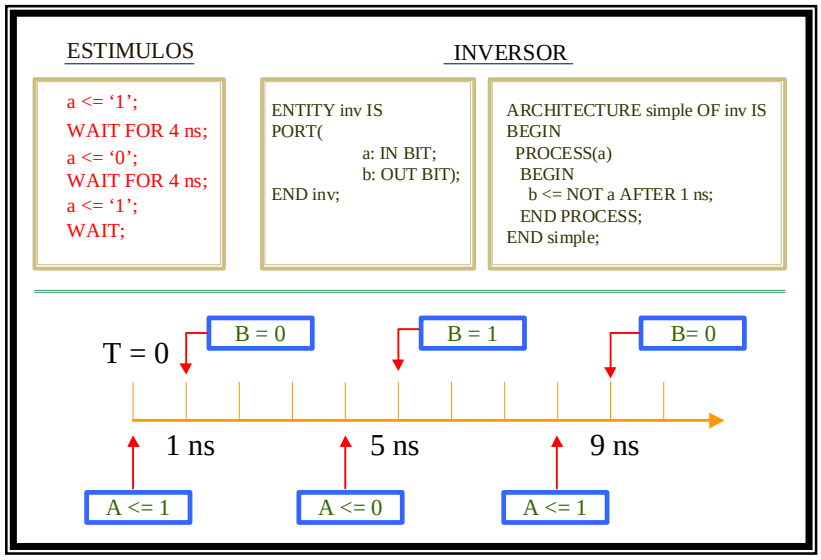
Las siguientes figuras resumen el mecanismo:





Los modelos de simulación que funcionan con un mecanismo como el que se ha descrito se denominan **manejados por eventos** (**Event Driven**).

El siguiente ejemplo ilustra el desarrollo de una simulación y nos permitirá introducir algún nuevo concepto.



Se trata de un ***test-bench*** para probar el funcionamiento del modelo de un inversor con un tiempo de propagación de 1 ns. El modelo se prueba asignando a la entrada del inversor los valores '1', '0' y '1' en los instantes 0, 4 y 8 ns, respectivamente, del **tiempo de simulación**. Por simplicidad se muestran sólo los detalles relevantes del código del ***test-bench***.

En T = 0 se inicializan las señales del modelo: son de tipo **BIT**, no tienen especificado valor inicial en la declaración y, por tanto, su valor inicial será '0'. A continuación se ejecutan todos los procesos del ***test-bench***: el que maneja los estímulos y el del modelo del inversor. Como consecuencia se proyecta la asignación del valor '1' para la entrada **A** del inversor con un retardo de 0 ns (para T = 0 ns) y del valor '1' para la salida **B** con un retardo de 1 ns (T = 1 ns). El proceso que maneja los estímulos se suspende hasta el instante T = 4 ns.

La simulación avanza hasta el tiempo en que está proyectada la primera asignación, que resulta volver a ser T = 0 ns. En este instante debe ejecutarse un **ciclo de simulación**. A estos **ciclos de simulación** que se ejecutan sin avance efectivo del valor de tiempo se los denomina **ciclos delta**; sirven fundamentalmente para que el modelo soporte la realización de simulaciones funcionales (sin retardos). Pueden entenderse como ciclos en los que el **tiempo de simulación** avanza una cantidad infinitesimal de tiempo.

En T = 0, por tanto, se actualiza la señal **A** con el valor proyectado, '1', en la primera parte del ciclo de simulación y, como consecuencia, en la segunda parte, se activa el proceso que modela el inversor y se proyecta la asignación a **B** del valor '0' para 1 ns después. Aquí surge un problema: ya existía una proyección de asignación, de valor '1', para este instante –realizada por el mismo proceso (**driver**, en términos del lenguaje), no tiene nada que ver con asignaciones realizadas por distintos ***drivers*** como sucede en las señales **resolved**–. El modelo de simulación resuelve este problema haciendo que la última proyección elimine la primera.

Resumiendo, después de este último ciclo de simulación en T = ‘0’ hay proyectada una asignación de valor a una señal en T = 1 ns y la reactivación de un proceso para T = 4 ns. Por tanto el **tiempo de simulación** avanza hasta T = 1 ns.

En T = 1 ns se ejecuta un ciclo de simulación: primero se actualiza **B** con el valor ‘0’ y después los procesos afectados por el evento en **B**, en este caso no hay ninguno y el tiempo avanza hasta 4 ns.

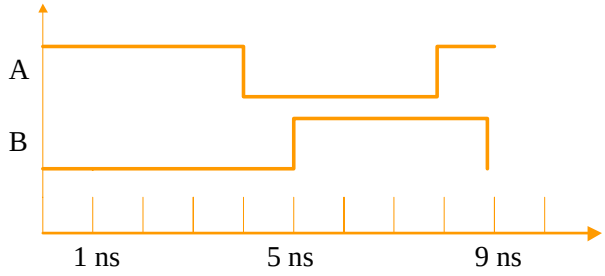
En T = 4 ns se reactiva el proceso que maneja los estímulos: en este caso no hay primera parte del ciclo de simulación. Se ejecuta la sentencia que proyecta la asignación del valor ‘0’ para 0 ns después y se suspende el proceso hasta T = 8 ns. Como consecuencia de la proyección se ejecuta un ciclo delta en el que se actualiza **A**, se activa el proceso que modela el inversor y se proyecta la asignación del valor ‘1’ para **B** en T = 5 ns.

La simulación continúa repitiendo el proceso descrito en los dos últimos párrafos para los instantes de tiempo 5 y 8 ns y finaliza en T = 9ns con una asignación del valor ‘0’ a la salida **B**.

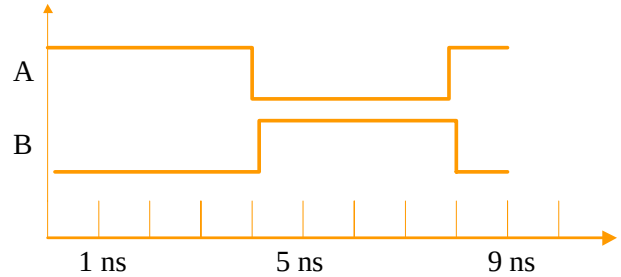
Es interesante señalar que si la sentencia que modela el inversor no incluyera una cláusula de retardo de asignación los ciclos de simulación que se ejecutan en 1, 5 y 8 ns se convertirían en ciclos delta ejecutados en 0, 4 y 8 ns, representando una simulación del funcionamiento de un inversor sin retardos.

El resultado de ambas simulaciones puede observarse en las siguientes figuras.

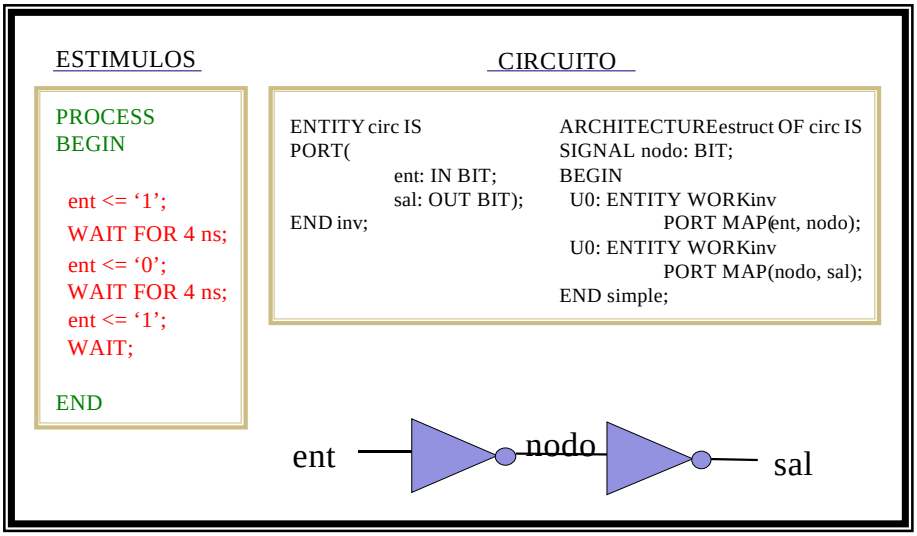
Simulación del inversor con un retardo de 1 ns:



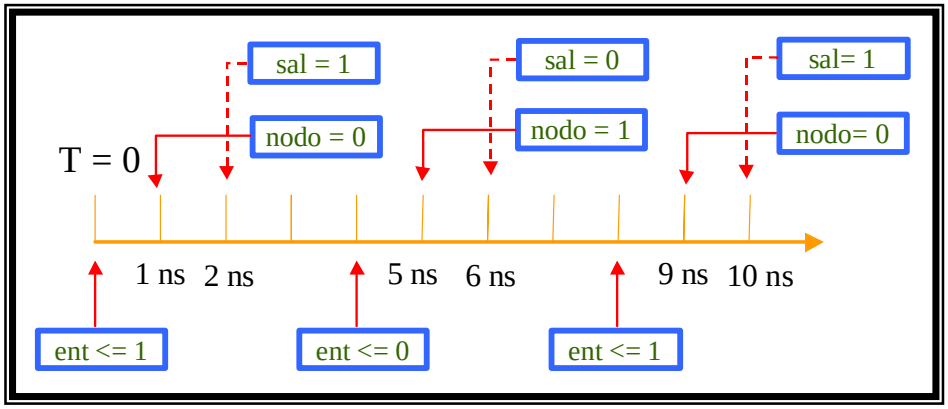
Simulación funcional del inversor:



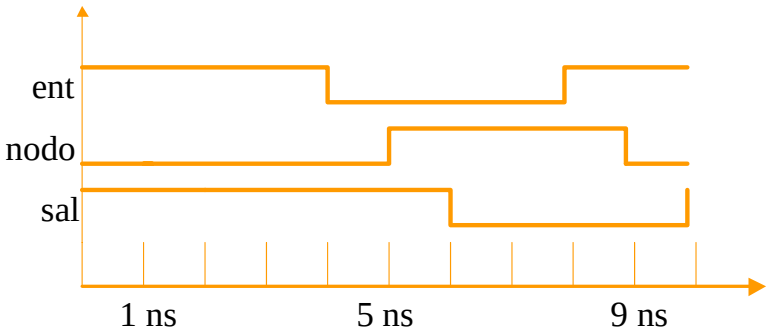
En general el modelo que se simula estará compuesto por una red de procesos conectados mediante señales. En este caso, los procesos del modelo que se ejecutan en la segunda parte del ciclo de simulación realizarán asignaciones que darán lugar a la activación de otros procesos del modelo. Si se conectan en serie dos inversores como los modelados en el último ejemplo:



La simulación para las señales **ent** y **nodo** se desarrollaría igual que como se describió anteriormente para **A** y **B**. Además, en los tiempos 1, 5 y 9 ns (cuando se producen eventos en **nodo** –antes **B**–), se ejecutaría en la segunda parte del ciclo de simulación el proceso que modela al segundo inversor, proyectando asignaciones de valor sobre **sal** para 2, 6 y 10 ns, respectivamente.

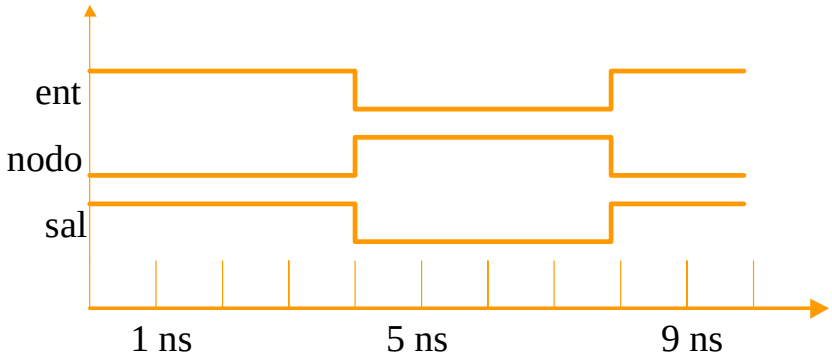


El resultado gráfico de la misma puede verse en la figura:



Es interesante hacer notar como el mecanismo de especificación de retardos combinado con el modelo de simulación permite la correcta representación del retardo total del circuito.

Si se hubiera empleado un modelo sin retardos para el inversor, los ciclos de simulación que se ejecutan en los tiempos 1 y 2 ns, 5 y 6 ns y 9 y 10 ns, se ejecutarían, en ese orden como ciclos delta, en 0, 4 y 8 ns, respectivamente. En ese caso el resultado no representaría retardos.



12.- APENDICE B

En este apéndice se presentan las sentencias que pueden utilizarse dentro de los procesos, los atributos de señales y las sentencias concurrentes. Todas las sentencias pueden llevar etiquetas pero, por simplicidad, no se muestra esta posibilidad en la sintaxis expuesta.

12.1.- Sentencias de asignación a señales

Normalmente los procesos ejecutan un algoritmo de procesamiento para determinar el valor que deben tomar una o varias señales. La sentencia que especifica la asignación de un valor a una señal puede tomar alguna de las siguientes formas:

```
nombre_de_señal <= valor after tiempo;

nombre_de_señal <= valor after tiempo,
                   valor after tiempo,
                   . . .,
                   valor after tiempo;
```

El valor que se asigna puede ser el de un objeto, el devuelto por una función o, directamente, cualquiera que pertenezca al tipo de dato sobre el que está declarada la señal.

