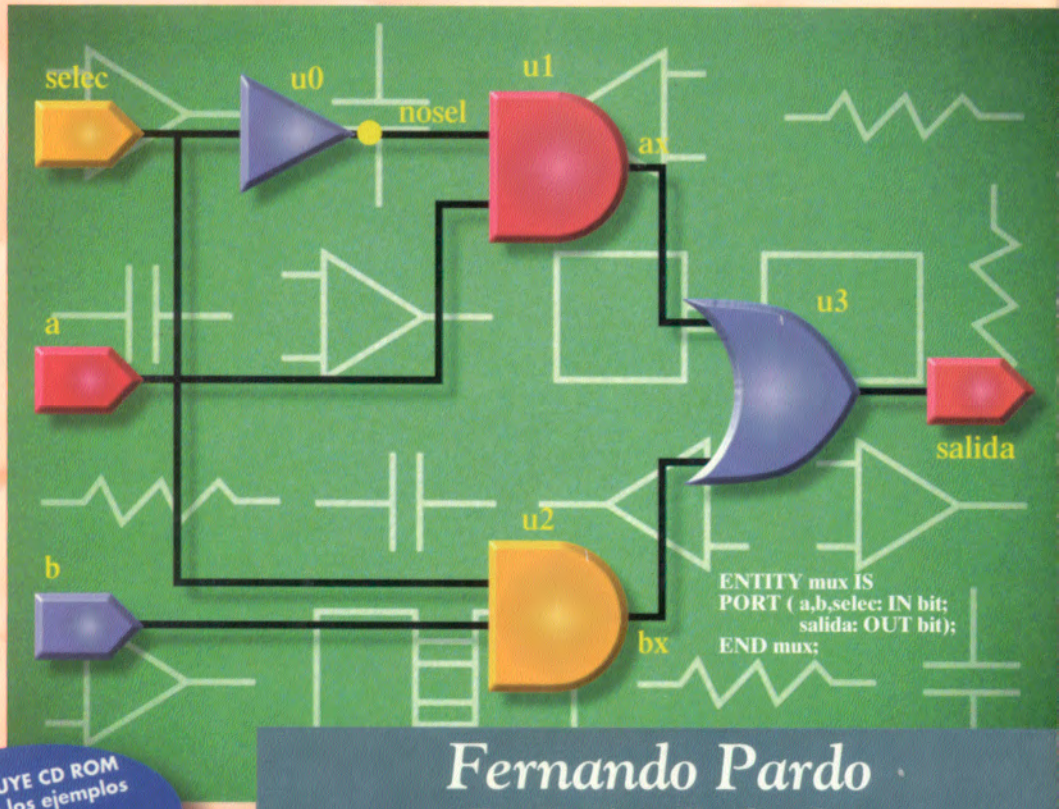
VHDL

Lenguaje para síntesis y modelado de circuitos



Fernando Pardo
José A. Boluda

INCLUYE CD ROM con los ejemplos del libro, un sintetizador (Altera) y un simulador (VeriBest) de VHDL

ra-ma®

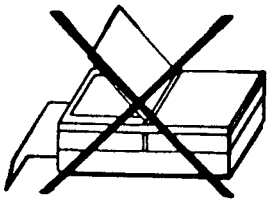
VHDL

Lenguaje para síntesis y modelado de circuitos

VHDL

Lenguaje para síntesis y modelado de circuitos

Fernando Pardo Carpio
José A. Boluda Grau
Dpto. Informática y Electrónica
Universitat València



La ley prohíbe
fotocopiar este libro

VHDL: Lenguaje para síntesis y modelado de circuitos

Editado por:
RA-MA Editorial
Ctra. de Canillas, 144
28043 MADRID
Teléfono: 91 381 03 00
Telefax: 91 381 03 72
Correo electrónico: rama@arrakis.es
Web Internet: **http://www.ra-ma.es**
Autoedición: Autores
ISBN: 84-7897-351-6
Depósito Legal: M-6531-1999
Filmación e impresión: Albadalejo, S.L.
Impreso en España
Primera impresión: Abril 1999

ÍNDICE

PRÓLOGO

En el campo de la realización de sistemas digitales existen dos líneas que han seguido una evolución muy pareja; por un lado la tecnología de realización de circuitos, y por otro, la metodología de diseño de estos circuitos. Se puede decir que la evolución de la tecnología se ha desarrollado hacia los circuitos de lógica programable y alta escala de integración, mientras que la evolución de la metodología ha llevado a la necesidad de utilizar lenguajes de descripción de circuitos.

Ambas líneas, metodología y tecnología, van necesariamente unidas; sin la aparición de la lógica programable, y el abaratamiento de fabricación de los circuitos integrados, no hubieran estado tan disponibles las herramientas que, a partir de una descripción mediante lenguaje, permiten la realización de un circuito. Por otro lado, sin la evolución de la metodología de diseño, no hubiera sido posible integrar y abordar diseños con la complejidad que los dispositivos actuales permiten.

La pieza clave, en cualquier metodología actual de diseño de circuitos, es la especificación y descripción del diseño. La captura de esquemas ya no sirve si se pretende abordar un sistema realmente complejo con miles de elementos. Se hace necesario, por tanto, la utilización de un lenguaje para la especificación del hardware.

Existen muchos lenguajes para la descripción de circuitos digitales, pero es precisamente el VHDL el que está alcanzando mayor popularidad, debido a que nació como un estándar de descripción de circuitos. El resto de lenguajes de descripción suelen ser propios de una determinada herramienta o fabricante de chips, por ello resultan a veces más óptimos, pero no son estándar. VHDL, aparte de ser estándar, tiene un amplio campo de aplicación, desde el modelado para simulación de circuitos, hasta la síntesis automática de circuitos.

Dada la importancia del VHDL, pero sobre todo de la metodología actual de diseño de circuitos digitales, se está incorporando la enseñanza de este lenguaje en la mayoría de titulaciones de informática, electrónica, y telecomunicaciones. Por tanto, el objetivo a la hora de escribir este libro ha sido doble: por un lado servir al estudiante y al profesor como guía en el aprendizaje y enseñanza del lenguaje y la metodología de diseño, por el otro, servir también al ingeniero que diseña circuitos digitales como referencia del lenguaje, así como de guía, si pretende empezar con él.

La estructura del libro, así como su contenido, está inspirado en la experiencia docente de un curso de diseño digital en la titulación de Ingeniería Informática de la Universidad de Valencia. El objetivo del curso es explicar a los alumnos una metodología de diseño, desde la descripción hasta la implementación, pasando por la simulación. El contenido del libro cubre completamente los aspectos de descripción, y también los de simulación e implementacion.

El curso contenido en estas páginas ha sido también impartido en otras instituciones fuera de España, como en el CINVESTAV (Centro de Investigación y Estudios Avanzados del IPN) de México. Impartir este curso fuera de España ha sido muy enriquecedor; la influencia recibida de alumnos y docentes de este centro es patente en muchas de las hojas de este libro.

La estructura del libro sigue, a grandes rasgos, la metodología típica de diseño de circuitos digitales. Esta estructura parece también la más adecuada para estructurar la enseñanza del lenguaje y la metodología. En el Capítulo 1 se describe la metodología de diseño de circuitos digitales. En el siguiente, Capítulo 2, se muestran las diferentes maneras de describir circuitos, desde la captura de esquemas hasta la descripción a partir de lenguajes.

Del Capítulo 3 hasta el 9, se explica el lenguaje en sí. Se empieza con una introducción donde se muestra un ejemplo y diferentes estilos para su descripción. Luego le sigue otro capítulo con la sintaxis del lenguaje. Posteriormente le siguen tres capítulos dedicados cada uno a los tres estilos de descripción utilizando VHDL. Luego viene un capítulo dedicado a las funciones y bibliotecas. Esta parte termina con el Capítulo 9 donde se explican aquellos conceptos de uso menos frecuente, pero no por ello menos importantes.

Los tres últimos capítulos están dedicados al uso del lenguaje. En el Capítulo 10 se explica el uso del lenguaje para el modelado de circuitos, es decir, para emplearlo en simulación. Como complemento se dan también algunas nociones sobre simulación digital. En el Capítulo 11 se explica el uso del lenguaje con el objetivo de realizar un

circuito; no sólo se explican las estructuras típicas de síntesis, sino que se dan además las tecnologías actuales para síntesis automática de circuitos. Por último, el Capítulo 12 muestra varios ejemplos comentados de problemas digitales de la vida cotidiana.

El Apéndice A recoge toda la sintaxis del VHDL utilizando la notación BNF. El Apéndice B contiene varias direcciones donde se pueden encontrar herramientas y documentos de uso público sobre VHDL. El Apéndice C explica el contenido del CD incluido en este libro.

La lista de agradecimientos podría ser tan larga que resultaría imposible nombrarlos a todos. No obstante queremos agradecer las sugerencias y comentarios recibidos, dentro del Departamento de Informática y Electrónica de la Universidad de Valencia, por parte de alumnos y docentes, durante estos años de enseñanza del VHDL y de la tecnología digital. Importante también ha sido la influencia recibida de las personas del CINVESTAV (Centro de Investigación y Estudios Avanzados del IPN) que tanto han contribuido a la difusión y uso del VHDL en México.

En la dirección *http://tapec.uv.es/VHDL/libro.html* se puede encontrar la presentación de este libro, así como ejercicios resueltos. Se han añadido también algunos enlaces muy interesantes para acceder a herramientas, documentación adicional, etc.

Fernando Pardo
José Antonio Boluda
Valencia, Enero 1999

CAPÍTULO 1

METODOLOGÍA DE DISEÑO

1.1 CONCEPTO DE HERRAMIENTAS CAD-EDA

En su sentido más moderno, CAD (diseño asistido por ordenador, del inglés *Computer Aided Design*) significa proceso de diseño que emplea sofisticadas técnicas gráficas de ordenador, apoyadas en paquetes de software para ayuda en los problemas analíticos, de desarrollo, de coste y ergonómicos asociados con el trabajo de diseño.

En principio, el CAD es un término asociado al dibujo como parte principal del proceso de diseño, sin embargo, dado que el diseño incluye otras fases, el término CAD se emplea tanto para el dibujo, o el diseño gráfico, como para el resto de herramientas que ayudan al diseño (por ejemplo, la comprobación de funcionamiento, análisis de costes, etc.)

El impacto de las herramientas de CAD sobre el proceso de diseño de circuitos electrónicos y sistemas procesadores es fundamental. No sólo por la adición de interfaces gráficas para facilitar la descripción de esquemas, sino por la inclusión de herramientas, como los simuladores, que facilitan el proceso de diseño y la conclusión con éxito de los proyectos.

EDA (*Electronic Design Automation*) es el nombre que se le da a todas las herramientas (tanto hardware como software) que sirven de ayuda en el diseño de sistemas electrónicos. Dentro del EDA, las herramientas de CAD juegan un importante papel. Sin embargo, no sólo el software es importante; computadoras cada día más veloces, elementos de entrada de diseño cada vez más sofisticados, etc., son también características que ayudan a facilitar el diseño de circuitos electrónicos.

El diseño hardware tiene un problema fundamental, que no existe, por ejemplo, en la producción del software. Este problema es el alto coste del ciclo diseño-prototipación-testeo-vuelta a empezar, ya que el coste del prototipo suele ser, en general, bastante elevado. Se impone la necesidad de reducir este ciclo de diseño para no incluir la fase de prototipación más que al final del proceso, evitando así la repetición de varios prototipos que es lo que encarece el ciclo. Para ello se introduce la fase de simulación y comprobación de circuitos utilizando herramientas de CAD, de forma que no es necesario realizar físicamente un prototipo para comprobar el funcionamiento del circuito, economizando así el ciclo de diseño. Este ciclo de diseño hardware se muestra con detalle en la figura 1.1.

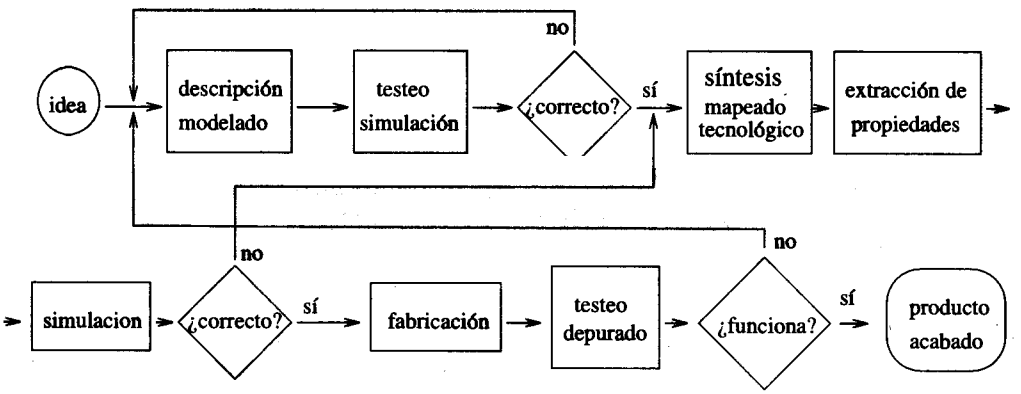


Figura 1.1: Flujo de diseño para sistemas electrónicos y digitales

En el ciclo de diseño hardware las herramientas de CAD están presentes en todos los pasos. En primer lugar en la fase de descripción de la idea, que será un esquema eléctrico, un diagrama de bloques, etc. En segundo lugar en la fase de simulación y comprobación de circuitos, donde diferentes herramientas permiten realizar simulación por eventos, funcional, digital o eléctrica de un circuito atendiendo al nivel de simulación requerido. Por último existen las herramientas de CAD orientadas a la fabricación. En el caso de diseño hardware estas herramientas sirven para la realización de PCBs (*Printed Circuit Boards* o placas de circuito impreso), y también para la realización de ASICs, o *Circuitos Integrados de Aplicación Específica* (*Application Specific Integrated Circuits*). Estas herramientas permiten la realización de microchips, así como la realización y programación de dispositivos programables.

Herramientas CAD para el diseño hardware:

Lenguajes de descripción de circuitos. Son lenguajes mediante los cuales es posible describir un circuito eléctrico o digital. La descripción puede ser estructural, donde se muestra la arquitectura del diseño, o bien de comportamiento, donde se

describe el comportamiento del circuito en vez de los elementos de los que está compuesto.

Captura de esquemas. Es la forma clásica de describir un diseño electrónico y la más extendida, ya que era la única usada antes de la aparición de las herramientas de CAD. La descripción está basada en un diagrama donde se muestran los diferentes componentes de un circuito y sus interconexiones.

Grafos y diagramas de flujo. Es posible describir un circuito, o sistema, mediante diagramas de flujo, redes de Petri, máquinas de estados, etc. En este caso sería una descripción *gráfica* pero, al contrario que en la captura de esquemas, la descripción sería comportamental en vez de una descripción de componentes.

Simulación de sistemas. Estas herramientas se usan sobre todo para la simulación global de sistemas. Los componentes de la simulación son elementos de alto nivel como discos duros, buses de comunicaciones, etc. Se aplica la teoría de colas para la simulación.

Simulación funcional. Bajando al nivel de circuitos digitales se puede realizar una simulación funcional. Este tipo de simulación comprueba el funcionamiento de circuitos digitales de forma funcional, es decir, a partir del comportamiento lógico de sus elementos (sin tener en cuenta problemas eléctricos como retrasos, etc.) se genera el comportamiento del circuito frente a unos estímulos dados.

Simulación digital. Esta simulación, también exclusiva de los circuitos digitales, es como la anterior con la diferencia de que se tienen en cuenta retrasos en la propagación de las señales digitales. Es una simulación muy cercana al comportamiento real del circuito y prácticamente garantiza el funcionamiento correcto del circuito a realizar.

Simulación eléctrica. Es la simulación de más bajo nivel donde las respuestas se elaboran a nivel del transistor. Sirven tanto para circuitos analógicos como digitales y su respuesta es prácticamente idéntica a la realidad.

Realización de PCBs. Con estas herramientas es posible realizar el trazado de pistas para la posterior fabricación de una placa de circuito impreso.

Realización de circuitos integrados. Son herramientas de CAD que sirven para el diseño de circuitos integrados. Las capacidades gráficas de estas herramientas permiten la realización de las diferentes máscaras que intervienen en la realización de circuitos integrados.

Realización de dispositivos programables. Con estas herramientas se facilita la programación de este tipo de dispositivos, desde las simples PALs (*Programmable*

And Logic) hasta las más complejas FPGAs (*Field Programmable Gate Arrays*), pasando por las PLDs (*Programmable Logic Devices*).

1.2 DISEÑO *BOTTOM-UP*

El término *diseño Bottom-Up* (diseño de abajo hacia arriba) se aplica al método de diseño mediante el cual se realiza la descripción del circuito, o sistema que se pretende realizar, empezando por describir los componentes más pequeños del sistema para, más tarde, agruparlos en diferentes módulos, y éstos a su vez en otros módulos hasta llegar a uno solo que representa el sistema completo que se pretende realizar. En la figura 1.2 se muestra esta metodología de diseño.

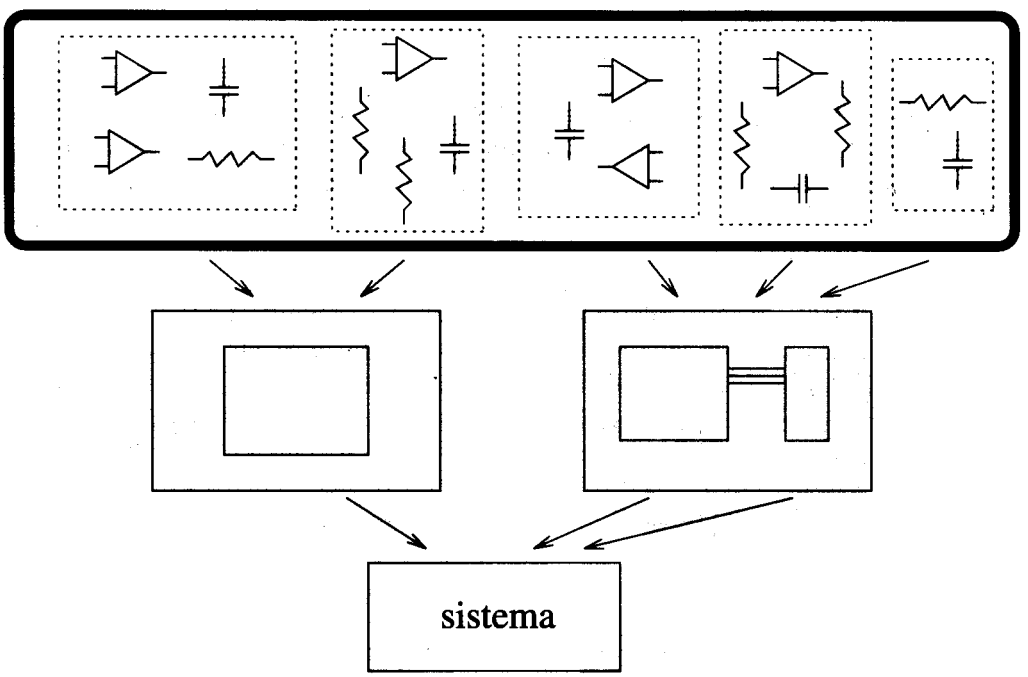


Figura 1.2: Metodología de diseño Bottom-Up

Esta metodología de diseño no implica una estructuración jerárquica de los elementos del sistema, sino que esta estructuración, al contrario de lo que ocurre en el diseño *top-down*, que se verá después, se realiza después de la descripción del circuito y, por tanto, no resulta necesaria.

En un diseño *bottom-up* se empieza por crear una descripción, con esquemas por ejemplo, de los componentes del circuito. Estos componentes pertenecen normalmente a una biblioteca que contiene chips, resistencias, condensadores, y otros elementos que representan unidades funcionales con significado propio dentro del diseño. A estas unidades se las puede conocer por el nombre de *primitivas*, puesto que no es necesario disponer de elementos de más bajo nivel para describir el circuito que se pretende realizar.

En general, esta forma de diseñar no es muy buena, ya que es un flujo de diseño bastante ineficiente. Para diseños muy grandes, como los actuales, no se puede esperar unir miles de componentes a bajo nivel y pretender que el diseño funcione adecuadamente. El hecho de unir un número elevado de componentes entre sí, sin una estructura más elevada que permita separarlos en bloques, hace que sea complejo el análisis del circuito, lo que provoca dificultades a la hora de detectar fallos en el circuito, anomalías de funcionamiento, etc. Con esto, la probabilidad de cometer errores de diseño se hace más elevada.

Esta metodología es la que se venía utilizando en los primeros tiempos de la automatización del proceso de diseño. Esto es así porque esta parte se había conseguido automatizar completamente. Para empezar las herramientas de diseño permitían una descripción sencilla a bajo nivel (captura de esquemas) y a partir de ahí, mediante otras herramientas integradas en el proceso, era posible su implementación, es decir, la realización de un ASIC (*Application Specific Integrated Circuit*) o un PCB (*Printed Circuit Board*).

Si el diseño era muy complejo, había una primera parte "manual" que consistía en la partición del diseño en bloques o módulos, es decir, se establecía una jerarquía, pero toda esta fase previa no había sido integrada como herramientas en el proceso de diseño. A esta parte previa se le llama diseño *Top-Down* y se cuenta a continuación.

1.3 DISEÑO *TOP-DOWN*

El diseño *Top-Down* es, en su más pura forma, el proceso de capturar una idea en un alto nivel de abstracción e implementarla partiendo de esa descripción abstracta, y después ir hacia abajo incrementando el nivel de detalle según sea necesario. Esta forma de diseñar se muestra gráficamente en la figura 1.3, donde el sistema inicial se ha dividido en diferentes módulos, cada uno de los cuales se encuentra a su vez subdividido hasta llegar a los elementos primarios de la descripción.

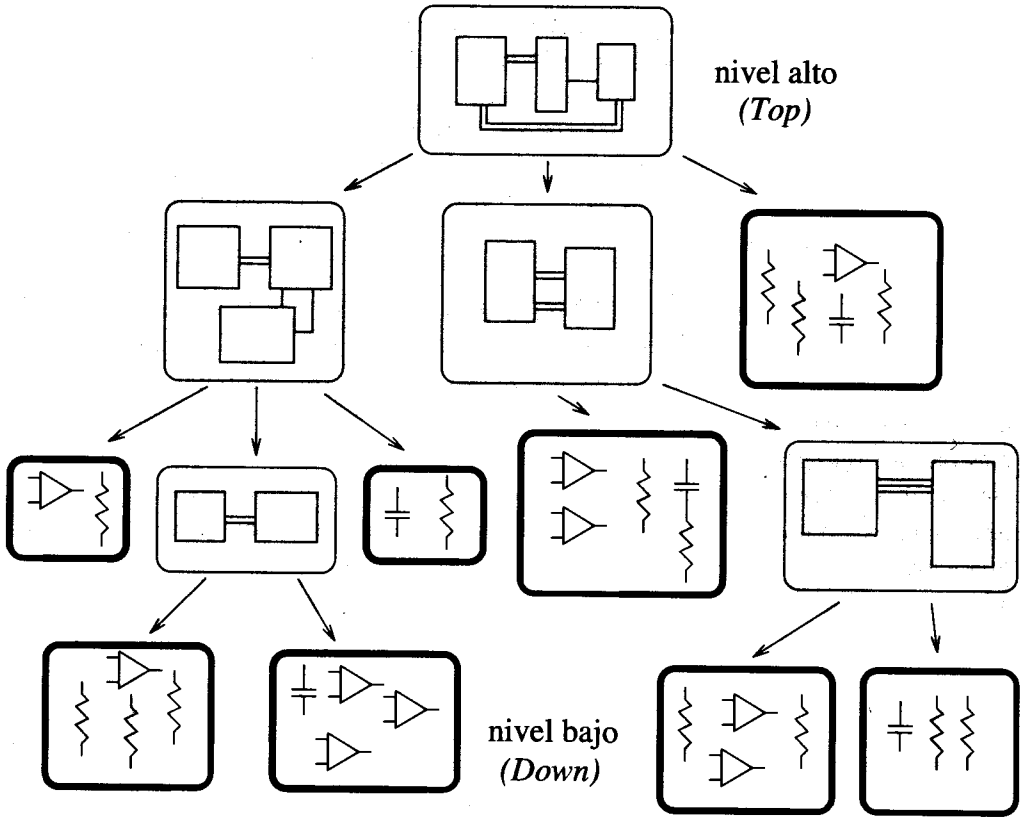


Figura 1.3: Metodología de diseño Top-Down

Los años ochenta trajeron una revolución de las herramientas para el diseño por ordenador. Aunque esto no modificó la forma de diseñar sí que mejoró la facilidad de hacerlo. Así, mediante el software disponible por ordenador, se podían diseñar circuitos más complejos en, comparativamente, cortos periodos de tiempo (aunque se siguiera utilizando el diseño *Bottom-Up*).

Pero hoy en día, es necesario hacer diseños más y más complicados en menos tiempo. Así, se puede descubrir que el flujo de diseño *Bottom-Up* es bastante ineficiente. El problema básico del diseño *Bottom-Up* es que no permite acometer con éxito diseños que contengan muchos elementos puesto que es fácil conectarlos de forma errónea. No se puede esperar unir miles de componentes de bajo nivel, o primitivas, y confiar en que el diseño funcione adecuadamente.

Para esto existe la metodología *Top-Down* que sigue un poco el lema de "divide y vencerás", de manera que un problema, en principio muy complejo, es dividido en

varios subproblemas que a su vez pueden ser divididos en otros problemas mucho más sencillos de tratar. En el caso de un circuito esto se traduciría en la división del sistema completo en módulos, cada uno de los cuales con una funcionalidad determinada. A su vez, estos módulos, dependiendo siempre de la complejidad del circuito inicial o de los módulos, se pueden dividir en otros módulos hasta llegar a los componentes básicos del circuito o primitivas.

La mejora de las herramientas de diseño electrónico en los últimos años ha ido por ese camino, es decir, el de ofrecer herramientas que permitan de forma automática la metodología de diseño *Top-Down*. Con esto, las herramientas actuales de síntesis permiten la implementación de un circuito a partir de una idea abstracta sin necesidad de que el diseñador tenga que descomponer su idea abstracta en componentes concretos.

1.3.1 Ventajas del diseño *Top-Down*

Incrementa la productividad del diseño. Este flujo de diseño permite especificar funcionalmente en un nivel alto de abstracción sin tener que considerar la implementación del mismo a nivel de puertas lógicas. Por ejemplo, se puede especificar un diseño en VHDL y el software utilizado generaría el nivel de puertas directamente. Esto minimiza la cantidad de tiempo utilizado en un diseño.

Incrementa la reutilización del diseño. En el proceso de diseño se utilizan tecnologías genéricas. Esto es, que la tecnología a utilizar no se fija hasta pasos posteriores en el proceso. Esto permite reutilizar los datos del diseño únicamente cambiando la tecnología de implementación. Así es posible crear un nuevo diseño de uno ya existente.

Rápida detección de errores. Como se dedica más tiempo a la definición y al diseño, se encuentran antes muchos errores en el proceso de descripción del circuito.

1.4 INGENIERÍA CONCURRENTE

En los años ochenta los suministradores de productos EDA se preocuparon sobre todo de realizar herramientas más veloces y computadoras más rápidas especialmente pensando en un entorno de diseño donde un producto es diseñado en serie. La competencia entre las diversas compañías se basaba en la rapidez con que podía realizarse cada paso de la cadena de diseño.

En los noventa la competencia se encuentra, no en lo rápido en que se puedan completar los diferentes pasos de un diseño, sino en que se pueda realizar *ingeniería concurrente*. La ingeniería concurrente permite que se puedan utilizar datos de un paso en el proceso de diseño antes de que el paso previo haya sido completado. Esto implica la existencia de *monitores* dentro del sistema de diseño para comunicar adecuadamente la actividad de diseño hacia todos los pasos del proceso.

La forma más sencilla de obtener un sistema concurrente es que todos los pasos del proceso de diseño compartan la misma base de datos. De esta manera, diferentes herramientas correspondientes a diferentes pasos en el proceso de diseño comparten los mismos datos. Un cambio realizado con una herramienta en una fase de diseño tiene efectos inmediatos sobre la ejecución de otra herramienta en otra fase de ciclo.

En general hay dos tipos diferentes de ingeniería concurrente:

Ingeniería concurrente personal. Viene referida a la posibilidad de realizar cambios en el diseño (esquema) sin tener que abandonar el análisis o simulación, o las herramientas de diseño de circuitos impresos, por ejemplo.

Ingeniería concurrente de grupo. Este tipo permite, a los diferentes equipos de expertos que trabajan en un diseño, el solapar la creación, análisis, y trazado de un diseño. Por ejemplo, un equipo puede estar simulando un circuito que otro equipo acaba de modificar, etc.

En general, el elemento más importante de un sistema EDA que permita diseño concurrente es la base de datos. En esta base de datos cada elemento es común a todas las herramientas que componen el sistema. Las diferencias entre una herramienta y otra vendrán de lo que la herramienta *ve* del elemento. Así, cada elemento de la base de datos estará compuesto por distintas *vistas* cada una asociada generalmente a una herramienta del sistema.

En una herramienta de CAD, donde se incluyan diferentes fases del proceso de diseño como captura de esquemas, simulación, etc., existe siempre la operación por la cual las herramientas posteriores del flujo de diseño (como simulación o diseño de PCBs) conocen los resultados de los pasos previos (como la captura de esquemas). A esta operación se le conoce con el nombre de *preanotación* o **forward-annotation** y consiste en que las herramientas anteriores dentro del flujo de diseño informan a las herramientas posteriores de los cambios realizados en el diseño.

En el caso de herramientas con capacidad para ingeniería concurrente se debe permitir una operación adicional. Esta operación, muy importante dentro de la ingeniería

concurrente, es la *retroanotación* o **back-annotation**. Uno de los objetivos de la ingeniería concurrente es la posibilidad de trabajar en fases del proceso de diseño sin haber completado previamente las fases anteriores. Para conseguir esto, no es únicamente necesario disponer de una base de datos única, sino también disponer de los mecanismos necesarios para que herramientas asociadas a fases anteriores del proceso de diseño puedan saber de los cambios realizados por herramientas posteriores e incorporarlos a su *visión* especial del diseño. Para esto existe el mecanismo de *back-annotation*, que simplemente sirve para que herramientas pertenecientes a fases finales del proceso de diseño puedan *anotar* cambios a las fases iniciales del diseño.

Por ejemplo, en un esquema se debe especificar el encapsulado de un chip, pero puede que en la fase inicial del diseño no se sepa todavía. Es posible que en el proceso de diseño de las pistas de un circuito impreso, que sería una fase posterior, ya se conozca dicho encapsulado. En este caso, la herramienta que realiza el diseño del circuito impreso puede retroanotar la información del encapsulado a la herramienta de captura de esquemas.

Otro ejemplo interesante es la retroanotación de los retrasos de las pistas, tanto en PCBs como en ASICs, para la simulación digital. En primer lugar se realiza una simulación digital del circuito con los retrasos propios de las puertas y componentes, pero sin conocer el retraso introducido por los cables. Cuando esta simulación previa es correcta, se realiza la implementación física del circuito, retroanotando los retrasos introducidos por las pistas. Se vuelve de nuevo a simular, pero esta vez se tendrá además la información del retraso de las pistas.

CAPÍTULO 2

DESCRIPCIÓN DEL DISEÑO

La primera tarea dentro del flujo de diseño electrónico, después de concebir la idea, es realizar una descripción de lo que se pretende hacer. Los ordenadores ofrecen hoy día herramientas especiales para la creación y verificación de diseños. Con dichas herramientas es posible describir tanto un sencillo circuito que represente una simple puerta lógica como un complejo diseño electrónico.

En un principio, las herramientas de CAD se limitaban a servir de meros instrumentos de dibujo para poder realizar el diseño; el diseñador de circuitos realizaba la descripción a bajo nivel sobre un papel, utilizando símbolos y componentes básicos que luego trasladaba a la computadora para obtener una representación más ordenada. Con la incorporación de herramientas de fabricación de PCBs, o circuitos integrados, o simuladores, etc., la descripción del circuito empezaba a desempeñar un papel más importante, ya que servía como entrada de información a las herramientas posteriores en el flujo de diseño. Esto, unido a la metodología Top-down de diseño de circuitos, llevó a la aparición de herramientas de descripción que permitían al diseñador definir el problema de una forma abstracta. De esta manera, es el ordenador quien se ocupaba de realizar la concretización de la idea.

Teniendo en cuenta esta evolución, las herramientas de CAD actuales permiten las siguientes posibilidades de abordar la descripción de una idea o diseño electrónico:

Descripción estructural. Consiste en enumerar los componentes de un circuito y sus interconexiones. Dependiendo de la herramienta que se utilice hay dos formas de hacerlo:

Esquemas. Es la forma tradicional en que los circuitos han sido diseñados desde los inicios de la electrónica. Consiste en la descripción gráfica de los componentes de un circuito.

Lenguaje. Se realiza una enumeración de los componentes de un circuito así como su conexionado.

Descripción comportamental. Es posible describir un circuito electrónico (generalmente digital) simplemente describiendo cómo se comporta. Para este tipo de descripción también se utiliza un lenguaje de descripción hardware específico.

2.1 CAPTURA DE ESQUEMAS

Por captura de esquemas se entiende el proceso de descripción, mediante un dibujo, de un circuito eléctrico. El dibujo del esquema puede incluir más que un simple diagrama de líneas. Puede incluir también información sobre tiempos, referencias, cables, conectores, notas, y muchas otras propiedades importantes y valores necesarios para la interpretación del mismo por el resto de aplicaciones.

Un esquema viene especificado en la base de datos por dos partes fundamentales: las **hojas** y los **símbolos**. En principio, un esquema puede estar formado por varias hojas que es donde se *dibujan* los diversos componentes o símbolos que forman el circuito. En las hojas se especifican también las interconexiones, así como informaciones adicionales para el uso posterior del esquema en otras aplicaciones.