Como se puede observar en las anteriores expresiones, en la sentencia de asignación se puede indicar un retardo que establece el período de tiempo que debe transcurrir desde la ejecución de la sentencia hasta el momento en que la asignación se haga efectiva.

Las señales a las que se asigna valor en los procesos serán puertos de la Declaración de Entidad o señales declaradas en el Cuerpo de Arquitectura. En los procesos no se pueden declarar señales – porque no tiene sentido, ya que en definitiva sirven para comunicar procesos -.

12.2.- Variables

Las sentencias de asignación a variables tienen la siguiente forma:

```
nombre_de_variable := valor;
```

Las variables son similares a las de los lenguajes de programación de alto nivel. Mantienen su valor cuando se suspende el proceso. Las variables son locales, es decir, se declaran en el propio proceso (el lenguaje permite la declaración de variables globales en los Cuerpos de Arquitectura, pero no se suelen utilizar debido a los problemas que conlleva su actualización por procesos que se ejecutan concurrentemente).

12.3.- Sentencia IF

Las posibles apariencias de la sentencia **IF** son las siguientes:

1.-

```
IF condición THEN
        SENTENCIAS
END IF;
```

2.-

```
IF condicion THEN
        SENTENCIAS
ELSE
        SENTENCIAS
END IF;
```

3.-

```
IF condición THEN
        SENTENCIAS
ELSIF condición THEN
        SENTENCIAS
ELSIF condición THEN
        SENTENCIAS
        . . .
END IF;
```

La condición de selección debe ser un valor o un objeto **BOOLEAN**.

12.4.- Sentencia CASE

La forma general de una sentencia case es:

```
CASE expresión IS
        WHEN valor => SENTENCIAS
         WHEN valor1|valor2 => SENTENCIAS
        WHEN rango_de_valores => SENTENCIAS
        WHEN OTHERS => SENTENCIAS
END CASE;
```

Los valores de selección deben cubrir todos los que pueda tomar la expresión de selección. Como se puede observar, pueden ser un valor o un conjunto de valores (especificados mediante un rango o unión de valores). La condición **OTHERS** resulta útil cuando el rango de valores que puede tomar la expresión de selección no es finito o simplemente, por comodidad, cuando muchos de los posibles valores de la expresión se corresponden con las mismas sentencias.

12.5.- Bucles

En VHDL se pueden utilizar tres tipos de bucles:

- bucles indeterminados,
- bucles **FOR**,
- bucles **WHILE**.

En los bucles pueden utilizarse sentencias **EXIT** y **NEXT** que sirven, respectivamente, para salir o finalizar una determinada iteración.

```
EXIT nombre_de_bucle;
o
        EXIT nombre_de_bucle WHEN condición;
```

y

```
        NEXT nombre_de_bucle;
o
        NEXT nombre_de_bucle WHEN condición;
```

Un bucle indeterminado es el que se ejecuta un número indefinido de veces. Su forma genérica es:

```
LOOP
        sentencias
END LOOP;
```

Es conveniente recordar que este tipo de construcciones suelen dar lugar a códigos mal estructurados, por lo que, en la medida de lo posible, debe evitarse, utilizándose en su lugar los bucles **FOR** y **WHILE**, que tienen la siguiente forma:

```
        FOR variable_del_bucle IN rango LOOP
                sentencias
        END LOOP;

WHILE condicion LOOP
        sentencias
END LOOP;
```

12.6.- Sentencia null

La sentencia **null** no realiza ninguna acción. Resulta útil en ramas de la sentencia case a las que no corresponde ningún grupo de sentencias. Su forma es:

```
null;
```

12.7.- Otras sentencias

Además de éstas, dentro de los procesos puede haber llamadas a procedimientos y sentencias de comprobación de condiciones que se verán en capítulos posteriores.

12.8.- Atributos de señales

Las señales y otras construcciones del lenguaje VHDL tienen, al igual que los tipos de datos, atributos. Los atributos de las señales se utilizan frecuentemente para la realización de descripciones de comportamiento, por lo que a continuación se presentan y explican algunos de los más interesantes.

En la explicación de los atributos se utilizan los términos **evento** y **transacción**. La diferencia entre un evento y una transacción consiste en que un evento ocurre cuando en una asignación de valor a una señal ésta cambia de valor; mientras que en una transacción la asignación puede ser la del valor previo de la señal. Por tanto, siempre que hay un evento se ha producido una transacción, pero no siempre que se realice una transacción habrá un evento.

1.- Atributos de clase señal:

Los siguientes atributos de señales son, a su vez, señales:

Señal'Delayed(tretardo): Es la misma señal retardada un tiempo **tretardo**.

Señal'Stable(tiempo): Es una señal booleana que vale **TRUE** si la señal no ha cambiado de valor en el tiempo especificado y **FALSE** en caso contrario.

Señal'Quiet(tiempo): Es una señal booleana que vale **TRUE** si a la señal no se le ha asignado un valor en el tiempo especificado y **FALSE** en caso contrario.

Señal'Transaction: Es una señal de tipo Bit que cambia de valor cada vez que se realiza una transacción sobre la señal.

2.- Atributos que no son señales:

Señal'event: Es un valor booleano que es **TRUE** si ha habido un evento en la señal en el ciclo de simulación.

Señal'active: Es un valor booleano que es **TRUE** si ha habido una transacción en la señal en el ciclo de simulación.

Señal'last_event: Es un valor de tipo **TIME** que indica el tiempo transcurrido desde el último evento en la señal

Señal'last_active: Es un valor de tipo **TIME** que indica el tiempo transcurrido desde la última transacción en la señal

Señal'last_value: Es el valor de la señal antes del último evento.

12.9.- Ejemplos

-- sumador de 1 bit

```
PROCESS(op1, op2, cin)

BEGIN

  sum <= op1 XOR op2 XOR cin  AFTER 3 NS;
  cout <= (op1 AND op2) OR (op2 AND cin)  OR (op1 AND cin) AFTER
2 NS;
END PROCESS;
```

-- puerta and

```
PROCESS(a,b)
  VARIABLE tmp: BIT;
BEGIN

IF a = '1' AND b = '1' THEN
  tmp:= '1';
ELSE
  tmp := '0'
END IF;

sal <= tmp AFTER 400 PS;

END PROCESS;
```

-- multiplexor con 4 entradas

```
PROCESS(sel, ent)
 CONSTANT tp: TIME := 2 ns;
BEGIN

CASE sel IS
        WHEN '00' => SAL <= ent(0) AFTER tp;
        WHEN '01' => SAL <= ent(1) AFTER tp;
        WHEN '10' => SAL <= ent(2) AFTER tp;
```

```
  WHEN '11' => SAL <= ent(3) AFTER tp;
END CASE;

END PROCESS;
```

-- autómata

```
                PROCESS(clk)
TYPE colores IS (ambar, rojo, verde);
VARIABLE estado: colores;

BEGIN
 WAIT UNTIL clk'event  AND clk = '1';

          IF cambiar = '1' THEN
  CASE estado IS
    WHEN verde => estado := ambar;
    WHEN ambar => estado := rojo;
    WHEN rojo => estado := verde;
 END CASE;
END IF;

END PROCESS;
```

-- contador

```
PROCESS
        VARIABLE cont: NATURAL;
        CONSTANT max_cont: NATURAL := 9;
BEGIN
 WHILE cont /= max_cont LOOP
  cont := cont + 1;
  q <= cont AFTER 1 NS;

  WAIT FOR 100 ns;
END LOOP;

cont := 0;
q <= cont AFTER 1 NS;
WAIT FOR 100 ns;

END PROCESS;
```

-- registro de desplazamiento

```
        PROCESS(reset, clk)
         BEGIN
```

```
IF reset = '1' THEN
 q <= (OTHERS => '0');

ELSIF clk'EVENT AND clk = '1' THEN
 IF desplazar = '1' THEN
  FOR i IN 15 DOWNTO 1 LOOP
    q(i) <= q(i –1);
  END LOOP;

  q(0) <= dato_in;
 END IF;
END IF;

END PROCESS;
```

12.10.- Sentencias concurrentes

Las sentencias concurrentes son construcciones equivalentes a los procesos. Su ejecución se realiza en paralelo a la de cualquier otro proceso o sentencia concurrente que pudiera haber en el modelo. Las sentencias concurrentes del lenguaje VHDL son:

- La **sentencia concurrente de asignación**.
- La **sentencia concurrente de asignación condicional**.
- La **sentencia concurrente de selección de condiciones**.
- La **sentencia concurrente de comprobación de condiciones**.

12.11.- Sentencia concurrente de asignación

La sentencia concurrente de asignación es equivalente a un proceso que realiza asignaciones sobre una o varias señales. Su sintaxis es idéntica a la de las sentencias de asignación de los procesos. Por ejemplo:

El proceso:

```
PROCESS(b,c)
BEGIN
 a <= b + c AFTER 4 NS;
END PROCESS;
```

es equivalente a la sentencia de asignación concurrente:

```
a <= b + c AFTER 4 ns;
```

La sentencia de asignación concurrente es sensible a las señales que aparecen en la misma y, por tanto, se ejecutará cada vez que haya un evento en una de ellas.

12.12.- Sentencia concurrente de asignación condicional

La sentencia de asignación condicional equivale a un proceso con una sentencia **IF**. Su sintaxis es la siguiente:

```
nombre_de_señal <= asignacion when condicion else
                   asignacion when condicion;
```

La sentencia:

```
q <= a AFTER 1 ns WHEN enable = '1' else
     b after 1 ns;
```

es equivalente al proceso:

```
PROCESS(enable, a, b)
BEGIN
        IF enable = '1' THEN
                q <= a AFTER 1 NS;
        ELSE
                q <= b AFTER 1 NS;
        END IF;
END PROCESS;
```

12.13.- Sentencia concurrente de selección de condiciones

Es una sentencia concurrente equivalente a un proceso con una sentencia **CASE**. Su forma general es:

```
WITH expresión SELECT

señal <=  asignacion WHEN valores,
          asignacion  WHEN valores,
          asignacion  WHEN valores,
          . . .
          asignacion WHEN valores;
```

La sentencia es sensible a la señal que forma la expresión de selección y a cualquier otra que se utilice para asignar valor en las opciones de selección.

El proceso:

```
PROCESS(din, sel)

BEGIN
 CASE sel IS
  WHEN "00" => dout <= din(0);
  WHEN "01" => dout <= din(1);
  WHEN "10" => dout <= din(2);
  WHEN "11" => dout <= din(3);
```

```
 END CASE;

END PROCESS;
```

equivale a la sentencia:

```
WITH sel SELECT

dout <=
        din(0) when "00",
        din(1) when "01",
        din(2) when "10",
        din(3) when "11";
```

12.14.- Ejemplos

```
-- decodificador

WITH datoin SELECT

datoout <=
                "00" after 3 ns when "0001",
                "01" after 3 ns when "0010",
                "10" after 3 ns when "0100",
                "11" after 3 ns when "1000";

-- puerta or

sal <= '0' AFTER 1 ns WHEN a = '0' AND b = '0'
        ELSE '1' after 1 ns;
```

12.15.- Resumen y Ejemplos

```
IF {Condición} THEN
  {Sentencias de ejecución secuencial};
END IF;
```

```
IF {Condición} THEN
  {Sentencias de ejecución secuencial};
ELSE
  {Sentencias de ejecución secuencial};
END IF;
```

```
IF {Condición} THEN
  {Sentencias de ejecución secuencial};
ELSIF {Condición} THEN
  {Sentencias de ejecución secuencial};
ELSIF {Condición} THEN
  {Sentencias de ejecución secuencial};
ELSE
  {Sentencias de ejecución secuencial};
END IF;
```

```
CASE {Cláusula de Selección} THEN
  WHEN {valor} =>
    {Sentencias de ejecución secuencial};
  WHEN {valor} =>
    {Sentencias de ejecución secuencial};
  WHEN {valor} =>
    {Sentencias de ejecución secuencial};
END CASE;
```

TIPO DISCRETO O ARRAY DE UNA DIMENSIÓN

VALOR O GRUPO DE VALORES U "OTHERS"

BUCLES FOR Y WHILE

```
FOR {Iteraciones} LOOP
  {Sentencias de ejecución secuencial};
END LOOP;
```

```
WHILE {Condición} LOOP
  {Sentencias de ejecución secuencial};
END LOOP
```

BUCLE INFINITO

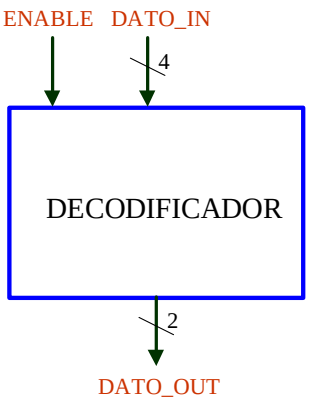
```
LOOP
  {Sentencias de ejecución secuencial};
END LOOP
```

SENTENCIA EXIT: SALIR DEL BUCLE.

SENTENCIA RETURN: PASAR A LA PRÓXIMA ITERACIÓN.

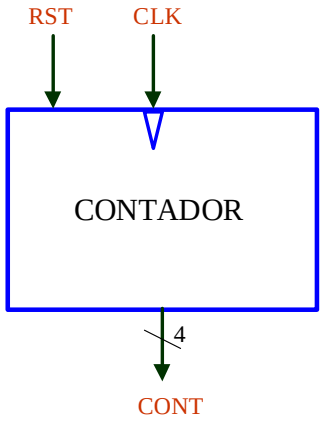
DECODIFICADOR

```
PROCESS(enable, dato_in)
 BEGIN
  IF enable = '1' THEN
   CASE (dato_in) IS
     WHEN "00" => dato_out <= "0001";
     WHEN "01" => dato_out <= "0010";
     WHEN "10" => dato_out <= "0100";
     WHEN "11" => dato_out <= "1000";
   END CASE;
  ELSE dato_out <= "0000";
  END IF;
 END PROCESS;
```



CONTADOR

```
 PROCESS(rst, clk)
 VARIABLE var_cnt: unsigned(3 downto 0);
  BEGIN
   IF rst = '1' THEN
         var_cnt := "0000";
   ELSIF clk'EVENT AND clk = '1' THEN
          var_cnt := var_cnt + 1;
   END IF;
          cnt <= var_cnt after 3 ns;
  END PROCESS;
```



PUERTA NOR

```
 PROCESS(ent)
 VARIABLE tmp: BIT;
  BEGIN
   tmp:= '0';
   FOR I IN ent'RANGE LOOP
    IF ent(I) = '1' THEN
         tmp := '1';
         EXIT;
    END IF;
   END LOOP;
   sal <=  NOT tmp after 1 ns;
  END PROCESS;
```

PUERTA AND

```
 PROCESS(ent)
 VARIABLE tmp: BIT;
  BEGIN
   tmp:= '1';
   FOR I IN ent'RANGE LOOP
    IF ent(I) = '0' THEN
         tmp := '0';
         EXIT;
    END IF;
   END LOOP;
   sal <=  tmp after 1 ns;
  END PROCESS;
```

Sentencia de Asignación de Valor a Señal

```
PROCESS(a, b)
BEGIN
   sal <= a + b after 6 ns;
END PROCESS;
```

⟺

```
sal <= a + b after 6 ns;
```

Sentencia Condicional de Asignación de Valor a Señal

```
PROCESS(a, b, sel)
BEGIN
  IF sel = '0' THEN sal <= a;
  ELSE sal <= b;
  END IF;
END PROCESS;
```

⟺

```
sal <= a WHEN sel = '0'
       ELSE b;
```

Sentencia de Asignación de Valor con Selección

```
PROCESS(a, b, c, d, sel)
BEGIN
   CASE sel IS
    WHEN "00" =>
                  sal <= a;
    WHEN "01" =>
                  sal <= b;
    WHEN "10" =>
                  sal <= c;
    WHEN "11" =>
                  sal <= d;
   END CASE;
END PROCESS;
```

⟺

```
WITH sel SELECT
sal <=  a WHEN "00",
        b WHEN "01",
        c WHEN "10",
        d WHEN "11";
```

AMPLIACIÓN DE CONCEPTOS

0.- Resumen del Capítulo

Conceptos Teóricos:

- *Sintaxis completa de la Declaración de Entidad.*
- *Genéricos.*
- *Subprogramas: procedimientos y funciones.*
- *Procesos Pasivos. Sentencias ASSERT.*
- *Declaración de Componentes y Especificaciones de Configuración.*
- *Sentencias GENERATE.*

Prácticas sobre el simulador VeriBest:

- *Verificación de modelos con sentencias ASSERT.*

En este capítulo se profundiza en el conocimiento de algunas construcciones ya presentadas: la Declaración de Entidad, la Declaración y Emplazamiento de Componentes y la Especificación de Configuración. Se presentan además los procedimientos y funciones en VHDL y las sentencias ASSERT y GENERATE. Todo ello para que se conozcan suficientemente los elementos del lenguaje más comúnmente usados.

Se realiza un ejercicio para comprobar la aplicación práctica de las sentencias ASSERT en el modelado del hardware.

1.- Sintaxis completa de la Declaración de Entidad

En la Declaración de Entidad, además del nombre del dispositivo y sus puertos, pueden especificarse parámetros, declararse objetos y subprogramas, que vayan a utilizarse en los Cuerpos de Arquitectura que pueda tener asociados, e incluirse procesos pasivos (procesos que no asignan valores a señales). La sintaxis completa de la Declaración de Entidad es:

```
ENTITY nombre IS
  GENERIC(
        lista_de_parámetros);
  PORT(
        lista_de_puertos);
  zona_de_declaración
BEGIN
  procesos_pasivos
END ENTITY;
```

Los parámetros (genéricos en terminología VHDL) pueden servir para realizar descripciones genéricas de módulos que se particularizan al emplazarlas en una descripción estructural. Por ejemplo, el siguiente es un modelo genérico de puertas **and**:

```
ENTITY and_generica IS
GENERIC(no_de_entradas: IN INTEGER := 2);
PORT(
        ent : BIT_VECTOR(no_de_entradas – 1 DOWNTO 0)
 sal: BIT);
END ENTITY;

ARCHITECTURE univ_and OF and_generica IS
BEGIN
  PROCESS(ent)
   VARIABLE tmp: BIT;
  BEGIN
    tmp := '1';
    FOR I IN ent'RANGE LOOP
      IF ent(I) = '0' THEN
        tmp := '0';
        EXIT;
      END IF;
    END LOOP;
    sal <= tmp;
  END PROCESS;
END univ_and;
```

Al emplazar el dispositivo en un Cuerpo de Arquitectura estructural se indica el valor que debe tomar el parámetro para cada instancia:

```
U0: ENTITY WORK.and_generica(univ_and)
GENERIC MAP (4);
        PORT MAP      (ent => A(3 DOWNTO 0), sal => nodo1);

U1: ENTITY WORK.and_generica(univ_and)
GENERIC MAP (10);
        PORT MAP      (ent => B(9 DOWNTO 0), sal => nodo2);

U2: ENTITY WORK.and_generica(univ_and)
        PORT MAP      (ent => C(1 DOWNTO 0), sal => nodo3);

U3: ENTITY WORK.and_generica(univ_and)
GENERIC MAP (OPEN);
        PORT MAP      (ent => D(1 DOWNTO 0), sal => nodo4);
```

En estos ejemplos, la instancia **U0** emplaza una puerta **and** de cuatro entradas y **U1** de diez. Si no se indica un valor, como en la instancia **U2**, el genérico toma el especificado en su declaración; también puede indicarse explícitamente que no se desea dar valor al genérico (**U3**) utilizando la palabra **OPEN**.

Otra aplicación interesante de los genéricos es utilizarlos para parametrizar retardos. Por ejemplo, en el modelo de la puerta **and** anterior podría haberse declarado una lista de parámetros como la siguiente:

```
GENERIC(no_de_entradas: IN INTEGER := 2;
            retardo: IN TIME := 5 ns);
```

y en el proceso que modela el funcionamiento haberse hecho la asignación:

```
sal <= tmp AFTER retardo;
```

De esta manera puede utilizarse un mismo modelo para describir puertas con distintos tiempos de propagación:

```
U3: ENTITY WORK.and_generica(univ_and)
GENERIC MAP (no_de_entradas => open, retardo => 2 ns);
        PORT MAP      (ent => D(1 DOWNTO 0), sal => nodo4);

U3: ENTITY WORK.and_generica(univ_and)
GENERIC MAP (no_de_entradas => open, retardo => 7 ns);
        PORT MAP      (ent => D(1 DOWNTO 0), sal => nodo4);
```

Las dos instancias anteriores modelan dos puertas **and** de dos entradas con retardos de 2 y 7 nanosegundos. En este ejemplo la asociación de valores a los genéricos se ha hecho “por nombre”, en lugar de “por posición”.

Para terminar con los genéricos hay que señalar que son, por definición, objetos de tipo **constante** –del mismo modo que, por ejemplo, los puertos de una Declaración de Entidad son **señales**–, pero que, a diferencia de las constantes normales, su valor se especifica en el momento en que se emplaza el dispositivo, debiéndose entender que el valor indicado en su declaración es un valor “por defecto”.

En Declaración de Entidad también pueden definirse objetos y subprogramas que se vayan a utilizar en los Cuerpos de Arquitectura asociados. Esta facilidad no se utiliza casi nunca –por no decir estrictamente NUNCA–. El motivo de su existencia es que podría haber varios Cuerpos de Arquitectura que compartan objetos o subprogramas y si se declaran en la Entidad no hay que hacerlo en cada Cuerpo de Arquitectura. Evidentemente no es una razón de mucho peso, ya que este problema se soluciona fácilmente utilizando Paquetes –y por eso no se usa jamás–, pero en este texto la vamos a utilizar para enlazar la presentación de las funciones y procedimientos VHDL.

2.- Subprogramas

Además de los operadores predefinidos, el lenguaje VHDL permite el uso de subprogramas para el diseño de operaciones de computación y el modelado hardware. El lenguaje proporciona dos tipos de subprogramas: las **funciones** y los **procedimientos**. Estas construcciones son semejantes a las existentes en los lenguajes de programación de alto nivel, diferenciándose por tanto entre sí en que:

- Los parámetros de las funciones son de entrada y por defecto se consideran constantes –no objetos de tipo constante, sino valores a los que no se puede asignar valor–, mientras que los de los procedimientos pueden ser de entrada, salida o bidireccionales (por defecto los parámetros de entrada se consideran constantes y los demás variables –aquí sí se trata de objetos de tipo constante o variable).

- La invocación de una función es una expresión (un operador que devuelve un valor); la de un procedimiento es una sentencia.

Para hacer uso de un subprograma hay que definir –es optativo– su interfaz, que consta de un nombre identificativo y de una lista de parámetros, en una **Declaración de Subprograma** y su funcionalidad –obligatoriamente– en el **Cuerpo del Subprograma**, donde se describe la operación del mismo mediante sentencias secuenciales. Frecuentemente se encapsulan en paquetes, de manera que la interfaz se declara en la Declaración de Paquete –la vista pública– y el cuerpo del subprograma se define en el Cuerpo de Paquete; en general pueden definirse en la zona de declaración de cualquier construcción VHDL (en la Declaración de Entidad, en el Cuerpo de Arquitectura, en procesos, en la zona de declaración de otro subprograma, etc.).

En las funciones se pueden declarar constantes y variables locales, pero no señales, cuyo valor se inicializa en cada llamada. Tampoco se puede asignar valores a señales visibles en la unidad donde se invocan. En los procedimientos, en cambio, sí se puede asignar valores a señales externas o declaradas en el propio procedimiento.

La sintaxis básica de la declaración de funciones y procedimientos es:

FUNCTION nombre_de_función (lista_de_parámetros) RETURN tipo_de_datos;

PROCEDURE nombre_de_procedimiento(lista_de_parámetros);

En el paquete **std_logic_arith** de la librería **ieee** se declaran varías funciones, como por ejemplo:

```
FUNCTION shl(arg: unsigned; count: unsigned) RETURN unsigned;

-- sign extend std_logic_vector (arg) to size,
-- size < 0 is same as size = 0
-- return std_logic_vector(size-1 downto 0)

FUNCTION sxt(arg: std_logic_vector; size: integer) RETURN
std_logic_vector;

FUNCTION "+"(l: unsigned; r: unsigned) RETURN unsigned;

FUNCTION "+"(l: signed; r: signed) RETURN signed;
```

En relación con estas declaraciones resulta interesante señalar lo siguiente:

- Es frecuente encontrar junto a la declaración algunos comentarios que informen sobre la operación realizada por el subprograma, de modo que no resulte necesario revisar el cuerpo de la función o procedimiento para utilizarlos.

- El nombre de la función puede ir entre comillas. En este caso se puede invocar de dos maneras:

  parámetro1 nombre_de_función parámetro2

  o

  nombre_de_función (parámetro1, parámetro2)

  esta modalidad resulta elegante para la representación de operaciones.

- El lenguaje soporta la sobrecarga de operadores. Esto significa que se puede utilizar el mismo nombre para varias funciones siempre y cuando tengan distinto número de parámetros o, teniendo el mismo, sean de distinto tipo. Cuando en una descripción se encuentra un nombre que puede corresponder a varias funciones, se identifica la adecuada por el tipo de parámetros que se le pasan. Teniendo en cuenta las funciones "+" del ejemplo anterior:

  C <= A + B;

  se corresponde con la primera función si **A** y **B** son de tipo **unsigned** y con la segunda si son de tipo **signed**.

Estas mismas consideraciones, exceptuando el uso de comillas en los nombres, se aplican también a los procedimientos. La siguiente declaración de procedimiento está sacada del paquete **TEXTIO** de la librería **STD:**

```
PROCEDURE READLINE(FILE f: TEXT; l: INOUT LINE);
```

Los parámetros que aparecen en las declaraciones se denominan **formales** y deben "conectarse" a objetos **actuales** cuando los subprogramas se invocan en una descripción. Los parámetros formales y actuales que se corresponden en una invocación tienen que coincidir en tipo de datos, dimensiones (si se trata de arrays) y deben respetar reglas de dirección (un formal de tipo **IN** debe asociarse a un actual que pueda leerse, uno de tipo **OUT** a un actual que pueda escribirse, etc.) y –sólo en los procedimientos– de clase de objeto (un formal de tipo señal debe asociarse con una señal, etc.).