Los símbolos son *cajas* que se interconectan entre sí en la hoja de diseño. Un símbolo es un objeto que contiene un conjunto de modelos usados para describir los aspectos funcionales, gráficos, temporales y tecnológicos del diseño electrónico.

Hay dos tipos de símbolos. El primer tipo está formado por los símbolos que representan componentes básicos, o *primitivas*. Estos componentes definen un elemento que se encuentra en el nivel más bajo de la jerarquía de diseño. Así, este tipo de componentes suelen ser resistencias, condensadores, transistores, puertas lógicas, procesadores, chips de memoria, etc., que suelen estar ubicados en bibliotecas propias de la herramienta.

Un segundo tipo de símbolos es el formado por aquellos que especifican, no un elemento simple, sino otro circuito completo, compuesto a su vez por símbolos, etc., es decir, elementos que están por encima de los símbolos básicos dentro de la jerarquía. Normalmente este tipo de símbolos suelen tener asociados una hoja que es la que describe sus componentes, aunque, con la aparición de las descripciones mediante lenguaje, es posible encontrar que dentro del símbolo en un esquema se tiene una descripción mediante lenguaje en vez de una hoja que sería lo esperable. Las posibilidades de las

herramientas de descripción actuales son tales que permiten, sin demasiados problemas, unir en un mismo diseño descripciones mediante gráficos y descripciones mediante lenguaje.

El método clásico para la interconexión de los distintos símbolos de una hoja son los **hilos** o *wires*. Un hilo en el esquema tiene una correspondencia inmediata con el circuito real, se trata de un cable físico que conecta un pin de un chip con un pin de otro. Sin embargo, dado que un esquema puede representar un nivel de abstracción elevado dentro de una jerarquía, un *cable* puede representar una conexión con un sentido más amplio, como por ejemplo una línea telefónica, o un enlace de microondas a través de satélite.

Un cable en un esquema es un elemento que indica conexión y, en principio, puede tratarse tanto de un hilo de cobre como de una pista en un circuito impreso, o bien un conjunto de hilos, o un cable de una interfase serie, etc. Sin embargo, en los comienzos del diseño electrónico, donde los esquemas correspondían en la mayoría de los casos al nivel más bajo de una jerarquía, los cables eran siempre hilos conductores, y para representar un conjunto de hilos conductores se introdujo otro elemento adicional, el **bus**. Un bus es una conexión que une dos componentes al igual que un cable, sin embargo se caracteriza por representar, no un único hilo, sino múltiples. La introducción de este elemento fue inmediata a partir del desarrollo de circuitos digitales, donde la conexión entre procesadores, memorias, etc., era fácilmente agrupable.

Actualmente, dada la gran complejidad de los diseños electrónicos, con miles de conexiones en una misma hoja, se hace necesario el uso de otras técnicas de interconexión de componentes. Una posibilidad que ofrecen la mayoría de herramientas de CAD es la utilización de etiquetas. Es posible poner etiquetas a los pines o a los cables, de manera que dos pines o cables con la misma etiqueta, o nombre, están físicamente interconectados. Esto evita el tener que trazar múltiples conexiones entre componentes evitando así una aglomeración de hilos que harían ilegible cualquier esquema.

Otro elemento importante dentro de una hoja o esquema son los **puertos**. Los puertos son conexiones al exterior de la hoja, y realizan la labor de interfase del circuito con el mundo exterior. En general, un esquema se puede ver como una caja negra donde los puertos son la única información visible. Esta caja negra, junto con sus puertos, forma un componente que puede ser usado en otra hoja, que a su vez es un componente que puede formar parte de otra hoja y así sucesivamente. Los puertos pueden ser de entrada, de salida, o de entrada/salida, dependiendo de la dirección del flujo de la información.

2.2 GENERACIÓN DE SÍMBOLOS

Como se ha comentado anteriormente, el concepto de símbolo tiene un sentido amplio; puede representar tanto un componente físico, como un transistor o un chip, o bien puede representar un elemento abstracto, como un sistema, etc., que a su vez se encuentra formado por distintos elementos o símbolos.

Los símbolos suelen estar formados por dos elementos fundamentales, el cuerpo y los puertos. El **cuerpo** es simplemente un dibujo que, en su forma más genérica, puede ser una simple caja. Los **puertos** son los elementos que realmente definen el componente, ya que indican la comunicación con el exterior. Un componente puede no tener cuerpo, aunque en principio resulta difícil trabajar con un elemento que no se *ve*; mientras se pueda interconexionar con otros elementos, no se necesita nada más.

Los símbolos más simples, que corresponden a un elemento físico, se encuentran normalmente agrupados en bibliotecas de componentes. Para ser usados es suficiente con copiarlos de la biblioteca e introducirlos en el diseño. Hay otros elementos que no representan primitivas, sino otros esquemas en un nivel más bajo de la jerarquía; en estos casos, los símbolos no suelen estar agrupados en bibliotecas sino que forman parte de la base de datos que contiene el diseño completo.

Las herramientas que soportan la metodología de diseño *bottom-up* o *top-down*, deben proveer algún mecanismo para convertir las hojas en símbolos, es decir, coger un esquema, con unas entradas y salidas, y generar una caja con las mismas entradas y salidas que pueda ser usado como un símbolo más en otras partes del diseño. En principio, es muy sencillo generar un símbolo a partir de un esquema, siempre y cuando en el esquema se especifiquen adecuadamente los puertos de interconexión. Estos símbolos, generados a partir de esquemas, sirven para la realización de diseños jerárquicos, ya que pueden ser usados en otros esquemas y así sucesivamente.

Aunque un símbolo sólo necesita los puertos y un cuerpo, hay otra serie de elementos que resultan de mucha utilidad. Un elemento muy importante es el nombre, ya que es una forma de identificar el símbolo y resulta de utilidad para leer un esquema. Dependiendo de la utilización del símbolo puede ser interesante la adición de otros elementos o propiedades, así, para símbolos que representen chips, es muy interesante añadirles información sobre el encapsulado del chip, una referencia para identificar individualmente a cada componente dentro del circuito, etc. Para otros componentes, dedicados a simulación por ejemplo, puede ser interesante añadirles propiedades sobre el retraso de la señal, etc.

Un mismo símbolo puede representar varias cosas dentro de un diseño. Lo que un símbolo representa depende de la herramienta particular que se esté utilizando. Supongamos el caso muy simple de un contador. El símbolo del contador será una caja cuadrada, con una serie de entradas y salidas, pero ¿qué representa realmente? Si por ejemplo se está realizando un circuito impreso, este símbolo del contador representa su encapsulado con sus diferentes pines, así como las partes de cobre asociadas. Si por el contrario, se quiere realizar una simulación para ver el comportamiento del contador, en realidad el símbolo estará haciendo referencia a una descripción del comportamiento del circuito. Y aún puede haber más representaciones, el mismo símbolo del contador puede representar a su vez una descripción estructural (realizada con un lenguaje de descripción hardware como VHDL) o incluso otro esquema formado por símbolos más simples como puertas lógicas o incluso transistores. El mismo símbolo representa varias características que conviven de forma concurrente en la misma base de datos. Lo que se *ve* del símbolo dependerá de la tarea que se realice en cada momento, así como de la herramienta que se esté utilizando.

2.3 DISEÑO MODULAR

El flujo de diseño *Top-Down* ofrece una ventaja adicional, y es que la información se estructura de forma modular. El hecho de empezar la realización de un diseño a partir del concepto de sistema, hace que las subdivisiones se realicen de forma que los diferentes módulos generados sean disjuntos entre sí y no se solapen. De esta forma, el diseño modular sería la realización de diseños ejecutando divisiones funcionalmente complementarias de los diversos componentes del sistema, permitiendo de esta manera una subdivisión clara y no solapada de las diferentes tareas dentro del diseño.

El diseño *Bottom-Up* no ofrece tanta facilidad para la división del diseño en partes funcionalmente independientes. Como se parte de los elementos básicos de los que se compone el sistema, no resulta tan sencillo agruparlos de forma coherente. Ésta es otra de las desventajas del flujo de diseño *Bottom-Up*, el resultado final puede resultar bastante confuso al no estar modularmente dividido.

2.4 DISEÑO JERÁRQUICO

Un complejo diseño electrónico puede necesitar cientos de miles de componentes lógicos para describir correctamente su funcionamiento. Estos diseños necesitan organizarse de tal forma que resulte fácil su comprensión. Una forma de organizar el diseño

es la creación de un diseño modular jerárquico tal y como se ha mostrado cuando se explicaba el flujo de diseño *top-down*.

Una jerarquía consiste en construir un nivel de descripción funcional de diseño debajo de otro de forma que cada nuevo nivel posee una descripción más detallada del sistema. La construcción de diseños jerárquicos es la consecuencia inmediata de aplicar el flujo de diseño *Top-Down*.

En la creación de diseños jerárquicos es muy útil la realización de bloques funcionales o módulos. Un bloque funcional es un símbolo que representa un grupo de elementos en alto nivel. Se puede pensar que un bloque funcional son particiones del diseño original con descripciones asociadas a las pequeñas unidades.

2.5 EL *NETLIST*

El *netlist*, o lista de conexiones, es la primera forma de describir un circuito mediante un lenguaje, y consiste en dar una lista de componentes, sus interconexiones y las entradas y salidas. No es un lenguaje de alto nivel, por lo que no describe como funciona el circuito, sino que simplemente se limita a describir los componentes que posee y las conexiones entre ellos.

2.5.1 El formato EDIF

Dada la gran proliferación de lenguajes para la comunicación de descripciones del diseño entre herramientas, fue necesario crear un formato que fuera estándar y que todas las herramientas pudieran entender. Así es como apareció el formato EDIF.

El formato EDIF (*Electronic Design Interchange Format*) es un estándar industrial para facilitar el intercambio de datos de diseño electrónico entre sistemas EDA (*Electronic Design Automation*). Este formato de intercambio está diseñado para tener en cuenta cualquier tipo de información eléctrica, incluyendo diseño de esquemas, trazado de pistas (físicas y simbólicas), conectividad, e información de texto, como por ejemplo, las propiedades de los objetos de un diseño.

El formato EDIF fue originalmente propuesto como estándar por Mentor Graphics, Motorola, National Semiconductor, Texas Instruments, Daisy Systems, Tektronix, y la Universidad de California en Berkeley, todos ellos implicados cooperativamente en su desarrollo. Desde entonces, el EDIF ha sido aceptado por más y más compañías. Fue

aprobado como estándar por la Electronic Industries Association (EIA) en 1987, y por el American National Standards Institute (ANSI) en 1988.

La sintaxis de EDIF es bastante simple y comprensible, sin embargo, no se pretende que sea exactamente un lenguaje de descripción de hardware con el cual los diseñadores puedan definir sus circuitos, aunque hay algunos que lo utilizan directamente como lenguaje de descripción. La filosofía del formato EDIF es más la de un lenguaje de descripción para el intercambio de información entre herramientas de diseño que un formato para intercambio de información entre diseñadores. En cualquier caso, siempre es posible describir circuitos utilizando este lenguaje.

Un ejemplo de cómo sería el fichero EDIF que describiría un símbolo, de nombre `pruotro`, con una entrada llamada `"in"` y una salida llamada `"out"`, se puede ver a continuación:

```
(edif EDIFFILENAME (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (status
    (written
      (timeStamp 1995 2 20 18 2 40)
      (author "Ejemplo")
      (program "traductor" (version "v1.0"))
    )
  )
  (library (rename &_cosa_pardo "/u/pardo/lib")
    (edifLevel 0)
    (technology
      (numberDefinition
        (scale 1 (e 1 -6) (unit distance)))
    )
    (cell pruotro (cellType generic)
      (view pruotro (viewType netlist)
        (interface
          (port in (direction INPUT)
            (property pin (string "in"))
            (property pintype (string "in"))
          )
          (port out (direction OUTPUT)
            (property pin (string "out"))
            (property pintype (string "out"))
          )
        )
      )
    )
  )
  (design pruotro (cellRef pruotro (libraryRef &_cosa_pardo)))
)
```

Una de las características de este formato es la gran cantidad de información que se puede recoger en un único texto. En realidad, en el ejemplo anterior se mostraba el

EDIF de un único símbolo con un pin de entrada y otro de salida. Todas las sentencias iniciales son para la definición de bibliotecas, mientras que sólo las últimas sirven para describir el símbolo. Esta descripción empieza con la sentencia `(cell pruotro (cellType generic)` donde se indica que se va a describir una célula llamada internamente `pruotro`. A continuación vendría la sección de interfase donde se indican las entradas y salidas. Estas entradas y salidas se indican mediante la sentencia `port` donde se indica además si el puerto es de entrada o salida. En cada descripción de puerto vienen además sentencias indicando propiedades de puerto. Por ejemplo, el primer pin tiene dos propiedades, una que indica el nombre, llamada `pin`, y otra que indica el tipo, llamada `pintype`. Tanto el nombre de las propiedades como su valor son definibles por el usuario. Estas propiedades son importantes, ya que sirven para que otras herramientas de diseño puedan extraer información adicional sobre el circuito. Por ejemplo, en la misma descripción de puerto se podría haber incluido otra propiedad que fuera `retraso`, de manera que esta información pudiera ser utilizada por una herramienta de simulación.

2.5.2 Otros formatos de *Netlist*

Aunque el EDIF es el formato de intercambio estándar, dada su complejidad se utilizan a veces otros lenguajes de *Netlist* mucho más sencillos. Esto lo suelen hacer así los fabricantes, ya que les resulta más sencillo interpretar una descripción especialmente pensada para sus herramientas que el formato EDIF, que es tan genérico que no resulta sencillo tener una interfase. Lo que suelen hacer los fabricantes es utilizar un lenguaje propio y proveer los programas traductores necesarios para pasar de su lenguaje al EDIF y viceversa, de esta manera se aseguran la compatibilidad con el resto de herramientas, siendo las suyas propias más sencillas de realizar.

Un ejemplo de lenguaje de descripción se tiene en el Tango, cuyo lenguaje de netlist es muy simple y contempla muchas posibles descripciones, incluida la especificación de propiedades. Tango es un entorno de trabajo para PC que incluye herramientas de descripción y diseño de PCBs. Más adelante se verá un ejemplo de esta descripción.

Otro formato de *netlist*, muy usado directamente y no a partir de esquemas, es el formato de descripción de Spice. Spice es un simulador eléctrico, es decir, simula transistores, resistencias, etc., aunque también permite la simulación eléctrica y digital combinada. Este lenguaje es utilizado por el simulador para saber exactamente cómo es el circuito a simular. Está solamente indicado para ser utilizado con este programa, por lo que está limitado su uso para otros propósitos. Como ejemplo de las limitaciones que presenta se puede decir que no permite la inclusión de propiedades en el diseño.

2.5.3 Ejemplo de diferentes *Netlist*

Se presenta a continuación un circuito y su descripción usando los tres formatos que se acaban de comentar. El circuito que se pretende describir aparece en la figura 2.1 y consiste en un esquema que ha sido generado a partir de la herramienta de captura de esquemas de Tango.

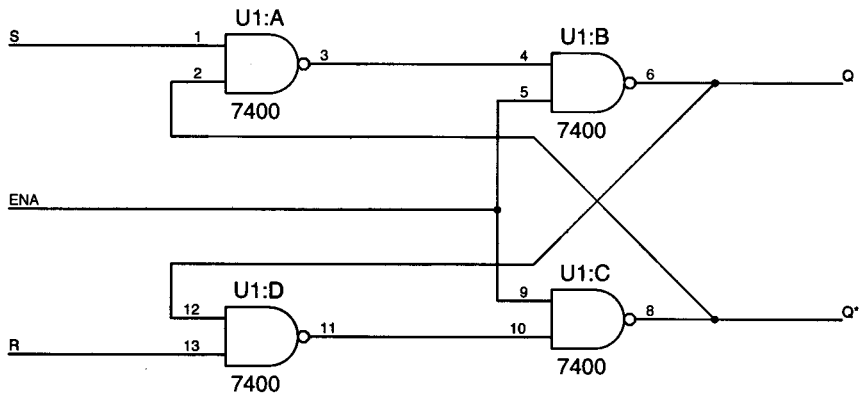


Figura 2.1: Ejemplo de esquema para su descripción Netlist

En primer lugar se presenta la descripción EDIF de este simple circuito:

```
(edif TI
 (edifVersion 2 0 0) (edifLevel 0) (keywordMap (keywordLevel 0))
 (status
  (written
   (timeStamp 1996 2 22 19 40 43)
   (dataOrigin "Esquema de Tango" (Version "1.30"))
   (comment "Ejemplo simple")
  )
 )
 (Design ROOT
  (CellRef TI
   (LibraryRef TI_LIB))
 )
 (Library TI_LIB (EdifLevel 0)
  (technology (numberDefinition (scale 1 (E 254 -7) (unit DISTANCE))))
  (cell U1 (cellType GENERIC)
   (property Type (string "7400"))
   (view S (viewType SCHEMATIC)
    (interface
     (Port A (Designator "1")
      (Direction INPUT )
     )
     (Port B (Designator "2")
      (Direction INPUT )
     )
     (Port Y (Designator "3")
      (Direction OUTPUT )
```

```
    )
    (Port A (Designator "4")
     (Direction INPUT )
    )
    (Port B (Designator "5")
     (Direction INPUT )
    )
    (Port Y (Designator "6")
     (Direction OUTPUT )
    )
    (Port GND (Designator "7")
     (property ElecType (string "Power") )
    )
    (Port Y (Designator "8")
     (Direction OUTPUT )
    )
    (Port A (Designator "9")
     (Direction INPUT )
    )
    (Port B (Designator "10")
     (Direction INPUT )
    )
    (Port Y (Designator "11")
     (Direction OUTPUT )
    )
    (Port A (Designator "12")
     (Direction INPUT )
    )
    (Port B (Designator "13")
     (Direction INPUT )
    )
    (Port VCC (Designator "14")
     (property ElecType (string "Power") )
    )
   )
  )
 )
 (cell TI (cellType GENERIC)
  (view N
   (viewType NETLIST)
   (interface)
   (Contents
    (net ENA
     (joined (portRef &B (viewRef S (cellRef U1)))
             (portRef &A (viewRef S (cellRef U1)))
     )
    )
    (net GND
     (joined (portRef &GND (viewRef S (cellRef U1)))
     )
    )
    (net NET_002
     (joined (portRef &B (viewRef S (cellRef U1)))
             (portRef &Y (viewRef S (cellRef U1)))
     )
    )
    (net NET_004
     (joined (portRef &Y (viewRef S (cellRef U1)))
             (portRef &A (viewRef S (cellRef U1)))
     )
    )
    (net Q
```

```
      (joined (portRef &Y (viewRef S (cellRef U1)))
              (portRef &A (viewRef S (cellRef U1)))
      )
     )
     (net Q*
      (joined (portRef &B (viewRef S (cellRef U1)))
              (portRef &Y (viewRef S (cellRef U1)))
      )
     )
     (net R
      (joined (portRef &B (viewRef S (cellRef U1)))
      )
     )
     (net S
      (joined (portRef &A (viewRef S (cellRef U1)))
      )
     )
     (net VCC
      (joined (portRef &VCC (viewRef S (cellRef U1)))
      )
     )
    )
   )
  )
 )
)
```

Se trata de una descripción bastante larga para lo pequeño que es el circuito, pero esto es necesario si se pretende una completa compatibilidad con el resto de herramientas. La primera parte es el encabezamiento y contiene básicamente comentarios. A continuación viene la parte de biblioteca donde se definen los componentes que aparecen. En este caso hay un único componente, representado por cuatro símbolos, que corresponde al 7400 de Texas Instruments y que contiene cuatro puertas NAND. El componente viene definido por sus puertos, cada uno con un nombre diferente y una dirección de entrada o salida. Además, en este caso, cada puerto contiene un `designator` que sirve para indicar la posición del pin dentro del encapsulado que es un DIP14. Conviene destacar que aunque en el esquema no aparecen las tensiones de alimentación, sí que aparecen en cambio en el *Netlist*. Esto es debido a que muchas veces estas tensiones, tanto VCC como GND, aparecen ocultas en los símbolos del esquema, pero son necesarias en el *Netlist* ya que éste debe contener absolutamente toda la información, incluida la alimentación del circuito.

Después de la definición del componente viene la definición del interconexionado. El emplazamiento del componente se realiza mediante la instrucción `cell`. Las interconexiones se realizan en el bloque indicado por la palabra clave `Contents`. Cada conexión se indica en un bloque `net` donde se especifica el nombre y qué cosas conecta mediante el bloque `joined`. Por ejemplo, la primera conexión (ENA) conecta los puertos A y B del chip U1 que es el 7400 especificado en la biblioteca. Y así se van conectando los diferentes elementos. Hay conexiones que no tienen nombre en el

esquema, pero todas las conexiones en un *Netlist* deben tener un nombre, así que lo que hace la herramienta en estos casos es ponerles nombres automáticamente. Éste es precisamente el caso de las conexiones NET_002 y NET_004 que son los nombres que la herramienta les ha puesto.

El listado que viene a continuación corresponde a la descripción del mismo circuito pero utilizando un netlist propio de Tango:

```
[
U1
DIP14
7400
]

(
ENA
U1-5
U1-9
)
(
GND
U1-7
)
(
NET_002
U1-10
U1-11
)
(
NET_004
U1-3
U1-4
)
(
Q
U1-6
U1-12
)
(
Q*
U1-2
U1-8
)
(
R
U1-13
)
(
S
U1-1
)
(
VCC
U1-14
)
```

Se observa que esta descripción es mucho más simple y fácil de entender que la anterior. Ello es debido a que este *Netlist* no necesita ser estándar ni ser exportado a ninguna otra herramienta, sino que debe servir únicamente para el entorno de Tango, por lo que es posible simplificar mucho más su descripción.

En la cabecera, que son las primeras líneas encerradas entre corchetes, se encuentra la parte de definición de los elementos. Aquí viene el nombre del chip (7400), su referencia dentro del esquema (U1) y una propiedad adicional que en el formato EDIF no se encontraba, y es la propiedad que indica el tipo de encapsulado del símbolo. En este caso, el valor de la propiedad de encapsulado es DIP14, que indica un encapsulado *Dual Inline Package* de catorce pines. Esto es necesario en Tango puesto que este *netlist* va a ser leído tal cual por la herramienta de diseño de PCBs, por lo que resulta interesante saber de antemano el encapsulado.

Después de la definición de los elementos que componen el esquema están las interconexiones, y vienen agrupadas entre paréntesis. La primera conexión es ENA y se conoce porque es el primer nombre después de abrir el paréntesis. A continuación del nombre vienen todos los *nodos* a los que está conectado. En el caso de ENA se ve claramente que está conectado a U1-5 y U1-9, es decir, ENA es una conexión que conecta los pines 5 y 9 del chip U1 que es el único en el esquema. Y el resto de interconexiones se realizan de la misma manera.

El último ejemplo corresponde a la descripción para Spice del mismo circuito. Se muestra que es la descripción más simple de todas, ya que sólo tiene un objetivo, y es el de ser utilizada como entrada para un programa en concreto, el simulador Spice:

```
*  TI CIRCUIT FILE
U1    S Q* 4 4 ENA Q 0 Q* ENA 2 2 Q R VCC  7400
.END
```

Toda la información del circuito se encuentra en la línea segunda, con lo que todavía es más simple de lo que parece. La primera es un comentario que además hace de título del *netlist*. En la segunda se encuentra la descripción y la última indica el final de la descripción.

La sintaxis es bien simple (línea segunda). La primera palabra indica el nombre, U1, y como empieza por la letra U, Spice ya sabe que se trata de un chip o componente. Además sabe que todos los nombres que siguen corresponden a nombres de nodos o conexiones y se corresponden con las entradas del chip. Sólo el último nombre indica de qué chip se trata, en este caso el 7400. En Spice dos nodos con el mismo nombre están conectados, así es fácil ver que la conexión ENA conecta los pines 5 y 9 del componente porque las posiciones quinta y novena del chip están marcadas como ENA.

Se han mostrado en esta sección diversos tipos de *netlist* y se han sacado algunas conclusiones. La más importante es que el *netlist* es un formato de intercambio de información en el ámbito de herramientas, cuya descripción se basa en enumerar los componentes del circuito y sus interconexiones. Otra conclusión importante es que existe un formato estándar que sirve casi para cualquier herramienta, como es el formato EDIF, y que la complejidad en la sintaxis depende de la generalidad del lenguaje utilizado. Así, el formato EDIF es el más complejo puesto que es el más genérico que existe. El resto de lenguajes, específicos para cada herramienta, pueden ser mucho mas simples, pero se pierde generalidad, ya que con la simplificación se está eliminando mucha información que puede ser útil para otro tipo de herramientas.

CAPÍTULO 3

INTRODUCCIÓN AL LENGUAJE VHDL

En el capítulo anterior se mostró que la forma más común de describir un circuito consiste en la utilización de esquemas, que son una representación gráfica de lo que se pretende realizar. Con la aparición de las herramientas de EDA cada vez más complejas, que integran en el mismo marco de trabajo tanto las herramientas de descripción como las de síntesis y realización; apareció también la necesidad de disponer de una descripción del circuito que permitiera el intercambio de información entre las diferentes herramientas que componen el ciclo de diseño.

En principio se utilizó un lenguaje de descripción que permitía, mediante sentencias simples, describir completamente un circuito. A estos lenguajes se les llamó *Netlist* puesto que eran simplemente eso, un conjunto de instrucciones que indicaban el interconexionado entre los componentes de un diseño, es decir, se trataba de una *lista de conexiones*.

A partir de estos lenguajes simples, que ya eran auténticos lenguajes de descripción hardware, se descubrió el interés que podría tener el describir los circuitos directamente utilizando un lenguaje en vez de usar esquemas. Sin embargo, se siguieron utilizando esquemas, puesto que desde el punto de vista del ser humano son mucho más sencillos de entender, aunque un lenguaje siempre permite una edición más sencilla y rápida.

Con una mayor sofisticación de las herramientas de diseño, y con la puesta al alcance de todos de la posibilidad de fabricación de circuitos integrados y de lógica programable, fue apareciendo la necesidad de poder describir los circuitos con un alto grado de abstracción, no desde el punto de vista estructural, sino desde el punto de vista funcional. Existía la necesidad de poder describir un circuito pero no desde el punto de vista de sus componentes, sino desde el punto de vista de su funcionamiento.

Este nivel de abstracción se había alcanzado ya con las herramientas de simulación. Para poder simular partes de un circuito era necesario disponer de un modelo que describiera el funcionamiento del circuito o sus componentes. Estos lenguajes estaban sobre todo orientados a la simulación, por lo que poco importaba que el nivel de abstracción fuera tan alto que no resultara sencilla una realización o síntesis a partir de dicho modelo.

Con la aparición de técnicas para la síntesis de circuitos a partir de un lenguaje de alto nivel, se utilizaron para la descripción precisamente estos lenguajes de simulación, que si bien alcanzan un altísimo nivel de abstracción, su orientación es básicamente la de simular. De esta manera, los resultados de una síntesis a partir de descripciones con estos lenguajes no es siempre la más óptima. En estos momentos no parece que exista un lenguaje de alto nivel cuya orientación o finalidad sea la de la síntesis automática de circuitos, por lo que todavía se utilizan estos lenguajes orientados a la simulación también para la síntesis de circuitos.

3.1 EL LENGUAJE VHDL

El significado de las siglas VHDL es VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language*, es decir, lenguaje de descripción hardware de circuitos integrados de muy alta velocidad. VHDL es un lenguaje de descripción y modelado diseñado para describir, en una forma en que los humanos y las máquinas puedan leer y entender la funcionalidad y la organización de sistemas hardware digitales, placas de circuitos y componentes.

VHDL fue desarrollado como un lenguaje para el modelado y simulación lógica dirigida por eventos de sistemas digitales, y actualmente se utiliza también para la síntesis automática de circuitos. El VHDL fue desarrollado de forma muy parecida al ADA debido a que el ADA fue también propuesto como un lenguaje que tuviera estructuras y elementos sintácticos que permitieran la programación de cualquier sistema hardware sin limitación de la arquitectura. El ADA tenía una orientación hacia sistemas en tiempo real y al hardware en general, por lo que se lo escogió como modelo para desarrollar el VHDL.

VHDL es un lenguaje con una sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento hardware. VHDL permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Uno de los objetivos del lenguaje VHDL es el modelado. Modelado es el desarrollo de un modelo para simulación de un circuito o sistema previamente implementado cuyo comportamiento, por tanto, se conoce. El objetivo del modelado es la simulación.

Otro de los usos de este lenguaje es la síntesis automática de circuitos. En el proceso de síntesis se parte de una especificación de entrada con un determinado nivel de abstracción y se llega a una implementación más detallada, menos abstracta. Por tanto, la síntesis es una tarea vertical entre niveles de abstracción, del nivel más alto en la jerarquía de diseño hacia el más bajo nivel de la jerarquía.

El VHDL es un lenguaje que fue diseñado inicialmente para ser usado en el modelado de sistemas digitales. Es por esta razón que su utilización en síntesis no es inmediata, aunque lo cierto es que la sofisticación de las actuales herramientas de síntesis es tal que permiten implementar diseños especificados en un alto nivel de abstracción.

La síntesis a partir de VHDL constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda de uso. Las herramientas de síntesis basadas en el lenguaje permiten en la actualidad ganancias importantes en la productividad de diseño.

Algunas ventajas del uso de VHDL para la descripción hardware son:

- VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de puertas.
- Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados por diversas herramientas de síntesis para crear e implementar circuitos.
- Los módulos creados en VHDL pueden utilizarse en diferentes diseños, lo que permite la reutilización del código. Además, la misma descripción puede utilizarse para diferentes tecnologías sin tener que rediseñar todo el circuito.
- Al estar basado en un estándar (IEEE Std 1076-1987, IEEE Std 1076-1993) los ingenieros de toda la industria de diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.
- VHDL permite diseño *Top-Down*, esto es, describir (modelar) el comportamiento de los bloques de alto nivel, analizarlos (simularlos) y refinar la funcionalidad en alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.
- Modularidad: VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.

3.1.1 VHDL describe estructura y comportamiento

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera se tiene especificado un circuito y se sabe cómo funciona. Ésta es la forma habitual en que se han venido describiendo circuitos, siendo las herramientas utilizadas para ello las de captura de esquemas y las de descripción *netlist*.

La segunda forma consiste en describir un circuito indicando lo que hace o cómo funciona, es decir, describiendo su comportamiento. Naturalmente esta forma de describir un circuito es mucho mejor para un diseñador puesto que lo que realmente le interesa es el funcionamiento del circuito más que sus componentes. Por otro lado, al encontrarse lejos de lo que es realmente un circuito, se pueden plantear algunos problemas a la hora de implementarlo a partir de la descripción de su comportamiento.

El VHDL va a ser interesante puesto que va a permitir los dos tipos de descripciones:

Estructura: VHDL puede ser usado como un lenguaje de *Netlist* normal y corriente donde se especifican por un lado los componentes del sistema y por otro sus interconexiones.

Comportamiento: VHDL también se puede utilizar para la descripción comportamental o funcional de un circuito. Esto es lo que lo distingue de un lenguaje de *Netlist*. Sin necesidad de conocer la estructura interna de un circuito es posible describirlo explicando su funcionalidad. Esto es especialmente útil en simulación, ya que permite simular un sistema sin conocer su estructura interna. Así, este tipo de descripción se está volviendo cada día más importante porque las actuales herramientas de síntesis permiten la creación automática de circuitos a partir de una descripción de su funcionamiento.

Muchas veces la descripción comportamental se divide a su vez en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones. Estas dos formas comportamentales de describir circuitos son la de *flujo de datos* y la *algorítmica*.

3.2 EJEMPLO BÁSICO Y ESTILOS DE DESCRIPCIÓN EN VHDL

VHDL presenta tres estilos de descripción de circuitos dependiendo del nivel de abstracción. El menos abstracto es una descripción puramente estructural. Los otros dos estilos representan una descripción comportamental o funcional, y la diferencia viene de la utilización o no de la ejecución serie.

Mediante un sencillo ejemplo, que además sirve para ilustrar la forma en que se escriben descripciones en VHDL, se van a mostrar estos tres estilos de descripción.

Ejemplo 3.1 *Describir en VHDL un circuito que multiplexe dos líneas de entrada (*`a` *y* `b`*) de un bit, a una sola línea (*`salida`*) también de un bit; la señal* `selec` *sirve para seleccionar la línea* `a` *(*`selec='0'`*) o* `b` *(*`selec='1'`*).*

En la figura 3.1 se muestra el circuito implementado con puertas lógicas que realiza la función de multiplexación.

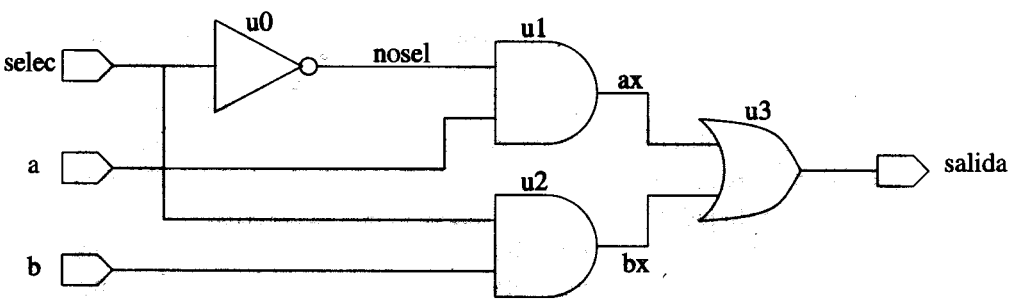


Figura 3.1: Esquema del ejemplo básico en VHDL

3.2.1 Descripción algorítmica

Lo que se va a realizar a continuación es la descripción comportamental algorítmica del circuito de la figura 3.1, después se realizará la de transferencia entre registros, que sigue siendo comportamental, y por último se verá la descripción estructural mostrando así las diferencias.