La llamada a un subprograma puede hacerse de manera secuencial, siempre, o concurrente, si alguno de sus parámetros actuales es una señal o si es un procedimiento que contiene sentencias **WAIT** –las funciones no pueden incluir este tipo de sentencias–. Por ejemplo, la función **shl** del ejemplo de la página anterior desplaza a la izquierda un objeto de tipo **unsigned**, **arg**, el número de posiciones especificado en **count**. Una invocación concurrente sería:

```
ARCHITECTURE simple OF sh_comb IS
BEGIN
  sal_unsigned <= shl( ent_unsigned, pos_unsigned);

END simple;
```

Suponiendo que los parámetros actuales son puertos de entrada de la Declaración de Entidad, la sentencia concurrente de asignación se ejecutaría cuando hubiera eventos en cualquiera de ellos.

El uso en una sentencia secuencial de la función anterior sería el siguiente:

```
ARCHITECTURE simple OF sh_comb IS
BEGIN

PROCESS
BEGIN
  sal_unsigned <= shl( ent_unsigned, pos_unsigned);
  WAIT ON ent_unsigned, pos_unsigned;
END PROCESS;

END simple;
```

La diferencia entre ambos usos es más clara en el caso de los procedimientos, pues estos son sentencias. Suponga que dispone de un procedimiento que desplaza una señal **BIT_VECTOR** a la izquierda:

```
PROCEDURE shl_bit( SIGNAL ent: IN BIT_VECTOR(NATURAL RANGE <>);
                   desp: IN INTEGER := 1;
                  SIGNAL sal: OUT BIT_VECTOR(NATURAL RANGE<>)
                 );
```

Un uso concurrente del procedimiento sería:

```
ARCHITECTURE simple OF shl_bit IS
BEGIN
 shl(ent_bit, 2, sal_bit);

END simple;
```

En este caso el procedimiento se ejecutaría cada vez que hubiera un evento en **ent_bit**.

El procedimiento se podría utilizar también dentro de un proceso:

```
PROCESS(clk)
BEGIN
 IF clk'EVENT AND clk = '1' THEN
   shl(ent_bit, 3, sal_bit);

 END IF;
END PROCESS;
```

Ahora el procedimiento se ejecutaría cada vez que se produjera un flanco positivo en la señal de reloj.

Como ya se ha dicho, la declaración de procedimientos y funciones es opcional. Esto se debe a que en la definición de los mismos se repite la especificación de su interfaz. La sintaxis de los cuerpos de procedimientos y funciones es:

```
FUNCTION nombre_funcion (lista_de_parámetros) RETURN tipo_datos IS

Zona_de_declaración
BEGIN
Sentencias

END FUNCTION nombre_de_función;

PROCEDURE nombre_procedimiento (lista_de_parámetros) IS

Zona_de_declaración
BEGIN
Sentencias

END PROCEDURE nombre_de_procedimiento;
```

La función **shl** utilizada en ejemplos anteriores está definida en el cuerpo del paquete **std_logic_arith**:

```
FUNCTION shl(arg: unsigned; count: unsigned) RETURN unsigned IS
```

```
CONSTANT control_msb: INTEGER := count'LENGTH - 1;
VARIABLE control: unsigned (control_msb DOWNTO 0);
CONSTANT result_msb: INTEGER := arg'LENGTH-1;
SUBTYPE rtype is unsigned (result_msb DOWNTO 0);
VARIABLE result, temp: rtype;
   BEGIN
control := make_binary(count);
IF (control(0) = 'x') THEN
   result := rtype'(OTHERS => 'x');
   RETURN result;
END IF;
result := arg;
FOR i IN 0 TO control_msb LOOP
   IF control(i) = '1' THEN
      temp := rtype'(OTHERS => '0');
      IF 2**i <= result_msb THEN
        temp(result_msb DOWNTO 2**i) :=   result(result_msb - 2**i DOWNTO
0);
      END IF;
      result := temp;
   END IF;
END LOOP;
RETURN result;
   END;
```

Puede observarse que se utilizan las mismas construcciones que en los procesos y que el valor devuelto por la función se indica en una sentencia **RETURN**, que , además, finaliza la ejecución de ésta.

El cuerpo del procedimiento **shl_bit** podría ser, por ejemplo, este:

```
PROCEDURE shl_bit( SIGNAL ent: IN BIT_VECTOR(NATURAL RANGE <>);
                          desp: IN INTEGER := 1;
                   SIGNAL sal: OUT BIT_VECTOR(NATURAL RANGE<>)
                  ) IS
  CONSTANT rsup: NATURAL := ent'LENGTH – 1;
  VARIABLE tmp: BIT_VECTOR(rsup DOWNTO 0);
BEGIN
tmp := ent;
FOR I IN 1 TO desp LOOP
 FOR J IN tmp'RANGE LOOP
   IF J /= 0 THEN
    tmp(J) := tmp(J-1);
   ELSE
    tmp(J) := '0';
   END IF;
 END LOOP;
END LOOP;
sal <= tmp;

END PROCEDURE shl_bit;
```

3.- Procesos pasivos. Sentencias ASSERT

En la Declaración de Entidad puede haber procesos pasivos. Un proceso pasivo es aquel en el que no se asigna valor a señales. Hay un aspecto del hardware cuyo modelado puede realizarse con procesos pasivos: la comprobación de condiciones de funcionamiento. El lenguaje VHDL proporciona una sentencia que permite realizar esta función: la sentencia **ASSERT**; puede ejecutarse secuencial y concurrentemente. Su sintaxis es:

```
ASSERT expresión_booleana REPORT string
valor_de_tipo_SEVERITY_LEVEL;
```

Un ejemplo de aplicación de esta sentencia puede ser la comprobación del tiempo de **set-up** en un **flip-flop**:

```
PROCESS(clk)
BEGIN
 IF clk'EVENT AND clk = '1' THEN
  ASSERT d'STABLE(tsu) REPORT ("violación de tiempo de set-up")
  SEVERITY WARNING;
 END IF;
END PROCESS;
```

Cuando la condición booleana se evalúa a **FALSE**, se escribe el **string** especificado en la consola del simulador. Si el valor de tipo **SEVERITY_LEVEL** incluido en la sentencia es **ERROR** o **FAILURE**, suele abortarse la simulación.

La versión concurrente del proceso anterior no es tan fácil de interpretar:

```
ASSERT NOT(clk'EVENT AND clk = '1') OR d'STABLE(tsu)
REPORT ("violación de tiempo de set-up")
SEVERITY WARNING;
```

Esta sentencia se ejecutaría cuando hubiera eventos en **clk** o **d**. Si recuerda la discusión mantenida sobre la fórmula **clk'EVENT AND clk = '1'** en el capítulo anterior, trate de dilucidar si **clk'EVENT** es redundante o no.

4.- Ejercicio IV.1

En el siguiente ejercicio se va a simular el modelo de un registro parametrizable en tamaño y tiempos. El modelo del registro es:

```
ENTITY reg_param IS
GENERIC(tp: IN TIME := 2 ns;
  tsu: IN TIME := 1 ns;
  th: IN TIME := 0 ns;
  Tclkmin: IN TIME := 3 ns;
  N: IN NATURAL := 8);

PORT(clk, reset: IN BIT;
        dato_in: IN BIT_VECTOR(N-1 DOWNTO 0);
        dato_out: OUT BIT_VECTOR(N-1 DOWNTO 0));

BEGIN
```

```
PROCESS(clk, dato_in)
 VARIABLE t_pos: TIME := 0 ns;
BEGIN
 IF clk'event and clk = '1' THEN
  IF NOW > Tclkmin THEN
   ASSERT (NOW - t_pos >= Tclkmin)
   REPORT ("La frecuencia de reloj es mayor que la mínima admisible")
   SEVERITY FAILURE;
  END IF;
  t_pos := NOW;

  ASSERT dato_in'stable(tsu)
  REPORT "Violación de tiempo de set-up del registro"
  SEVERITY WARNING;

 END IF;

 IF dato_in'EVENT THEN
   ASSERT (NOW - t_pos) >= th
   REPORT "Violación de tiempo de hold del registro"
   SEVERITY WARNING;
 END IF;
END PROCESS;

END ENTITY;



ARCHITECTURE ctrl_tiempos OF reg_param IS
BEGIN

 PROCESS(clk, reset)
 BEGIN
  IF reset = '1' THEN
   dato_out <= (OTHERS =>'0') AFTER tp;
  ELSIF clk'EVENT AND clk = '1' THEN
   dato_out <= dato_in AFTER tp;
  END IF;
 END PROCESS;
END ctrl_tiempos;
```

Para modelar el control de los tiempos del registro se utilizan atributos de señal y la función **NOW**, declarada en el paquete **STANDARD**, que devuelve el valor del tiempo actual de simulación. El código del ***test-bench*** es el siguiente:

```
ENTITY test_reg IS
END ENTITY;

LIBRARY ieee;
USE ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
ARCHITECTURE test OF test_reg IS

 CONSTANT tp: TIME := 4 ns;
 CONSTANT tsu: TIME := 2 ns;
 CONSTANT th: TIME := 1 ns;
 CONSTANT Tclkmin: TIME := 6 ns;
 CONSTANT Tclktest: TIME := 10 ns;
 CONSTANT N_test: NATURAL := 4;

 SIGNAL reset, clk: BIT := '0';
 SIGNAL dato_in, dato_out: BIT_VECTOR( N_test-1 DOWNTO 0);

 PROCEDURE gen_clk(signal clk: INOUT BIT; Tclk: IN TIME) IS
 BEGIN
```

```
  clk <= NOT clk AFTER Tclk/2;
 END PROCEDURE gen_clk;

 FUNCTION inc( ent: IN BIT_VECTOR)
 return BIT_VECTOR IS

  CONSTANT rsup: NATURAL:= ent'length;
  VARIABLE tmp:std_logic_vector(rsup-1 DOWNTO 0);
 BEGIN
  tmp := To_StdLogicVector(ent);
  tmp := tmp + 1;
  RETURN To_bitvector(tmp);
 END FUNCTION inc;
BEGIN

 gen_clk(clk => clk, Tclk => Tclktest);

 PROCESS
  VARIABLE t_ini: TIME := th;
 BEGIN
   WAIT UNTIL clk'EVENT AND clk = '1';
    reset <= '1' AFTER 2 ns;
   WAIT UNTIL clk'EVENT AND clk = '1';
    reset <= '0' AFTER 2 ns;

   LOOP
    WAIT UNTIL clk'EVENT AND clk = '1';
    dato_in <= inc(dato_in) AFTER t_ini;
    t_ini := t_ini + 500 ps;
   END LOOP;
 END PROCESS;

 DUT: ENTITY WORK.reg_param(ctrl_tiempos)
     GENERIC MAP(tp, tsu, th, Tclkmin, N_test)
     PORT MAP(clk, reset, dato_in, dato_out);

END test;
```

En el ***test-bench*** se define un procedimiento para generar un reloj con una determinada frecuencia y una función que incrementa valores de tipo **BIT_VECTOR**. Observe que en este ejemplo se utilizan constantes para asignar valor a los genéricos de la instancia del registro.

1. Edite los ficheros en un nuevo espacio de trabajo, compílelos y arranque una simulación.

2. Abra una ventana para revisar las formas de onda y ordene que la simulación avance **300 ns**.

3. Lea los mensajes que aparecen en la parte inferior de la ventana (figura e1). Corresponden a informes de sentencias **ASSERT**.

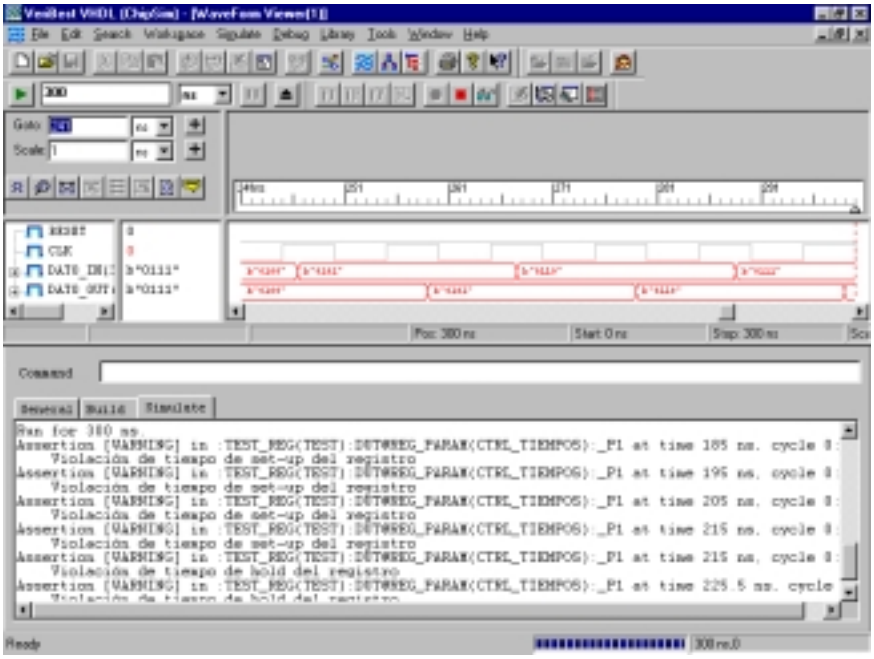


Figura e1

4. Compruebe con los cursores de medida, contrastando valores con los parámetros especificados en el ***test-bench***, que los mensajes detectan correctamente violaciones en los tiempos de ***hold*** y ***set-up*** del modelo.