La sintaxis del VHDL no es sensible a mayúsculas o minúsculas, por lo que se puede escribir como se prefiera. A lo largo de las explicaciones se pondrán siempre las palabras clave del lenguaje en mayúsculas para distinguirlas de las variables y otros elementos. Esto no significa que durante la descripción de diseños se tenga que hacer así, de hecho resulta más rápido escribir siempre en minúsculas. Se ha hecho así en todos los ejemplos para mayor claridad del código.

En primer lugar, sea el tipo de descripción que sea, hay que definir el símbolo o **entidad** del circuito. En efecto, lo primero es definir las entradas y salidas del circuito, es decir, la caja negra que lo define. Se le llama entidad porque en la sintaxis de VHDL

esta parte se declara con la palabra clave **ENTITY**. Esta definición de entidad, que suele ser la primera parte de toda descripción VHDL, se expone a continuación:

```
-- Los comentarios empiezan con dos guiones
ENTITY mux IS
PORT ( a:      IN bit;
       b:      IN bit;
       selec:  IN bit;
       salida: OUT bit);
END mux;
```

Esta porción del lenguaje indica que la entidad `mux` (que es el nombre que se le ha dado al circuito) tiene tres entradas de tipo `bit` y una salida también del tipo `bit`. Los tipos de las entradas y salidas se verán más adelante. El tipo `bit` simplemente indica una línea que puede tomar los valores `'0'` o `'1'`.

La entidad de un circuito es única. Sin embargo, se mostró que un mismo símbolo, en este caso entidad, podía tener varias *vistas*, que en el caso de VHDL se llaman **arquitecturas**. Cada bloque de arquitectura, que es donde se describe el circuito, puede ser una representación diferente del mismo circuito. Por ejemplo, puede haber una descripción estructural y otra comportamental, ambas son descripciones diferentes, pero ambas corresponden al mismo circuito, símbolo, o entidad. Se muestra a continuación la descripción comportamental del multiplexor:

```
ARCHITECTURE comportamental OF mux IS
BEGIN
   PROCESS(a,b,selec)
   BEGIN
      IF (selec='0') THEN
         salida<=a;
      ELSE
         salida<=b;
      END IF;
   END PROCESS;
END comportamental;
```

Más adelante se verá lo que hace un bloque **PROCESS**, de momento, y como primera aproximación, se considerará que es una especie de subrutina cuyas instrucciones se ejecutan secuencialmente cada vez que alguna de las señales de la *lista sensible* cambia. Esta lista sensible es una lista de señales que se suele poner junto a la palabra clave **PROCESS**, y en el caso del ejemplo es `(a,b,selec)`.

Esta descripción comportamental es muy sencilla de entender, ya que sigue una estructura parecida a los lenguajes de programación convencionales. Es por lo que se dice

que se trata de una descripción comportamental algorítmica. Lo que se está indicando es simplemente que si la señal `selec` es cero, entonces la salida es la entrada `a`, y si `selec` es uno, entonces la salida es la entrada `b`. Esta forma tan sencilla de describir el circuito permite a ciertas herramientas sintetizar el diseño a partir de una descripción comportamental como la que se acaba de mostrar. La diferencia con un *Netlist* es directa: en una descripción comportamental no se están indicando ni los componentes ni sus interconexiones, sino simplemente lo que hace, es decir, su comportamiento o funcionamiento.

3.2.2 Descripción flujo de datos

La descripción anterior era puramente comportamental, de manera que con una secuencia sencilla de instrucciones se podría describir el circuito. Naturalmente, a veces resulta más interesante describir el circuito de forma que esté más cercano a una posible realización física del mismo. En ese sentido VHDL posee una forma de describir circuitos que además permite la paralelización de instrucciones,[1] y que se encuentra más cercana a una descripción estructural del mismo, siendo todavía una descripción funcional. A continuación se muestran dos ejemplos de una descripción concurrente, también llamada de flujo de datos o de *transferencia entre registros*:

```
ARCHITECTURE flujo1 OF mux IS
SIGNAL nosel,ax,bx: bit;
BEGIN
  nosel<=NOT selec;
  ax<=a AND nosel;
  bx<=b AND selec;
  salida<=ax OR bx;
END flujo1;
```

```
ARCHITECTURE flujo2 OF mux IS
BEGIN
  salida<=a WHEN selec='0' ELSE
          b;
END flujo2;
```

En la descripción de la izquierda hay varias instrucciones todas ellas concurrentes, es decir, se ejecutan cada vez que cambia alguna de las señales que intervienen en la asignación. Este primer caso es casi una descripción estructural, ya que de alguna manera se están describiendo las señales (cables) y los componentes que la definen; aunque no es estructural, ya que en realidad se trata de asignaciones a señales y no una lista de componentes y conexiones. El segundo caso (derecha) es también una descripción de flujo de datos, aunque basta una única instrucción de asignación para definir el circuito.

[1] Un lenguaje que describa hardware debe permitir ejecución paralela o, lo que es lo mismo, instrucciones concurrentes.

3.2.3 Descripción estructural

Aunque no es la característica más interesante del VHDL, también permite ser usado como *Netlist* o lenguaje de descripción de estructura. En este caso esta estructura también estaría indicada dentro de un bloque de arquitectura, aunque la sintaxis interna es completamente diferente:

```
ARCHITECTURE estructura OF mux IS
SIGNAL ax,bx,nosel: bit;
BEGIN
  u0: ENTITY inv PORT MAP(e=>selec,y=>nosel);
  u1: ENTITY and2 PORT MAP(e1=>a,e2=>nosel,y=>ax);
  u2: ENTITY and2 PORT MAP(b,selec,bx);
  u3: ENTITY or2  PORT MAP(e1=>ax,e2=>bx,y=>salida);
END estructura;
```

Se observa fácilmente que esta descripción es un poco más larga y mucho menos clara que las anteriores. En el cuerpo de la arquitectura se hace lo que en un *netlist* normal, es decir, se ponen los componentes y sus interconexiones. Para los componentes se utilizarán entidades que estarán definidas en alguna biblioteca, y para la conexiones se usarán señales que se declararán al principio de la arquitectura.

Al igual que ocurre en cualquier *netlist*, las señales o conexiones deben tener un nombre. En el esquema se le han puesto nombres a las líneas de conexión internas al circuito. Estas líneas hay que declararlas como SIGNAL en el cuerpo de la arquitectura y delante del BEGIN. Una vez declaradas las señales que intervienen se procede a conectar entre sí las señales y entidades que representan componentes. Para ello la sintaxis es muy simple. Lo primero es identificar y poner cada componente, que es lo que comúnmente se conoce como *replicación*, es decir, asignarle a cada componente concreto un símbolo. En este ejemplo se le ha llamado `u` a cada componente y se le ha añadido un número para distinguirlos, en principio el nombre puede ser cualquier identificador válido y la única condición es que no haya dos nombres iguales. A continuación del nombre viene la entidad correspondiente al componente, que en este caso puede ser una `and2`, una `or2` o una puerta inversora `inv`. Después se realizan las conexiones poniendo cada señal en su lugar correspondiente con las palabras PORT MAP. Así, los dos primeros argumentos en el caso de la puerta `and2` son las entradas, y el último es la salida. De esta forma tan simple se va creando el *netlist* o definición de la estructura.

Aunque los dos primeros ejemplos se ajustan al VHDL'87 y al VHDL'93, el último sólo es válido para el VHDL'93. En la primera versión de VHDL no se permite la referencia directa a la entidad, por lo que se hace necesaria la definición de un componente

y luego el enlace, en un bloque de configuración, del componente con la entidad que se quiera. Como es mucho más largo, y en este ejemplo sólo se pretendían mostrar las posibilidades del VHDL como lenguaje puramente estructural, se ha preferido utilizar esta última forma que además es más moderna y actual. Es el mismo ejemplo estructural, con componentes y bloque de configuración, que se encuentra en la sección 7.2.1.

3.3 VHDL'87 Y VHDL'93

VHDL es efectivamente un lenguaje estándar. Sin embargo, y como ocurre en la mayoría de lenguajes estándar, existen varias versiones.

Después de años de trabajo y definición apareció el primer VHDL estándar, fue en el año 1987 y de ahí su nombre, VHDL'87. Esta primera especificación fue estandarizada por el IEEE y su número de registro fue el 1076, así que la referencia correcta de esta versión es IEEE Std 1076-1987.

Esta primera versión pronto mostró algunas carencias, especialmente en lo relacionado con la síntesis de circuitos. Esto fue debido sobre todo a la rápida evolución de las herramientas que utilizaban el VHDL, y también a la diseminación del lenguaje en toda la comunidad científica. Durante varios años se recogieron las experiencias obtenidas del uso del primer estándar y se planteó un segundo que sigue vigente hoy día. Fue en 1993 y se trata del IEEE Std-1076-1993, que es conocido comúnmente como VHDL'93.

Ambas especificaciones son bastante parecidas y sólo tienen pequeñas diferencias. Se puede decir que el VHDL'87 es un subconjunto del VHDL'93, de manera que cualquier descripción antigua va a ser entendida por herramientas nuevas que, en general, poseen la capacidad de entender las nuevas construcciones del VHDL'93. Una excepción a esta regla general se encuentra en la declaración y uso de los ficheros.

En este libro se ha intentado siempre utilizar la sintaxis del VHDL'87, ya que es compatible con la del 93; sin embargo, en aquellos sitios donde la estructura del VHDL'93 es más clara, se ha utilizado la versión del 93. A lo largo de los diferentes capítulos se han ido comentando también las diferencias entre el 87 y el 93, pero si bien no se resaltan todas, sí que se han escogido aquellas que van a ser más importantes. En general, se puede decir que es más sencillo utilizar la versión del 93 ya que impone menos restricciones, pero no todas las herramientas actuales de simulación o síntesis están todavía preparadas para entenderlo.

En la siguiente lista se enumeran la mayoría de las incorporaciones realizadas al VHDL'93 frente a la versión del 87:

- Se ha dado consistencia a la delimitación de varias estructuras, en concreto a las entidades, arquitecturas, configuraciones, paquetes, componentes, procedimientos, funciones, procesos, senencias de generación, sentencias condicionales y de selección, bucles y registros.
- Referencia directa de entidades y configuración, lo que hace innecesario el uso de los componentes.
- Se añade la posibilidad de abrir y cerrar ficheros, así como la de abrir para añadir.
- Es posible el uso de variables compartidas.
- Se definen nuevos atributos como `'image` y `'path_name`.
- Sentencia de informe `report` para escribir mensajes.
- Nuevos operadores que incluyen todos los de rotación y desplazamiento, además del XNOR.
- Generalización de las formas Octal y Hexadecimal, que sólo se podían usar con el tipo `bit_vector`, y que ahora también sirven para el tipo `std_logic_vector`.
- Incorporación de la posibilidad UNAFFECTED en las asignaciones condicionales.
- Los puertos en modo entrada (IN) se pueden conectar a constantes en el PORT MAP.
- Se añade la posibilidad REJECT para las asignaciones a señal con retraso inercial.
- Se admite la posibilidad de posponer la ejecución de procesos al final del ciclo de simulación, son los procesos POSTPONED.
- Las funciones se pueden declarar como puras o impuras.
- Se pueden realizar declaraciones dentro de las sentencias de referencia.
- A cualquier elemento sintáctico se le puede poner un alias.
- Se añaden los grupos para permitir el paso de información entre herramientas.
- Se permiten etiquetas para las sentencias en serie.
- Los identificadores pueden contener casi cualquier carácter imprimible.
- Los ficheros se usan y declaran de forma diferente.

CAPÍTULO 4

ELEMENTOS SINTÁCTICOS DEL VHDL

VHDL es verdaderamente un lenguaje, por lo que tiene sus elementos sintácticos, sus tipos de datos y sus estructuras como cualquier otro tipo de lenguaje. El hecho de que sirva para la descripción hardware lo hace un poco diferente de un lenguaje convencional. Una de estas diferencias es probablemente la posibilidad de ejecutar instrucciones a la vez de forma *concurrente*, aunque naturalmente existen muchos lenguajes que permiten esto.

Algunos de los elementos sintácticos son los siguientes:

Comentarios: Cualquier línea que empieza por dos guiones "`--`" es un comentario.

Símbolos especiales Además de las palabras reservadas, que se enumeran más adelante, existen unos símbolos o caracteres especiales. Los hay de un sólo carácter: `+ - / * ( ) . , : ; & ' " < > = | #` y los hay de dos caracteres: `** => := /= >= <= <> --`

Identificadores: Es lo que se usa para dar nombre a los diferentes objetos del lenguaje como variables, señales, nombres de rutina, etc. Puede ser cualquier nombre compuesto por letras y números, incluyendo el símbolo de subrayado "_", nunca puede contener ninguno de los símbolos especiales ni puede empezar por un número o subrayado; tampoco se permite que el identificador acabe con un subrayado ni que haya dos seguidos. Por último, no debe haber ningún identificador que coincida con alguna de las palabras clave del VHDL. Las mayúsculas y minúsculas son consideradas iguales, así que `HOLA`, `hola` y `HoLa` representan el mismo objeto.

Números: Cualquier número se considera que se encuentra en base 10. Se admite la notación científica convencional para números en coma flotante. Es posible

poner números en otras bases utilizando el símbolo del sostenido "#". Ejemplo: `2#11000100#` y `16#C4#` representan el entero `196`.

Caracteres: Es cualquier letra o carácter entre comillas simples: `'1','3','t'`.

Cadenas: Son un conjunto de caracteres englobados por comillas dobles: `"Esto es una cadena"`.

Cadenas de bits: Los tipos `bit` y `bit_vector` son en realidad de tipo carácter y matriz de caracteres respectivamente. En VHDL se tiene una forma elegante de definir números con estos tipos, y es mediante la cadena de bits. Dependiendo de la base en que se especifique el número se puede poner un prefijo B (binario), O (octal), o X (hexadecimal). Ejemplo: `B"11101001", O"126", X"FE"`.

Palabras reservadas: Las palabras reservadas en VHDL, o palabras clave, son aquellas que tienen un significado especial. Son las instrucciones, órdenes y elementos que permiten definir sentencias. Por esta razón, no se deben utilizar como identificadores, ya que tienen un significado diferente.

Las palabras clave del VHDL'87 son las siguientes:

```
ABS             CONFIGURATION   INOUT       OR          THEN
ACCESS          CONSTANT        IS          OTHERS      TO
AFTER           DISCONNECT      LABEL       OUT         TRANSPORT
ALIAS           DOWNTO          LIBRARY     PACKAGE     TYPE
ALL             ELSE            LINKAGE     PORT        UNITS
AND             ELSIF           LOOP        PROCEDURE   UNTIL
ARCHITECTURE    END             MAP         PROCESS     USE
ARRAY           ENTITY          MOD         RANGE       VARIABLE
ASSERT          EXIT            NAND        RECORD      WAIT
ATTRIBUTE       FILE            NEW         REGISTER    WHEN
BEGIN           FOR             NEXT        REM         WHILE
BLOCK           FUNCTION        NOR         REPORT      WITH
BODY            GENERATE        NOT         RETURN      XOR
BUFFER          GENERIC         NULL        SELECT
BUS             GUARDED         OF          SEVERITY
CASE            IF              ON          SIGNAL
COMPONENT       IN              OPEN        SUBTYPE
```

En el VHDL'93 se completó el lenguaje con las siguientes palabras clave:

```
GROUP           POSTPONED       ROR         SRA
IMPURE          PURE            SHARED      SRL
INERTIAL        REJECT          SLA         UNAFFECTED
LITERAL         ROL             SLL         XNOR
```

4.1 OPERADORES Y EXPRESIONES

Las expresiones en VHDL son prácticamente iguales a como pudieran ser en otros lenguajes de programación o descripción, por lo que se expondrán brevemente los existentes en VHDL y su utilización.

4.1.1 Operador de concatenación

& (concatenación) Concatena matrices de manera que la dimensión de la matriz resultante es la suma de las dimensiones de las matrices sobre las que opera: Por ejemplo, `punto<=x&y` construye la matriz `punto` con la matriz `x` en las primeras posiciones, y la matriz `y` en las últimas.

4.1.2 Operadores aritméticos

** (exponencial) Sirve para elevar un número a una potencia: `4**2` es 4^2. El operador de la izquierda puede ser entero o real, pero el de la derecha sólo puede ser entero.

ABS() (valor absoluto) Como su propio nombre indica esta función devuelve el valor absoluto de su argumento que puede ser de cualquier tipo numérico.

* (multiplicación) Sirve para multiplicar dos números de cualquier tipo numérico (los tipos `bit` o `bit_vector` no son numéricos).

/ (división) También funciona con cualquier dato de tipo numérico.

MOD (módulo) Calcula el módulo de dos números. Se define el módulo como la operación que cumple: `a=b*N+(a MOD b)` donde `N` es un entero. Los operandos sólo pueden ser enteros. El resultado toma el signo de `b`.

REM (resto) Calcula el resto de la división entera y se define como el operador que cumple: `a=(a/b)*b+(a REM b)`, siendo la división entera. Los operandos sólo pueden ser enteros. El resultado toma el signo de `a`.

+ (suma y signo positivo) Este operador sirve para indicar suma, si va entre dos operandos, o signo si va al principio de una expresión. La precedencia es diferente en cada caso. Opera sobre valores numéricos de cualquier tipo.

– (resta y signo negativo) Cuando va entre dos operandos se realiza la operación de sustracción, y si va delante de una expresión le cambia el signo. Los operandos pueden ser numéricos de cualquier tipo.

4.1.3 Operadores de desplazamiento

Estos operadores no existen en el VHDL'87, pero sí en el VHDL'93.

SLL, SRL (desplazamiento lógico a izquierda y a derecha) Desplaza un vector un número de bits a izquierda (SLL) o derecha (SRL) rellenando con ceros los huecos libres. Se utiliza en notación infija de manera que la señal a la izquierda del operador es el vector que se quiere desplazar y el de la derecha es un valor que indica el número de bits a desplazar. Por ejemplo `dato SLL 2` desplaza a izquierda el vector `dato`, es decir, lo multiplica por 4.

SLA, SRA (desplazamiento aritmético a izquierda y derecha). La diferencia con el anterior es que este desplazamiento conserva el signo, es decir, conserva el valor que tuviera el bit más significativo del vector.

ROL, ROR (rotación a izquierda y a derecha) Es como el de desplazamiento, pero los huecos que se forman son ocupados por los bits que van saliendo.

4.1.4 Operadores relacionales

Devuelven siempre un valor de tipo booleano (`true` o `false`). Los tipos con los que pueden operar dependen de la operación:

`=, /=` (igualdad) El primero devuelve `true` si los operandos son iguales y `false` en caso contrario. El segundo indica desigualdad, así que funciona justo al revés. Los operandos pueden ser de cualquier tipo con la condición de que ambos sean del mismo tipo.

`<, <=, >, >=` (menor mayor) Tienen el significado habitual. La diferencia con los anteriores es que los tipos de datos que pueden manejar son siempre de tipo escalar o matrices de una sola dimensión de tipos discretos.

4.1.5 Operadores lógicos

Son NOT, AND, NAND, OR, NOR, XOR y en el VHDL'93 se añadió XNOR. El funcionamiento es el habitual para este tipo de operadores. Actúan sobre los tipos `bit`, `bit_vector` y `boolean`. En el caso de realizarse estas operaciones sobre un vector, la operación se realiza bit a bit, incluyendo la operación NOT que nada tiene que ver con el signo.

4.1.6 Precedencia de operadores

La precedencia de operadores se presenta en la tabla 4.1, encontrándose ordenados de mayor precedencia (arriba) a menor (abajo). Los operadores que se encuentran en la misma fila tienen la misma precedencia, por lo tanto, en una expresión, se evaluará primero siguiendo el orden de izquierda a derecha de la expresión. Al igual que en otros lenguajes, los paréntesis sirven para evaluar primero operadores con menor precedencia.

Mayor	**	ABS	NOT				
	*	/	MOD	REM			
	+(signo)	-(signo)					
	+	-	&				
	=	/=	<	<=	>	>=	
Menor	AND	OR	NAND	NOR	XOR	XNOR	

Tabla 4.1: Operadores ordenados por precedencia

Se ha visto que, por ejemplo, los operadores lógicos sólo operaban sobre unos tipos de datos determinados. Existe en VHDL la posibilidad de *sobrecargar* operadores y funciones, como se verá más adelante, de manera que es posible extender la aplicación de estos operadores para que trabajen con otros tipos aparte de los predefinidos. Así se podrían redefinir los operadores lógicos para que pudieran trabajar sobre enteros.

4.2 TIPOS DE DATOS

La sintaxis del VHDL es estricta con respecto a los tipos. Cualquier objeto en VHDL debe tener un tipo. En VHDL no existen tipos propios del lenguaje como pueden ser el tipo *real* o *integer*, lo que tiene son los mecanismos para poder definir cualquier tipo incluidos éstos.

Cuando se compila el programa se carga una parte de código previa que se encuentra en una biblioteca. Esta parte de código es común a todas las herramientas de VHDL y contiene una serie de definiciones de tipos y funciones que, al ser comunes a todas las herramientas, compiladores, simuladores, etc., casi parecen formar parte del propio lenguaje, pero no es así. De esta forma, por ejemplo, existe un tipo que se llama precisamente `integer`, pero este tipo no es propio del lenguaje sino que se carga al inicio junto con otros tipos predefinidos. A continuación se muestran las posibles declaraciones de tipos que se pueden hacer y se presenta cómo están especificados estos tipos predefinidos.

Como en cualquier lenguaje, VHDL tiene dos grupos de tipos de datos. Por un lado están los *escalares*, con los que se pueden formar el otro grupo, que son los *compuestos*.

4.2.1 Tipos escalares

Son tipos simples que contienen algún tipo de magnitud. A continuación se muestra cómo se pueden crear tipos escalares y se verán algunos de los predefinidos en VHDL:

Enteros: Son datos cuyo contenido constituye un valor numérico entero. La forma en que se definen estos datos es mediante la palabra clave RANGE, es decir, no se dice que un dato es de tipo entero, sino que está comprendido en cierto intervalo especificando los límites de dicho intervalo con valores enteros.

Ejemplos:

```
TYPE byte IS RANGE 0 TO 255;
TYPE index IS RANGE 7 DOWNTO 1;
TYPE integer IS RANGE -2147483648 TO 2147483647;  -- Predefinido
```

Este último tipo viene ya predefinido en el lenguaje; su utilización es interesante, pero hay que tener cuidado y especificar mejor un rango, especialmente pensando en la posterior síntesis del circuito.

Reales: Conocidos también como coma flotante son los tipos que definen un número real. Al igual que los enteros se definen mediante la palabra clave RANGE, con la diferencia de que los límites son números reales.

Ejemplos:

```
TYPE nivel IS RANGE 0.0 TO 5.0;
TYPE real IS RANGE -1.0E38 TO 1.0E38;  -- Predefinido
```

Físicos: Como su propio nombre indica se trata de datos que se corresponden con magnitudes físicas, es decir, tienen un valor y unas unidades.

Ejemplo:

```
TYPE longitud IS RANGE 0 TO 1.0e9
  UNITS
     um;
     mm=1000 um;
     m=1000 mm;
     inch=25.4 mm;
  END UNITS;
```

Hay un tipo físico predefinido en VHDL que es `time`. Este tipo se utiliza para indicar retrasos y tiene todos los submúltiplos, desde `fs` (femtosegundos) hasta `hr` (horas). Cualquier dato físico se escribe siempre con su valor seguido de la unidad: `10 mm, 1 um, 23 ns`.

Enumerados: Son datos que pueden tomar cualquier valor especificado en un conjunto finito o lista. Este conjunto se indica mediante una lista encerrada entre paréntesis de elementos separados por comas.

Ejemplos:

```
TYPE nivel_logico IS (nose,alto,bajo,Z);
TYPE bit IS ('0','1');          -- Predefinido en el lenguaje
```

Hay varios tipos enumerados que se encuentran predefinidos en VHDL. Estos tipos son: `severity_level, boolean, bit` y `character`.

4.2.2 Tipos compuestos

Son tipos de datos que están compuestos por tipos escalares como los vistos anteriormente.

Matrices: Son una colección de elementos del mismo tipo a los que se accede mediante un índice. Su significado y uso no difiere mucho de la misma estructura presente en casi todos los lenguajes de programación. Los hay monodimensionales (un índice) o multidimensionales (varios índices). A diferencia de otros lenguajes, las matrices en VHDL pueden estar enmarcadas en un rango, o el índice puede ser libre teniendo la matriz una dimensión teórica infinita.

Ejemplos:

```
TYPE word IS ARRAY(31 DOWNTO 0) OF bit;
TYPE transformada IS ARRAY(1 TO 4, 1 TO 4) OF real;
TYPE positivo IS ARRAY(byte RANGE 0 TO 127) OF integer;
TYPE string IS ARRAY(positive RANGE <>) OF character;   -- Predef.
TYPE bit_vector IS ARRAY(natural RANGE <>) OF bit;      -- Predef.
TYPE vector IS ARRAY(integer RANGE <>) OF real;
```

Estos tres últimos ejemplos muestran una matriz cuyo índice no tiene rango sino que sirve cualquier entero. Más tarde, en la declaración del dato, se deberán poner los límites de la matriz: `SIGNAL dato: vector(1 TO 20);`

A los elementos de una matriz se accede mediante el índice. Así `dato(3)` es el elemento 3 del vector dato. De la misma manera se puede acceder a un rango: `datobyte<=datoword(2 TO 9)`. También se pueden utilizar en las asignaciones

lo que se conoce como agregados o conjuntos (*aggregate*), que no es más que una lista separada por comas de manera que al primer elemento de la matriz se le asigna el primer elemento de la lista, y así sucesivamente. A continuación se muestran algunos ejemplos:

```
semaforo<=(apagado,encendido,apagado);
dato<=(datohigh,datolow);
triestado<=(OTHERS=>'Z');
```

En este último se ha empleado la palabra clave OTHERS para poner todos los bits del triestado a 'Z', de esta forma ponemos un vector a un valor sin necesidad de saber cuántos bits tiene la señal.

Registros: Es equivalente al tipo registro o *record* de otros lenguajes.

Ejemplo:

```
TYPE trabajador IS
  RECORD
    nombre: string;
    edad: integer;
  END RECORD;
```

Para referirse a un elemento dentro del registro se utiliza la misma nomenclatura que en Pascal, es decir, se usa un punto entre el nombre del registro y el nombre del campo: `persona.nombre="Carlos"`

4.2.3 Subtipos de datos

VHDL permite la definición de subtipos que son restricciones o subconjuntos de tipos existentes. Hay dos tipos. El primero es el formado por subtipos obtenidos a partir de la restricción de un tipo escalar a un rango. Ejemplos:

```
SUBTYPE raro IS integer RANGE 4 TO 7;
SUBTYPE digitos IS character RANGE '0' TO '9';
SUBTYPE natural IS integer RANGE 0 TO entero_mas_alto;  -- Predefinido
SUBTYPE positive IS integer RANGE 1 TO entero_mas_alto; -- Predefinido
```

El segundo tipo de subtipos es el formado por aquellos que restringen el rango de una matriz:

```
SUBTYPE id IS string(1 TO 20);
SUBTYPE word IS bit_vector(31 DOWNTO 0);
```

Los subtipos sirven además para crear *tipos resueltos*, que es una clase especial de tipos que se explicará en detalle en la sección 9.1.

La ventaja de utilizar un subtipo es que las mismas operaciones que servían para el tipo también sirven para el subtipo. Una función que opere sobre enteros funcionará igual de bien sobre un subrango de enteros (siempre que la función implemente el chequeo de límites). En cambio, una función definida sólo para un rango sólo podrá usarse para ese rango, con lo que se pierde generalidad.

Esto tiene especial importancia cuando se describe un circuito para ser sintetizado, ya que si se utiliza `integer` sin más, esto se interpretará como un bus de 32 líneas (puede cambiar dependiendo de la herramienta) y lo más probable es que en realidad se necesiten muchas menos.

Otro caso se da cuando se tiene una lista de elementos y se les quiere asignar un entero a cada una, dependiendo de las operaciones que se quieran realizar puede resultar más conveniente definir un subtipo a partir de `integer` que crear un tipo enumerado.

4.3 ATRIBUTOS

Los elementos en VHDL, como señales, variables, etc., pueden tener información adicional llamada *atributos*. Estos atributos están asociados a estos elementos del lenguaje y se manejan en VHDL mediante la comilla simple " ' ". Por ejemplo, `t'left` indica el atributo `'left` de `t` que, en este caso, es un tipo escalar (este atributo indica el límite izquierdo del rango).

Hay algunos de estos atributos que están predefinidos en el lenguaje y a continuación se muestran los más interesantes. Suponiendo que `t` es un tipo enumerado, entero, flotante, o físico, se tienen los siguientes atributos:

`t'left` Límite izquierdo del tipo `t`.

`t'right` Límite derecho del tipo `t`.

`t'low` Límite inferior del tipo `t`.

`t'high` Límite superior del tipo `t`.

Suponiendo un tipo `t` como el anterior, un miembro `x` de ese tipo, y un entero `N`, se pueden utilizar los siguientes atributos:

`t'pos(x)` Posición de `x` dentro del tipo `t`.

`t'val(N)` Elemento `N` del tipo `t`.

`t'leftof(x)` Elemento que está a la izquierda de `x` en `t`.

`t'rightof(x)` Elemento que está a la derecha de `x` en `t`.

`t'pred(x)` Elemento que está delante de `x` en `t`.

`t'succ(x)` Elemento que está detrás de `x` en `t`.

Si `a` es un tipo matriz, u elemento de éste, y `N` es un entero desde 1 hasta el número de dimensiones de la matriz, se pueden usar los siguientes atributos:

`a'left(N)` Límite izquierdo del rango de dimensión `N` de `a`.

`a'right(N)` Límite derecho del rango de dimensión `N` de `a`.

`a'low(N)` Límite inferior del rango de dimensión `N` de `a`.

`a'high(N)` Límite superior del rango de dimensión `N` de `a`.

`a'range(N)` Rango del índice de dimensión `N` de `a`.

`a'length(N)` Longitud del índice de dimensión `N` de `a`.

Suponiendo que `s` es una señal, se pueden utilizar los siguientes atributos (se han elegido los más interesantes):

`s'event` Devuelve `true` si se ha producido un cambio en la señal `s`.

`s'stable(tiempo)` Devuelve `true` si la señal estuvo estable durante el último periodo `tiempo`.

El atributo `'event` es especialmente útil en la definición de circuitos secuenciales para detectar el flanco de subida o bajada de la señal de reloj. Por esto es probablemente el atributo más utilizado en VHDL.

4.3.1 Atributos definidos por el usuario

En muchas ocasiones resulta muy interesante agregar información adicional a los objetos que se están definiendo en VHDL. Estas informaciones adicionales, propiedades o atributos sirven para pasar información a las herramientas de diseño que se estén utilizando en VHDL. Así, algunas herramientas de síntesis permiten la inclusión de determinados atributos en el diseño que van a permitir guiar la síntesis en una determinada dirección, por ejemplo, unas partes del diseño se pueden optimizar para tener un tamaño reducido, y otras para que los retrasos sean menores. Otro ejemplo son las herramientas de diseño de PCBs donde es importante saber el encapsulado que tiene cada componente, el pin que ocupa cada señal en el chip, etc.

La diferencia con los parámetros declarados en GENERIC, aparte de que se pueden llamar desde el programa como si fueran atributos normales, es que se pueden aplicar a muchos más objetos. Así, se pueden poner atributos a las entidades, arquitecturas, configuraciones, procedimientos, funciones, paquetes, tipos, subtipos, constantes, señales, variables, componentes, y etiquetas de instrucción.

Lo que devuelve un atributo definido por el usuario es siempre una constante. Para definir el atributo primero hay que declarar ese atributo, diciendo qué tipo de elemento devuelve, y luego especificar el valor que devuelve en cada caso. La forma general es por tanto:

```
-- Declaración:
ATTRIBUTE nombre : tipo;

-- Especificación:
ATTRIBUTE nombre OF id_elemento : clase_elemento IS valor;
```

El `id_elemento` es un objeto concreto del VHDL declarado en alguna parte, aunque puede incluir las palabras ALL u OTHERS para referirse a todos, o resto, de los elementos de la `clase_elemento`, que es la clase de objeto del que se trata. Esta clase se refiere a los elementos sintácticos que se declaran en el código, como por ejemplo señales, componentes, procedimientos, variables, configuraciones, entidades, y un largo etcétera. Por último se le da un `valor` al atributo que será del tipo especificado en la declaración.

Un ejemplo típico de utilización de los atributos se encuentra en añadir propiedades de encapsulado y pin a la entidad y puertos que contiene.

```
  SIGNAL control: std_logic;
  COMPONENT ne555 ... END COMPONENT;
  ATTRIBUTE pin: integer;
  ATTRIBUTE encapsulado: string;
  ATTRIBUTE pin OF control: SIGNAL IS 5;
  ATTRIBUTE encapsulado OF ne555: COMPONENT IS "dip8";
BEGIN
  IF control'pin>4 THEN ...
  IF ne555'encapsulado="dip8" THEN ...
END;
```

4.4 DECLARACIÓN DE CONSTANTES, VARIABLES Y SEÑALES

Un elemento en VHDL contiene un valor de un tipo especificado. Hay tres tipos de elementos en VHDL: las variables, las señales y las constantes. Las variables y constantes se parecen mucho a las variables y constantes que se encuentran en cualquier lenguaje. Las señales, en cambio, son elementos cuyo significado es bastante diferente y es consecuencia directa de que, aunque VHDL es un lenguaje muy parecido a los convencionales, no deja en ningún momento de ser un lenguaje de descripción hardware, por lo que cabe esperar algunas diferencias.

4.4.1 Constantes

Una constante es un elemento que se inicializa a un determinado valor que no puede ser cambiado una vez inicializado, sino que se conserva para siempre. Ejemplos:

```
CONSTANT e: real := 2.71828;
CONSTANT retraso: time := 10 ns;
CONSTANT max_size: natural;
```

En la última sentencia la constante `max_size` no tiene ningún valor asociado. Esto se permite siempre y cuando el valor sea declarado en algún otro sitio, y se hace así para las declaraciones en *paquetes* que se verán más adelante.