5. Termine la simulación.

6. Modifique en el ***test-bench*** el valor de la constante **Tclktest** de modo que valga **4 ns**.

7. Recompile el código y arranque una simulación.

8. Ordene que la simulación avance **300 ns**.
9. Observe que se abre una ventana indicando que por petición de una sentencia **ASSERT** hay que abortar la simulación (figura e2).

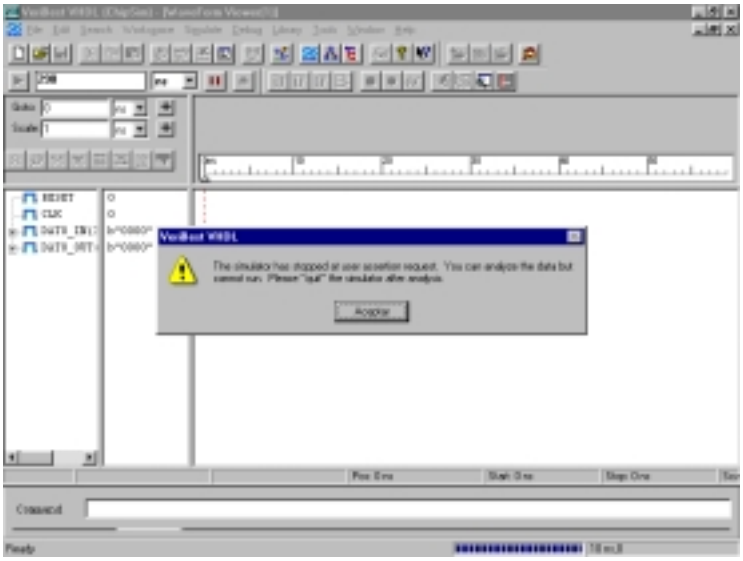


Figura e2

10. Si revisa la ventana de mensajes verá que se ha detectado que la frecuencia de reloj es inferior a la mínima de funcionamiento del registro. Compruébelo y salga de la simulación.

    El simulador **VeriBest** permite que el usuario pueda determinar cual es el valor **SEVERITY_LEVEL** que una sentencia **ASSERT** activa debe incluir para que se aborte la simulación. Puede especificarse en la opción **Settings** del menú **Workspace** (figura e3).

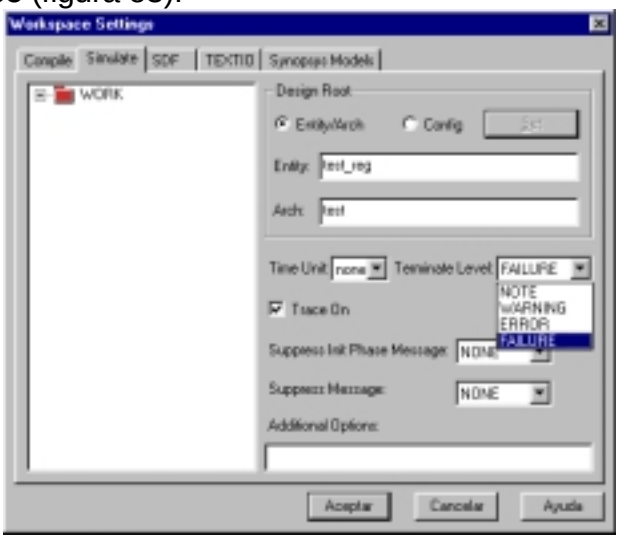


Figura e3

11. Puede comprobar que si repite la última simulación después de cambiar en el control de la frecuencia de reloj **FAILURE** por **ERROR**, simplemente aparecen mensajes en la ventana. Si a continuación modifica el nivel de terminación de la simuilación a **ERROR**, el simulador volverá a repetir el mensaje de la figura e2.

12. Si lo desea, puede realiza simulaciones dando distintos valores a los parámetros. Cuando termine, cierre la herramienta.

5.- Sentencias de descripciones estructurales

La sintaxis de una declaración de componente puede incluir una lista de genéricos para poder emplazar dispositivos parametrizables (en el último ejercicio se ha utilizado en el emplazamiento directo del registro):

```
COMPONENT nombre_de_componente IS
GENERIC lista_de_genéricos;
PORT   lista_de_puertos;
END COMPONENT;
```

El componente utilizado para emplazar un dispositivo no tiene porque corresponderse exactamente con la interfaz de éste. Observe el siguiente código:ç

```
COMPONENT reg_univ IS
 GENERIC(tpLH, tpHL: IN TIME := 0 ns;
     tsetup, thold: IN TIME := 0 ns;
    Tclkmin: IN TIME := 0 ns;
    N: IN NATURAL := 8);

 PORT( clk: IN BIT;
          reset, preset: IN BIT := '0';
ctrl_func: IN BIT_VECTOR(1 DOWNTO 0) := "00";
din_izqda, din_dcha: IN BIT := '0';
din_paralelo: IN BIT_VECTOR;
dout: OUT BIT_VECTOR);

 END COMPONENT;
```

Este componente puede utilizarse, por ejemplo, para emplazar instancias del modelo del registro **reg_param**. Para ello hay que especificar en la configuración de las instancias la conexión entre formales de la Declaración de Entidad emplazada y actuales del componente. En el caso del registro parametrizable, la especificación de configuración quedaría:

```
FOR DUT: reg_univ USE ENTITY WORK.reg_param(ctrl_tiempos)
    GENERIC MAP (tp => tpLH,
                 tsu => tsetup,
                 th => thold,
                 Tclkmin => Tclkmin,
                 N => N)
    PORT MAP( clk => clk,
              reset => reset,
              dato_in => din_paralelo,
              dato_out => dout);
```

Observe que la lista de asociación indica, en definitiva, la correspondencia entre parámetros y puertos del componente y la entidad. En la

configuración por defecto, que se explicó en el capítulo III, la asociación se establece automáticamente y por eso no se precisa configurar las instancias de componentes. Las especificaciones de configuración pueden referirse, como en el ejemplo anterior, a una sola instancia, a varias o a todas las que haya de un determinado componente; las fórmulas a emplear son:

```
FOR U0, U1, U2: nombre_de_componente...

FOR ALL: nombre_de_componente...
```

Al emplazar el componente hay que especificar la conexión de todos sus puertos:

```
DUT: reg_univ
    GENERIC MAP(tpLH => tp,
                tpHL => OPEN,
                tsetup => tsu,
                thold => th,
                Tclkmin => Tclkmin,
                N => N_test)
    PORT MAP(clk => clk,
          reset => reset,
          preset => OPEN,
          ctrl_func => OPEN,
          din_izqda => OPEN,
          din_dcha => OPEN,
          din_paralelo => dato_in,
          dout => dato_out);
```

Observe que los puertos del componente que no están conectados a ninguno de la Declaración de Entidad del dispositivo emplazado están asociados a **OPEN**.

También podría emplazarse el registro parametrizable con el siguiente componente:

```
COMPONENT reg IS
GENERIC(tp: IN TIME := 0 ns;
    N: IN NATURAL := 8);
PORT(clk: IN BIT;
din_paralelo: IN BIT_VECTOR;
dout: OUT BIT_VECTOR);
END COMPONENT;
```

En este caso, la especificación de configuración sería:

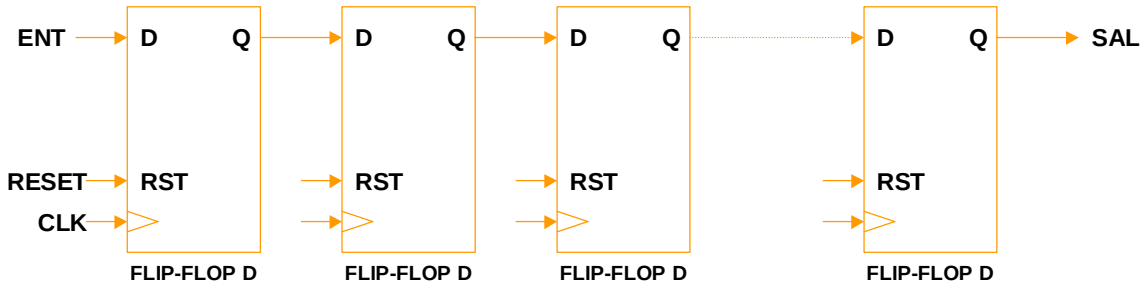
```
FOR DUT: reg USE ENTITY WORK.reg_param(ctrl_tiempos)
   GENERIC MAP(tp => tp,
                  N => N)
   PORT MAP(clk => clk,
               reset => '0',
               dato_in => din_paralelo,
               dato_out => dout);
```

Los parámetros de la Declaración de Entidad que no se corresponden con puertos de los componentes toman los valores especificados en su declaración. Observe que el puerto **reset**, que no se usa, se conecta a **'0'**; podría haberse dejado sin conectar (**OPEN**) si en su declaración se le hubiera asignado un valor inicial. La sentencia de emplazamiento de una instancia del componente sería:

```
DUT: reg
                GENERIC MAP(tp => tp,
                            N => N_test)
                PORT MAP(clk => clk,
                      din_paralelo => dato_in,
                      dout => dato_out);
```

6.- Sentencias GENERATE

Para terminar con la revisión de las sentencias utilizadas en las descripciones estructurales se va a presentar la sentencia **GENERATE**. Esta sentencia sirve para generar estructuras regulares de conexión con un estilo compacto y parametrizable; por ejemplo, si se desea modelar un registro de desplazamiento como el de la figura con un estilo estructural.



Suponiendo que se dispone del modelo de un **flip-flop** tipo D cuyo modelo VHDL es:

```
ENTITY ffd IS
PORT( clk, reset: IN BIT;
 D: IN BIT;
 Q: OUT BIT);
END ENTITY;

ARCHITECTURE rtl OF dff IS
BEGIN
 PROCESS(clk, reset)
 BEGIN
  IF reset = '1' THEN
   Q <= '0';
  ELSIF clk'EVENT AND clk = '1' THEN
   Q <= D;
  END IF;
 END PROCESS;
END RTL;
```

El registro de desplazamiento podría describirse de la siguiente manera:

```
ENTITY reg_desp IS
GENERIC(N: IN NATURAL := 4);
port(clk, reset: IN bit;
    d_in: IN bit;
    d_out: OUT bit);
END ENTITY;


ARCHITECTURE estructural OF reg_desp IS

COMPONENT dff IS
 PORT(clk, reset: in BIT;
     d: in BIT;
     q: out BIT);
 END COMPONENT;

SIGNAL nodo_int: BIT_VECTOR(N-1 DOWNTO 0);

BEGIN
gen_reg:FOR I IN 0 TO N-1 GENERATE
 gen_izqda:IF I = 0 GENERATE
               U0: dff PORT MAP(clk, reset, d_in, nodo_int(0));
              END GENERATE;

  gen_dcha:IF I = N-1 GENERATE
                U0: dff PORT MAP(clk, reset, nodo_int(I-1), d_out);
               END GENERATE;

   gen_int:IF I /= 0 AND I /= (N - 1) GENERATE
             U0: dff PORT MAP(clk, reset, nodo_int(I -1), nodo_int(I));
            END GENERATE;

          END GENERATE;

END estructural;
```

En el ejemplo se muestran los dos tipos de sentencias **GENERATE** que soporta el lenguaje. La sentencia **FOR** se utiliza para generar la estructura regular; la sentencia **IF** para especificar excepciones a la regla de conexión. En las sentencias **GENERATE** es obligatorio incluir la etiqueta.

En el ejemplo anterior es interesante observar que el modelo no es válido si el valor del parámetro **N** es 1, ya que entonces el bus **nodo_int** tiene como rango **(0 DOWNTO 0)** –esto sí es válido– y la referencia **nodo_int(I-1)** es incorrecta (**I-1** vale **–1**, que está fuera del rango). Si intentara emplazar una instancia de **reg_desp** con **N = 1**, un simulador VHDL reportaría un error al elaborar el modelo para simularlo (no al compilar el modelo).

El ejemplo anterior muestra las características generales de este tipo de sentencias. Sólo hay que añadir dos detalles de interés:

- En la versión del lenguaje de 1993, se admite que la sentencia tenga una zona de declaración –para declarar localmente, por ejemplo, las señales que se utilizan para generar la arquitectura–. En este caso la sintaxis de la sentencia es:

```
etiqueta: FOR/IF... GENERATE
zona_de_declaración
BEGIN
 sentencias concurrentes;
END GENERATE;
```

- En la zona de descripción puede haber no sólo sentencias de emplazamiento de instancias de componentes, sino cualquier tipo de sentencia concurrente –en VHDL se considera que la sentencia de emplazamiento de instancias es también una sentencia concurrente, como los procesos o las propiamente denominadas sentencias concurrentes, puesto que en definitiva coloca los procesos que describen el comportamiento del dispositivo emplazado–. Por ejemplo, una batería de puertas **and** se modelaría del siguiente modo:

```
-- A, B y C están declaradas como BIT_VECTOR(1 TO N)

Puertas_and: FOR I IN 1 TO N GENERATE
              BEGIN
             C(I) <= A(I) AND B(I);
              END GENERATE;
```

MODELADO PARA SÍNTESIS

0.- Resumen del Capítulo

Conceptos Teóricos:

- *Reglas generales y Tipos de Datos.*
- *La Declaración de Entidad*
- *Cuerpos de Arquitectura Combinacionales.*
- *Cuerpos de Arquitectura Secuenciales.*
- *Cuerpos de Arquitectura Estructurales.*
- *Consejos Prácticos.*

Este capítulo aborda la realización de modelos sintetizables. Se presentan reglas de carácter general que permiten que los modelos puedan ser sintetizados por la mayoría de las herramientas actualmente existentes. Se aborda por separado el modelado de circuitos combinacionales y secuenciales y se hace un énfasis especial en la descripción de autómatas. Además se ofrecen una serie de consejos que pueden facilitar la realización eficiente de este tipo de modelos.