4.4.2 Variables

Una variable en VHDL es similar al concepto de variable en otros lenguajes. Se diferencia de una constante en que su valor puede ser alterado en cualquier instante. A las variables también se les puede asignar un valor inicial.

```
VARIABLE contador: natural := 0;
VARIABLE aux: bit_vector(31 DOWNTO 0);
```

Es posible, dado un elemento previamente definido, cambiarle el nombre o ponerle nombre a una parte. Esto se realiza mediante la instrucción ALIAS que suele resultar muy útil. Ejemplo:

```
VARIABLE instruccion: bit_vector(31 DOWNTO 0);
ALIAS codigo_op: bit_vector(7 DOWNTO 0) IS instruccion(31 DOWNTO 24);
```

4.4.3 Señales

Las señales se declaran igual que las constantes y variables con la diferencia de que las señales pueden ser *normal*, *register* y *bus*. Si en la declaración no se especifica nada, se entiende que la señal es de tipo *normal*; para que sea de tipo *bus* o *register* se debe declarar explícitamente con las palabras clave BUS o REGISTER.

Una señal no es lo mismo que una variable. La señal no es exactamente un objeto del lenguaje que guarda un valor, sino que lo que hace en realidad es guardar un valor (o varios como se verá más adelante) y hacerlo visible en el momento adecuado. Esto es, se puede decir que la señal tiene dos partes, una donde se escribe y que almacena el valor, y otra que se lee y que no tiene por qué coincidir con lo que se acaba de escribir.

Internamente se establece una conexión entre la parte que se escribe y la que se lee. Es posible en VHDL desconectar una parte de la otra, de manera que al leer la señal no se tenga acceso a lo que se escribió. La desconexión de una señal se hace asignándole el valor NULL. La diferencia entre una señal *normal*, *register* o *bus* viene del comportamiento al desconectarse. Las de tipo normal no se pueden desconectar. Las de clase *bus* y *register* se pueden desconectar, ya que o bien se trata de señales de tipo resuelto, que admiten varios o ningún valor escrito a un tiempo, o bien de tipo *vigiladas*, que tienen una condición de desconexión. La diferencia entre ambas es que la señal de tipo *bus* tiene un valor por defecto cuando todas las fuentes de la señal están desconectadas, y las de clase *register* no tienen un valor por defecto pero conservan el último valor que se escribió.

Al igual que en variables y constantes, a las señales también se les puede dar un valor inicial si se quiere. Ejemplos:

```
SIGNAL selec: bit := '0';
SIGNAL datos: bit_vector(7 DOWNTO 0) BUS := B"00000000";
```

4.4.4 Comparación entre constantes, señales y variables

Constantes, señales y variables son cosas diferentes. Las variables, por ejemplo, sólo tienen sentido dentro de un bloque (PROCESS) o un subprograma, es decir, sólo tienen sentido en entornos de programación donde las sentencias son ejecutadas en serie, por tanto las variables sólo se declaran en los procesos o subprogramas. Las señales pueden ser declaradas únicamente en las arquitecturas, paquetes (PACKAGE), o en los bloques concurrentes (BLOCK). Las constantes pueden ser habitualmente declaradas en los mismos sitios que las variables y señales.

Las variables son elementos abstractos del lenguaje, lo que significa que no tienen por qué tener una concordancia física real inmediata. Sin embargo, las señales poseen un significado físico inmediato, y es el de representar conexiones reales en el circuito. Las señales pueden ser usadas en cualquier parte del programa o descripción y son declaradas siempre en la parte de arquitectura antes del BEGIN. Esto indica que las señales son visibles por todos los procesos y bloques dentro de una arquitectura, por lo que en realidad representan interconexiones entre bloques dentro de la arquitectura.

Desde un punto de vista software, las señales representan el mecanismo que va a permitir ejecutar en paralelo las instrucciones concurrentes, es decir, VHDL implementa el mecanismo de *sincronización de procesos por monitorización* para la ejecución paralela de instrucciones.

En un diseño las conexiones físicas entre unos elementos y otros son habitualmente declaradas como señales. Las entradas y salidas, definidas en la entidad, son, por lo tanto, consideradas señales. Aunque estas entradas y salidas son en realidad señales, hay algunas diferencias; por ejemplo, las salidas no se pueden leer, es decir, no pueden formar parte del argumento de una asignación. De la misma manera, a una entrada no se le puede asignar un valor en la descripción.

La diferencia principal entre variables y señales es que una asignación a una variable se realiza de forma inmediata, es decir, la variable toma el valor que se le asigna en el mismo momento de la asignación. La señal, en cambio, no recibe el valor que se le está asignando hasta el siguiente paso de simulación, es decir, cuando el proceso se acaba al encontrar una sentencia WAIT, o al final de éste si no tiene sentencias de espera. Esta forma extraña en que se les asignan valores a las señales se entenderá mejor cuando se explique el significado de los procesos, la ejecución concurrente y secuencial, y los pasos de simulación en el capítulo 10.

4.5 DECLARACIÓN DE ENTIDAD Y ARQUITECTURA

Ya se ha visto, en anteriores ejemplos, cómo se declaran tanto las entidades como las arquitecturas. Se verá a continuación que en la declaración de estas estructuras se pueden incluir otros elementos, aunque en la mayoría de los casos tanto la entidad como la arquitectura se declaran de las formas vistas hasta el momento.

4.5.1 Declaración de entidad

La entidad es la parte del programa que define el símbolo. Es decir, define las entradas y salidas del circuito. Además, la entidad es la estructura que permite realizar diseños jerárquicos en VHDL, ya que un diseño jerárquico es generalmente una colección de módulos interconectados entre sí. En VHDL estos módulos se definen mediante la palabra clave ENTITY.

A lo largo de las explicaciones se pondrá la estructura general de las diferentes órdenes y definiciones del lenguaje. Se utiliza la notación BNF (*Backus Naur Form*) por ser la más extendida en la descripción de lenguajes. En el anexo A se puede encontrar una explicación de esta notación, así como la descripción completa de la sintaxis del VHDL'93. Teniendo en cuenta este formato de descripción, la forma general de declaración de entidad es:

```
ENTITY nombre IS
   [GENERIC(lista de parámetros);]
   [PORT(lista de puertos);]
   [declaraciones]
[BEGIN
   sentencias]
END [ENTITY] [nombre];
```

Se observa que la declaración de entidad es algo más compleja de lo que se había visto en un principio.

Las sentencias GENERIC y PORT son las que más significado tienen a pesar de ser opcionales. La instrucción GENERIC sirve para definir y declarar propiedades o constantes del módulo que está siendo declarado en la entidad. Las constantes declaradas aquí tienen el mismo significado que las constantes declaradas como parámetros en las funciones y procedimientos que se verán más adelante. Es decir, a la entidad se le pueden pasar como parámetros las constantes definidas en GENERIC, si se pasan valores entonces la constante tomará el valor que se le pasa, y si no se le pasa ningún valor, la

constante tomará el valor por defecto que se le hubiera asignado en GENERIC, si es que se le asignó alguno. Al igual que con PORT MAP se le pasan las entradas y salidas a la entidad, con GENERIC MAP se le pasan los parámetros que definen a esa entidad.

Con la palabra clave PORT, también opcional como el resto de partes de la entidad, se definen las entradas y salidas del módulo que está siendo definido. Esta forma de declarar estas entradas y salidas ya se ha visto, y simplemente consiste en un nombre, seguido por el modo de conexión, y seguido por el tipo de datos de la línea. Se habían visto dos modos de conexiones que eran IN, para indicar entrada, y OUT para indicar salida.

La diferencia entre IN y OUT es importante: las señales de entrada se pueden leer pero **no puede asignárseles ningún valor**, es decir, no se puede cambiar su valor en el programa, y vienen a ser como constantes. Las señales de salida pueden cambiar y se les pueden asignar valores, pero **no pueden leerse**, es decir, no pueden ser usadas como argumentos en la asignación de cualquier elemento del VHDL.

Junto a los modos IN y OUT existen otros que también pueden ser usados. Uno de estos modos es el INOUT, que sirve tanto de entrada como de salida, por lo que pueden ser usados en el programa como de lectura y escritura.

Hay ocasiones en las que interesa poder leer una señal que en el circuito es de salida, como en la definición de un contador por ejemplo. La solución a esto no es la de definir esta salida como INOUT, sino que hay que definir una nueva señal, interna al circuito, que será la que finalmente se conectará a la salida, y que podemos leer y escribir libremente. Esto es, sólo se utilizará la clase INOUT cuando la señal sea realmente de entrada-salida como en el bus de datos de un microprocesador, o señales de este estilo donde además se tienen los mecanismos de contención de señales que se verán en la sección 9.1.

Otro modo de puerto es el BUFFER, que es equivalente al OUT visto con anterioridad, con la diferencia de que se puede leer. Aunque se parece al INOUT, en realidad son muy diferentes, ya que externamente no se le puede dar ningún valor. Este modo también soluciona el problema de leer una salida, aunque no es aconsejable, ya que la propia definición de *buffer* está indicando una bidireccionalidad que puede no ser apropiada. Es importante que al realizar un diseño complejo, compuesto por varias unidades, el interfase sea común a todos. Así, hay que decidir de antemano si las salidas se definirán como de modo OUT o modo BUFFER. Lo normal es definir las salidas en modo OUT y luego utilizar una señal auxiliar interna para poder leer lo que se saca por ese puerto.

El último tipo, muy poco usado y poco aconsejable, es el LINKAGE que es también como el INOUT, pero que sólo puede ser usado con elementos de tipo LINKAGE. Este modo se utiliza para enlazar con otros módulos que no tienen por qué estar escritos en VHDL, es decir, se utiliza para hacer de interfase con otros modelos de otras herramientas.

Si no se especifica el tipo de puerto se supone el tipo IN por defecto.

Por último, la parte de declaraciones es opcional, como todo lo que va dentro de la entidad, y sirve para realizar algunas declaraciones de constantes, etc. A continuación le sigue un bloque BEGIN, también opcional, donde se pueden incluir sentencias. Esta parte se puede usar en modelado, pero tiene poco sentido en síntesis. El tipo de sentencias que se pueden utilizar en esta parte son muy restringidas y se limitan a sentencias de indicación de errores o comprobación de alguna cosa.

Ejemplos de declaración de entidad:

```
ENTITY rom IS
   GENERIC(tamano, ancho: positive);
   PORT(enable : IN bit;
        address : IN bit_vector(tamano-1 DOWNTO 0);
        data: OUT bit_vector(ancho-1 DOWNTO 0));
END rom;

ENTITY procesador IS
   GENERIC(max_freq: frequency := 30 MHz);
   PORT(clk: IN bit;
        address: OUT integer;
        data: INOUT word_32;
        control: OUT proc_control;
        ready: IN bit);
END procesador;
```

4.5.2 Declaración de arquitectura

En la arquitectura es donde se define el funcionamiento del módulo definido en la entidad. Una arquitectura siempre está referida a una entidad concreta, por lo que no tiene sentido hacer declaraciones de arquitectura sin especificar la entidad. Una misma entidad puede tener diferentes arquitecturas, es en el momento de la simulación, o la síntesis, cuando se especifica qué arquitectura concreta se quiere simular o sintetizar.

La declaración de la arquitectura se realiza con la palabra clave ARCHITECTURE, y su sintaxis completa es:

```
ARCHITECTURE nombre OF nombre_entidad IS
   [declaraciones]
BEGIN
   [sentencias concurrentes]
END [ARCHITECTURE] [nombre];
```

La estructura de esta declaración es parecida a la que ya se había visto en los ejemplos.

Antes de definir la funcionalidad en el bloque BEGIN..END, hay una parte declarativa donde se definen los subprogramas (funciones, procedimientos, etc.), declaraciones de tipo, declaraciones de constantes, declaraciones de señales, declaraciones de alias, declaraciones de componentes, etc. Es importante destacar que uno de los pocos sitios donde se pueden declarar señales es dentro de la parte declarativa de una arquitectura.

A continuación, después del BEGIN, se encuentran todas las instrucciones que definen el comportamiento, estructura y funcionalidad del circuito. Hay que destacar que dentro de la arquitectura las instrucciones son de dos tipos: o concurrentes, o de referencia de componentes, que en realidad también es una construcción concurrente. En el caso de definir estructura las instrucciones serán de referencia, es decir, de colocación de componentes y las conexiones entre ellos. En el caso de querer una descripción más abstracta se pueden utilizar las asignaciones concurrentes *flujo de datos* que se ven en el próximo capítulo. Hay que reseñar que una de estas instrucciones concurrentes es el bloque PROCESS dentro del cual la ejecución es secuencial.

CAPÍTULO 5

DESCRIPCIÓN FLUJO DE DATOS

Se vio en el ejemplo 3.1, en el capítulo de introducción al VHDL, que el lenguaje VHDL no sólo servía para la descripción estructural de un circuito sino también para su descripción funcional.

En VHDL existen dos aproximaciones a la descripción comportamental de un circuito. Por un lado se pueden especificar las ecuaciones de transferencia entre diferentes objetos en VHDL. Esta posibilidad de descripción de un circuito se llama *descripción flujo de datos*, o también, refiriéndose al nivel de abstracción *descripción a nivel de transferencia entre registros*, conocido por las siglas RTL (*Register Transfer Level*). La otra forma de describir circuitos en un nivel de abstracción todavía más elevado se conoce como *descripción comportamental* o descripción algorítmica. Esta segunda posibilidad incluye a la primera y permite al diseñador de circuitos describir la funcionalidad en un nivel de abstracción alto.

La diferencia más importante entre un estilo de descripción y el otro es que la ejecución o interpretación de sentencias en flujo de datos es concurrente, es decir, las sentencias, más que mandatos u órdenes, indican conexiones o leyes que se cumplen, por tanto es como si se ejecutaran continuamente. En la descripción comportamental algorítmica es posible describir partes con instrucciones que se ejecutan en serie de la misma manera que se ejecutan las instrucciones en un lenguaje como el C o Pascal.

A nivel de síntesis siempre es más sencillo sintetizar un circuito descrito al nivel de flujo de datos que otro descrito en un nivel más abstracto. Esto es debido a que la mayoría de estructuras de la descripción de flujo de datos tienen una correspondencia casi directa con su implementación hardware correspondiente. En un nivel más abstracto, la

síntesis automática del circuito es más compleja, especialmente debido a la ejecución en serie de las instrucciones, que tienen mucho sentido en la ejecución de programas, pero cuyo significado hardware es algo menos directo.

5.1 EJECUCIÓN CONCURRENTE Y EJECUCIÓN SERIE

En lenguajes como el C, Pascal, Fortran, etc., la ejecución de las sentencias es en serie. Esto significa que las sentencias son ejecutadas una tras otra por el microprocesador. Naturalmente esta ejecución es únicamente válida para arquitecturas basadas en un único procesador con memoria principal. Éste es un caso particular de sistema, muy utilizado por otra parte, pero no se trata del caso que engloba a cualquier sistema. En general, cualquier sistema hardware está compuesto por numerosas unidades procesadoras con unidades de almacenamiento distribuidas. En estos sistemas se puede hacer una programación en serie, pero resulta poco efectiva puesto que no se explota el paralelismo que una arquitectura así puede proveer. Para especificar la funcionalidad de un sistema que permita paralelismo, es necesario utilizar un lenguaje que permita una especificación concurrente, o paralela, de sus instrucciones, siendo la ejecución serie un caso particular de la ejecución concurrente.

Un sistema digital pequeño, por ejemplo un sumador o multiplicador, se puede ver como un sistema compuesto por múltiples unidades procesadoras (puertas NAND y NOR) y por tanto se puede decir que se trata de un sistema *multiprocesador*, si bien no es lo que comúnmente se entiende como tal. En este caso, cada procesador es una unidad muy simple, una puerta lógica, pero no por ello deja de ser un sistema con múltiples unidades funcionales; por lo tanto, la descripción de tal sistema, aun siendo tan simple, debe ser concurrente. Por esto cualquier lenguaje que pretenda describir hardware debe ser como mínimo concurrente, es decir, sus sentencias no se ejecutan cuando les llega el turno dado por el orden en que se escribieron, sino que son como aseveraciones que se deben cumplir siempre, de manera que se activarán cuando sea necesaria su intervención.

La arquitectura típica monoprocesadora es un circuito hardware, por tanto, internamente tiene muchas unidades funcionales que procesan en paralelo; sin embargo, desde el punto de vista del programador del sistema, sólo existe un procesador que es el que ejecuta el programa que se le introduce. Por esto sólo existe una instrucción concurrente, que es precisamente el programa serie que se está ejecutando.

Es conocida la facilidad algorítmica de una ejecución serie, y es por esto por lo que resulta tan sencillo programar un sistema monoprocesador, debido, en parte, al nivel

de abstracción que se consigue con una descripción algorítmica serie, al encontrarse cercana a la forma en que los seres humanos abordamos los problemas. Por lo tanto, se puede decir que la ejecución serie se encuentra en un nivel alto de abstracción, o más cercano al pensamiento humano, que la concurrente. Por esto, en un alto nivel de abstracción de un circuito debe poderse incluir una descripción serie de lo que se quiere describir.

Lenguajes como el VHDL o el ADA, pensados para programar y describir sistemas (ADA), o circuitos (VHDL), deben ser en primer lugar de ejecución concurrente, y tener como caso particular de sentencia concurrente la posibilidad de ejecución serie de alguna parte de código. En el caso del VHDL, cuyo objetivo es más bien la descripción de los circuitos poco complejos de nivel medio, es bastante común que toda la descripción sea a base de sentencias concurrentes. A una descripción de este tipo, muy cercana al hardware físico, se le llama descripción de transferencia entre registros. Si además la descripción incluye sentencias que se ejecutan en serie, entonces el nivel de abstracción con el que se está describiendo el circuito es más alto, y se considera que la descripción es puramente comportamental y algorítmica. La descripción de transferencia de registros se encontraría a mitad de camino entre una descripción puramente estructural y la puramente comportamental.

5.2 DESCRIPCIÓN CONCURRENTE FLUJO DE DATOS

Este estilo de descripción se encuentra a mitad de camino entre una descripción estructural del circuito y una descripción completamente abstracta del mismo. En cierto sentido es parecido a un *netlist*, pero en este caso las interconexiones son entre objetos abstractos del lenguaje en vez de entre componentes físicos. Además, se permiten estructuras de tipo condicional que facilitan la abstracción de ciertos comportamientos.

A esta descripción se le llama de *flujo de datos* puesto que las instrucciones son todas de asignación, siendo precisamente los datos los que gobiernan el flujo de ejecución de las instrucciones.

En el ejemplo 3.1 se mostró la descripción de un multiplexor sencillo. Entonces se presentaron tres posibles descripciones, la primera, que era completamente estructural, y las otras dos comportamentales, una de ellas con ejecución serie, y la otra, de la que se dieron dos ejemplos, RTL.

De los dos ejemplos de descripción RTL había uno con una única instrucción y otro con tres instrucciones y que se encontraba más cercano al hardware. Es fácil comprobar

que la descripción flujo de datos, con una única instrucción, es más simple que la vista en el tercer ejemplo, el correspondiente a la descripción algorítmica. En este ejemplo había, además, otras sentencias y bloques como el PROCESS, cuyo significado se entenderá mejor más adelante cuando se explique el estilo de descripción completamente comportamental o algorítmico.

Se observa que la ejecución en estas descripciones RTL no es serie, es decir, no se ejecuta instrucción por instrucción, sino que la única que hay se ejecuta continuamente, esto es, como si siempre estuviera *activa*. A esta forma de ejecutarse las instrucciones se la conoce como ejecución **concurrente** y es propia de las descripciones RTL. Se muestra a continuación un ejemplo de esta ejecución concurrente.

Ejemplo 5.1 *Realizar una descripción RTL, transferencia entre registros, de un comparador de dos buses (*`a` *y* `b`*) de 11 bits. El comparador tendrá tres salidas activas a nivel alto. Una se activará si* `a=b`*, otra si* `a>b`*, y la última si* `a<b`*.*

```
ENTITY comp IS
  PORT(a,b: IN bit_vector(10 DOWNTO 0);
       amayb,aeqb,amenb: OUT bit);
END comp;

ARCHITECTURE flujo OF comp IS
BEGIN
  amayb<='1' WHEN a>b ELSE '0';
  aeqb <='1' WHEN a=b ELSE '0';
  amenb<='1' WHEN a<b ELSE '0';
END flujo;
```

En esta ocasión se tienen tres instrucciones en vez de una. Estas instrucciones no se ejecutan en serie, sino de forma concurrente. Es decir, cuando se simula este circuito se leen estas instrucciones de manera que si alguna de las señales que intervienen cambia (`a` o `b`) entonces se ejecutan las instrucciones que se vean afectadas por este cambio, en este caso todas.

5.3 ESTRUCTURAS DE LA EJECUCIÓN FLUJO DE DATOS

La instrucción básica de la ejecución concurrente es la asignación entre señales que viene gobernada por el operador `<=`. Para facilitar la tarea de realizar asignaciones algo complejas, VHDL introduce algunos elementos de alto nivel como son instrucciones condicionales, de selección, etc. En los ejemplos anteriores ya se han mostrado algunas de estas instrucciones como la instrucción condicional básica que se hacía mediante la

construcción WHEN..ELSE. A continuación se muestran otras estructuras propias de la ejecución concurrente o descripción RTL.

5.3.1 Asignación condicional: WHEN..ELSE

Ya se ha visto anteriormente y su utilización es muy sencilla. En toda expresión condicional que describa hardware de forma concurrente, es importante incluir todas las opciones posibles y contemplar todos los casos posibles de variación de una señal. De hecho, en el VHDL'87 era obligatorio siempre acabar esta expresión condicional con un ELSE, pero en la versión del 93 esta condición se ha relajado bastante tal y como muestra la sintaxis genérica de esta instrucción:

```
[id_instr:]
senal<=[opciones] {forma_de_onda WHEN condición ELSE}
       forma_de_onda [WHEN condición];
```

El id_instr es una etiqueta que sirve para identificar las instrucciones concurrentes. Su uso es opcional y se suele usar para comentar lo que hace la instrucción, para identificar por dónde va la ejecución durante la simulación, o para establecer unos atributos propios para dicha instrucción. Hay algunas instrucciones concurrentes donde esta etiqueta es obligatoria.

Las opciones que se ponen justo delante del valor a asignar tienen que ver con una expresión de vigilancia (guarda) y con el tipo de retraso que se quiera realizar sobre dicha señal. La expresión de vigilancia se comenta más adelante en la explicación de la sentencia BLOCK, entonces se verá que para que la señal responda a esta expresión de vigilancia debe hacerse la asignación mediante la opción GUARDED. Las opciones de los retrasos se verán en el capítulo 10 y son básicamente tres: TRANSPORT, INERTIAL y REJECT.

Se pueden anidar varias condiciones en una misma asignación. Ejemplo:

```
s<='1' WHEN a=b ELSE
   '0' WHEN a>b ELSE
   'X';
```

Aunque es obligatorio asignar algo, sea cual sea el resultado de la condición, en el VHDL'93 se introdujo una palabra clave para permitir el caso en que no se realiza ninguna acción. Esta palabra clave es UNAFFECTED. Una forma de solucionar esto, sin

recurrir al VHDL'93, es asignar la misma señal que está siendo asignada, así, al asignarle el mismo valor que tiene, es como si no pasara nada. A continuación se muestran dos descripciones que se pueden suponer equivalentes pero que en realidad no lo son:

```
q1<=d1 WHEN nivel='1' ELSE UNAFFECTED;          -- VHDL'93
q2<=d2 WHEN nivel='1' ELSE q2;                  -- VHDL'87
```

Aparentemente ambas líneas son equivalentes, pero en realidad hay una sutil diferencia. En el biestable `q1` no se realiza ninguna transacción, mientras que en q2, aunque todo se quede igual, sí que se produce una transacción. Hay que tener cuidado con esta diferencia, ya que, por ejemplo, existe un atributo parecido al `'event` (llamado `'transaction`) que se activa cuando se produce una transacción aunque la señal se quede igual. El resultado de este atributo es diferente para cada caso.

5.3.2 Asignación con selección: `WITH..SELECT..WHEN`

Es una ampliación del condicional y es similar a las construcciones *case* o *switch* del Pascal o C. La asignación se hace según el resultado de cierta expresión. La expresión debe devolver siempre un tipo discreto que sea enumerado o entero, o bien un tipo vector de una dimensión.

La forma general de esta instrucción es:

```
WITH expresión SELECT
     senal <= [opciones] forma_onda WHEN caso
                       {, forma_onda WHEN caso} ;
```

Aquí, al igual que en cualquier asignación a señal, las posibilidades para `opciones` son la palabra GUARDED y el mecanismo de retraso que puede incluir TRANSPORT y REJECT. Hay varias posibilidades para el `caso`, por un lado puede ser alguno de los posibles valores que puede tomar la expresión, por otro lado se puede también especificar un rango de valores construido con TO o DOWNTO; también se puede dar una lista de valores separados por el símbolo `'|'`, por último, también puede ser la palabra clave OTHERS.

Ejemplo de utilización de la orden de selección:

```
WITH estado SELECT
   semaforo<="rojo"          WHEN "01",
             "verde"         WHEN "10",
             "amarillo"      WHEN "11",
             "no funciona"   WHEN OTHERS;
```

Es obligatorio, al igual que ocurría con la asignación condicional, incluir todos los posibles valores que pueda tomar la expresión. Por lo tanto, si no se especifican todos los valores en las cláusulas **WHEN**, entonces hay que incluir la cláusula **WHEN OTHERS** que necesariamente debe venir al final. Esto es así ya que de lo contrario la expresión podría tomar valores frente a los cuales no se sabe qué respuesta dar. Si se desea no realizar ninguna asignación en determinados casos, se puede utilizar también la palabra clave **UNAFFECTED**.

5.3.3 Bloque concurrente: BLOCK

En muchas ocasiones es interesante agrupar sentencias de ejecución concurrente en bloques. Estos bloques son el mecanismo que tiene VHDL para la realización de diseños modulares, y de alguna manera tienen cierta equivalencia con las *hojas* de la captura de esquemas. Estos bloques van a permitir además subdividir un mismo programa en una jerarquía de módulos, ya que estos bloques o módulos pueden estar unos dentro de otros. Estos bloques están definidos dentro de la arquitectura en entornos de ejecución concurrente y de alguna manera son equivalentes a entidades, ya que se les pueden definir entradas y salidas, aunque quizá su uso más normal es el agrupamiento de instrucciones para separar el diseño en módulos.

La estructura general de la declaración de bloque se muestra a continuación:

```
id_block:
BLOCK [(expresión de vigilancia)] [IS]
   [cabecera]
   [declaraciones]
BEGIN
   {sentencias concurrentes}
END BLOCK [id_block];
```

El nombre `id_block` es obligatorio y se utiliza para nombrar a diferentes bloques en un mismo diseño y así ayudar en la depuración, simulación, legibilidad del programa y, sobre todo, para poder referenciar y replicar el bloque en otras partes del diseño.

En la cabecera se tiene por un lado la posibilidad de declarar entradas y salidas al bloque, y también la de declarar constantes "genéricas" que se le pasan al bloque lo mismo que se hacía en la entidad. Las entradas y salidas se declaran mediante la palabra PORT y su conexión con las señales en el exterior del bloque se realiza mediante la orden PORT MAP. Lo mismo ocurre con los parámetros que se definan mediante la palabra GENERIC; hay que pasárselos al bloque mediante un GENERIC MAP.

El hecho de poder declarar puertos de entrada y salida en un bloque, y así conectarlos con un nivel superior dentro del programa, es especialmente interesante, ya que va a permitir el uso de un bloque, que por ejemplo se tiene en otro diseño, e incorporarlo al nuevo sin necesidad de cambiar todas las señales internas del bloque. Es decir, el bloque permite descomponer el diseño en varios módulos y por tanto realizar diseños jerárquicos. De alguna manera equivale a la hoja y símbolo de los esquemas.

Por ejemplo, se supone que se tiene una memoria ROM definida como bloque con entradas `direccion` y `enable`, y salida `dato`. Estas señales se definen con PORT como en la entidad, y la conexión entre estas señales y todo lo de fuera, que se supondrá que son las señales `rom_dir`, `rom_ena` y `rom_dato`, mediante PORT MAP, es decir:

```
rom: BLOCK
 PORT(direccion: IN bit_vector(15 DOWNTO 0);
      enable: IN bit;
      dato: OUT bit_vector(7 DOWNTO 0));
 PORT MAP(direccion=>rom_dir,enable=>rom_ena,dato=>rom_dato);
BEGIN
...
END BLOCK rom;
```

Antes del BEGIN del bloque queda todavía la parte de declaración. En esta parte se puede incluir casi cualquier cosa como subprogramas, señales, tipos, atributos, etc. Lo que resulta importante es que la visibilidad de todo lo que se declara aquí es local al bloque, lo que permite la modularidad, reutilización y portabilidad del código.

La expresión de vigilancia es opcional, y permite la habilitación o deshabilitación de la asignación de determinadas señales dentro del bloque, en concreto aquellas que empleen la palabra clave GUARDED (vigilado) en su asignación. La expresión de vigilancia es de tipo booleano, de manera que si es cierta la condición se realizan todas las asignaciones, y si es falsa se realizarán todas menos las vigiladas. Un ejemplo sencillo, aunque no es lo habitual, es definir un registro activo por nivel alto de la señal de habilitación, es decir:

```
latch: BLOCK(enable='1')
BEGIN
  q<=GUARDED d;
END BLOCK latch;
```

Sólo cuando `enable` sea uno, la entrada pasará a la salida, en caso contrario la salida no cambia por mucho que cambie la entrada, por tanto, se trata efectivamente de un cerrojo activo por nivel alto, aunque como se verá a continuación hay que tener cuidado con la clase de señal que se tiene; si por ejemplo fuera de clase BUS, el ejemplo anterior no funcionaría bien.

Lo que se acaba de comentar sirve para las señales "normales", es decir, aquellas que no son ni BUS ni REGISTER. Para el resto de señales la expresión de vigilancia (guardia) no dice qué asignaciones se van a realizar y cuáles no; lo que hace en realidad es *desconectar* una señal de su fuente. El símil físico es que la señal es un cable que se ha dejado al aire y que por tanto podría tomar cualquier valor. Para que una señal se pueda desconectar y quedar "al aire" es necesario que sea de un tipo resuelto, es decir, una señal que permita que ninguna, una, o varias fuentes escriban a un tiempo sobre ella. La explicación de los tipos resueltos se ofrece con más detalle en 9.1.

Una vez se tiene una señal de tipo resuelto, existen varias posibilidades para el caso en que ninguna fuente escriba sobre ella, es decir, para el caso en que se desconecta. Si se declaró como REGISTER, entonces el valor que tendrá al desconectarse será el que ya tenía en el momento de la desconexión. Si se declaró como BUS, entonces tendrá que ser la función de resolución la que decida qué valor se le tiene que dar.

Con todo esto, y volviendo al ejemplo del *latch* anterior, para que funcionase bien habría que declarar la señal `q` de forma normal, o como REGISTER, debiendo ser en este caso de un tipo resuelto.

Por último, encerrados entre el BEGIN..END del bloque se encuentran las instrucciones concurrentes a ejecutar, por lo que se pueden incluir otros bloques dentro de un bloque, dando como resultado un diseño jerárquico. A continuación se da un ejemplo de descripción de flujo de datos típica.

5.4 EJEMPLOS DE DESCRIPCIÓN FLUJO DE DATOS

Ejemplo 5.2 *Realizar la descripción RTL de un circuito que desplace 1 bit, a derecha o izquierda, un bus de entrada de 4 bits. El circuito está controlado por una señal de dos bits de manera que cuando esta señal es "00" no desplaza. Si es "01" el despla-*

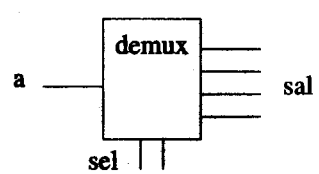
zamiento es hacia la izquierda. Si es "10" el desplazamiento es hacia la derecha, y si es "11" se produce una rotación a derechas. En el caso de desplazamientos se introduce un cero en el hueco que quede libre. Realizar una descripción con la estructura `WHEN..ELSE` *y otra con la* `WITH..SELECT`.

```
ENTITY shifter IS
PORT( shftin:  IN  bit_vector(0 TO 3);
      shftout: OUT bit_vector(0 TO 3);
      shftctl: IN  bit_vector(0 TO 1));
END shifter;

ARCHITECTURE flujo1 OF shifter IS
BEGIN
 shftout<=shftin                    WHEN shftctl="00" ELSE
          shftin(1 TO 3)&'0'        WHEN shftctl="01" ELSE
          '0'&shftin(0 TO 2)        WHEN shftctl="10" ELSE
          shftin(3)&shftin(0 TO 2);
END flujo1;

ARCHITECTURE flujo2 OF shifter IS
BEGIN
  WITH shftctl SELECT
   shftout<=shftin                     WHEN "00",
            shftin(1 TO 3)&'0'         WHEN "01",
            '0'&shftin(0 TO 2)         WHEN "10",
            shftin(3)&shftin(0 TO 2) WHEN "11";
END flujo2;
```

Ejemplo 5.3 *Realizar la descripción flujo de datos de un demultiplexor. El demultiplexor que se propone tiene dos entradas de selección, y otra entrada que se conectará a una de las cuatro salidas indicadas por los dos bits de selección.*

La entidad y el símbolo son los siguientes:

```
ENTITY demux IS
PORT(a: IN bit;
     sel: IN bit_vector(2 DOWNTO 0);
     sal: OUT bit_vector(0 TO 7));
END demux;
```



La estructura que debe tener la arquitectura de este simple circuito es la de una selección. Una primera forma en la que se puede describir es la siguiente:

```
ARCHITECTURE flujo OF demux IS
  SIGNAL aux: bit_vector(0 TO 3);
BEGIN
  WITH sel SELECT
  aux<="0001" WHEN "00",
       "0010" WHEN "01",
       "0100" WHEN "10",
       "1000" WHEN "11";
  sal<=aux AND a&a&a&a;
END flujo;
```

No es imprescindible el uso de una señal auxiliar para describir este circuito, de hecho se puede utilizar también el operador & para ello:

```
ARCHITECTURE otro_flujo OF demux IS
BEGIN
  WITH sel SELECT
  sal<="000"&a WHEN "00",
       "00"&a&"0" WHEN "01",
       "0"&a&"00" WHEN "10",
       a&"000" WHEN "11";
END otro_flujo;
```

CAPÍTULO 6

DESCRIPCIÓN COMPORTAMENTAL ALGORÍTMICA

La descripción RTL es bastante cercana al hardware real. Las estructuras condicionales vistas para esta ejecución concurrente le dan al programador un cierto nivel de abstracción, bastante más elevado que el que se puede alcanzar con un *Netlist* o una descripción estructural pura de un circuito. Sin embargo, para sistemas mucho más complejos que los ejemplos que se están mostrando aquí, se hace necesario un grado de abstracción todavía más elevado.

Los lenguajes de programación software son un ejemplo claro de lenguajes de descripción algorítmica de alto nivel, si bien el objetivo es bastante diferente. La diferencia fundamental entre el lenguaje VHDL que se ha visto hasta ahora, y estos lenguajes de software, es el modo de ejecución. Mientras el VHDL, hasta ahora, se ejecutaba de forma concurrente, los programas software se ejecutan en serie, lo que permite la utilización de estructuras como bucles que no son posibles, de forma directa, en una ejecución concurrente. En realidad, no todos los lenguajes de programación son en serie, un ejemplo de lenguaje de programación cuya ejecución es concurrente es el Prolog. En este lenguaje se declaran *reglas* que se ejecutan en paralelo al ser requeridos por otras reglas, etc. La ejecución concurrente en VHDL es declarativa al igual que en Prolog, es decir, lo que se hace es declarar *reglas* o *leyes* que en el caso del VHDL son reglas eléctricas que se traducen casi directamente en conexiones; cuando cambian algunos de los argumentos de la asignación, entonces se ejecuta la instrucción.

Como la programación concurrente no es siempre la más cómoda para la descripción de ideas, VHDL también permite una programación en serie o algorítmica. En VHDL esta programación serie se define dentro de bloques indicados con la palabra clave PROCESS. Por tanto, siempre que en VHDL se precise de una descripción en serie, se deberá utilizar un bloque de tipo PROCESS.

En un mismo programa puede haber múltiples bloques PROCESS. En el caso de que ocurra, cada uno de ellos equivale a una instrucción concurrente. Es decir, internamente a los PROCESS la ejecución de las instrucciones es serie, pero entre los propios bloques PROCESS, que pueden convivir con otras instrucciones concurrentes, la ejecución es concurrente.

Se plantea una cuestión de forma inmediata, y es que si un bloque PROCESS es en realidad una instrucción concurrente ¿cómo se activa esta instrucción?; es decir, ¿cuándo se ejecuta? En la ejecución concurrente se había visto que una instrucción se activaba o ejecutaba cuando alguno de los argumentos que intervenían en la asignación cambiaba. En el caso de un bloque PROCESS esto es mucho más complejo, ya que dentro de un bloque de este tipo puede haber muchas instrucciones, asignaciones, condiciones, bucles, y no parece que tenga que existir un convenio para la ejecución. Para poder indicarle al programa cuándo activar o ejecutar un bloque PROCESS existen dos procedimientos. Lo normal es utilizar una *lista sensible*, es decir, una lista de señales, de manera que cuando se produzca un cambio en alguna de las señales, entonces se ejecute el PROCESS. La otra opción consiste en utilizar una sentencia WAIT en algún lugar dentro del bloque PROCESS. Más adelante se explicarán con mayor detalle estos mecanismos.

Ejemplo 6.1 *Realizar la descripción comportamental serie del comparador del ejemplo 5.1.*

```
ARCHITECTURE abstracta1 OF comp IS
BEGIN
  PROCESS(a,b)   -- Se ejecuta cuando a o b cambian
  BEGIN
     IF a>b THEN
       amayb<='1';
       aeqb <='0';
       amenb<='0';
     ELSIF a<b THEN
       amayb<='0';
       aeqb <='0';
       amenb<='1';
     ELSE
       amayb<='0';
       aeqb <='1';
       amenb<='0';
     END IF;
  END PROCESS;
END abstracta1;
```

Esta descripción del ejemplo es correcta, pero no hay que olvidar que dentro de un PROCESS la ejecución es serie, por tanto este mismo ejemplo se puede describir de forma algo más sencilla como:

```
ARCHITECTURE abstracta2 OF comp IS
BEGIN
  PROCESS(a,b)  -- se ejecuta cuando a o b cambian
  BEGIN
     amayb<='0';
     aeqb <='0';
     amenb<='0';
     IF a>b THEN
       amayb<='1';
     ELSIF a<b THEN
       amenb<='1';
     ELSE
       aeqb <='1';
     END IF;
  END PROCESS;
END abstracta2;
```

Aunque sintácticamente es correcto, y una simulación de este ejemplo daría los resultados que se esperan, lo más probable es que una herramienta de síntesis no permitiera usar una descripción así. La cuestión es que los algoritmos de síntesis tienen problemas cuando a una señal se le asignan varios valores en un mismo proceso, aunque sea uno detrás de otro.

6.1 DIFERENCIAS ENTRE VARIABLE Y SEÑAL

Una de las cosas que más chocan al programador experimentado con otros lenguajes, y que se enfrenta por primera vez a algún lenguaje de descripción hardware, y más concretamente al VHDL, es la diferencia entre señal y variable.

Hasta ahora, en la ejecución concurrente las variables no existían y sólo se disponía de las señales. En la ejecución concurrente no hay mucha diferencia entre lo que es una señal y lo que es una variable normal en cualquier otro lenguaje. Aparentemente, cuando en la ejecución concurrente una señal recibe un valor, la señal toma inmediatamente este valor. Cuando se explique más adelante el VHDL para simulación, se verá que éste *inmediatamente* es en realidad un *paso de simulación* que si no se especifica ningún retraso, implicará que la señal toma ese valor en el momento de la asignación.

En realidad, lo que ocurre en la asignación de una señal es que en el momento actual, o en el *paso de simulación actual*, se indica que se quiere que en el próximo paso de simulación (o después de un tiempo si se especifica un retraso), la señal adquiera el valor que se le está asignando en este momento. En una asignación concurrente esto equivale a que la asignación se realice de forma instantánea, debido a que cuando se acaba de ejecutar una instrucción concurrente se pasa inmediatamente al siguiente paso de simulación.

Un bloque de tipo PROCESS es equivalente a una única instrucción concurrente formada por numerosas instrucciones en serie. Como se trata de una única instrucción concurrente, todas las instrucciones serie internas ocurren en el mismo paso de simulación, y no se pasa al siguiente paso de simulación hasta que se haya completado la ejecución del PROCESS. Esto quiere decir que *dentro de un bloque* PROCESS, *las señales conservan su valor y no cambian hasta que el bloque termina de ejecutarse*, momento en el cual toman el valor que se les haya asignado, durante la ejecución del proceso, debido a que se encuentra en el siguiente paso de simulación.

Para verlo un poco más claro se puede decir que una señal es como una caja con dos secciones. Una, que es la sección que se suele ver, y que es la que contiene el valor actual, y otra, separada, que contiene el valor o valores futuros. Cuando se *lee* una señal se está accediendo a la sección donde se guarda el valor actual. Cuando se le asigna algo a una señal, se está *escribiendo* sobre la sección dedicada al valor futuro. Sólo cuando se acaba la ejecución de la instrucción concurrente, lo que se encuentra en la sección de valor futuro pasa a la sección de valor actual.

De ahora en adelante, a la sección del valor futuro, donde se escriben siempre los valores, se le llamará *fuente* de la señal, y a la sección de valor actual se le llamará simplemente *señal* o valor de la señal. En realidad, y se verá mejor en el capítulo de simulación, hay un único *valor actual*; sin embargo, pueden haber varios *valores futuros* ya que se puede hacer una lista de los valores que tomará la señal en el futuro. Esto, que ahora no tiene mucho sentido puesto que las señales toman su valor inmediatamente en la ejecución concurrente, será muy útil cuando se expliquen los retrasos en la simulación.

Las variables son diferentes en comportamiento, ya que tienen exactamente el mismo significado que las variables en cualquier otro lenguaje. Para empezar, sólo pueden ser usadas en entornos serie, por lo que únicamente se las puede definir dentro de procesos o en subprogramas. Para continuar, y ésta es la diferencia más importante, *las variables toman inmediatamente su valor en el momento de la asignación*. Es decir, son iguales que las variables de cualquier otro lenguaje.

Estas diferencias entre variable y señal se ven mejor en la figura 6.1, donde se muestran las dos partes de código que a continuación se comentan.

En el código de la izquierda sólo se utilizan señales y la ejecución tiene lugar de la siguiente manera: En primer lugar se hace la fuente de `c` igual a `a`, lo cual sólo está indicando que *tomará ese valor en el próximo paso de simulación, pero no en el presente*. A continuación se hace lo mismo con `x` asignándosele a su fuente el valor

```
-- Uso incorrecto de las señales
ARCHITECTURE ejemp1 OF entidad IS
SIGNAL a,b,c,x,y: integer;
BEGIN
 p1: PROCESS(a,b,c)
 BEGIN
    c<=a;  -- Se ignora
    x<=c+2;
    c<=b;  -- Se mantiene
    y<=c+2;
 END PROCESS p1;
END ejemp1;
```

```
-- Uso correcto de las variables
ARCHITECTURE ejemp1 OF entidad IS
SIGNAL a,b,x,y: integer;
BEGIN
 p1: PROCESS(a,b)
 VARIABLE c: integer;
 BEGIN
    c:=a;  -- Inmediato
    x<=c+2;
    c:=b;  -- Inmediato
    y<=c+2;
 END PROCESS p1;
END ejemp1;
```

Figura 6.1: Ejemplo donde se muestra la diferencia entre variable y señal

de la expresión `c+2`, es decir, el valor que contuviera `c` antes de empezar la ejecución, porque el valor que se le asignó en el paso anterior todavía no está presente. Luego se hace `c<=b`, es decir, que se está sustituyendo el valor de la fuente de `c`, que era `a`, por la señal `b`. Esto quiere decir que el valor futuro de `c` ya no será `a` como estaba previsto sino `b`. A continuación se hace `y<=c+2` de manera que a `y` se le asigna el valor `c+2`, pero cogiendo como `c` el valor de la señal antes de empezar el proceso.

En definitiva, supongamos que antes de iniciarse la ejecución `c=2`, `a=4` y `b=6` por ejemplo. Entonces, al final de la ejecución de este PROCESS, se tiene que `c=b=6`, `x=4` y `y=4`. De todas formas la ejecución no ha terminado todavía puesto que `c` ha cambiado y, como se encuentra en la lista sensible, se volverá a ejecutar el bloque. Esto quiere decir que se repiten de nuevo las operaciones, es decir, primero se hace la fuente de `c` igual a `a`, luego se hace `x<=c+2`, con lo que `x=8`, luego se pone `b` en `c`, y por último, se hace `y<=c+2` con lo que `y=8`. Se observa que `c` conserva su valor puesto que sigue siendo `c=b=6`, por lo tanto la ejecución se detiene.

Se ha visto que la primera instrucción (`c<=a`) ha sido ignorada completamente en toda la ejecución. Esto es debido a que la tercera instrucción (`c<=b`), al venir después, sobreescribe la fuente de `c`, por lo que en realidad `c` nunca puede tomar el valor `a`. En el ejemplo de la derecha se puede ver cómo hacer para que `c` tome valores instantáneos y que la ejecución sea como en un principio se espera.

En el programa de la derecha del ejemplo anterior primero se define `c` como una variable en vez de como señal. Esta definición se hace dentro del PROCESS puesto que las variables sólo tienen sentido dentro de la ejecución serie. También `c` desaparece de la lista sensible puesto que las variables son internas a los PROCESS y nunca pueden formar parte de las listas sensibles.

La ejecución es muy simple. Primero `c` toma el valor de `a`, y como es una variable toma este valor de forma inmediata, es decir, `c` toma el valor `4` justo en este momento. A continuación se hace la fuente de `x` igual a `c+2`, como `c` vale `4`, entonces `x` tomará el valor `6` en el próximo paso de ejecución. A continuación se hace `c:=b` de manera que ahora `c` vale `8`. Después viene `y<=c+2`, por lo que `y` valdrá, cuando se acabe la ejecución, `8`. Al finalizar el **PROCESS** se tiene que `x=6` y `y=8`, y no se volverá a ejecutar puesto que ni `a` ni `b` han cambiado.

6.2 ESTRUCTURAS DE LA EJECUCIÓN SERIE

A continuación se verán las estructuras más comunes de la ejecución serie así como algunas características típicas de este tipo de ejecución que es, sin duda, el más utilizado en VHDL por permitir un alto grado de abstracción y encontrarse más cerca del lenguaje natural.

6.2.1 El bloque de ejecución serie: PROCESS

La instrucción **PROCESS** es concurrente ya que el único sitio donde se puede emplear es en entornos concurrentes. Sin embargo, se ha incluido en este capítulo dedicado a la descripción algorítmica ya que es la puerta de entrada a la ejecución serie; es decir, la manera de especificar instrucciones serie dentro de un programa en VHDL es mediante la definición de un bloque **PROCESS**. Su estructura y funcionamiento se han visto ya pero se comentarán a continuación con un poco más de detalle. La forma en que se declara un bloque **PROCESS** es la siguiente:

```
[id_proc:]
[POSTPONED] PROCESS [(lista_sensible)] [IS]
   declaraciones
BEGIN
   instrucciones_serie
END [POSTPONED] PROCESS [id_proc];
```

El `id_proc` es simplemente una etiqueta opcional que sirve para ponerle nombre a los diferentes procesos de una descripción. La `(lista_sensible)` es también opcional y contiene una lista de señales separadas por comas y encerradas entre paréntesis. La ejecución del **PROCESS** se activa cuando se produce un *evento*, o cambio, en alguna de las señales de la lista sensible. En el caso de no existir lista sensible, la ejecución se controla mediante el uso de sentencias **WAIT** dentro del **PROCESS**. Esta sentencia **WAIT** se verá más adelante. En cualquier caso debe existir, o bien una lista sensible, o una o

más sentencias WAIT; de lo contrario se ejecutaría el proceso una y otra vez entrando la simulación en un bucle infinito del que no se sale.

La parte de declaración es parecida a la de otras estructuras, de forma que se pueden definir aquí variables, tipos, subprogramas, atributos, etc., pero en ningún caso señales. En interesante destacar que éste es el único lugar, aparte de en los subprogramas, donde se pueden definir las variables, cosa que no se puede hacer en otras estructuras como entidades, bloques, arquitecturas o paquetes. A continuación, y entre el BEGIN..END, vienen todas las instrucciones serie que, como se verá, presentan sus propios elementos sintácticos, siendo la asignación simple el único elemento común con la ejecución concurrente.

Por último, existe la posibilidad de *posponer* el proceso. Un proceso POSTPONED, significa que debe posponerse su ejecución al final del paso de simulación actual. Esto se entiende mejor después de entender la simulación en VHDL. Como resumen se puede decir que, al cambiar una señal, se activan varios procesos al mismo tiempo; la ejecución de estos procesos no puede ser paralela ya que normalmente el simulador se ejecuta sobre un único procesador. No existe forma de decir qué proceso se tiene que ejecutar primero y cuál después. El único mecanismo que hay para controlar esto es declarar el proceso como pospuesto, que significa que la ejecución de ese proceso será posterior a la ejecución del resto de procesos no pospuestos.

6.2.2 Sentencia de espera: WAIT

La ejecución de un bloque PROCESS, si no se indica nada más, se realiza de forma continuada como si de un bucle se tratara. Es decir, se ejecutan todas las instrucciones y se vuelve a empezar. Esto resulta peligroso en simulación, puesto que nunca se podría salir de un proceso y la simulación no acabaría nunca. Por lo tanto, debe existir un mecanismo que permita detener la ejecución del bloque serie. Una de las formas de detener la ejecución es mediante la inclusión de la *lista sensible*. La inclusión de esta lista equivale a la adición de una sentencia de espera al final del proceso, para que detenga la ejecución del PROCESS hasta que cambie alguna de las señales de la lista sensible.

Aunque la utilización de la lista sensible es suficiente para la mayoría de procesos en un programa, existe una posibilidad más compleja de detener la ejecución del bloque serie resultando la lista sensible un caso particular de esta operación. La forma genérica de detener la ejecución en un proceso se realiza mediante la palabra clave WAIT que detiene la ejecución hasta que se cumple una condición o evento especificado en la propia sentencia. La sintaxis es como sigue:

```
[id_wait:]
WAIT [ON lista_sensible] [UNTIL condición] [FOR tiempo_límite];
```

En su forma más sencilla se puede utilizar WAIT a solas. En este caso, cuando la ejecución llega a esta instrucción, se detiene para siempre. Esto es útil en simulación ya que muchas veces se necesitan realizar unas acciones iniciales, detenerse, y continuar con el resto de la simulación.

Los tres elementos que se ven en la instrucción de espera indican las condiciones bajo las cuales se reanudará la ejecución del bloque.

La `lista_sensible` es simplemente una lista de señales separadas por comas. La `condición` es una condición que cuando se cumple permite que siga la ejecución. El `tiempo_límite` indica un tiempo durante el cual la ejecución está detenida, cuando ese tiempo se acaba sigue la ejecución. Estas tres posibilidades, es decir, la de la lista sensible, la condición y el tiempo límite, son opcionales y pueden especificarse una, dos o las tres en una misma sentencia WAIT. Cualquiera que ocurra primero permitirá que se continúe con la ejecución del proceso.

Ahora se aprecia claramente que la lista sensible del bloque PROCESS es completamente equivalente a poner la sentencia `WAIT ON lista_sensible;` al final del bloque PROCESS.

Ejemplos de uso de la sentencia WAIT:

```
WAIT ON pulso;
WAIT UNTIL contador>7;
WAIT FOR 1 ns;
WAIT ON interrupcion FOR 25 ns;
WAIT ON clk,sensor UNTIL contador=3 FOR 100 ns;
```

Hay varios elementos a tener en cuenta con el uso de la sentencia WAIT. Un proceso debe tener una lista sensible o al menos una sentencia WAIT, de lo contrario, éste se ejecuta indefinidamente una y otra vez. Si el proceso ya tiene una lista sensible, entonces no se puede utilizar la sentencia WAIT en su interior, ni siquiera en algún subprograma que se pudiera llamar desde ese proceso. En un mismo proceso pueden haber varias sentencias WAIT con varias condiciones diferentes. A continuación se muestra un ejemplo donde ambos procesos son equivalentes:

```
-- Con lista sensible
p1: PROCESS(b,a)
BEGIN
  a<=b+a+2;
END PROCESS p1;
```

```
-- Con WAIT
p2: PROCESS
BEGIN
  a<=b+a+2;
  WAIT ON a,b;
END PROCESS p1;
```

Un ejemplo algo más elaborado se puede realizar intentando la descripción de alguna función lógica mediante sentencias `WAIT` como el ejemplo que sigue:

Ejemplo 6.2 *Realizar un proceso que describa el comportamiento de una puerta OR utilizando sentencias de espera* `WAIT`.

```
-- Puerta OR complicada
PROCESS
BEGIN
   s<='0';
   WAIT UNTIL (a='1' OR b='1');
   s<='1';
   WAIT UNTIL (a='0' AND b='0');
END PROCESS;
```

Éste es un claro ejemplo de las descripciones que se pueden realizar en VHDL pero que es mejor no hacer. En el caso de la simulación, este proceso ralentiza la ejecución ya que debe monitorizar `a` y `b` y comprobar sus valores; con una lista sensible sería más rápido. En cuanto a la síntesis es todavía peor. Probablemente una herramienta de síntesis sería incapaz de entender esa descripción como un circuito digital, y si fuera capaz de entenderla, quizá no descubriría que en realidad se trata de una simple puerta lógica.

6.2.3 Sentencia condicional: `IF..THEN..ELSE`

Es la estructura típica para realizar una acción u otra según el resultado de una expresión booleana, siendo equivalente en significado a estructuras del mismo tipo en otros lenguajes. La forma general es:

```
[id_if:]
IF condición THEN
    sentencias
{ELSIF condición THEN
    sentencias}
[ELSE
    sentencias]
END IF [id_if];
```

Es importante resaltar que esta estructura tiene la posibilidad de anidar IFs consecutivos mediante la palabra ELSIF, de esta manera se evita tener que poner al final de cada nuevo IF un END IF, ganando el programa en legibilidad.

Tanto el ELSE como el ELSIF son opcionales, pero conviene asegurarse de si hay que poner ELSE al final o no, y sobre todo, saber por qué. El no hacerlo así va a tener impacto sobre el tipo de hardware que se está describiendo; si por ejemplo no se pone ELSE, el hardware correspondiente será secuencial y necesitará registros. En programación software esto puede tener poca importancia, pero pensando en la síntesis posterior del circuito, es importante tener esto en cuenta.

Esta sentencia, cuando sólo se realiza una asignación, tiene su equivalente en la ejecución concurrente. De hecho, cualquier instrucción concurrente se puede poner de forma serie mediante un proceso. En la figura 6.2.3 se muestran dos partes de programa que son equivalentes entre sí.

```
-- Ejecución serie
PROCESS(a,b,c)
BEGIN
   IF a>b THEN
      p<=2;
   ELSIF a>c THEN
      p<=3;
   ELSIF (a=c AND c=b) THEN
      p<=4;
   ELSE p<=5;
   END IF;
END PROCESS;
```

```
-- Ejecución concurrente
p<=2.WHEN a>b ELSE
   3 WHEN a>c ELSE
   4 WHEN (a=c AND c=b) ELSE
   5;
```

Figura 6.2: Ejemplo de equivalencia entre serie y concurrente

Para estos casos particulares resulta más simple la ejecución concurrente. La regla general para que ambas estructuras sean equivalentes es que el PROCESS contenga, en su lista sensible, todas las señales que intervienen como argumentos en las asignaciones de la ejecución concurrente.

6.2.4 Sentencia de selección: CASE

Es la estructura típica que permite ejecutar unas sentencias u otras dependiendo del resultado de una expresión. Su forma sintáctica es la siguiente:

```
[id_case:]
CASE expresion IS
   WHEN caso => sentencias;
   {WHEN caso => sentencias;}
   [WHEN OTHERS => sentencias;]
END CASE [id_case];
```

Esta instrucción es equivalente a la de WITH..SELECT vista en el entorno de ejecución concurrente. Lo mismo que se había comentado para aquella instrucción sirve ahora para ésta, con la salvedad de que en este caso la selección no está limitada a una asignación.

La expresión de selección tiene que ser o bien de tipo discreto o una matriz monodimensional de caracteres. Dependiendo de la expresión se ejecutarán unas sentencias u otras. No pueden haber dos casos duplicados porque daría error, y todos los posibles casos de valores de la expresión deben estar contemplados en los diferentes WHEN. Es por esto conveniente el uso de la palabra OTHERS, para indicar que se ejecuten ese conjunto de instrucciones, si la expresión toma un valor que no se contempla en ninguno de los casos anteriores.

Los casos se pueden especificar o bien con un valor simple, o bien con un rango de valores mediante las palabras TO o DOWNTO, o una lista de valores separados por el símbolo "|". La otra posibilidad, explicada anteriormente, es utilizar OTHERS. Se muestra a continuación un ejemplo simple de esta estructura:

```
CASE puntuacion IS
   WHEN 9    TO  10 => acta<="Sobresaliente";
   WHEN 8 DOWNTO 7  => acta<="Notable";
   WHEN 5    |   6  => acta<="Aprobado";
   WHEN 0           => acta<="No presentado";
   WHEN OTHERS      => acta<="Suspenso";
END CASE;
```

Naturalmente en este ejemplo se supone que la puntuación es un entero, ya que el tipo de la expresión a evaluar siempre tiene que ser un tipo discreto o una matriz monodimensional.

6.2.5 Bucles: FOR y WHILE LOOPS

En VHDL existen las dos posibilidades típicas para bucles, es decir, los de tipo WHILE y FOR. La parte repetitiva del bucle siempre viene especificada por la palabra

clave LOOP..END LOOP, y será la parte anterior del bucle lo que indique si es de tipo *for* o *while*. La sintaxis general es la siguiente:

```
[id_bucle:]
[esquema_iterativo] LOOP
     sentencias;
END LOOP [id_bucle];
```

El esquema_iterativo indica si es un WHILE o FOR. Es opcional, por lo que se puede definir un bucle sin más, de manera que las sentencias en ese bucle se repiten para siempre.

Las construcciones de los bucles FOR y WHILE son entonces:

```
[id_for:]
FOR identificador IN rango
LOOP
   sentencias;
END LOOP [id_for];
```

```
[id_while:]
WHILE condición
LOOP
   sentencias;
END LOOP [id_while];
```

En el caso de la sentencia FOR, el bucle se repite para cada valor que toma el identificador en el rango especificado. Esto quiere decir que el tipo del identificador debe ser discreto, es decir, o un entero o un enumerado.

El bucle en la instrucción WHILE se repite mientras la condición sea cierta, si no lo es deja de repetirse.

A continuación se muestra un ejemplo donde se ven dos bucles, uno FOR y el otro WHILE, que realizan exactamente la misma operación:

```
-- Bucle FOR
FOR cuenta IN 5 DOWNTO 0 LOOP
   tabla(cuenta)<=cuenta*2;
END LOOP;
```

```
-- Bucle WHILE
cuenta:=5;
WHILE cuenta>=0 LOOP
   tabla(cuenta)<=cuenta*2;
   cuenta:=cuenta-1;
END LOOP;
```

Interrupción en bucles: NEXT y EXIT

Junto a las instrucciones de bucles, se dan dos más que permiten interrumpir el flujo normal de ejecución. Estas sentencias son NEXT y EXIT. La primera permite detener la ejecución de la iteración actual y pasar a la siguiente. La segunda detiene la ejecución

en ese instante y sale del bucle. En el caso de tener varios bucles anidados, la salida se realiza del bucle donde se encuentre la instrucción o bien del bucle indicado por la etiqueta después de la instrucción, si es que se especifica. Estas instrucciones aceptan opcionalmente una condición, de manera que si se cumple la condición se interrumpe el lazo, y si no, no.

La sintaxis general de estas dos instrucciones es la siguiente:

```
[id_next:]                                  | [id_exit:]
NEXT [id_bucle] [WHEN condición];           | EXIT [id_bucle] [WHEN condición];
```

En el siguiente ejemplo se muestran dos lazos anidados y el uso de una de las operaciones de interrupción de lazo:

```
fuera:
WHILE a<10 LOOP
  -- varias sentencias
  dentro:
  FOR i IN 0 TO 10 LOOP
     -- varias sentencias
     NEXT fuera WHEN i=a;   -- Interrumpe el FOR y sigue en el WHILE
  END LOOP dentro;
END LOOP fuera;
```

6.3 EJEMPLOS DE EJECUCIÓN SERIE

Como ejemplos de descripciones serie resulta interesante dar la descripción de elementos secuenciales sencillos como son los registros y todas sus variedades. En el capítulo 12 ya se verán descripciones más complejas que incluyen máquinas de estados.

6.3.1 Descripción de cerrojos o *latches*

Un *latch* es uno de los elementos de memoria más sencillos y, por tanto, constituye uno de los componentes primarios con los que se construyen los sistemas secuenciales. Se caracteriza por tener una señal que cuando está inactiva guarda el estado del biestable, mientras que cuando está activa su salida toma y sigue el de la entrada. Existen muchos tipos de cerrojos, pero el más utilizado, y que comúnmente es lo que se conoce como *latch*, es el de tipo D. A estos biestables también se les conoce como biestables activos por nivel, puesto que dependiendo del nivel de la señal se comporta de una manera u otra.

La descripción de un *latch* con una entrada `d`, una salida `q`, y una señal de habilitación `ena` sería:

```
PROCESS(ena,d)
BEGIN
  IF ena='1' THEN q<=d;
  END IF;
END PROCESS;
```

Si estas señales en vez de ser de un bit fueran un vector, se tendrían tantos cerrojos de un bit puestos en paralelo como la dimensión del vector.

6.3.2 Descripción de registros

Si se cogen dos cerrojos y se colocan en serie con las señales de habilitación invertidas uno respecto del otro, entonces se tiene un biestable *maestro-esclavo*. Esto significa que el biestable no es sensible al nivel de la señal de control, sino que es sensible al flanco de esta señal; será sensible al flanco de subida o bajada dependiendo de cómo estén invertidas las señales de habilitación.

En VHDL no hay que poner dos cerrojos en serie para definir un registro; como se está utilizando una descripción comportamental, lo que hay que hacer es definir su función, es decir, hay que decir que la salida capture la entrada cuando se produzca uno de los flancos de reloj, y que no haga nada en caso contrario. La forma en que se define esto en VHDL es la siguiente:

```
PROCESS(clk)
BEGIN
  IF clk='1' THEN q<=d;
  END IF;
END PROCESS;
```

La única diferencia con el cerrojo, activo por nivel, es que no se ha puesto `d` en la lista sensible. Esto significa que el PROCESS sólo se ejecutará cuando cambie el reloj, y, además, sólo se realizará la asignación cuando este reloj sea uno, es decir, sólo se realiza la asignación cuando cambie la señal de reloj y sea uno, lo que sólo se da en el caso de flanco de subida del reloj.

La afirmación que se acaba de hacer, y la descripción del registro que se acaba de dar, son solamente ciertas suponiendo que el circuito se va a sintetizar. Si lo que se quiere hacer es un modelo hay que tener en cuenta los diferentes valores que puede

tomar el reloj. En efecto, si en vez del tipo `bit` se utiliza otro con más niveles, como el `std_logic`, resulta que se activará el registro cuando el reloj cambie y pase a `'1'`, lo cual no quiere decir que se ha producido un flanco de subida del reloj. Por lo tanto, y sólo para el caso del modelado, una definición más correcta del registro sería la siguiente:

```
PROCESS(clk)
BEGIN
  IF clk='1' AND clk'last_value='0' THEN q<=d;
  END IF;
END PROCESS;
```

En esta descripción el atributo `'last_value` da el último valor que tenía la señal antes de cambiar. De esta manera cualquier otro tipo de transición (como de `'Z'` a `'1'`, etc.) no es tenido en cuenta.

Registro con señal de réset

Es muy normal incluir señales de *réset* (puesta a cero) o *set* (puesta a uno) en un registro, de manera que se pueda poner a cero o uno en cualquier momento de forma asíncrona. La manera normal de hacerlo es poniendo el réset en la lista sensible y luego un `IF` que ponga a cero la salida si el réset está activo. Después de la comprobación de la señal de réset vendría la condición de flanco de subida:

```
PROCESS(clk,reset)
BEGIN
  IF reset<='1' THEN q<='0';
  ELSIF clk='1' AND clk'event THEN q<=d;
  END IF;
END PROCESS;
```

Si no se indica la condición `clk'event` entonces casi se comporta como un registro salvo en un sólo caso particular, y es cuando la señal de `reset` pasa de `'1'` a `'0'`. En este caso, si el reloj es uno, la entrada pasaría a la salida, y esto en un registro real no ocurre. Lo que ocurre en un registro real es que la entrada no pasará a la salida hasta que no llegue el primer flanco de subida de reloj. Por lo tanto, la inclusión del `clk'event` es imprescindible y se debe especificar siempre. Además, una herramienta de síntesis tendría problemas en sintetizar una descripción que no tuviera la condición de flanco de subida completa.

Registro con réset asíncrono y carga síncrona

A esta estructura básica de memoria, que es el registro, se le pueden añadir todas las señales de control que se quieran. Si son síncronas con el reloj deberán venir siempre encerradas en el `IF` donde se detecta el flanco y no deben estar en la lista sensible, y si son asíncronas habrá que ponerlas antes del `IF` y en la lista sensible.

En este ejemplo se ha añadido una señal de carga síncrona, que en este caso es equivalente a una habilitación del reloj:

```
PROCESS(clk,reset)
BEGIN
  IF reset='1' THEN q<='0';
  ELSIF clk='1' AND clk'event THEN
    IF load='1' THEN q<=d;
    END IF;
  END IF;
END PROCESS;
```

La comparación `load='1'` se podía haber puesto en la misma condición que el reloj, pero el hacerlo como en el listado tiene ventajas; por una lado se separa claramente lo que es la señal de reloj del resto del circuito, y por otro lado, es una forma de indicarle a un sintetizador que intente no utilizar la señal de `load` para deshabilitar el reloj.

La importancia de no poner lógica combinacional sobre la señal de reloj viene del hecho de que si se pone se obtienen pulsos o *glitches* sobre esta señal. Estos micropulsos se pueden permitir en las señales de entrada a un biestable, pero nunca en la señal de reloj, ya que pueden producir un falso disparo del biestable. Una forma de habilitar o deshabilitar el reloj, sin utilizar lógica sobre esta señal, consiste en colocar un multiplexor en la entrada del biestable; la señal de habilitación entonces sirve para elegir el valor de entrada o el que ya tenía el biestable, de esta manera se tiene una señal de habilitación sin necesidad de tocar la señal del reloj. Un sintetizador suele hacer esto automáticamente aunque conviene asegurarse.

A partir de las estructuras vistas aquí se pueden describir registros de desplazamiento, contadores, etc., sin más que añadir alguna señal adicional.

Motor controlado por botón

Otra forma de utilización de los registros se da en el siguiente ejemplo que se encuentra también en el capítulo 12 (Ejemplo 12.1). Se ha incluido aquí porque el acercamiento al problema se hace de forma un poco diferente. Se verá entonces que la

descripción que se da aquí es más sencilla, e incluso más adecuada, si se desea sintetizar el circuito.