1.- Modelos VHDL para síntesis lógica

La aplicación más importante de los lenguajes de descripción hardware es la elaboración de modelos sintetizables automáticamente. Los modelos que es capaz de interpretar una herramienta de síntesis lógica automática actual deben atenerse a unas reglas que restringen el estilo que se puede emplear en la descripción del hardware; en general cada herramienta impone sus propias normas, pero hay un conjunto de esquemas de codificación que son aceptados por cualquiera: se van a presentar en este capítulo. Además se dará algún consejo de carácter práctico que puede ayudar a los diseñadores con poca experiencia en el uso de estas metodologías de diseño.

2.- Reglas de carácter general

Las herramientas de síntesis lógica pueden clasificarse en dos grupos:

- Sintetizadores de "propósito general" (**Synopsys**, **Synplicity**, etc.), que permiten sintetizar circuitos para un amplio abanico de tecnologías.
- Sintetizadores integrados en entornos de CAD de fabricantes de silicio (**MAX+plus II** de **ALTERA**, etc.).

Los primeros son más complejos y suelen tener una mejor calidad que los segundos. En cualquier caso, todos ellos proporcionan una guía de estilo para la realización de modelos sintetizables cuya lectura es obligatoria. En estas guías se encuentran consejos que permiten optimizar el funcionamiento de la herramienta (por ejemplo, comentarios VHDL que permiten dirigir el proceso de síntesis, como los **pragma** de **Synopsys**), así como, en casi todos los casos, "defectos" o "rarezas" específicas de la herramienta (en el entorno **MAX+plus II** se informa, por ejemplo, que no se soporta la configuración de componentes ni el emplazamiento directo de componentes de la versión del año 1993).

Además, debe revisarse en cada caso la versión que se soporta de los paquetes de la librería **IEEE**. En casi todos los casos se dispone de la versión de **Synopsys**, pero esto no tiene porqué ocurrir siempre.

En muchas ocasiones será suficiente con echar un vistazo a la información de la herramienta para poder sintetizar con éxito modelos que se atengan a las normas que se exponen a lo largo de este capítulo; cuando los resultados no sean satisfactorios habrá que profundizar en su estudio. En general, al margen de lo ya señalado, para obtener un buen rendimiento de un sintetizador lógico hay que conocer con detalle las restricciones que se pueden especificar, en los propios modelos o en la configuración de cada herramienta, para optimizar la velocidad o el área del circuito u obtener cualquier otra característica deseable. Si se dispone de un sintetizador de propósito general, merece la pena invertir esfuerzo en aprender sus secretos y peculiaridades, ya que podrá aplicarse, con las librerías de síntesis adecuadas, a distintas tecnologías.

Resumiendo, deben siempre consultarse tres cuestiones relativas a la herramienta de síntesis que se utilice:

1. Los contenidos de la guía de estilo.

2. Los paquetes que contiene la librería IEEE.

3. Las restricciones disponibles para dirigir el proceso de síntesis.

3.- Tipos de Datos en los modelos sintetizables

Se deben utilizar los tipos de datos declarados en los paquetes de síntesis de **IEEE**: **STD_LOGIC,** para señales, y **STD_LOGIC_VECTOR** para grupos de señales. Ya se han descrito sus ventajas en capítulos anteriores.

Como ya se sabe, deberán incluirse antes de la cabecera de las Declaraciones de Entidad, las sentencias de visibilidad:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

Si en la especificación se van a utilizar operaciones aritméticas o de desplazamiento y suponiendo que se dispone de la versión de **Synopsys** de los paquetes, deberá añadirse una cláusula de visibilidad adicional para el paquete que proceda:

- **std_logic_unsigned** si se desea que los vectores **STD_LOGIC** representen números sin signo;
- **std_logic_signed** si se desea que los vectores **STD_LOGIC** representen números con signo.

Si en algún caso –raro– no hay más remedio que mezclar en un mismo módulo aritmética con y sin signo, deberán utilizarse los tipos **SIGNED** y **UNSIGNED**. En este caso hay que obtener visibilidad sobre el paquete **std_logic_arith**.

Resulta interesante señalar que en muchas guías de estilo se aconseja el uso de subtipos enteros para modelar grupos de señales que representan números. Por ejemplo, un número binario natural de tres bits se declararía:

```
numero: INTEGER RANGE 0 TO 7;
```

Esto puede ser aceptable en ocasiones; el problema que presenta es que se haga así y haya que segregar algún bit del grupo: no puede hacerse sin utilizar funciones de conversión de tipos, lo que puede acarrear problemas (algunos sintetizadores no son capaces de procesarlas). En definitiva, es aconsejable desechar esta recomendación.

4.- Declaraciones de Entidad

La realización de la Declaración de Entidad requiere que se tomen sólo un par de precauciones especiales. Al margen de estas, que se indicarán al final, es conveniente recordar algunas reglas básicas que deben guardarse siempre:

1. Es conveniente que el nombre de la Declaración de Entidad sea el mismo que el del fichero que la contiene y haga referencia, en la medida de lo posible, a la funcionalidad del circuito.

2. Los nombres de los puertos deben estar relacionados con su funcionalidad (**clk**, **reset**, etc.); es conveniente disponer de algún código que permita identificar el nivel de actividad de las señales (por ejemplo, que el nombre de las señales activas a nivel bajo termine en ***n*** *o* ***_n***: **reset*n*** o **reset_*n***). También resulta aconsejable comentar brevemente la función de cada puerto.

3. La dirección del puerto se determina simplemente a partir de las características del circuito (los pines de entrada se definen como puertos de entrada, **IN**, los de salida, **OUT** y los bidireccionales **INOUT**. Aquí debe recordarse que en ocasiones resulta conveniente definir algunos puertos de salida con dirección **BUFFER**. Esto suele ocurrir en circuitos secuenciales en que hay que utilizar la salida para el cálculo del próximo estado y en circuitos combinacionales o secuenciales en los que alguna salida se describe fácilmente a partir de otra (circuitos con dos salidas, por ejemplo, en la que una es la negada de la otra).

4. Es muy aconsejable que los rangos de los puertos que sean grupos de señales se definan siempre con el mismo sentido, ascendente o descendente; preferentemente debe escogerse el sentido descendente, de este modo si el vector representa un número binario, el elemento correspondiente al índice más alto del *array* será el que represente el bit de mayor peso.

Todas estas consideraciones sobre el estilo de descripción de la Declaración de Entidad son aplicables a cualquier modelo. En el caso de descripciones sintetizables hay que tener en cuenta, además, lo siguiente:

- No debe indicarse un valor inicial en la Declaración de los puertos; algunos sintetizadores detectan esto como un error.

- Se pueden utilizar modelos parametrizables (con lista de genéricos), pero debe tenerse en cuenta que los que afectan al proceso de síntesis son los que manejan dimensiones del circuito; los genéricos que se utilizan para controlar retardos se ignoran, ya que las herramientas de síntesis lógica no procesan los retardos especificados en sentencias de asignación de valor a señal –admiten sin mayores problemas que aparezcan en el código pero los ignoran–.

A continuación se comentan algunos ejemplos de Declaraciones de Entidad.

SUMADOR DE CUATRO BITS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;

ENTITY sum4b IS
PORT(
        sum1, sum2: in std_logic_vector(3 downto 0);
        res : out std_logic_vector(3 downto 0);
        acarreo : out std_logic
    );
END sum4b;
```

En este primer ejemplo, un sumador de números binarios de cuatro bits con acarreo se cumplen todas las normas.

En este otro ejemplo, un contador, se muestra una práctica indeseable, la mezcla caprichosa de tipos de datos.

CONTADOR DE CUATRO BITS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all, ieee.std_logic_arith.all;

ENTITY cont4b IS
PORT(
        clk, rst            : IN BIT;
        up_down, enable     : IN std_logic;
        cont                : BUFFER std_logic_vector(3 downto 0)
    );
END cont4b;
```

Por último, se muestra un ejemplo en el que se hace uso de una lista de genéricos. La descripción es correcta y sintetizable. Si el módulo se sintetiza por separado, se generará un circuito con las dimensiones por defecto del

parámetro; si se coloca en un Cuerpo de Arquitectura, con el valor que se especifique en la sentencia de emplazamiento.

COMPARADOR PARAMETRIZABLE

```
LIBRARY ieee;
USE ieee.std_logic_1164.all, ieee.std_logic_arith.all;

ENTITY comp IS
GENERIC( n: IN integer RANGE 0 TO 31 := 4);
PORT(
            agbin, aeqbin, aminbin: IN std_logic;
            a, b : IN  std_logic_vector(n DOWNTO 0);
            agb, aeqb, aminb  : OUT std_logic
      );
END comp;
```

5.- Modelado del funcionamiento

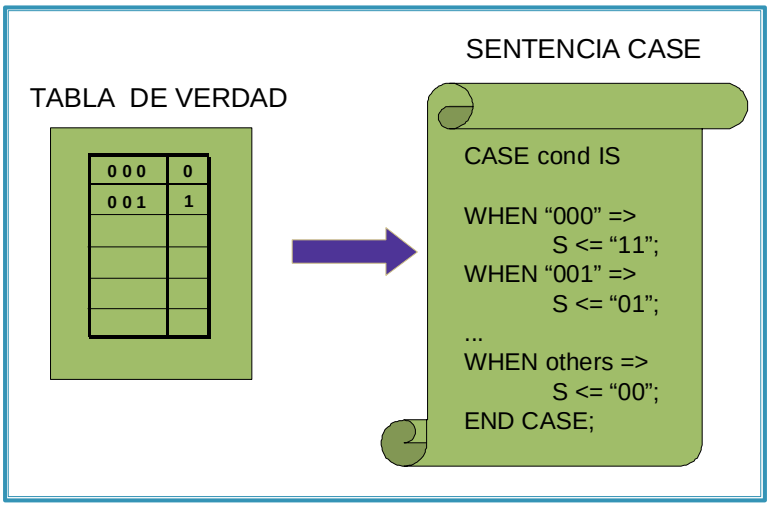
A continuación se describen las reglas generales que deben seguirse para la realización de modelos sintetizables RTL de circuitos combinacionales y secuenciales. Más adelante se darán algunos consejos para la realización de descripciones estructurales.

6.- Realización de Arquitecturas sintetizables de Circuitos Combinacionales

Los Cuerpos de Arquitectura de modelos de circuitos combinacionales contendrán procesos (o sentencias concurrentes) que describan el funcionamiento del circuito. Para realizar un proceso sintetizable de un circuito combinacional deben seguirse las siguientes reglas:

1. El proceso debe ser sensible a todas las entradas del circuito combinacional. Es aconsejable utilizar el formato con lista de sensibilidad en la cabecera del proceso; si se prefiere usar sentencias **WAIT ON**, deberá haber una única sentencia.

2. Dentro del proceso deberán utilizarse las sentencias necesarias de modo que se especifique para cada combinación de las entradas el valor de las salidas; es decir, debe traducirse a código VHDL la tabla de verdad completa del circuito. Esto puede hacerse muy fácilmente con combinaciones de sentencias **CASE** e **IF**.



La clave para realizar eficientemente procesos combinacionales sintetizables consiste en repetir siempre el mismo patrón de codificación. Normalmente una sentencia **CASE** o varias sentencias **CASE** anidadas permiten describir muy fácilmente cualquier circuito combinacional. Si el circuito dispone de entradas de habilitación puede utilizarse una sentencia **IF** auxiliar:

```
IF condición_de_habilitación THEN
        CASE
                Modelado_de_habilitación
        END CASE;
ELSE
        Modelado_de_no_habilitación
END IF;
```

Importante: Los errores más comunes en la descripción de procesos combinacionales suelen consistir en la omisión de alguna de las entradas del circuito en la lista de sensibilidad –suelen ser, por cierto, las entradas de control las que se “esfuman” más fácilmente–, la ausencia de la especificación de alguna salida para alguna combinación de entrada y, más raramente, la no especificación de algún conjunto de combinaciones de entrada. Los tres errores pueden ser o no detectados por la herramienta de síntesis –depende de cada una–; si lo son, se corrigen, si no pueden ser muy peligrosos, sobre todo los dos últimos, porque pueden pasar desapercibidos y provocan que se sintetice un circuito combinacional realimentado o con ***latches***. Hay que tener mucho cuidado para no cometerlos nunca.

Para no tener problemas es muy conveniente repetir siempre los mismos patrones de codificación y no confundir elegancia o brevedad con eficiencia. También resulta aconsejable huir del uso de bucles y subprogramas en la medida de los posible, aunque todos los sintetizadores permiten su uso con mayores o menores restricciones. Y no confundir el lenguaje VHDL con un lenguaje de programación. Muchas descripciones algoritmicas perfectamente válidas (desde el punto de vista del modelado del procesamiento) son ineficientes o imposibles de sintetizar.