Ejemplo 6.3 *Un motor eléctrico viene controlado por un botón. Cada vez que se pulsa el botón cambia de encendido a apagado.*

Este problema se resuelve sencillamente mediante un *flip-flop* tipo D, activo por flanco de subida, y un inversor con el botón haciendo de reloj. Si se sabe cómo se describe esto en VHDL entonces el problema está resuelto. Si se desea una aproximación más abstracta se puede decir que lo que se quiere es que la salida cambie justo cuando se pulsa el botón, es decir, cuando la entrada pasa de cero a uno. En ambos casos la descripción coincide y se podría poner así:

```
ENTITY conmutador IS
PORT(boton: IN bit; motor: OUT bit);
END conmutador;

ARCHITECTURE serie OF conmutador IS
SIGNAL motoraux: bit:='0';
BEGIN
  PROCESS(boton)
  BEGIN
    IF boton='1' THEN motoraux<=NOT motoraux; END IF;
  END PROCESS;
  motor<=motoraux;
END serie;
```

Como la señal que activa todo es `boton`, es la única que se ha puesto en la lista sensible. Al ser la única señal, el proceso sólo se ejecutará si la señal del botón cambia, esto permite el ahorro de la comprobación de flanco (que se realiza con el atributo `'event`) que tendría que ponerse en el `IF` si la lista sensible hubiera contenido varias señales.

La señal auxiliar `motoraux` es necesaria ya que `motor` es una salida y, por lo tanto, no se puede leer; es decir, no se puede usar en la parte derecha de ninguna asignación. Siempre que se tienen que manejar valores de salida se hace lo mismo, es decir, se pone una instrucción en la que a la salida se le asigna la señal auxiliar, y en el resto del programa se usa únicamente la señal auxiliar. Es interesante darle un valor inicial a estas señales ya que luego, a la hora de sintetizar o simular, sirven para que el sistema se encuentre en un estado inicial conocido; en este caso se ha inicializado a cero para que el motor esté parado cuando empiece la simulación o se encienda el circuito.

Esta misma descripción se podría haber realizado totalmente concurrente y el resultado sería:

```
ARCHITECTURE concurrente OF conmutador IS
SIGNAL motoraux: bit:='0';
BEGIN
  motoraux<=NOT motoraux WHEN boton='1' AND boton'event
            ELSE motoraux;
  motor<=motoraux;
END concurrente;
```

Lo único que se ha hecho es poner el proceso anterior, que era un IF, en la forma concurrente equivalente del IF que es el WHEN. Al contrario que en el proceso, aquí es necesario indicar lo que se ha de hacer en todas las posibilidades de la condición. Como en el caso en que el botón está a cero no se quiere que haga nada, se le asigna a la señal el valor que ya tiene y así se queda como está. Ya se vio en su momento que otra forma de hacer esto era mediante el uso de la palabra clave UNAFFECTED.

Conviene destacar que el uso de `boton'event`, que detecta que la señal del botón ha cambiado, es necesario, ya que de lo contrario, para este caso particular, la instrucción podría entrar en un bucle infinito; por ejemplo, supongamos que `motoraux` está a cero y que de pronto se pulsa el botón, entonces la señal `motoraux` pasa a valer `'1'` y como cambia, se vuelve a ejecutar otra vez la instrucción, cambiando de nuevo, y así sucesivamente. Si se sintetizara el circuito, mientras se pulsara el botón, la salida cambiaría de 0 a 1 con un periodo impuesto por el retraso en el inversor y el registro. En simulación sería peor ya que el tiempo nunca avanzaría y entraría en un bucle infinito (a no ser que se pusiera una cláusula AFTER en cuyo caso pasaría como en síntesis).

Esta última forma de especificar el registro, aunque posible en simulación, no es posible en muchas herramientas de síntesis, que sencillamente se negarán a sintetizar cualquier instrucción concurrente que tenga el atributo `'event` u otro similar.

CAPÍTULO 7

DESCRIPCIÓN ESTRUCTURAL

En los capítulos anteriores se han mostrado los dos estilos funcionales para la realización de descripciones en VHDL: descripción flujo de datos y descripción algorítmica. Estos estilos son los más utilizados por ser más cercanos al pensamiento humano. Existe otro estilo de descripción en VHDL más cercano al *Netlist*, que es la descripción estructural. Este estilo lo utilizan sobre todo las herramientas de CAD para intercambiar información entre ellas.

A pesar de que la mayoría de descripciones van a ser de flujo de datos o algorítmicas, muchas veces resulta interesante conocer la descripción estructural ya que va a permitir la incorporación al diseño de elementos de biblioteca, realización de diseños jerárquicos a partir de componentes, etc. Esto es así porque la descripción estructural incluye la definición de la interfase entre unos elementos y otros.

Un lenguaje de *Netlist*, o de descripción estructural, debe ser capaz al menos de dos cosas: por un lado la definición y referencia de componentes, y por otro, la especificación de interconexiones entre unos y otros. En VHDL se dispone de estos y otros mecanismos de descripción estructural:

Definición de componentes: En VHDL los componentes se declaran con la palabra clave especial COMPONENT. Es algo muy parecido a la entidad.

Enlace entre componentes y entidades: Cada componente debe enlazarse con una entidad y una arquitectura predefinidas. De esta manera el componente es como una interfase que se puede referenciar en el diseño tantas veces como se necesite. Este enlace entre componentes, entidades y arquitecturas se realiza mediante sentencias FOR.

Definición de señales: Las señales sirven para conectar unos componentes con otros. Se declaran igual que las señales de cualquier otro tipo de descripción y tienen exactamente el mismo significado.

Referencia de componentes: Un mismo componente se puede referenciar o *instanciar* tantas veces como se necesite. La forma en que se hace es mediante las instrucciones de referencia que se verán más adelante en este mismo capítulo.

Configuración: Algunas de las definiciones anteriores, especialmente la de enlace, se pueden especificar en un bloque especial llamado configuración.

7.1 COMPONENTES, REFERENCIA Y ENLACE

7.1.1 Referencia de componentes

La reproducción de componentes, (replicación, instanciación, referencia, etc.) consiste en copiar un componente en la arquitectura, tantas veces como se necesite, completando así el diseño.

Es una instrucción concurrente y su forma general es la siguiente:

```
ref_id:
  [COMPONENT] nombre_componente                    |
   ENTITY nombre_entidad [(nombre_arquitectura)] |
   CONFIGURATION nombre_configuración
  [GENERIC MAP (parámetros)]
  [PORT MAP (puertos)] ;
```

El `ref_id` es el nombre concreto que se le da a la copia particular del componente, y es su referencia.

Después del nombre viene el elemento que se desea reproducir. A partir de la declaración se ve que es posible reproducir tres cosas: un componente, una entidad o una configuración. En los inicios del VHDL sólo existía la posibilidad de referenciar componentes, por eso la cláusula COMPONENT es opcional. Sin embargo, esto obliga a que finalmente el componente que se referencia tenga que ser asociado con una entidad, que es la que realmente tiene sentido ya que posee al menos una arquitectura con la descripción de lo que hace. En el caso de que se referencie una configuración, ésta deberá llevar asociada una entidad.

La definición de los parámetros del componente, si los tiene, y el paso de los puertos de conexión, si los tiene, se especifican con las cláusulas **GENERIC MAP** y **PORT MAP**. Son opcionales ya que o bien no son necesarios para un determinado componente, o bien, se han definido en otro sitio como en una configuración o en la parte declarativa de la arquitectura.

Ejemplos:

```
u1: and2 PORT MAP(e1=>a,e2=>b,s=>c);
u2: ENTITY or2(comportamental) PORT MAP(a,b,c);
u3: ENTITY rom GENERIC MAP(palabra=>15);
```

El primero reproduce una puerta AND de dos entradas, como no dice nada se supone que es un componente, esto significa que habrá que realizar el enlace entre componente y entidad en algún otro sitio. El segundo es una puerta OR pero esta vez se referencia directamente la entidad y además la arquitectura. El tercer ejemplo referencia directamente la entidad de una ROM donde se le pasa el tamaño de palabra como parámetro; podría igualmente tener puertos de entrada y salida, pero se supone que se definen estas conexiones en otro lugar.

7.1.2 Definición de componentes

Un lenguaje de *Netlist* debe poseer dos construcciones básicas, una es la de componente y la otra es la forma en que se interconectan. El VHDL'87 estructural también incluye estos dos mecanismos, el de declaración de componente y luego el de dar las interconexiones entre ellos.

Normalmente un componente se corresponde precisamente con una de las entidades declaradas en algún sitio en el diseño, o bien, en alguna biblioteca. Es necesario por tanto enlazar el componente con la entidad correspondiente. Esto al final acaba confundiendo el componente con la entidad a la que hace referencia. Para facilitar la descripción, y como al final lo que se quiere referenciar es la entidad, el VHDL'93 permite la referencia directa, réplica, reproducción o instanciación (*instantiation*) de entidades sin necesidad de declarar componentes.

La declaración de componente no es estrictamente necesaria si se tiene un compilador de VHDL'93, pero como se sigue utilizando mucho es conveniente conocerla:

```
COMPONENT nombre [IS]
  [GENERIC(lista_parámetros);]
  [PORT(lista_puertos);]
END COMPONENT [nombre];
```

Si se compara con la declaración de entidad vista ya en 4.5.1 se puede comprobar que son bastante similares.

Ejemplos

```
COMPONENT and2
  PORT(e1,e2: IN bit; s: OUT bit);
END COMPONENT and2;

COMPONENT rom
  GENERIC(palabra: integer);
  PORT(dir: IN bit_vector(7 DOWNTO 0);
       dat: OUT bit_vector(palabra DOWNTO 0));
END COMPONENT rom;
```

El primer componente es una AND de dos entradas y una salida de un bit. El segundo es una ROM a la cual se le ha incluido el tamaño de palabra como parámetro. Como se ve, no hay mucha diferencia con la definición de entidad.

7.1.3 Enlace entre componentes y entidades. Sentencia FOR

Un componente a solas no es más que una interfase sin ninguna funcionalidad asociada. Para que el componente tenga algún sentido más allá de la descripción de las entradas y salidas, hay que asociarlo o enlazarlo con una entidad y una arquitectura.

La forma de declarar este enlace entre componente y entidad se realiza mediante la sentencia FOR. La sintaxis de esta sentencia, para el caso de componentes, es la siguiente:

```
FOR lista_refs : nombre_componente
   USE ENTITY nombre_entidad [ (nombre_arquitectura) ] |
   USE CONFIGURATION nombre_configuración
  [GENERIC MAP (parámetros)]
  [PORT MAP (puertos)] ;
```

La `lista_refs` es una lista, separada por comas, de parte de las referencias realizadas del componente `nombre_componente`. En esta lista se pueden utilizar también

las palabras clave ALL y OTHERS para seleccionar todas las referencias hechas del componente, o todas las que queden por enlazar con una entidad. Normalmente todas las referencias de un componente están enlazadas con la misma entidad y arquitectura, pero en ocasiones puede ser interesante elegir una entidad, o una arquitectura, diferente para algún componente crítico.

Después del FOR viene la cláusula USE que permite dos posibilidades. Lo normal es enlazar con una entidad, de manera que se utiliza habitualmente la cláusula USE ENTITY seguida del nombre de la entidad y, opcionalmente, de una arquitectura, perteneciente a esa entidad, entre paréntesis. La otra posibilidad es enlazar directamente con un bloque de configuración donde habrá información sobre la entidad a enlazar.

Por último existe la posibilidad de incluir el paso de parámetros al componente (GENERIC MAP) o la definición de las conexiones a los puertos (PORT MAP). Lo normal es que estas dos últimas definiciones se den en la propia referencia del componente en la arquitectura, pero también existe la posibilidad de especificarlas en esta instrucción.

La sentencia FOR es en realidad mucho más compleja, ya que no sólo sirve para especificar componentes sino que también se puede usar para dar detalles sobre bloques, arquitecturas y sentencias de generación. En el apéndice A se encuentra la especificación BNF del VHDL'93; un análisis de su sintaxis nos demuestra la amplitud de posibilidades de la sentencia FOR.

La sintaxis del enlazado entre componente y entidad también depende de dónde se coloque la sentencia. La sintaxis que se acaba de describir es para enlazar componentes con sus referencias en la parte declarativa de la arquitectura. A esto se le llama *especificación de configuración*. Existe la posibilidad de declarar este enlace en un bloque especial, que se verá más adelante, que es específico para configuración. En ese caso la declaración del enlace entre componente y entidad se llama *declaración de configuración*, y la sintaxis es un poco diferente aunque el significado es el mismo.

Ejemplos:

```
FOR ALL: inv USE ENTITY work.inversor;
FOR u8,u23: and2 USE ENTITY work.puerta_and2(concurrente);
FOR OTHERS: and2 USE ENTITY work.puerta_and2(serie);
```

En el primer ejemplo se enlazan todas las copias del componente `inv` con una entidad que se llama `inversor`, como no se especifica arquitectura se usará la última compilada para esa entidad. El segundo y tercer ejemplos van juntos ya que, en el pri-

mero, se enlazan las copias `u8` y `u23` del componente `and2` con la entidad `puerta_and2` y la arquitectura `concurrente`; en el tercer ejemplo se enlazan el resto de componentes `and2` que quedan con la misma entidad pero con la arquitectura `serie`.

7.1.4 Repetición de estructuras: GENERATE

En muchas ocasiones el hardware está compuesto por estructuras que se repiten una y otra vez. Para poder describir estructuras repetitivas, el VHDL tiene la instrucción **GENERATE** que repite, tantas veces como se diga, las sentencias especificadas en su interior. Las dos formas generales de esta sentencia son las siguientes:

```
id_generate:
FOR parámetro_repetitivo GENERATE
   [declaraciones
BEGIN]
   [sentencias]
END GENERATE [id_generate] ;
```

```
id_generate:
IF condición GENERATE
   [declaraciones
BEGIN]
   [sentencias]
END GENERATE [id_generate] ;
```

La etiqueta, al igual que en los bloques **BLOCK**, es también obligatoria. De alguna manera la sentencia de generación equivale a la del bloque, con la diferencia de que dentro de un **GENERATE** las cosas se encuentran repetidas varias veces.

Esta sentencia es concurrente, por lo que no debe venir dentro de los **PROCESS** o subprogramas. Las sentencias que puede incluir son cualquier instrucción concurrente. Esto significa que se pueden poner también sentencias de generación dentro de sentencias de generación. También entran aquí sentencias de bloque y, naturalmente, el **PROCESS**, lo que permite repetir y generar descripciones algorítmicas.

La forma más común de la sentencia **GENERATE** es la que se mostraba a la izquierda, es decir, la formada por la palabra **FOR**. En ese caso se reproduce todo lo que viene entre **GENERATE** y **END GENERATE** tantas veces como indique la especificación del **FOR**. Lo que distingue una copia de la siguiente es el parámetro cambiante especificado en el **FOR**.

Uno de los ejemplos más simples en los que se puede pensar es la descripción de un registro de varios bits a partir de un *flip-flop* de un solo bit. Ejemplo:

```
ENTITY registro IS
GENERIC(ancho: integer);
PORT (clock,ena: IN bit;
      r_in: IN bit_vector(0 TO ancho-1);
      r_out: OUT bit_vector(0 TO ancho-1));
END registro;
```

```
ARCHITECTURE estructural OF registro IS
  COMPONENT ff IS PORT(clk,d: IN bit; q: OUT bit);
  END COMPONENT;
  SIGNAL aux_r_out: bit_vector(0 TO ancho-1); -- Para leer la salida.

BEGIN
  r_out<=aux_r_out;

  vector:
  FOR bit_idx IN 0 TO ancho-1 GENERATE
     SIGNAL interna: bit;
  BEGIN
    interna<=r_in(bit_idx) WHEN ena='1' ELSE aux_r_out(bit_idx);
    flipflop: ff PORT MAP(clk=>clock,d=>interna(bit_idx),
                          q=>aux_r_out(bit_idx));
  END GENERATE;

END estructural;
```

En este ejemplo se repite todo lo que viene en el **GENERATE** un número `ancho` de veces; eso incluye no sólo el biestable sino también la señal `interna` declarada dentro del bucle. Esto quiere decir que en una sentencia de generación todo lo declarado en el bucle no sólo se repite para cada copia, sino que además es local a cada copia.

En el ejemplo mostrado la diferencia entre un *flip-flop* y el siguiente es su conexión, que depende del valor del índice `bit_idx`. Normalmente es el índice especificado en el bucle el que distingue unos elementos de otros.

En la descripción de la sintaxis de esta sentencia se habían mostrado dos posibilidades: una se acaba de ver y consiste en la repetición de elementos iguales (**FOR**); la otra se muestra a continuación y consiste en que si una condición se cumple se genera lo especificado, y si no, no.

En principio esta posibilidad condicional se puede utilizar para generar o no cierta estructura a voluntad. Lo que ocurre es que esto no tiene demasiado sentido por sí solo. Donde tiene sentido, y es la forma en que se suele utilizar, es en la generación de estructuras repetitivas donde algún elemento es diferente al resto.

El ejemplo más típico es el de un registro de desplazamiento donde el primer y último *flip-flops* son diferentes. Suponiendo una señal `intercon` de `ancho-1` bits declarada en la arquitectura, y que `s_en` y `s_sal` son las entradas y salidas serie del registro, se puede obtener la descripción del registro de desplazamiento:

```
reg_desplaza:
FOR bit_idx IN 0 TO ancho-1 GENERATE
  primero: IF (bit_idx=0) GENERATE
             priff: ff PORT MAP(clock,s_en,intercon(bit_idx));
```

```
            END GENERATE;
  medios:   IF (bit_idx/=0 AND bit_idx/=N-1) GENERATE
               medff: ff PORT MAP(clock,intercon(bit_idx-1),
                                  intercon(bit_idx);
            END GENERATE;
  ultimo:   IF (bit_idx=N-1) GENERATE
               ultff: ff PORT MAP(clock,intercon(bit_idx-1),s_sal);
            END GENERATE;
END GENERATE reg_desplaza;
```

El ejemplo es una sentencia de generación de bucle que lleva tres sentencias de generación condicionales en su interior. Si el índice es cero se genera el primer *flip-flop*, si es `ancho-1` entonces el último, y si es cualquier otro se genera la celda normal de desplazamiento. Otra forma de realizar esto es mediante la utilización de un **GENERATE** sin más, y luego conectar los extremos de la señal `intercon` a `s_en` y `s_sal`.

Hay que destacar que, tal y como se muestra en el ejemplo, no es necesario incluir la palabra clave **BEGIN** si no hay declaraciones.

7.2 LA UNIDAD DE CONFIGURACIÓN

A lo largo de este capítulo se ha puesto de manifiesto la importancia que tiene el enlazar, o hacer corresponder, un componente con una entidad y, por tanto, con una arquitectura. Este enlace se realiza mediante un **FOR** y un **USE**.

Hasta ahora la definición de enlace se realizaba en la parte declarativa de la arquitectura. A esto es a lo que se le llama una especificación de configuración. Sin embargo, existe otra posibilidad de enlazar componentes con entidades y arquitecturas, y es lo que se llama declaración de configuración.

La declaración de configuración se realiza en un bloque especial del VHDL llamado *configuración*. En este bloque se pueden especificar los mismos enlaces componente-entidad que se habían visto en la parte declarativa de la arquitectura.

La forma general de la declaración de la configuración de una entidad es la siguiente:

```
CONFIGURATION nombre_conf OF nombre_entidad IS
  {sentencia_use |
   def_atributo  |
   def_grupos}
   Configuración_bloque
END [CONFIGURATION] [nombre_conf] ;
```

Hay que destacar que un bloque de configuración, al igual que ocurría con la arquitectura, está siempre asociado a una entidad. Todo lo que viene en la configuración se refiere a la entidad indicada y a los elementos de ésta. Lo primero que se puede poner es una cláusula USE tal y como se muestra en la sección 8.2. También caben aquí las definiciones de atributos que se mostraron en la sección 4.3.1.

El tercer elemento, la definición de grupos, es algo que no se ha comentado todavía. Un grupo es una colección de elementos al que se le da un nombre. Un patrón de grupo define el tipo de elementos que pueden pertenecer a ese grupo. Los grupos no tienen mucho significado en simulación, pero se pueden usar, al igual que los atributos, para pasar información de unas herramientas a otras (ejemplo: síntesis). La palabra clave para su definición es GROUP y se usa realmente poco.

El cuarto elemento del bloque configuración es el más importante y, realmente, el único obligatorio. Aunque se llama configuración de bloque, no se refiere al bloque definido con la instrucción BLOCK sino que se refiere a las posibles estructuras que se encuentran en una arquitectura, y se puede decir que hay tres niveles: los elementos que se encuentran en la propia arquitectura, los que se encuentran dentro de bloques, y los que se encuentran dentro de bloques GENERATE. Como estos dos últimos admiten varios niveles jerárquicos, la propia definición de la configuración de bloque también será jerárquica.

La forma general de definición de la configuración de bloque es la siguiente:

```
FOR especificación_bloque
  {sentencia_use}
  {configuración_bloque | configuración_componente}
END FOR;
```

Hay tres elementos que pueden constituir la `especificación_bloque`: lo normal es poner aquí el nombre de una arquitectura, pero también puede ser el identificador de un bloque BLOCK o el de un bloque GENERATE. En cada caso, los elementos definidos en el FOR serán aquellos pertenecientes a la arquitectura, bloque, o bloque de generación según se hayan definido. Al igual que en la configuración, aquí también se pueden poner cláusulas USE. A continuación se tiene la posibilidad de definir otra vez una `configuración_bloque`, o bien, la configuración de un componente.

La posibilidad de incluir de nuevo la configuración del bloque es debido a que el diseño puede tener una estructura jerárquica, lo que significa que puede haber unos bloques dentro de otros, unas sentencias de generación dentro de otras, y lo que es aun más complejo, las unas mezcladas dentro de las otras.

La configuración de componente es prácticamente igual que lo que ya se había mostrado para enlazar el componente con su entidad. Pero existen sutiles diferencias en la definición. La especificación de la configuración del componente se realiza de esta manera:

```
FOR lista_refs : nombre_componente
   USE ENTITY nombre_entidad [ (nombre_arquitectura) ] |
   USE CONFIGURATION nombre_configuración
  [GENERIC MAP (parámetros)]
  [PORT MAP (puertos)] ;
  [configuración_bloque]
END FOR ;
```

Prácticamente es lo mismo que se había visto en la especificación de configuración de la página 86, pero hay diferencias: la primera es que aquí se admite la configuración de bloque, lo que permite especificar jerarquía; la segunda diferencia es que esta sentencia termina con END FOR en vez del punto y coma de la especificación de configuración.

Toda esta complejidad de declaración en la configuración viene del hecho de que los diseños pueden ser jerárquicos, lo que complica la visibilidad de determinados elementos.

A continuación se da una posible configuración de una entidad que tiene bloques, sentencias de generación, etc.:

```
CONFIGURATION tal OF cual IS
FOR funcional  -- Este es el nombre de la arquitectura
  FOR bloque1
    FOR u1: y2 USE ENTITY gates.and2(functional);
    END FOR;
    FOR gen2
      FOR u3: suma USE CONFIGURATION operadores.suma_func;
      END FOR;
    END FOR;
  END FOR;
  FOR gen1
    ...
  END FOR;
  ...
END FOR;
END CONFIGURATION tal;
```

En este ejemplo se da la configuración de la arquitectura `funcional` de la entidad `cual`. La arquitectura tiene un bloque `bloque1` con un componente de referencia `u1`, y un bloque de generación que a su vez tiene un componente con referencia `u3`, que se ha enlazado con una configuración en vez de con una entidad. La arquitectura tam-

bién tiene otro bloque de generación que se llama `gen1`, que a su vez puede tener más componentes, bloques, etc.

Si el diseño que se está realizando es complejo, es conveniente realizar todos los enlaces entre componentes y entidades en un bloque de configuración. La ventaja frente a la especificación de configuración, donde el enlace se hace directamente en la declaración de arquitectura, es que en la configuración se puede especificar cualquier componente en cualquier nivel de jerarquía. Además, en caso de querer modificar los enlaces, es mejor modificar el fichero que contenga la configuración antes que editar el fichero con la arquitectura.

Normalmente se enlaza un componente con una entidad, pero se había visto que es también posible enlazar un componente con una configuración que estará asociada naturalmente a una entidad. El hacerlo así tiene muchas ventajas, ya que en diseños jerárquicos todos los subniveles de esta entidad estarán contenidos en la configuración de esa entidad, de otra manera, habría que referenciar los subniveles de esa entidad en la configuración de nivel alto. Otro uso es que permite decir qué arquitectura utilizar para esa determinada entidad. En el ejemplo anterior se enlazaba el componente `suma` con la configuración `suma_func`, un posible contenido de esta configuración, a fin de definir qué arquitectura elegir, podría ser éste:

```
CONFIGURATION suma_func OF suma IS
  FOR funcional            -- Elige la arquitectura 'funcional'
  END FOR;                 -- como la arquitectura a utilizar.
END CONFIGURATION suma_func;
```

Si la entidad sólo tuviera esa arquitectura no habría sido necesario crear una configuración. Tampoco habría sido necesario crearla si esa arquitectura fuera la última de todas las posibles, y es que las herramientas de síntesis o simulación suelen coger la última arquitectura compilada de una entidad.

7.2.1 Ejemplo

Ejemplo 7.1 *Realizar la descripción estructural, con componentes y configuración, del ejemplo 3.1.*

En aquel ejemplo ya se dio una descripción estructural del multiplexor, pero entonces se referenciaba directamente la entidad, por lo que no era necesaria una especificación de configuración. Ésta es una de las ventajas del VHDL'93, pero como no siempre se tiene esta posibilidad, se va a mostrar cómo se haría en el caso de tener componentes.

```
-- Lo primero de todo es la definición de las entidades y sus
-- arquitecturas. Esto se podría poner en un fichero aparte o
-- definirlo dentro de una biblioteca:

ENTITY inv IS PORT  (e: IN bit; y: OUT bit); END inv;
ENTITY and2 IS PORT  (e1,e2: IN bit; y: OUT bit); END and2;
ENTITY or2 IS PORT  (e1,e2: IN bit; y: OUT bit); END or2;

ARCHITECTURE rtl OF inv IS BEGIN  y<=NOT e; END rtl;
ARCHITECTURE rtl OF or2 IS BEGIN  y<=e1 OR e2; END rtl;
ARCHITECTURE rtla OF and2 IS BEGIN  y<=e1 AND e2; END rtla;
ARCHITECTURE rtlb OF and2 IS  -- dos arquitecturas diferentes
BEGIN
   y<='0' WHEN (e1='0' OR e2='0') ELSE '1';
END rtlb;

-- A continuación ya viene la entidad y arquitectura del mux:

ENTITY mux IS
PORT ( a,b,selec:  IN bit;
       salida: OUT bit);
END mux;

ARCHITECTURE estructura OF mux IS
  COMPONENT inv IS PORT(e: IN bit; y: OUT bit);     END COMPONENT;
  COMPONENT y2  IS PORT(e1,e2: IN bit; y: OUT bit); END COMPONENT;
  COMPONENT o2  IS PORT(e1,e2: IN bit; y: OUT bit); END COMPONENT;
  SIGNAL ax,bx,nosel: bit;
BEGIN
  u0: inv PORT MAP(e=>selec,y=>nosel);
  u1: y2  PORT MAP(e1=>a,e2=>nosel,y=>ax);
  u2: y2  PORT MAP(b,selec,bx);
  u3: o2  PORT MAP(e1=>ax,e2=>bx,y=>salida);
END estructura;

-- Por último, la configuración sería:

CONFIGURATION estru OF mux IS
-- poniendo USE work.ALL; aquí, no haría falta poner 'work' cada vez.
  FOR estructura
    FOR ALL:    inv USE ENTITY work.inv;         END FOR;
    FOR u1:      y2 USE ENTITY work.and2(rtla);  END FOR;
    FOR OTHERS: y2 USE ENTITY work.and2(rtlb);  END FOR;
    FOR ALL:    o2 USE ENTITY work.or2;          END FOR;
  END FOR;
END estru;
```

Si en vez de una declaración de configuración se hubiera hecho una especificación de configuración, es decir, dar el enlace en la parte declarativa de la arquitectura, se habría obtenido exactamente el mismo resultado.

Aunque es conveniente realizar el enlace entre componente y entidad, no siempre es necesario. Si por ejemplo el componente se llama igual que la entidad a la que viene asociada, no será necesario realizar el enlace. En el ejemplo que se acaba de ver, por tanto, no era necesario realizar el enlace de la puerta `inv` ya que existe una entidad con el mismo nombre.

CAPÍTULO 8

PONIENDO ORDEN: SUBPROGRAMAS, PAQUETES Y BIBLIOTECAS

En descripciones complejas de un circuito o programa se hace necesaria una organización que permita al diseñador trabajar con grandes cantidades de información. Ya se han visto los bloques como forma de estructurar una descripción en forma modular permitiendo establecer una jerarquía. Hay otras formas de organizar la información y es la introducción de funciones y procedimientos, llamados *subprogramas*, que hacen más legibles las descripciones. Por otro lado, y a un nivel más elevado, se pueden agrupar subprogramas, definiciones de tipos, bloques, etc. en estructuras por encima de la propia descripción; estos elementos son los que formarían los paquetes que a su vez, junto con otros elementos de configuración, etc., formarían las bibliotecas. De todos estos elementos es de los que trata este capítulo.

8.1 SUBPROGRAMAS

Al igual que ocurre en la mayoría de los lenguajes de programación, también el VHDL se puede estructurar mediante el uso de subprogramas. Un subprograma no es más que una función o procedimiento que contiene una porción de código.

Las funciones y procedimientos en VHDL son estructuras muy parecidas entre sí aunque existen algunas diferencias. Estas diferencias se especifican a continuación:

- Una función siempre devuelve un valor, mientras que un procedimiento sólo puede devolver valores a través de los parámetros que se le pasen.
- Los argumentos de una función son siempre de entrada, por lo que sólo se pueden leer dentro de la función. En el procedimiento pueden ser de entrada, de salida o

de entrada y salida, por lo que pueden sufrir modificaciones.

- Una función no tiene efectos colaterales, pero un procedimiento sí, es decir, puede provocar cambios en objetos externos a él debido a que se pueden cambiar las señales aunque no se hubieran especificado en el argumento. Es decir, en los procedimientos se permite realizar asignaciones sobre señales declaradas en la arquitectura y, por tanto, externas al procedimiento.
- Las funciones, como devuelven un valor, se usan en expresiones, mientras que los procedimientos se llaman como una sentencia secuencial o concurrente.
- La función debe contener la palabra clave RETURN seguida de una expresión puesto que siempre devuelve un valor, mientras que en el procedimiento no es necesario.
- Una función jamás puede tener la instrucción WAIT, mientras que un procedimiento sí.

8.1.1 Declaración de procedimientos y funciones

Las declaraciones de procedimientos y funciones se realizan en las partes declarativas de las arquitecturas, bloques, paquetes, etc. Las declaraciones de una estructura y otra son muy parecidas. A continuación se muestra la definición de un procedimiento:

```
PROCEDURE nombre[(parámetros)] IS
   [declaraciones]
BEGIN
   [sentencias_serie]
END [PROCEDURE] [nombre] ;
```

La declaración de funciones es muy similar, con la diferencia de que hay que indicar el tipo del valor que devuelve:

```
[PURE | IMPURE]
FUNCTION nombre[(parámetros)] RETURN tipo IS
   [declaraciones]
BEGIN
   [sentencias serie]     -- debe incluir al menos un RETURN
END [FUNCTION] [nombre] ;
```

La lista de parámetros, tanto en la función como en el procedimiento, es opcional. Si no hay parámetros tampoco es necesario usar los paréntesis. Su significado es el

mismo en procedimientos que en funciones. Esta lista no es más que el conjunto de parámetros que se le pasan al subprograma, y se declaran de forma muy parecida a como se declaraban los puertos en una entidad; primero se pone la clase de objeto de que se trata, es decir, una señal, una variable o una constante; a continuación se pone el nombre del objeto, y después dos puntos seguidos por el tipo de puerto, que será IN, OUT o INOUT. Al igual que en la entidad, el tipo IN sólo se puede leer, el tipo OUT sólo se puede escribir, y el tipo INOUT se puede escribir y leer. Por último se especifica el tipo de objeto que es.

Dependiendo de si se trata de funciones o procedimientos, los parámetros pueden ser una cosa u otra:

Funciones:

- Sólo es válido el modo IN, por lo que no es necesario especificarlo.
- El parámetro puede ser CONSTANT o SIGNAL. El valor por defecto es CONSTANT; si se desea SIGNAL hay que ponerlo explícitamente.

Procedimientos:

- Por defecto el modo es IN, pero el OUT e INOUT son también válidos.
- Por defecto la clase es CONSTANT si el modo es IN, o VARIABLE en el resto de casos. La clase SIGNAL hay que declararla explícitamente.

No es aconsejable el uso de señales como parámetros puesto que pueden llevar a confusión dada la especial forma en que estos objetos se asignan, sin embargo, su uso es posible. En el caso de las señales hay que tener especial cuidado en el uso de los atributos ya que algunos no están permitidos en el interior de funciones; éste es precisamente el caso de `'stable`, `'quiet`, `'transaction` y `'delayed`.

En el caso de las funciones se debe especificar, además, el tipo del objeto que devuelve la función. Como las funciones siempre devuelven algo, esto implica que debe existir una instrucción RETURN en el interior del cuerpo de la función y, además, esta instrucción debe estar seguida por una expresión que es precisamente lo que se devuelve. El uso del RETURN en procedimientos es posible, pero no debe llevar una expresión puesto que los procedimientos no devuelven nada. Si se usa en procedimientos simplemente se interrumpe la ejecución del procedimiento y vuelve.

Las declaraciones dentro de una función o procedimiento pueden incluir las mismas que incluiría un PROCESS, ya que se trata también de bloques de ejecución serie. Por lo tanto, y al igual que sucede en un proceso, no se pueden declarar señales en una función o procedimiento. Naturalmente, todo lo que se declara en esta parte sólo es visible en el cuerpo de la función.

En una función, y siempre con carácter opcional e informativo, existe la posibilidad de declararla como PURE o IMPURE. Un función *pura* es una función que devuelve siempre el mismo valor para unos parámetros de entrada dados. Una función es *impura* si para unos mismos parámetros puede devolver valores diferentes.

Normalmente las funciones impuras lo son porque dependen de alguna variable o señal global. Poner PURE o IMPURE en la declaración de la función no hace a esa función pura o impura, en realidad no hace nada, es sólo un comentario.

Un ejemplo típico de función impura es la función predefinida now. Esta función devuelve el tiempo que ha pasado desde el inicio de la simulación. Su definición es la siguiente:

```
IMPURE FUNCTION now RETURN delay_length;
```

El tipo delay_length es un subtipo del predefinido time. Esta función es evidentemente impura, ya que, dependiendo del tiempo que haya pasado durante la simulación, devuelve una cosa u otra.

Especificación de función y procedimiento

En muchas ocasiones resulta útil declarar la función antes de declarar su cuerpo por motivos de visibilidad, etc. En estos casos la declaración se hace igual que se ha visto pero se sustituye el IS por un punto y coma dando por finalizada la especificación:

```
-- Procedimiento:
PROCEDURE nombre [ (parámetros) ] ;

-- Función:
[PURE | IMPURE] FUNCTION nombre [ (parámetros) ] RETURN tipo ;
```

A continuación se presenta un ejemplo donde se define un procedimiento que calcula los valores máximo y mínimo de los números contenidos en un vector cuyo rango viene especificado en el propio tipo que se llama vec.

```
-- más corto: PROCEDURE lims(cjt: vec; min,max: OUT integer) IS
PROCEDURE lims(CONSTANT cjt: IN vec; VARIABLE min,max: OUT integer) IS
VARIABLE ind: integer;
BEGIN
```

```
   min:=cjt(cjt'left);    -- valores iniciales de min y max
   max:=cjt(cjt'right);
   FOR ind IN cjt'range LOOP
      IF min>cjt(ind) THEN min:=cjt(ind); END IF;
      IF max<cjt(ind) THEN max:=cjt(ind); END IF;
   END LOOP;
END lims;
```

En este ejemplo se muestra el encabezamiento completo y, comentado, otro encabezamiento que también es válido por tomar los valores por defecto, de hecho se aconseja utilizar la forma abreviada.

El funcionamiento es muy simple. Por un lado utiliza la variable de entrada `cjt` y no la modifica puesto que es de entrada. Como devuelve dos resultados, `min` y `max`, no se puede utilizar una función, ya que ésta sólo puede devolver un resultado. El uso de un procedimiento permite que se puedan devolver dos parámetros a través de sus argumentos. En el caso de que sea sólo necesaria la devolución de un parámetro, es mejor elegir una función.

8.1.2 Llamadas a subprogramas

La forma de invocar un subprograma consiste solamente en indicar el nombre seguido por los argumentos entre paréntesis, si los tiene. A las funciones sólo se las puede invocar como parte de una expresión, mientras que los procedimientos se llaman como si fueran una sentencia, secuencial o concurrente.

Hay tres formas de pasar parámetros a un subprograma. La primera es poniendo los parámetros en el mismo orden en que se declaran. Ésta es la forma normal en que suelen funcionar los lenguajes de programación, pero VHDL permite dos más: una es mediante la asociación explícita, que permite poner los parámetros en cualquier orden, y la otra permite dejar parámetros por especificar, de manera que se toman unos valores por defecto. Aparte de éstas, existe otra forma para el caso de los operadores con notación infija, pero esto se verá más adelante cuando se explique la sobrecarga de operadores.

A continuación se dan unas llamadas al procedimiento del ejemplo anterior junto con otro procedimiento que no tiene parámetros:

```
lims(conj(4 TO 20),valmin,valmax);
reset;              -- llamada a un procedimiento sin argumentos
-- Asociación explícita:
lims(min=>valmin,max=>valmax,cjt=>conj(4 TO 20));
-- Las funciones siempre forman parte de una expresión:
```

```
tiempo_lleva:=now;
resultado:=calcula(a,b,c);
```

En la declaración del procedimiento o función se pueden dar valores por defecto a algunos parámetros. Esto permite que se pueda llamar a ese subprograma sin necesidad de pasarle este parámetro; si se le pasa toma el que se le pase, y si no, toma el que tenga por defecto. Ejemplo:

```
PROCEDURE lee_bloque(longitud: positive := 256);
-- Con esta definición, las siguientes llamadas son equivalentes:
lee_bloque;
lee_bloque(256);
```

Aunque la ejecución dentro de los subprogramas es siempre serie como en un proceso, éstos pueden ser llamados tanto en entornos serie (procesos) como en entornos concurrentes. En el caso de ser invocados en entornos concurrentes, los subprogramas se ejecutan igual que un proceso cuya lista sensible estuviera compuesta por aquellos argumentos del subprograma que fueran de tipo IN o INOUT.

Un procedimiento llamado desde un entorno concurrente se comporta exactamente igual que un bloque PROCESS, de manera que la ejecución externamente es concurrente, pero internamente la ejecución es en serie. Como en el PROCESS, es necesario incluir alguna sentencia que permita suspender la ejecución del procedimiento, de otra manera se ejecutaría indefinidamente. Si un procedimiento tiene argumentos de tipo IN o INOUT, se considerará a éstos como la lista sensible de activación del procedimiento. Si no existen argumentos en el procedimiento y la llamada se produce en un entorno concurrente, será necesario incluir una sentencia de espera (WAIT) dentro del procedimiento, de lo contrario se entraría en un bucle infinito.

En llamadas en entornos serie hay que tener cuidado de no incluir ninguna instrucción WAIT si es que en el proceso que llamó al procedimiento existe una lista sensible. Si existe una lista sensible en un nivel superior no se puede poner el WAIT, pero si no hay ninguna, sí que se puede. En ese caso hay que tener cuidado de que el procedimiento no se detenga para siempre, a no ser de que sea eso lo que se desea hacer.

8.1.3 Sobrecarga de operadores

La sobrecarga de funciones permite que puedan existir funciones con el mismo nombre, siendo la diferencia el tipo de datos que devuelven, el tipo de datos de sus argumentos, o el número y orden de éstos. Esto es especialmente útil cuando se desea ampliar la

cobertura de algunos operadores predefinidos. En principio no es necesario indicar que se ha sobrecargado una función, ya que en tiempo de compilación o ejecución el compilador o intérprete del programa elegirá una función u otra dependiendo de los tipos de datos que se estén utilizando en ese momento.

Tal y como se ha dicho, lo que marca la diferencia entre una función y otra con el mismo nombre es el *tipo* de sus argumentos o lo que devuelve. No hay que confundir el *tipo* con el *subtipo* ya que, en principio, se puede pensar que dos funciones con *subtipos* diferentes son diferentes, en cambio, esto no tiene por qué ser así. En efecto, si un parámetro tiene subtipos diferentes en una y otra función, pero ambos subtipos vienen del mismo tipo, entonces la sobrecarga no es posible.

Los procedimientos en VHDL se pueden llamar con menos de los parámetros que se tengan especificados siempre que se hayan declarado unos valores por defecto. Hay que tener un poco de cuidado con esto ya que, por ejemplo, pueden haber dos funciones que se llamen igual y tengan un parámetro que las diferencie, pero precisamente este parámetro tiene un valor por defecto; si se llama a esa función y no se especifica el parámetro que las diferencia, no habrá forma de saber a qué subprograma se está refiriendo la llamada. Ejemplo:

```
PROCEDURE lee_bloque(longitud: positive := 256);
PROCEDURE lee_bloque(factor: real := 1.0);
-- Con estas definiciones correctas, la siguiente llamada da error:
lee_bloque;
```

Lo normal es que la sobrecarga se emplee en operadores. Los operadores, a diferencia de las funciones normales, siguen una notación infija que además no necesita paréntesis. En VHDL se pueden definir también este tipo de operadores de la misma manera que una función normal; la inclusión de unas comillas dobles alrededor del nombre indicará que se trata de un función que permite notación infija. A continuación se muestra un ejemplo donde se sobrecarga la operación suma para que funcione también sobre un tipo `bit_vector` de 8 bits definido como `byte`. (Se supone que se han declarado en otros sitios unas funciones que pasan de `bit_vector` a integer y viceversa.):

```
FUNCTION "+"(a,b: byte) RETURN byte IS
BEGIN
   RETURN inttobyte(bytetoint(a)+bytetoint(b));
END "+";
```

En este ejemplo se le ha dado otra definición a la operación suma que convive con la función suma anterior que sólo servía para enteros. De hecho, se ha utilizado la

operación suma anterior para definir esta nueva. No hay confusión posible puesto que los tipos de datos son diferentes, así que se realizará una u otra según el tipo de datos.

Tal y como se ha definido la función, su nombre es en realidad `"+"`, por lo que se puede usar como una función normal utilizando su nombre completo. De esta manera, las expresiones `X"FF"+X"80"` y `"+"(X"FF",X"80")` son equivalentes.

Ejemplo 8.1 *Se define el tipo enumerado* `logico` *como* `('X','0','1','Z')`*, es decir, como el tipo* `bit` *pero añadiendo los valores* `'X'` *(desconocido) y* `'Z'` *(alta impedancia). Sobrecargar el operador* AND *para que se pueda usar esta operación con este nuevo tipo.*

Una definición conservadora es aquella que dará desconocido siempre que alguna de las entradas sea `'X'` o `'Z'`:

```
FUNCTION "and"(a,b: logico) RETURN logico IS
BEGIN
   CASE a&b IS
      WHEN "00" => RETURN '0';
      WHEN "01" => RETURN '0';
      WHEN "10" => RETURN '0';
      WHEN "11" => RETURN '1';
      WHEN OTHERS => RETURN 'X';
   END CASE;
END "and";
```

Una definición más realista podría tener en cuenta una familia lógica determinada y, por ejemplo, suponer que una entrada `'Z'` es equivalente a un `'1'`. Esto se propone como ejercicio.

8.2 BIBLIOTECAS, PAQUETES Y UNIDADES

Hasta ahora se han mostrado las diferentes estructuras del lenguaje VHDL para la descripción de circuitos. En esta sección se verá cómo se unen todos los elementos anteriores para formar una descripción completa de un sistema digital.

Los elementos que se han visto hasta ahora eran las entidades y las arquitecturas, la entidad servía para definir el interfase de un módulo o sistema, mientras que la arquitectura describía el comportamiento del circuito. A este tipo de estructuras se las conoce como *unidades*, y a continuación se verá que hay algunas más que las que se han visto hasta ahora.

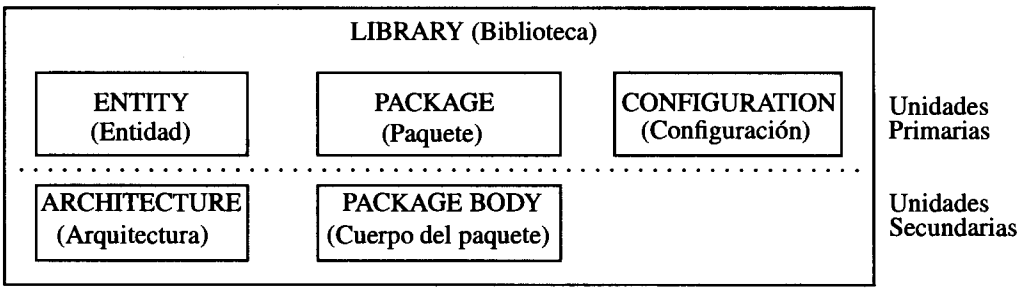


Figura 8.1: Las bibliotecas y las unidades que la componen

Al realizar una descripción en VHDL, estas unidades se suelen introducir en un mismo fichero o en varios. Cada uno de estos ficheros es lo que se llama un *fichero de diseño*. Normalmente, antes de simular o sintetizar un circuito descrito en VHDL, estos *ficheros de diseño* se compilan previamente. El resultado de la compilación, aparte de realizarse la correspondiente comprobación de sintaxis, es lo que se llama una *biblioteca de diseño*. Es decir, lo que inicialmente constituye uno o varios *ficheros de diseño* de la descripción hardware, pasa a ser una única *biblioteca de diseño* después de la compilación, de manera que esta biblioteca contiene las descripciones de todos los elementos que componen el circuito. Posteriormente, la simulación o síntesis del circuito se realizará sobre esta biblioteca de diseño. A esta biblioteca, donde se guardan los elementos de la descripción después de la compilación, se le llama `work`.

Los elementos que componen una biblioteca es lo que se llaman *unidades*. Ya se han visto tres unidades hasta ahora, la entidad, la arquitectura, y la configuración, pero hay dos más que son los paquetes y los cuerpos de los paquetes.

A las unidades de tipo declarativo, que son la entidad, paquete y configuración, se las conoce como *unidades primarias*. Al resto de unidades, que son de tipo ejecutivo, es decir, a las arquitecturas y cuerpos de los paquetes, se las llama *unidades secundarias*. Una unidad secundaria tiene siempre asociada una unidad primaria, por lo que las primarias deben ser evaluadas en primer lugar, ésta es la razón por la que también se suelen poner al principio de una descripción las unidades primarias, a excepción de las configuraciones que hacen referencia también a las arquitecturas.

En la figura 8.1 se muestra de forma esquemática cómo se ordenan las unidades dentro de la biblioteca.

Al realizar la compilación se van analizando unas tras otras las unidades que vayan apareciendo en el fichero de diseño. Como unas unidades dependen de otras, es necesario respetar un cierto orden. Así, lo primero será analizar los paquetes y sus cuerpos,

seguirá el análisis de las entidades y sus correspondientes arquitecturas, y por último, se analizarán las configuraciones. Este orden es lógico, ya que las configuraciones dependen de las arquitecturas y entidades, las arquitecturas dependen de las entidades y lo definido en los paquetes, las entidades dependen de los paquetes, y los paquetes pueden depender de otros paquetes.

La forma que tiene el *fichero de diseño* es siempre la misma, ya que se trata de un fichero texto con los comandos de VHDL, sin embargo, la forma que puede tomar la biblioteca, con todas las unidades después de la compilación, puede ser muy diversa; dependiendo de la herramienta de compilación utilizada y del sistema operativo se puede organizar de una forma u otra. Esto quiere decir que no existe un mecanismo estándar en VHDL para la creación de bibliotecas, siendo ésta una tarea de la herramienta que se esté utilizando.

Sí que existe, sin embargo, un mecanismo para incorporar elementos de otras bibliotecas y hacerlos visibles al diseño que se está llevando a cabo. Este mecanismo consiste en la inclusión, delante de la entidad, de una cláusula **LIBRARY**. A continuación de esta sentencia se pone la lista de bibliotecas que se desea que sean visibles. Estas bibliotecas se referencian mediante un nombre lógico, de manera que la herramienta traduce este nombre lógico al correspondiente sistema de almacenamiento que tenga; puede ocurrir que la biblioteca sea un fichero, o un directorio, o que todas las bibliotecas estén en un único fichero, etc., esto es algo que depende de la herramienta en particular.

Junto a la cláusula de biblioteca puede haber también cláusulas que permitan hacer visibles los elementos internos a los paquetes. La sentencia que permite hacer esto se llama **USE**. Seguido del **USE** se pone el paquete y a continuación la unidad o elemento que se quiere referenciar dentro del paquete precedido por un punto. Si se quieren referenciar todos los elementos de un paquete se puede utilizar la palabra **ALL**. Ejemplos:

```
LIBRARY componentes;          -- Hace visible la biblioteca componentes.
USE componentes.logic.and2;   -- Hace visible la puerta "and2" del
                              --  paquete "logic" al resto del programa.
USE componentes.arith.ALL;    -- Hace visibles todos los elementos del
                              --  paquete "arith".
```

En cualquier sistema basado en VHDL siempre existen dos bibliotecas que no necesitan ser invocadas puesto que son cargadas por defecto. Una de estas bibliotecas es `work`, es decir, la que contiene las unidades del diseño que se está compilando. La otra biblioteca es la `std` que contiene dos paquetes, el `standard` y el `textio`. El paquete `standard` dentro de esta biblioteca contiene todas las definiciones de tipos y constantes vistas hasta ahora, como por ejemplo los tipos `bit` y `bit_vector`. El paquete `textio` contiene tipos y funciones para el acceso a ficheros de texto.

Junto a estas bibliotecas suele venir en las herramientas de simulación y síntesis otra biblioteca que se usa tanto que prácticamente también es estándar. Esta biblioteca se llama `ieee` y contiene algunos tipos y funciones que completan los que vienen incorporados por defecto. Dentro de esta biblioteca hay inicialmente un paquete, el `std_logic_1164`, que contiene la definición de tipos y funciones para trabajar con un sistema de nueve niveles lógicos que incluyen los de tipo bit con sus fuerzas correspondientes, así como los de desconocido, alta impedancia, etc. El nombre de este tipo es el `std_ulogic`, y en el mismo paquete viene otro tipo, el `std_logic`, que es exactamente como el anterior sólo que éste tiene asociada una función de resolución (ver la sección 9.1). Junto a este paquete existe otro que no es más que una extensión del anterior y se llama `std_logic_1164_ext`. Este paquete es como el anterior pero incorpora alguna función de resolución más, así como operaciones aritméticas y relacionales.

Con el fin de clarificar cómo vienen definidos estos tipos, se presenta a continuación el comienzo de la parte declarativa del paquete `std_logic_1164` de la biblioteca `ieee`, donde se pueden ver los diferentes niveles lógicos disponibles:

```
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 funct
---------------------------------------------------------------------
   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;
```

Posteriormente a estas definiciones viene la sobrecarga de operadores, declaración de funciones de conversión, etc.

Como la biblioteca `ieee` es la empleada mayoritariamente en la industria, a partir de este momento se utilizarán indistintamente estos tipos y los definidos en la biblioteca `std`. En todos los ejemplos que siguen se supondrá por tanto que la biblioteca `ieee` ha sido cargada y que los paquetes `std_logic_1164` y `std_logic_1164_ext` son visibles.

8.2.1 Paquetes: PACKAGE y PACKAGE BODY

Un paquete es una colección de declaraciones de tipo, constantes, subprogramas, etc., normalmente con la intención de implementar algún servicio en particular o aislar un grupo de elementos relacionados. De esta manera se pueden hacer visibles las interfaces de algunos elementos como funciones o procedimientos quedando ocultas las descripciones de estos elementos.

Los paquetes están separados en dos partes, una es la parte de declaraciones y la otra es la de cuerpo. La parte de cuerpo, donde se encuentran por ejemplo algunas definiciones de funciones y procedimientos, puede ser omitida si no hay ninguno de estos elementos. A continuación se muestra la declaración de paquete y de cuerpo del paquete:

```
-- Declaración de paquete
PACKAGE nombre IS
   declaraciones
END [PACKAGE] [nombre];
```

```
-- Declaración del cuerpo
PACKAGE BODY nombre IS
   declaraciones
   subprogramas, etc.
END [PACKAGE BODY] [nombre];
```

El nombre dado al PACKAGE y al PACKAGE BODY debe coincidir para que se entienda que ambos forman un conjunto. A continuación se muestra un ejemplo de este tipo de declaraciones, donde al principio se declaran unos tipos y cabeceras de función, y a continuación se definen las funciones en un PACKAGE BODY:

```
PACKAGE tipos_mios IS
   SUBTYPE direcc IS bit_vector(23 DOWNTO 1);
   SUBTYPE dato IS bit_vector(15 DOWNTO 0);
   CONSTANT inicio: direcc;  -- Habra que definirlo en el BODY
   FUNCTION datotoint(valor: dato) RETURN integer;
   FUNCTION inttodato(valor: integer) RETURN dato;
END tipos_mios;
```

Como la constante y las funciones no han sido definidas se debe hacer esto en el cuerpo del paquete:

```
PACKAGE BODY tipos_mios IS
   CONSTANT inicio: direcc:=X"FFFF00";
   FUNCTION datotoint(valor: dato) RETURN integer IS
      el cuerpo de la funcion datotoint
   END datotoint;
   FUNCTION inttodato(valor: integer) RETURN dato IS
      el cuerpo de la funcion inttodat
   END inttodato;
END tipos_mios;
```

Una vez se han declarado los paquetes de esta manera, los elementos de los que están compuestos pueden ser referenciados con el nombre del paquete y del elemento separados por un punto. Por ejemplo, para hacer visibles la constante o los tipos del ejemplo anterior se haría simplemente:

```
VARIABLE pc: work.tipos_mios.direcc;
pila:=work.tipos_mios.inicio+X"FF";
desp:=work.tipos_mios.datotoint(registro);
```

Aunque ésta es una forma posible de utilizar los elementos de un paquete, no es la forma usual de referenciarlos. Lo normal es hacer *visible* el paquete de manera que se puedan referenciar algunos o todos sus elementos sin necesidad del punto. Los elementos de un paquete se pueden hacer visibles para el fichero de diseño actual mediante el comando USE tal y como se había mostrado anteriormente. De esta manera, el ejemplo anterior se puede simplificar empleando un USE en la cabecera del programa:

```
USE work.tipos_mios.ALL
VARIABLE pc: direcc;
pila:=inicio+X"FF";
desp:=datotoint(registro);
```

CAPÍTULO 9

CONCEPTOS AVANZADOS EN VHDL

Hasta este momento se ha intentado ofrecer una visión general de lo que es el VHDL. Las explicaciones vistas cubren buena parte de las posibilidades del lenguaje, pero todavía quedan algunas cosas que resultan interesantes en determinadas ocasiones. No es que en esta sección se vayan a cubrir el resto de características del lenguaje, pero sí que se expondrán algunos conceptos importantes. Como parte del lenguaje en sí se explicará la utilización de buses y funciones de resolución, así como la utilización de punteros y ficheros en VHDL.

9.1 BUSES Y RESOLUCIÓN DE SEÑALES

Normalmente un bus es un conjunto de hilos que se agrupan juntos por poseer un significado común, como por ejemplo un bus de datos, de direcciones, etc. Se ha visto que estos buses podían definirse de forma sencilla mediante la definición de matrices, o vectores en este caso. Por lo tanto, no es sobre la creación de buses sobre lo que trata esta sección.

Muchas veces, en un bus real, y se puede dar también en una señal única, ocurre la circunstancia de que hay varios elementos conectados a la misma línea. Se puede pensar por ejemplo en el bus de datos de un microprocesador sencillo. En ese bus de datos pueden escribir tanto el procesador, como la memoria, como elementos periféricos, etc. Resulta evidente que si varios dispositivos escriben al mismo tiempo sobre el bus, aparte de que no se sabe qué valor lógico resulta, se pueden destruir, o como poco, calentar mucho los circuitos de salida de los dispositivos que escriben sobre el bus. Cuando varios dispositivos escriben al mismo tiempo sobre una misma señal, se le llama *contención*, o lo que es lo mismo, que hay una lucha o contienda por el bus.

Para evitar que las contenciones en el bus deterioren los dispositivos conectados a él, éstos escriben en el bus a través de *buffers* que, dependiendo del tipo que sean (triestado, colector abierto, etc.), administrarán la contención de una forma u otra. En el caso de procesadores y memorias se suele utilizar el *buffer triestado*, que tiene una señal de habilitación de manera que cuando esta señal está activa el buffer escribe un valor fuerte sobre el bus, y cuando está inactiva también escribe pero un valor débil. Este último suele ser lo que se llama *alta impedancia* (en el tipo `std_logic` es el valor `'Z'`) que tiene una fuerza en el bus muy débil, de manera que si algún otro dispositivo escribe un `'1'` o un `'0'`, éste será el valor que se tome, ya que el `'1'` o el `'0'` son valores más fuertes que el `'Z'`. La precaución, a la hora de realizar el circuito, es cuidar de que sólo haya un dispositivo a un tiempo que escribe el valor fuerte, mientras que el resto de dispositivos deben estar en alta impedancia. Otros buses, normalmente los dedicados al arbitraje, suelen utilizar salidas en colector abierto de manera que varios dispositivos pueden escribir a la vez, y no pasa nada porque uno escriba un `'0'` y otro un `'1'` al mismo tiempo, ya que el cero siempre gana. Es una forma, además, de realizar lo que se llama una AND cableada, ya que simplemente conectando juntas varias señales en colector abierto el resultado en el bus será una AND lógica sobre todas las señales que escriben en el bus.

Hasta ahora, en todos los ejemplos que se han visto, sólo se asignaba un valor a una señal. De hecho, es imposible en VHDL asignar dos veces una misma señal (de tipo no resuelto) en dos instrucciones concurrentes. Dicho de otra manera, sólo una fuente puede escribir sobre una señal (en el caso de poner retrasos, en una multiasignación, son varias fuentes consecutivas, pero sólo hay una que escribe sobre la señal). Como corolario de este principio que se acaba de dar se puede decir que **no se puede asignar dos veces un valor a una señal en procesos diferentes**, y conviene insistir en este punto, ya que es un error muy común a la hora de describir circuitos usando VHDL. Sin embargo, existe una manera de que varias fuentes escriban sobre una misma línea, tal y como se da en los buses, y eso es lo que se explica a continuación.

Para solucionar el problema de los buses en VHDL existen los *tipos resueltos*, que van a ser un tipo de señales que tienen asociada una función de resolución, que es precisamente la que resuelve el conflicto que se da cuando varias fuentes escriben sobre una misma señal y decide qué valor asignarle.

Un tipo resuelto se define con la declaración de subtipo añadiendo el nombre de la función. Para aclarar esto se muestra una aplicación evidente. Supongamos que tenemos una lógica con tres niveles que son el uno `'1'`, el cero `'0'`, y el de alta impedancia `'Z'`. Se va a suponer que se tienen señales sobre las que se pueden dar accesos múltiples a un tiempo (por ejemplo, un bus de datos). Con estas consideraciones se muestra cómo se describiría un bus de este tipo para que soportara múltiples fuentes en una misma señal.

Para empezar habría que definir estos tipos:

```
TYPE logico IS ('0','1','Z');
TYPE vector_logico IS ARRAY (integer range <>) OF logico;
```

Para continuar hay que definir una función de resolución que calcule el valor de la fuente en función de todas las asignaciones que se están haciendo. En el ejemplo, cualquier señal que contenga 'Z' no interviene, y para el resto, será cero si al menos hay uno que es cero (AND cableada o *wire AND logic*). Con estas consideraciones, la función de resolución será:

```
FUNCTION resolver(senales: IN vector_logico) RETURN logico IS
VARIABLE index: integer;
VARIABLE escribe: boolean:=false;
BEGIN
   FOR index IN senales'range LOOP
      IF senales(index)='0' THEN RETURN '0';
      END IF;
      IF senales(index)='1' THEN escribe:=TRUE;
      END IF;
   END LOOP;
   IF escribe RETURN.'1';
   END IF;
   RETURN 'Z';
END resolver;
```

A continuación se debe declarar un subtipo para que todas las señales que se declaren con ese subtipo puedan ser usadas como descripción de un bus:

```
SUBTYPE logico_resuelto IS resolver logico;
```

La forma en que se usa este nuevo tipo resuelto es exactamente igual que el no resuelto. Es decir, la declaración de señales se hace de la misma manera que siempre. También se admite, en la declaración de la señal, la función de resolución que se desea utilizar, así las siguientes instrucciones son equivalentes:

```
SIGNAL linea: logico_resuelto;
SIGNAL linea: resolver logico;
```

Como un ejemplo de la utilización de los tipos resueltos se aplica el tipo anterior (`logico_resuelto`) para solucionar el problema de la contención en el bus. Supongamos que se tiene un bus de datos compartido por una memoria y un microprocesador

que se llama `datos`, que el bus de datos del micro se llama `micro_datos` y el de la memoria `mem_datos` y que ambos son las entradas internas a sendos buffers triestados que tienen como señales de habilitación `micro_ena` y `mem_ena`. Por último se ha supuesto que es el procesador el que lo controla todo a partir de su señal de `read`. La parte de código referida al bus quedaría:

```
-- triestado del microprocesador:
datos<=micro_datos WHEN micro_ena='1' ELSE (OTHERS => 'Z');

-- triestado de la memoria:
datos<=memo_datos WHEN memo_ena='1' ELSE (OTHERS => 'Z');

-- Control con la señal de read:
micro_ena<=NOT read;
memo_ena<=read;
```

Cuando el microprocesador lee de la memoria, su buffer está deshabilitado, ya que es la memoria la que escribe por el bus. Cuando el micro escribe ocurre al revés. De esta manera nunca hay dos señales que escriben valores *fuertes* sobre el bus, ya que siempre una de las dos estará en alta impedancia. Las instrucciones concurrentes anteriores, que se podían haber puesto como procesos, no serían posibles si la señal `datos` no fuera de tipo resuelto, ya que se tendrían dos instrucciones concurrentes que escriben al mismo tiempo sobre la misma señal.

Aunque ya se vio en las asignaciones a matrices (capítulo 4), resulta interesante comentar que la cláusula `(OTHERS => 'Z')` es una agregado o conjunto (*aggregate*) que significa que se le asigna `'Z'` a todos los bits que tenga la señal `datos`, si se supone un bus de 8 bits, la cláusula anterior sería equivalente a poner `"ZZZZZZZZ"`, con la ventaja de que la misma descripción sirve para cualquier tamaño de bus.

En el paquete `standard`, que se encuentra en la biblioteca `std`, se definían los tipos `bit` y `bit_vector` como no resueltos, es decir, no tienen función de resolución y, por lo tanto, no se les puede usar en buses de datos ni señales donde varios procesos escriban a un tiempo. Para evitar esto están los tipos `std_logic` y `std_logic_vector`, que además de poseer un número más realista de niveles lógicos (ver el capítulo 8), poseen funciones de resolución. Estos tipos resueltos venían definidos en el paquete `std_logic_1164` que se encuentra en la biblioteca `ieee`, que incluye además los tipos `std_ulogic` y `std_ulogic_vector` que son los tipos no resueltos equivalentes.

9.2 PUNTEROS EN VHDL

Dentro de la mayoría de lenguajes de programación existe la posibilidad de crear, reservando espacio de memoria, variables de forma dinámica. Cuando se conoce previamente el *tamaño* del problema, se pueden definir las variables de los tipos adecuados que permitan tratarlo. Por el contrario hay sistemas que por su naturaleza pueden tener un tamaño variable no definido *a priori*. Ejemplos de estos sistemas son las típicas estructuras de datos de listas, colas FIFO (*First In First Out*) y colas LIFO (*Last In First Out*), también llamadas pilas. Siempre es posible definir un tamaño máximo de estas estructuras, ajustando los algoritmos de almacenamiento y lectura de datos para no superar este límite, y reservando espacio que puede llevar a una mala utilización de los recursos del sistema. Es mejor definir estos sistemas de manera que se utilice sólo la capacidad de memoria necesaria en cada instante, sin reservas previas y sin problemas de desbordamiento.

Los lenguajes de programación disponen de la posibilidad de reservar espacio de memoria de manera dinámica mediante **punteros**. Un puntero no es más que una dirección de memoria que *apunta* a una variable de nueva creación. Sólo hay que organizar de manera adecuada, mediante estructuras de tipo registro, las variables a almacenar y los punteros o enlaces a éstas.

En VHDL el tipo de datos que permite la reserva de memoria de forma dinámica se denomina ACCESS y se define de la siguiente manera:

```
TYPE tipo_puntero IS ACCESS tipo_elemento;
VARIABLE p : tipo_puntero;
```

La variable `p` es un puntero que apunta a variables del tipo `tipo_elemento`, que puede ser cualquier tipo de datos escalar o compuesto. Por su naturaleza los punteros sólo pueden ser variables (nunca señales), por tanto, su utilización se enmarca siempre en la ejecución serie dentro de un PROCESS.

La creación de un puntero se realiza mediante la palabra clave NEW. Un ejemplo sería:

```
TYPE apunta_enteros IS ACCESS integer RANGE 0 TO 255;
VARIABLE p : apunta_enteros;
...
p:= NEW integer RANGE 0 TO 255;
p.ALL:= 45;
```

En este caso el puntero `p` apuntará a un entero dentro del rango de 0 a 255. Para definir el contenido de la dirección apuntada por `p` se utiliza la palabra clave **ALL**, de manera que en el caso anterior `p` apunta a un entero cuyo valor es 45.

Si se crea un puntero pero no se define su valor éste toma el valor por defecto que es **NULL**. Cuando ya no es necesaria la variable apuntada y se desea liberar la memoria utilizada, se utilizará el procedimiento implícito `deallocate`. De esta manera, para liberar la memoria utilizada por la nueva variable del ejemplo anterior, se escribiría:

```
Deallocate(p);
```

El puntero desaparece y, por tanto, ya no se tiene acceso a la variable por él apuntada.

La utilización más interesante del tipo **ACCESS** y los punteros es, tal como se ha citado anteriormente, la construcción de estructuras de datos complejas. Como ejemplo se va a abordar la construcción de una pila, descrita a continuación.

Ejemplo 9.1 *Realizar el modelo de una pila o cola LIFO de tamaño ilimitado que almacene enteros (rango de 0 a 255). La pila tendrá un puerto de entrada donde se pondrán los valores a apilar, y un puerto de salida por donde saldrán los valores de la pila. Para apilar un valor entero éste se capturará en el flaco de subida de la señal* `apilar`. *Cuando se detecte un flanco de subida en la señal* `desapilar`, *se desapilará el valor entero que esté más alto en la pila, presentándolo en el puerto de salida. En el caso de que se activen ambas señales a la vez (apilar y desapilar) se apilará. Habrá también una señal de réset que eliminará toda la pila.*

Este ejemplo permite mostrar de forma completa todos los conceptos asociados a los punteros en VHDL. El código que resolvería el problema propuesto se muestra seguidamente:

```
ENTITY pila IS PORT(
  apilar, desapilar, reset : IN bit;
  entrada : IN integer RANGE 0 TO 255;
  salida  : OUT integer RANGE 0 TO 255);
END pila;

ARCHITECTURE comportamiento OF pila IS
BEGIN
  PROCESS (apilar, desapilar, reset)
   TYPE tipo_elemento;            -- Declaración de tipos incompleta
```

```
  TYPE tipo_puntero IS ACCESS tipo_elemento;
  TYPE tipo_elemento IS RECORD  -- Ahora se completa la definición
       siguiente : tipo_puntero;
       numero : integer RANGE 0 TO 255;
       END RECORD;
  VARIABLE top : tipo_puntero := NULL;
  VARIABLE puntero : tipo_puntero ;
  VARIABLE elemento : tipo_elemento;

 BEGIN
   IF reset='1' THEN salida<=0; -- Se borra la pila completa
      WHILE top /= NULL LOOP
        puntero:=top.siguiente;
        deallocate(top);
        top:=puntero;
      END LOOP;
   ELSIF apilar='1' AND apilar'event THEN
     puntero:=NEW tipo_elemento;
     puntero.numero:=entrada;
     puntero.siguiente:=top;
     top:=puntero;
   ELSIF desapilar='1' AND desapilar'event THEN
     IF top /= NULL THEN          -- Se comprueba que no está vacía
           salida<=top.numero;
           top:=top.siguiente;
       ELSE ASSERT false REPORT "La pila está vacía"
           SEVERITY warning;
     END IF;
   END IF;
 END PROCESS;
END;
```

En la definición de tipos del listado anterior aparece una declaración de tipos en la que sólo se especifica el nombre del tipo. Esto es lo que se conoce como **declaración de tipos incompleta**.