Por último, hay que señalar algunas cuestiones importantes:

1. Los circuitos combinacionales aritméticos se modelan con las operaciones correspondientes. Los sintetizadores lógicos son capaces de procesar las operaciones de suma, resta, multiplicación y, normalmente, división por una potencia de 2 (en esencia, un desplazamiento). También entienden cualquier operación lógica básica (si se quiere definir la función combinacional algebraicamente) y las de desplazamiento y rotación.

2. Si el proceso equivale a una sentencia concurrente puede utilizarse ésta última sin ningún problema.

3. Los retardos especificados en asignaciones a señal son ignorados por las herramientas de síntesis, pero puede especificarlos sin mayores problemas.

Los siguientes ejemplos ilustran las reglas anteriores.

Ejemplo: Multiplexor de 4 Canales

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux4a1 IS
PORT(
  datos: IN std_logic_vector(3 DOWNTO 0 );
  dir: IN std_logic_vector (1 DOWNTO 0 );
  sal: OUT std_logic);
END mux4a1 ;
```

```
ARCHITECTURE rtl OF mux4a1 IS
BEGIN
    PROCESS( datos, dir)
    BEGIN
    CASE dir IS
    WHEN “00” => sal <= datos(0);
    WHEN “01” => sal <= datos(1);
    WHEN “10” => sal <= datos(2);
    WHEN “11” => sal <= datos(3);
    WHEN OTHERS =>
                    sal <= ‘X’;
    END CASE;
    END PROCESS;
END rtl;
```

El modelo de la figura describe un multiplexor. Observe que una sentencia **CASE** permite representar muy fácilmente la tabla de verdad del circuito. En muchas ocasiones al utilizar sentencias **CASE** se agrupan varias combinaciones de entrada –si tienen asociada la misma salida– para resumir el código (en el ejemplo, en la cláusula **OTHERS**). La cláusula **OTHERS** es obligatoria, puesto que en las sentencias **CASE** deben especificarse todas las combinaciones posibles –una ventaja adicional de las sentencias CASE es que permiten detectar el tercero de los errores mencionados antes– y la selección se hace con un vector de tipo **STD_LOGIC_VECTOR** que puede tomar más valores que las combinaciones de **‘0’** y **‘1’**. Para expresar indiferencia en los valores de salida puede utilizarse el valor **‘-‘** o **‘X’**. El primero es el más apropiado de acuerdo con la semántica del tipo de datos, pero su uso se evita en ocasiones porque al simular el modelo puede resultar complicada su monitorización en los visores de formas de onda.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all, ieee.std_logic_signed.all;
ENTITY sum_rest IS
PORT (
        ctrl : IN std_logic ;
        sum1, sum2 : IN std_logic_vector(3 downto 0);
        result : OUT std_logic_vector(3 downto 0);
END ENTITY;

ARCHITECTURE rtl OF sum_rest IS
BEGIN
  result <= sum1 + sum2 WHEN ctrl = '0'
       ELSE sum1 - sum2 ;
END rtl ;
```

Este otro es un ejemplo de un circuito sumador-restador de cuatro bits. La sentencia concurrente es equivalente al proceso:

```
PROCESS(sum1, sum2, ctrl)
BEGIN
 IF ctrl = '0' THEN
   result <= sum1 + sum2;
ELSE
   result <= sum1 – sum2
 END IF;
END PROCESS;
```

En el siguiente ejemplo se comete uno de los errores que se señalaron antes: no se asignan valores a todas las salidas para cada combinación de entrada. El modelo implica la existencia de memoria en el circuito –que por lo demás terminaría con todas sus salidas a **'1'**–.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
ENTITY comp IS
PORT (
       A, B : IN std_logic_vector(3 downto 0);
       a_mn_b, a_eq_b, a_my_b : OUT std_logic );
END ENTITY;

ARCHITECTURE simple OF comp IS
BEGIN
  PROCESS(A, B );
     BEGIN
       IF A > B THEN a_my_b <= '1';
           ELSIF A < B THEN a_mn_b <= '1';
           ELSE a_eq_b <= '1';
       END IF;
  END PROCESS;
END simple ;
```

Una versión correcta del proceso que modela el circuito comparador sería:

```
PROCESS(a, b)
 BEGIN
  IF a > b THEN
   a_my_b_o <= '1';
   a_mn_b_o <= '0';
   a_eq_b_o <= '0';

  ELSIF a < b THEN
   a_my_b_o <= '0';
   a_mn_b_o <= '1';
   a_eq_b_o <= '0';

  ELSE
   a_my_b_o <= '0';
   a_mn_b_o <= '0';
   a_eq_b_o <= '1';
 END IF;
END PROCESS;
```

Es interesante observar como se agrupan combinaciones de entrada mediante condiciones lógicas en ramas de la sentencia IF.

El siguiente ejemplo muestra el tipo de complicaciones de estilo que deben evitarse.

```
ENTITY Det_Paridad IS
PORT(
        d_in: IN BIT_VECTOR(7 TO 0);
        par, impar: OUT BIT);
END Det_Paridad;

ARCHITECTURE Comport OF Det_Paridad IS
BEGIN
 PROCESS(d_in)
  VARIABLE aux: BIT;
  BEGIN
    aux := '0';
    FOR I IN d_in'RANGE LOOP
     aux := aux XOR d_in(I);
    END LOOP;
     impar <= aux;
     par <= NOT aux;
 END PROCESS;
END Comport;
```

El mismo circuito podría modelarse fácilmente declarando el puerto **par** con dirección **BUFFER** y sustituyendo el proceso por las sentencias concurrentes:

```
impar <= d_in(0) XOR d_in(1) XOR d_in(2) ... XOR d_in(7);
```

par <= NOT impar;

7.- Salidas con control de tercer estado

En ocasiones habrá que modelar salidas con control de tercer estado (alta impedancia). En estos casos es conveniente –muchas veces resulta obligatorio– separar el modelado de la operación lógica del de las salidas, como se muestra en el siguiente ejemplo.

```
ARCHITECTURE Compor OF Bus IS
 SIGNAL A : std_logic;
 BEGIN

  A <= B AND C;
  bus_Z1  <= A WHEN ctrl1 = '1'
                ELSE 'Z';
END Compor
```

B, **C**, **ctrl1** y **bus_Z1** son puertos de la declaración de entidad. La señal **A** sirve para separar el modelado de las salidas de la función combinacional habilitada. En general, para modelar las salidas con control de alta impedancia debe utilizarse cualquiera de las siguientes fórmulas:

```
salida <= objeto_asignado WHEN condición_lógica
                          ELSE 'Z'; -- o "ZZZ...Z" si es un vector

PROCESS(objeto_asignado, lista_señales_condición)
 BEGIN
  IF condición_logica THEN
   salida <= objeto_asignado;
  ELSE
   salida <= 'Z'; -- o "ZZZZ..Z" si es un vector
  END IF;
 END PROCESS;
```

A veces las salidas con control de tercer estado se desean utilizar como salidas en colector/drenador abierto; por ejemplo:

```
Sal_oc <= '0' WHEN ctrl = '0'
          ELSE 'Z';
```

8.- Realización de arquitecturas sintetizables de circuitos secuenciales síncronos

En este apartado se va a mostrar las fórmulas generales para la realización de procesos que modelen circuitos secuenciales síncronos. Además

se tratará detalladamente la descripción de autómatas finitos por lo usual que resulta en la práctica la necesidad de realizarlos.

La estructura general de estos procesos es:

```
PROCESS (clk, {lista de señales asíncronas})
{declaración de variables};
BEGIN
   IF {señales asíncronas activas} THEN
   {asignaciones}
   ELSIF clk'event AND CLK = '1' THEN
   {algoritmo de funcionamiento síncrono}
   END IF;
END PROCESS;
```

El proceso es siempre sensible a la señal de reloj y, en el caso de que existan, a las entradas asíncronas de inicialización del circuito. Si hay entradas asíncronas, su prioridad de operación sobre el funcionamiento síncrono se modela por la precendencia en la evaluación de su nivel sobre la condición de operación síncrona (la ocurrencia de un flanco activo de reloj). El esquema anterior corresponde, evidentemente, a un circuito activo por flanco de subida, para modelar la actividad por flanco de bajada basta con cambiar **clk = '1'** por **clk = '0'**. En la rama de la sentencia **IF** correspondiente a la condición de flanco se modela el funcionamiento síncrono, normalmente con sentencias **IF** y **CASE**. Por ejemplo:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
ENTITY cont4b IS
PORT (
       rst, clk, enable : IN std_logic;
       cont : OUT std_logic_vector(3 downto 0);
        );
END ENTITY;
ARCHITECTURE rtl OF cont4b IS
BEGIN
 PROCESS(clk, rst);
   BEGIN
    IF rst = '1' THEN cont <= (OTHERS => '0') AFTER 5ns;
    ELSIF clk'EVENT AND  clk = '1' THEN
        IF enable = '1' THEN cont <= cont + '1' AFTER 5ns;
        END IF;
    END IF;
 END PROCESS;
END rtl ;
```

El modelo anterior corresponde a un contador de cuatro bits con habilitación de reloj y es un ejemplo típico de modelado de contadores y registros –los modelos para este tipo de circuitos tienen aspectos muy

parecidos a este–. Con este esquema resulta muy fácil implementar distintas funcionalidades; por ejemplo, si se desea modificar la descripción anterior para que el contador disponga de cuenta ascendente y descendente, carga de datos en paralelo y *preset* síncrono, el proceso podría modificarse así:

```
PROCESS(clk, rst)
BEGIN
 IF rst = '1' THEN  cnt  <= "0000";
 ELSIF clk'EVENT AND clk = '1' THEN
   IF enable = '1' THEN
     IF preset = '1' THEN cnt <= "1111";
     ELSIF load = '1' THEN cnt <= dato_in;
     ELSIF up_down = '0' THEN cnt <= cnt + 1;
     ELSE cnt <= cnt – 1;
     END IF;
   END IF;
 END IF;
END PROCESS;
```

En este esquema es interesante observar que la prioridad de operación de las entradas síncronas se establece también por el orden de sucesión en la evaluación de las mismas en las ramas de la sentencia **IF** correspondiente. En el anterior ejemplo **preset** es prioritaria respecto a **load** y ambas están validadas por **enable**. Si se deseará, por ejemplo, independizar su operación de la señal **enable** y, además, alterar su prioridad podría modificarse el orden de las sentencias **IF** de la siguiente manera:

```
PROCESS(clk, rst)
BEGIN
 IF rst = '1' THEN  cnt  <= "0000";
 ELSIF clk'EVENT AND clk = '1' THEN
  IF load = '1' THEN cnt <= dato_in;
  ELSIF preset = '1' THEN cnt <= "1111";
  ELSIF enable = '1' THEN
   IF up_down = '0' THEN cnt <= cnt + 1;
   ELSE cnt <= cnt – 1;
   END IF;
  END IF;
 END IF;
END PROCESS;
```

En definitiva, nótese que el orden de sucesión de la evaluación de las entradas síncronas condiciona la funcionalidad del hardware que se sintetiza –a diferencia, por ejemplo, de los procesos combinacionales donde el orden de evaluación de las combinaciones de entrada es irrelevante–.

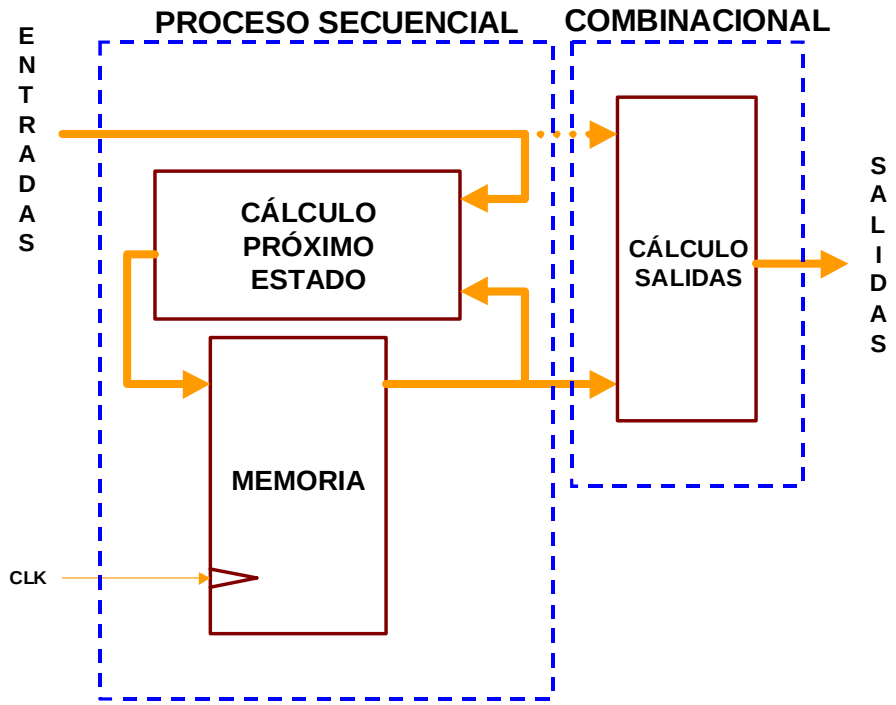
A modo de ejemplo, se muestra el código del contador original modificado para modelar un contador de décadas.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
ENTITY cont4b IS
PORT (
        rst, clk, enable : IN std_logic;
        cont : OUT std_logic_vector(3 downto 0);
         );
END ENTITY;
ARCHITECTURE rtl OF cont4b IS
BEGIN
  PROCESS(clk, rst);
    BEGIN
      IF rst = '1' THEN cont <= (OTHERS => '0') AFTER 5ns;
      ELSIF clk'EVENT AND  clk = '1' THEN
          IF enable = '1' THEN
            IF cont /= "1001" THEN cont <= cont + '1' AFTER 5ns;
            ELSE cont <= (OTHERS => '0') AFTER 5ns;
            END IF;
          END IF;
      END IF;
  END PROCESS;
END rtl ;
```

9.- Descripción de Autómatas

Una de las aplicaciones más frecuentes de los modelos orientados a la síntesis lógica es la descripción de autómatas finitos. Existen distintos estilos para realizarlos. Se describen a continuación.

En la figura se muestra la arquitectura genérica de un autómata.



Como bien se sabe, la transición entre estados (el cálculo del estado futuro) se realiza a partir del estado actual y de las entradas del circuito. El modelado de estas transiciones puede hacerse utilizando un único proceso secuencial:

```
PROCESS(reloj, entradas_asíncronas)
BEGIN
 IF entradas_asincronas THEN
  estado <= estado_inicial;

ELSIF clk'event AND clk = '1' THEN
  CASE estado IS
   WHEN estado1 =>
        Calculo_estado_en _función de entradas
   WHEN estado2 =>
        Calculo_estado_en _función de entradas
   WHEN estado3 =>
        Calculo_estado_en _función de entradas
   . . .

   WHEN estadon =>
        Calculo_estado_en _función de entradas
  END CASE;
 END IF;

END PROCESS;
```

La señal **estado** debe declararse en el Cuerpo de Arquitectura donde se modele el autómata. En principio puede ser de cualquier tipo con un rango finito de valores (**std_logic_vector**, por ejemplo), de modo que cada estado se represente por un valor. Habitualmente se define un tipo de datos de usuario para esta señal:

```
TYPE mis_estados IS (estado1, estado2,..., estadon);
signal estado: mis_estados;
```

Con este sistema la especificación de los estados puede hacerse cómodamente y resulta muy fácil interpretar el código.

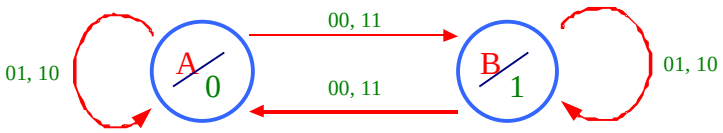
Las salidas del autómata son función:

- Únicamente del estado actual si se trata de un autómata de Moore.
- Del estado actual y las entradas (de las transiciones) si es de Mealy.

Para modelar la lógica que genera las salidas hay que realizar un proceso combinacional sensible a la señal de estado o a ésta y las entradas según el autómata sea de Moore o Mealy. Su estructura es la siguiente:

```
Moore:PROCESS(estado)
        BEGIN
        CASE estado IS
          WHEN estado1 =>
                Salidas <= valor;
          WHEN estado2 =>
        Salidas <= valor;

          WHEN estado3 =>
        Salidas <= valor;
          .....

          WHEN estadon =>
        Salidas <= valor;
         END CASE;
        END PROCESS;

Mealy:PROCESS(estado, entradas)
        BEGIN
        CASE estado IS
          WHEN estado1 =>
                Cálculo_de_salidas_en_función_del_estado;
          WHEN estado2 =>
        Cálculo_de_salidas_en_función_del_estado;

          WHEN estado3 =>
        Cálculo_de_salidas_en_función_del_estado;
          .....

          WHEN estadon =>
        Cálculo_de_salidas_en_función_del_estado;

        END CASE;
        END PROCESS;
```

Observe que la señal **estado**, además de almacenar el estado actual del autómata, conecta ambos procesos dentro de la arquitectura. El siguiente ejemplo muestra un autómata de Moore:



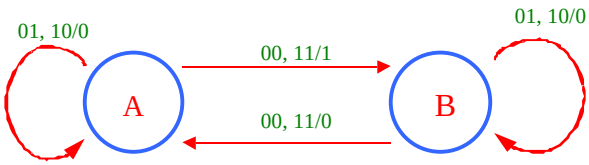

```
ARCHITECTURE rtl OF automata IS
  TYPE ESTADO IS (A, B);
  SIGNAL s_estado : ESTADO;
BEGÍN
  PROCESS (clk)
    BEGIN
      IF clk'EVENT AND clk = '0' THEN
        CASE s_estado IS
          WHEN A =>
              IF (ent1 XOR ent2) = '0' THEN s_estado <= B;
              END IF;
          WHEN B =>
              IF (ent1 XOR ent2) = '0' THEN s_estado <= A;
              END IF;
        END CASE;
      END IF;
  END PROCESS;
```

-- sigue al lado …

```
  PROCESS (s_estado)
    BEGIN
      CASE s_estado IS
        WHEN A => sal <= '0';
        WHEN B => sal <= '1';
      END CASE;
  END PROCESS;
END rtl;
```

DOS PROCESOS QUE SE COMUNICAN MEDIANTE UNA SEÑAL

El siguiente ejemplo se corresponde con un autómata de Mealy. , las sentencias **CASE** “degeneran” en **IF** porque sólo existen dos estados.




```
ARCHITECTURE rtl OF automata IS
  TYPE ESTADO IS (A, B);
  SIGNAL s_estado : ESTADO;
BEGÍN
  PROCESS (clk)
    BEGIN
     IF clk'EVENT AND clk = '0' THEN
       CASE s_estado IS
         WHEN A =>
            IF (ent1 XOR ent2) = '0' THEN s_estado <= B;
            END IF;
         WHEN B =>
            IF (ent1 XOR ent2) = '0' THEN s_estado <= A;
            END IF;
       END CASE;
     END IF;
  END PROCESS;
                              -- sigue al lado …
```

```
PROCESS (s_estado, ent1, ent2)
  BEGIN
     CASE s_estado IS
       WHEN A => sal <= NOT(ent1 XOR ent2);
       WHEN B => sal <= '0';
     END CASE;
 END PROCESS;
END rtl;
```

Cuando, como en los ejemplo anteriores, el número de estados en reducido, las sentencias **CASE** pueden sustituirse por **IF**, tal como muestra la figura, para el caso del autómata de Mealy.

```
ARCHITECTURE rtl OF Automata IS
  TYPE ESTADO IS (A, B);
  SIGNAL s_estado: ESTADO;
BEGIN
 PROCESS(clk)
   BEGIN
        IF clk'EVENT AND clk = '0'THEN
          IF s_estado = A THEN
            IF ent1 XOR ent2  = '0' THEN s_estado = B;
            END IF;
           ELSIF s_estado = B THEN
            IF ent1 XOR ent2  = '0' THEN s_estado = A;
            END IF;
          END IF;
        END IF;
  END PROCESS;
 -- Sigue al lado…….
```

```
 PROCESS(s_estado, ent1, ent2)
    BEGIN
      IF s_estado =  A THEN
           sal <= NOT(ent1 XOR ent2);
      ELSIF s_estado = B THEN
           sal <= '0';
      END IF;
    END PROCESS;
END rtl;
```

Hay dos variantes a este estilo que se puedan usar si se desea. La primera consiste en descomponer el proceso secuencial en dos:

- Uno síncrono que modele los ***flip-flops***.
- Otro asíncrono que modela el cálculo del estado futuro.

El esquema de estos procesos es:

```
PROCESS(reloj, entradas_asíncronas)
BEGIN
 IF entradas_asincronas THEN
   estado_actual <= estado_inicial;

 ELSIF clk'event AND clk = '1' THEN
  Estado_actual <= estado_futuro;

 END IF;
END PROCESS;


PROCESS(estado_actual, entradas)
BEGIN
CASE estado_actual IS
   WHEN estado1 =>
     Cálculo_de_estado_futuro_en_función_de entradas
   WHEN estado2 =>
     Cálculo_de_estado_futuro_en_función_de entradas
     . . .
   WHEN estadon =>
     Cálculo_de_estado_futuro_en_función_de entradas

END CASE;
END PROCESS;
```

En esta modalidad el proceso que modela el cálculo de las salidas debe ser, evidentemente, sensible a la señal **estado_actual**. Tanto ésta como **estado_futuro** deben estar declaradas en el Cuerpo de Arquitectura. No presenta diferencias sensibles respecto a la propuesta inicialmente, de modo que puede elegirse la que se prefiera.

Para los autómatas de Mealy con salidas registradas y para los de Moore existe una tercera alternativa que consiste en modelar el autómata mediante un único proceso síncrono:

```
PROCESS(reloj, entradas_asíncronas)

BEGIN
 IF entradas_asincronas THEN
   estado <= estado_inicial;
   ELSIF clk'event AND clk = '1' THEN
  CASE estado IS
   WHEN estado1 =>
         Calculo_estado_en _función de entradas
Asignación_valor_salidas_próximo_estado
   WHEN estado2 =>
         Calculo_estado_en _función de entradas
Asignación_valor_salidas_próximo_estado
   WHEN estado3 =>
         Calculo_estado_en _función de entradas
Asignación_valor_salidas_próximo_estado
   . . .
```

```
    WHEN estadon =>
          Calculo_estado_en _función de entradas
Asignación_valor_salidas_próximo_estado
  END CASE;
 END IF;

END PROCESS;
```

La peculiaridad de esta alternativa –que, por cierto, se usa con cierta frecuencia– es que los sintetizadores implementan salidas registradas, es decir, además de los ***flip-flops*** para la memoria del autómata, utilizan uno más por cada bit de salida.

Para terminar con el tema de los procesos síncronos, se van a hacer algunas consideraciones:

- En las guías de estilo de algunas herramientas se propone utilizar sentencias **WAIT UNTIL** para la detección del flanco activo de reloj, en lugar de la fórmula con lista de sensibilidad y sentencia **IF**. No presenta mayores problemas si el circuito no tiene entradas asíncronas, en caso contrario sí; por ello es preferible usar siempre la indicada en este texto –es curioso que en algunos manuales de herramientas ésta última no aparece, sin embargo todas las conocidas por el autor la aceptan sin ningún problema–.

- En algunos sintetizadores se acepta la descripción de autómatas implícitos; sin querer entrar aquí en un análisis de este estilo descriptivo, se desaconseja su uso.

10.- Descripciones estructurales para síntesis lógica

Para la realización de descripciones estructurales no hay que tomar precauciones adicionales; sólo resulta interesante tener en cuenta lo siguiente:

- Hay herramientas de síntesis que no soportan los mecanismos de configuración, especialmente en el caso de las Declaraciones de Configuración, así que es mejor evitarlos y recurrir al emplazamiento directo o al mecanismo de configuración por defecto. El primero es preferible al segundo –por su simplicidad– pero hay herramientas que no lo soportan.

- Si se utilizan Declaraciones de Componentes no surge ningún problema porque se encapsulen en paquetes y esta solución “aligera” el código.

- Las herramientas de síntesis lógica soportan los dispositivos declarados con lista de genéricos, lo que permite aprovechar un modelo para instancias de la misma función con distintas dimensiones. La dimensión se especifica al emplazar la instancia.

11.- Algunas consideraciones finales

Además de las reglas de codificación, es conveniente tener en cuenta algunas otras consideraciones a la hora de realizar modelos sintetizables:

1. Si el buen sentido obliga a que en un Cuerpo de Arquitectura haya que utilizar varios procesos y deben conectarse, puede hacerse perfectamente mediante señales declaradas en la propia Arquitectura (véase el ejercicio del capítulo III). No resulta conveniente realizar Arquitecturas demasiado complejas (con muchos procesos); tampoco hay que pasarse en la jerarquización del diseño. Por ejemplo, si se desea hacer una unidad con varios procesos combinacionales, más o menos independientes, y algún secuencial, no hay que alarmarse: se hace sin mayores reparos; el límite puede venir dado por la facilidad para elaborar un test de lo que hay dentro y por no complicarle demasiado el trabajo al sintetizador. Un valor límite para una Arquitectura puede ser no pasarse de unas 300 líneas, si bien hay que señalar que los módulos grandes son a veces, además, los más complicados, sufren modificaciones y su tamaño puede crecer. Por otra parte, si se hace un diseño muy modular resulta bastante engorrosa la realización (y modificación) de los Cuerpos de Arquitectura estructurales.

2. Mientras no se tenga suficiente experiencia en la realización de modelos sintetizables y el manejo de las herramientas de síntesis es obligatorio revisar cuidadosamente los resultados de la síntesis de cada módulo (todas las herramientas ofrecen una buena información al respecto) y simular la lógica generada. Con esto se obtendrán dos beneficios importantes: por un lado se irá comprendiendo el modo en que los sintetizadores interpretan el código y, por otro, se tendrán garantías sobre el “producto” obtenido. Cuando se disponga de suficiente confianza, se podrán obviar frecuentemente ambas operaciones y realizar simulaciones a nivel de sistema una vez se hayan sintetizado todos los módulos