El uso de tipos incompleto sólo se permite conjuntamente con la declaración de un tipo ACCESS posterior que apunte a este tipo incompleto. Después ya se realiza la definición completa del tipo que inicialmente era incompleto. Esto se permite porque al construir estructuras de datos del tipo RECORD, que contienen a su vez un puntero que apunta a esa misma estructura, se produce una especie de definición cruzada. Si no se hubiera puesto la línea inicial `TYPE tipo_elemento`, el compilador de VHDL habría generado un error al no existir ese tipo en la definición del puntero: `TYPE tipo_puntero IS ACCESS tipo_elemento`.

El resto del código simplemente reproduce la estructura habitual de una pila diseñada con punteros de un lenguaje de alto nivel. El truco consiste en crear una estructura del tipo registro que contenga el dato a apilar y el puntero al siguiente dato. De esta manera se accederá al puntero contenido en el registro con `puntero.siguiente` y al dato con `puntero.numero`.

El proceso de apilar consiste solamente en crear un elemento nuevo en la pila con la asignación `puntero:= NEW tipo_elemento`, capturar el número a apilar mediante `puntero.numero:=entrada`, y mover el puntero `top` para que apunte al nuevo elemento más alto en la pila con `top:=puntero`.

Análogamente se realiza el movimiento de punteros para desapilar un elemento. Lo único que hay que tener en cuenta es que la pila no esté vacía o, equivalentemente, que el puntero al elemento más alto de la pila (`top`) no apunte a NULL. En el caso de que la pila esté vacía se utiliza una sentencia ASSERT para notificar el error, tal como se explica en la sección 10.3.

Cuando se activa la señal de `reset` simplemente se realiza un bucle que recorre la pila *borrando* todos los elementos con una sentencia `deallocate`.

Este ejemplo muestra la potencia del tipo ACCESS combinado con el tipo incompleto y el tipo registro. De manera similar se pueden realizar estructuras de datos como colas FIFO o listas, sólo con redefinir los procedimientos de inserción y borrado.

9.3 FICHEROS

La sección 10.5.2 se centra en la generación de una banco de pruebas, utilizando para ello ficheros externos al código VHDL. En esta sección se va a describir el modo de acceso a ficheros que permite VHDL.

Puede ser interesante, o bien leer información de ficheros, o bien almacenarla en ficheros durante una simulación VHDL. De hecho, la utilización de ficheros externos como entrada y salida en VHDL se restringe en la práctica a su uso en simulación de bancos de pruebas, no siendo apropiado su uso en síntesis.

Un fichero puede almacenar cualquier tipo de datos VHDL, siendo la declaración de ficheros diferente en VHDL'87 y VHDL'93, tal como se muestra a continuación:

```
TYPE nombre_tipo IS FILE OF integer;                    -- Común
FILE nombre_logico :
            nombre_tipo IS [modo] "fichero.ext";        -- Sólo VHDL'87
FILE nombre_logico :
            nombre_tipo [OPEN modo] IS "fichero.ext"; -- Sólo VHDL'93
```

En la primera línea se define el tipo `nombre_tipo` como un fichero de enteros de manera común para ambas versiones de VHDL. Es después, en la declaración de los fi-

cheros, cuando aparecen las divergencias. El nombre lógico es el visible para el código VHDL. Éste está vinculado al fichero cuyo nombre en el árbol de directorios se entrecomilla.

La indicación entre corchetes `[]` indica que esa parte de código es opcional. El `modo` en VHDL'87 puede ser **IN** (por defecto) si se va a leer de él, u **OUT** para escribir en él. En VHDL'93 puede ser `write_mode`, `read_mode` (por defecto) o `append_mode`, indicando cada uno su significado obvio.

Un fichero lógico puede tener sólo un modo, permitiéndose que múltiples procesos concurrentes accedan a un fichero lógico. Un fichero físico también puede ser escrito y leído desde procesos concurrentes que accedan a ficheros lógicos mapeados en él, tal como permita el sistema operativo.

La definición de tipos de ficheros lleva asociada la definición implícita de unos procedimientos. Así a partir de la definición de ficheros anterior quedarían automáticamente definidos:

```
PROCEDURE read(VARIABLE nombre: INOUT nombre_tipo; valor: OUT integer);
PROCEDURE write(nombre: INOUT nombre_tipo; valor: IN integer);
FUNCTION endfile(VARIABLE nombre: IN nombre_tipo; RETURN boolean);
```

Para acceder a un fichero se podrían utilizar los procedimientos `read` y `write` automáticamente definidos. Un ejemplo sería:

```
TYPE tipo_fichero IS FILE OF integer;
FILE f1 : tipo_fichero OPEN modo IS "fichero.ext";
VARIABLE n : integer;
...
read(f1,n); -- La variable n contiene el primer valor del fichero
            -- lógico f1
```

Se puede trabajar sin problemas utilizando ficheros de tipos permitidos por VHDL, lo cual puede resultar útil pero bastante complicado. Efectivamente, si se quiere realizar en un banco de pruebas los estímulos de entrada para una cierta simulación, se puede desear tener ficheros lo más *legibles* posible. Es decir, lo que se va a almacenar en el fichero no van a ser los números enteros, sino su equivalente texto. Por tanto, un fichero introducido con un procesador de texto con el aspecto:

```
23 15  3  0  0  7
 0  0 12  0 15  8
```

se trata en realidad de un fichero del tipo `text`, definido en el paquete `textio` de la biblioteca estándar `std`. Si se desean leer estos valores deberán definirse ficheros del tipo `text` y posteriormente realizar la conversión de tipos para el banco de pruebas.

Afortunadamente en el paquete `textio` de la biblioteca estándar (`std`) se encuentran unos procedimientos y tipos que ayudan a realizar esta tarea. A continuación se muestran algunos de los más interesantes, que se usarán en el ejemplo 9.2.

```
PACKAGE textio IS

TYPE line IS ACCESS string;
TYPE text IS FILE OF string;

FILE input: text OPEN read_mode IS "std_input";
FILE output: text OPEN write_mode IS "std_output";

PROCEDURE readline (VARIABLE f: IN text; l: INOUT line);
PROCEDURE read (l: INOUT line; valor: OUT bit; ok: OUT boolean);
PROCEDURE read (l: INOUT line; valor: OUT bit);
PROCEDURE read (l: INOUT line; valor: OUT bit_vector; ok: OUT boolean);
PROCEDURE read (l: INOUT line; valor: OUT bit_vector);
PROCEDURE read (l: INOUT line; valor: OUT character; ok: OUT boolean);
PROCEDURE read (l: INOUT line; valor: OUT character);
-- Lo mismo para integer, real, string y time
-- Análogamente se definen los procedimientos writeline y write
```

En el paquete `textio` se define el tipo `line` como un puntero al tipo `string`, y el tipo `text` como un fichero de `string`. Con los procedimientos `readline` y `writeline` se leen y escriben líneas de los ficheros, y con los procedimientos `read` y `write` se leen y escriben datos de las líneas. Los tipos para los que están definidos las rutinas son el `bit`, `bit_vector`, `character`, `integer`, `real`, `string` y `time`.

La sobrecarga de operadores permite definir el mismo procedimiento para cada tipo de datos. Incluso para el mismo tipo se han definido dos procedimientos de lectura, uno que lee de una línea devolviendo solamente el dato, y otro que devuelve además un valor booleano indicando si lo que se ha leído corresponde a ese tipo.

Lo habitual a la vista del paquete `textio` será acceder a ficheros legibles con los estímulos escritos como texto. Se leerá una línea con el procedimiento `readline`, leyendo de esta línea con el procedimiento `read`, que devolverá el dato del tipo adecuado (dentro de los existentes). Después de aplicar los estímulos se podrá escribir en una línea con el procedimiento `write`, y una vez completada la línea, se podrá escribir en el fichero de texto con el procedimiento `writeline`.

Para ver de manera práctica todos estos conceptos, se propone al lector el siguiente ejercicio.

Ejemplo 9.2 *Escribir un banco de pruebas, para el ejemplo 9.1, que acceda a un fichero de tipo texto con los vectores de test ordenados en 12 líneas de forma consecutiva. El programa creará otro fichero con dos columnas: los valores del resultado esperado de la simulación (leídos del fichero de estímulos) y el resultado de la simulación.*

El código VHDL que implementaría la solución al ejemplo 9.2 podría ser el siguiente, aunque dependerá del formato exacto de la entrada y la salida.

```
ENTITY test2_pila IS
END test2_pila;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE std.textio.ALL;
USE work.paquete_pila.all;

USE work.paquete_pila.ALL;
ARCHITECTURE test OF test2_pila IS
 SIGNAL apilar, desapilar, reset : bit;
 SIGNAL entrada, salida : integer RANGE 0 TO 255;
 BEGIN
  pila1 : pila PORT MAP(apilar=>apilar, desapilar=>desapilar,
              reset=>reset, entrada=>entrada, salida=>salida);

PROCESS
  TYPE fichero IS FILE OF integer;
  VARIABLE errores: boolean:= false;
  VARIABLE linea1, linea2 : line;
  VARIABLE v_apilar, v_desapilar, v_reset : bit;
  VARIABLE v_entrada, v_salida : integer;
  FILE f_entrada : text OPEN read_mode IS "in_pila";
  FILE f_salida : text OPEN write_mode IS "out_pila";
 BEGIN
 FOR i IN 0 TO 11 LOOP
   readline(f_entrada, linea1);
   read(linea1, v_apilar); read(linea1, v_desapilar);
   read(linea1, v_reset);
   read(linea1, v_entrada); read(linea1, v_salida);
   apilar<=v_apilar; desapilar<=v_desapilar; reset<=v_reset;
   entrada<=v_entrada;
   WAIT FOR 20 ns;
   write(linea2, v_salida); write(linea2, ' '); write(linea2, salida);
   writeline(f_salida, linea2);
 END LOOP;
 WAIT;
END PROCESS;
END;
```

La arquitectura del código es clara. Lo primero que se hace es definir los ficheros lógicos `f_entrada` y `f_salida` a partir de los ficheros que van a servir de entrada y salida respectivamente para la simulación. Una vez se lee una línea del fichero de entrada, se obtiene de ahí el valor para cada entrada de la pila, y se obtiene también el valor

esperado de la simulación. Se aplican las entradas y mediante las sentencias `WAIT FOR 20 ns;` se espera la estabilización del circuito. Posteriormente se escriben el valor esperado de la simulación y el valor real obtenido en el fichero de salida.

En este caso se ha supuesto que los valores en el fichero de entrada son consecutivos (sin espacios). En el caso de que el formato fuera distinto debería tenerse esto en cuenta. La escritura se realiza en dos columnas separadas por un espacio.

En este caso han sido necesarias todas las funciones de conversión y lectura de texto, porque el fichero de estímulos era de ese tipo, a pesar de que almacenaba caracteres que iban a ser interpretados como `bit` o `integer`.

CAPÍTULO 10

VHDL PARA SIMULACIÓN

El lenguaje VHDL sirve tanto para síntesis automática de circuitos como para descripción de modelos para simulación. El estilo de descripción en uno y otro caso es diferente. Por un lado la simulación de un programa en VHDL no tiene demasiadas restricciones, lo único que se necesita es un intérprete de las instrucciones VHDL. La síntesis, en cambio, tiene muchas más restricciones puesto que al final se debe obtener un circuito real que realice la misma función que el circuito descrito en el código. Si el nivel de abstracción es muy alto, la síntesis será muy difícil, llegando a ser imposible sintetizar un circuito a partir de la especificación.

En simulación, aparte de que el nivel de abstracción importa poco, habrá una serie de elementos que sólo tienen significado en un entorno de simulación. Estos elementos son retrasos, señalización de errores, descripciones de bancos de pruebas, etc. En síntesis, algunos de estos elementos, especialmente los retrasos, no tienen ningún sentido y deben ser evitados.

10.1 ASIGNACIÓN CON RETRASOS

El elemento más importante en simulación, el retraso de las líneas, no se ha descrito hasta ahora. En todo circuito digital, aparte de la funcionalidad que se pueda implementar, existe siempre un retraso entre que se producen los cambios en los estímulos de entrada y la salida cambia.

10.1.1 Sintaxis completa de la asignación

La sintaxis de una asignación es más compleja de lo que se ha visto. Esto es debido a que la asignación debe incluir la posibilidad de especificar un retraso. La sintaxis completa de una asignación se describe a continuación:

```
[id_asig:]
senal <= [ TRANSPORT | [ REJECT tiempo ] INERTIAL ]
         forma_onda {,forma_onda} | UNAFFECTED ;
```

Como `forma_onda` es algo que se puede repetir, y tiene también sus propias palabras clave, es mejor ponerlo aparte. La `forma_onda` es por tanto:

```
valor_expresión [AFTER tiempo] | NULL [AFTER tiempo]
```

Las palabras clave `NULL` y `UNAFFECTED` ya se explicaron en otros capítulos; el resto, que son las que especifican el comportamiento temporal de la asignación, se explican a continuación en este mismo capítulo.

10.1.2 Especificación de retrasos

Para poder poner en práctica el concepto de retraso, los lenguajes de descripción de modelos suelen utilizar lo que se llaman *fuentes*[1]. El concepto de *fuente* es algo que ya apareció cuando se explicaron las asignaciones a señales y las diferencias entre señal y variable en la ejecución serie.

La forma en que funciona la asignación de una señal es como sigue: cuando se le asigna un valor a una señal, no se le asigna este valor de forma inmediata, sino que se le asigna a su *fuente*. En la sentencia se ha especificado el tiempo en el cual se realizará realmente la asignación, de esta manera la información de la *fuente* pasa a la señal cuando la simulación llega a este tiempo. Hasta ahora nunca se ha especificado ningún tiempo en las asignaciones; en estos casos se considera que el retraso es cero, y, por tanto, la asignación debe producirse de forma inmediata. Esta asignación inmediata indica en realidad que se produzca la asignación al final del presente paso de simulación, siendo un paso de la simulación la ejecución de una instrucción concurrente, o, en el caso de los procesos, el paso que transcurre entre dos sentencias `WAIT`. Aunque transcurre la

[1] En inglés recibe el nombre de *driver*.

simulación en un paso de simulación, no transcurre el tiempo. Más adelante se explica la simulación guiada por eventos donde se aclara el mecanismo de paso de tiempo y de paso de simulación.

El retraso en la asignación a una señal de un determinado valor se especifica en la propia sentencia de asignación. Este retraso va a indicar que se le asigne el valor dado a una señal, pero cuando haya transcurrido el tiempo especificado. Esto quiere decir que la información va a permanecer en la *fuente* hasta que haya pasado el tiempo descrito, por lo que la señal no será actualizada hasta después de transcurrido este tiempo.

Para indicar este retraso en las asignaciones se emplea la palabra AFTER como en el siguiente ejemplo:

```
senal<='0' AFTER 15 ns;
```

Esto quiere decir que cuando hayan pasado 15 ns. desde la asignación, entonces la señal tomará el valor `'0'`, y hasta entonces conservará el que tenga en ese momento. Es decir, si el tiempo actual de la simulación, en el momento de producirse la asignación, es de 400 ns., entonces esto significa que la señal `senal` pasará a valer `'0'` cuando se cumpla un tiempo de simulación de 415 ns.

Gracias al concepto de *evento* es fácil entender que en una asignación se pueden *programar* varios eventos o sucesos que tendrán lugar en el futuro. En el ejemplo siguiente se muestra cómo realizar varias asignaciones a una misma señal en la misma sentencia:

```
senal<='1' AFTER 4 ns, '0' AFTER 20 ns;
```

Cuando se produce la ejecución de esta instrucción, la fuente quedará formada por dos elementos, uno será un `'1'`, y el otro un `'0'`. En principio no hay conflicto puesto que tienen lugar en tiempos diferentes. Con esta asignación la señal tomará el valor `'1'` a los 4 ns. de su ejecución, y 16 ns. después tomará el valor `'0'`.

10.1.3 Retrasos inerciales y transportados

Se ha visto que en la asignación de una señal en realidad lo que se hace es poner en una lista sus valores futuros ordenados por tiempo. En cada asignación que se hace, los

eventos se van añadiendo a la lista ordenándolos según les corresponda, es decir, según el retraso asociado. No obstante, el momento en que se ejecuta la asignación, es decir, el tiempo en el que se introducen nuevos eventos, desempeña un papel importante, ya que permite definir dos formas de introducir los eventos en la lista, cada forma con un significado físico concreto tal y como se muestra a continuación.

Dada una puerta lógica cualquiera, se puede suponer que tendrá un retraso asociado, de manera que la salida cambiará un momento después de que haya cambiado la entrada, por ejemplo, se puede suponer un retraso de 50 ns. Un inversor, por ejemplo, se describiría de la siguiente forma:

```
sal<=NOT ent AFTER 50 ns;
```

Al principio de la simulación, y cada vez que la entrada `ent` cambie, se ejecuta esa instrucción, es decir, se introduce, en la lista de eventos de la salida, la entrada invertida con un retraso de 50 ns. Si la salida cambia en menos de 50 ns. (por ejemplo suponer un pulso en la entrada de 30 ns.), entonces se está introduciendo un nuevo evento en la lista de eventos de esa señal. En principio pueden ocurrir dos cosas, o este evento se añade a la lista que ahora estaría formada por dos eventos, o bien simplemente elimina el evento que hubiera y se pone él. Tal y como se ha especificado la instrucción anterior, el simulador decide eliminar todos los eventos que hubiera en cola ya que se trata de un retraso *inercial*.

La idea de la sustitución de la lista con el nuevo evento se ve con más claridad en la simulación que aparece en la figura 10.1, donde la columna de la izquierda supone un pulso de entrada de 30 ns. y la de la derecha uno de 60 ns.; en ambos casos se representa la salida según se considere un retraso *inercial*, que es el caso que se discute ahora, o *transportado*, que se verá más adelante.

A partir de la figura 10.1 se observa cómo al inicio está todo estabilizado, por tanto, para la entrada cero la salida es uno. Después de 10 ns. la entrada pasa a uno, es el comienzo del pulso. Esto quiere decir que se introduce el evento "la salida se tiene que poner a `'0'` después de 50 ns.", o lo que es lo mismo, "la salida se tiene que poner a `'0'` a los 60 ns. de tiempo absoluto". Este evento se introduce en la lista de *fuentes* de la señal de salida. Si transcurren 30 ns. más (duración del pulso) y la entrada cambia pasando a cero otra vez, se está produciendo un nuevo evento que es "la salida debe ponerse a `'1'` después de 50 ns." Lo que se espera normalmente es que se ponga en la lista, detrás del evento anterior, y conforme vaya llegando el momento de ejecutarse los eventos éstos se ejecuten sin más, pero esto no ocurre así, al menos definiendo los retrasos como se ha hecho hasta ahora.

Lo que ocurre en esta asignación, en la que se ha especificado un retraso sin más, es que este último evento va a sustituir al anterior, por lo que el primer evento desaparece y no se procesa nunca, y la señal de salida nunca pasará a cero a pesar de que la entrada ha sido uno durante 30 ns.

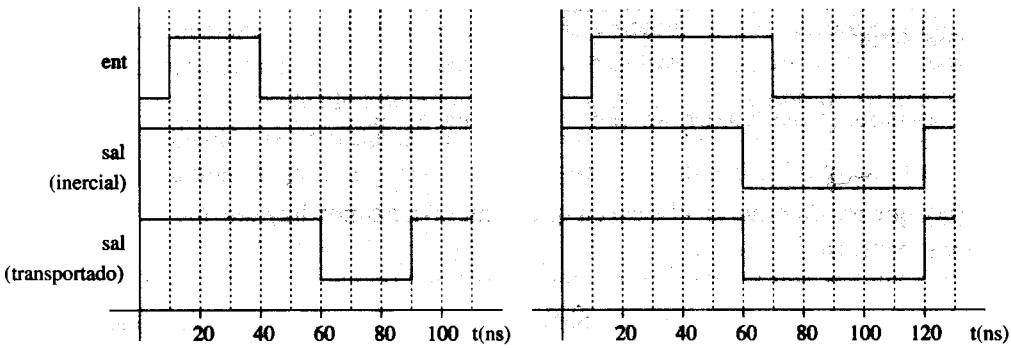


Figura 10.1: Retrasos inerciales y transportados

A esta forma de gestionar los eventos en la lista y generar los retrasos se le llama **retraso inercial**, y es el retraso por defecto en VHDL. En el ejemplo anterior, cualquier pulso de entrada menor de 50 ns. no tendrá ningún efecto sobre la salida puesto que siempre se producirá un evento antes de que se pueda ejecutar el primero.

Si no se especifica nada, se entiende que los retrasos son inerciales. Es posible, sin embargo, declararlo explícitamente mediante la plabra clave `INERTIAL` al principio de la asignación. Es decir, la siguiente línea es equivalente a la del inversor realizada anteriormente:

```
sal<=INERTIAL NOT ent AFTER 50 ns;
```

La justificación de este tipo de retrasos se encuentra en el modelado de puertas lógicas. Esto es debido a que lo que suele pasar en este tipo de dispositivos, y en muchos otros, es que si un pulso de entrada no tiene una duración mínima, nunca llega a modificarse la salida.

Ésta es una forma un poco inexacta de describir el filtrado de picos de una puerta. Es cierto que ciertos pulsos no se transmiten a la salida de las puertas lógicas debido a su corta duración, pero no es cierto que el tamaño máximo de los pulsos que se filtran coincida con el retraso de la puerta. Normalmente, estos pulsos son algo más pequeños. Para realizar un modelado más exacto de las puertas lógicas, se puede emplear un parámetro adicional que sirve para expresar el tamaño máximo de los pulsos que serán filtrados,

es decir, el intervalo de tiempo durante el cual los eventos que se produzcan eliminarán los eventos que haya en cola. Esta especificación se realiza mediante la palabra REJECT.

Suponiendo que la puerta anterior tiene efectivamente un retraso de 50 ns., pero sólo filtra aquellos pulsos que sean menores de 10 ns., entonces el modelo correcto para ese inversor sería:

```
sal<=REJECT 10 ns INERTIAL NOT sal AFTER 50 ns;
```

Hay que resaltar que en el caso en que se incluya REJECT hay que incluir también la palabra INERTIAL.

Hay dispositivos donde el mecanismo de retraso inercial no es adecuado. Si por ejemplo se piensa en una línea de transmisión, no importa la duración del pulso que se pueda introducir o el retraso que pueda tener la línea, a la salida siempre se obtiene el pulso de entrada tal cual (suponiendo claro está que la atenuación de la línea no sea grande). En este caso, al retraso se le llama **retraso transportado**, puesto que en realidad la señal de entrada se transporta a la salida sin modificaciones. En cuanto al tratamiento de la lista de eventos o *fuentes*, lo que se hace es simplemente introducir el evento en la lista, en el lugar que le corresponde según el retraso, procesándose éste cuando le corresponda.

En VHDL hay que indicar explícitamente este tipo de retraso transportado puesto que no es el que se utiliza por defecto. Esta indicación se realiza mediante la palabra clave TRANSPORT. Así, si en el ejemplo anterior se deseara que los pulsos menores que el retraso se propagaran a la salida se debería escribir:

```
sal<=TRANSPORT NOT ent AFTER 50 ns;
```

En el caso de asignaciones múltiples sólo la primera es inercial, mientras que las siguientes se consideran transportadas. Es evidente que si esto no fuera así, sólo la última asignación sería válida y el resto serían ignoradas, lo cual no tiene mucho sentido.

10.1.4 Simulación guiada por eventos

Es interesante advertir que cuando se simula un circuito descrito en VHDL, y siempre que aparezcan especificaciones de retraso, aparece el concepto de *tiempo de simulación*. Este tiempo transcurre gracias a la sucesión de eventos. Cuando una instrucción asigna `'0'` a una señal después de 15 ns., en realidad se está produciendo un evento que

tendrá lugar dentro de 15 ns. Esta información temporal le sirve a un simulador para hacer avanzar el tiempo.

El simulador de VHDL guarda una lista de todos los eventos que se generan y los ordena según el momento en que se tengan que procesar. La simulación tiene lugar al procesarse secuencialmente los diferentes eventos, es decir, después de que el simulador procesa el evento actual, pasa al evento siguiente, este evento siguiente tendrá asociado un tiempo que si no coincide con el actual provocará que el tiempo de simulación se incremente. Cuando se procese ese nuevo evento, se producirán nuevos eventos que se colocarán al final de la lista, y así sucesivamente se simula el circuito y el tiempo va transcurriendo.

Es decir, el avance de tiempo viene dado por el tiempo en el que se tiene que procesar el próximo evento. La diferencia entre el tiempo actual y el tiempo del próximo evento es lo que se conoce como ΔT.

Por otro lado, es posible que se generen eventos, con retraso nulo, en el mismo instante de la simulación y que, por tanto, deben ejecutarse sin que transcurra el tiempo real. Para estos casos, es decir, cuando la simulación corre pero el tiempo no avanza, resulta interesante definir lo que se llama un intervalo *delta*, es decir, un $\Delta\delta$. Normalmente la lista de eventos se actualiza al llegar a una sentencia `WAIT` o al acabar una instrucción concurrente. Con esto, un paso de simulación es lo que ocurre entre dos actualizaciones de listas consecutivas, y es a esto a lo que se le llama $\Delta\delta$. El tiempo puede transcurrir o no dependiendo de si los nuevos eventos se deben ejecutar de forma inmediata o en el futuro.

Una descripción del flujo que puede seguir un simulador en VHDL, guiado por eventos, se muestra en la figura 10.2, donde $\Delta\delta$ es el paso de simulación descrito anteriormente, y ΔT es un paso donde el tiempo corre realmente. Este flujo corresponde a lo que se conoce como *simulación guiada por eventos*.

Esta simulación comienza por los eventos iniciales (estos eventos iniciales vienen dados por los estímulos de entrada al sistema). Los estímulos de entrada, o patrones de test, son la lista inicial de eventos. La simulación comienza por el evento primero de la lista. Al procesarse este evento, lo cual implica la ejecución de una o varias sentencias concurrentes, incluidos los procesos, se van a generar nuevos eventos que se colocarán en la lista según el retraso asociado. De esta manera se van ejecutando y creando nuevos eventos haciendo avanzar el tiempo a cada nuevo evento. La simulación termina cuando no quedan más eventos en la lista, o bien cuando se llega a un tiempo especificado de antemano.

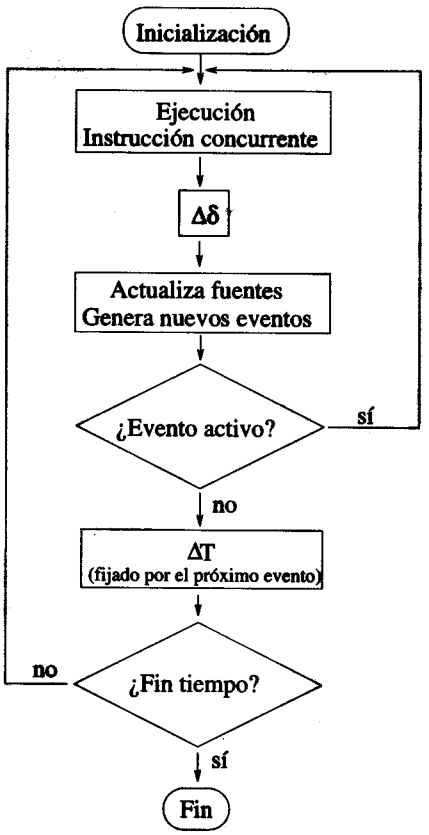


Figura 10.2: Flujo de simulación por eventos en VHDL

10.1.5 Avance de tiempo por incremento fijo

Aunque el mecanismo de avance de tiempo de la sección anterior resulta el más intuitivo y claro, no siempre es el más utilizado.

Los simuladores digitales suelen emplear otro tipo de simulación en la que el tiempo, en vez de incrementarse por el próximo evento que vaya a ocurrir, se incrementa un intervalo de tiempo fijo cada vez. De esta manera, si hay algún evento en ese intervalo de tiempo, se procesa. La figura 10.3 muestra el flujo de simulación con avance de tiempo por incremento fijo.

El inconveniente de este mecanismo de simulación es que por un lado se pierde precisión en cuanto al tiempo (el error lo da el propio paso de simulación), y por otro, la imprecisión que se produce al tratar eventos en el mismo paso de simulación, cuando en

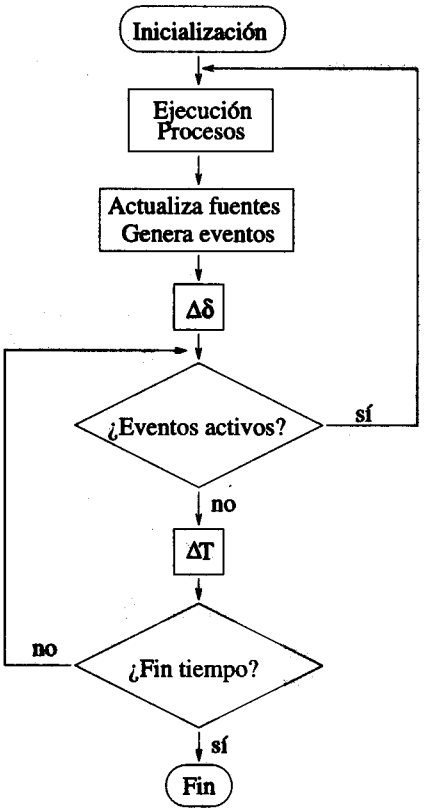


Figura 10.3: Flujo de simulación por incremento fijo

realidad se han podido producir en tiempos diferentes y, por tanto, existe una relación causal entre ellos que no ha sido considerada.

La ventaja que tiene este tipo de mecanismo es que la ejecución del simulador es mucho más veloz, lo que permite la simulación de largos periodos en menor tiempo real. Si se piensa en sistemas de gran tamaño, que además requieren largos tiempos de inicialización, etc., esta técnica resulta la más adecuada. Por esta razón, muchos de los simuladores de sistemas digitales utilizan este mecanismo.

La organización de un simulador que utilice esta técnica es un poco diferente. Para empezar no suele haber una lista con los eventos a procesar, sino que más bien existe una estructura de datos dividida en sectores. Cada uno de estos sectores se corresponde con un intervalo de tiempo, de manera que cada sector contiene los eventos que deben procesarse en ese intervalo. Naturalmente es posible que un sector se encuentre vacío o que tenga más de un evento a evaluar.

Con esta estructura de datos la simulación consistiría en ir recorriendo los sectores uno por uno, y a cada paso de sector el tiempo se incrementa un periodo fijo predeterminado. Si el sector contiene algún evento se procesa, lo que provoca la aparición de nuevos eventos que se colocarán en sectores futuros o en el propio sector si su retraso era nulo. Cuando se hayan procesado todos los eventos de ese sector, se pasa al siguiente.

Como mantener una estructura de este estilo requiere mucha memoria, algunos simuladores suelen tener lo que se conoce como *rueda de tiempos*, donde la lista de sectores es en realidad como un carrusel sin principio ni fin. Los eventos se van colocando de forma relativa en sectores futuros de la rueda; si un evento cae en un sector que implica darle la vuelta a la rueda, entonces se almacena en otra estructura lineal e infinita pero de acceso más lento.

10.1.6 Ejemplo de modelado de un registro

Ejemplo 10.1 *Realizar el modelo de simulación de un registro tipo D, activo por flanco de subida, que tiene un retraso de 10 ns., desde el flanco de subida del reloj hasta que la salida cambia, y un tiempo de establecimiento* (set-up) *de 5 ns. (Para el tiempo de establecimiento se supondrá que si se produce una violación se captura el valor anterior de la señal de entrada y no el que haya en el momento del flanco.)*

Éste es un ejemplo típico donde se especifica el retraso en una señal de salida respecto del reloj, y un retraso en una señal de entrada para realizar el tiempo de establecimiento. Para los tiempos de establecimiento lo que se suele hacer es retrasar la señal de entrada justo el tiempo de establecimiento, y usar esta señal interna retrasada como si no hubiera tiempo de establecimiento. La descripción tendría la forma:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL
ENTITY ff IS
PORT(d,clk: IN std_logic;
     q: OUT std_logic);
END ff;

ARCHITECTURE ejemplo OF ff IS
SIGNAL daux: std_logic;
BEGIN
  PROCESS(clk)
  BEGIN
    IF clk='1' THEN
      q<=daux AFTER 10 ns;  -- Retraso respecto del reloj
    END IF;
  END PROCESS;
  daux<=d AFTER 5 ns;       -- Tiempo de establecimiento (set-up)
END ejemplo;
```

10.2 NIVELES LÓGICOS PARA SIMULACIÓN

En VHDL existe el tipo `bit` como predefinido en el lenguaje. Este tipo es un enumerado que puede tomar los valores `'0'` y `'1'`. En principio puede bastar este tipo para la simulación de muchos circuitos, pero es en realidad insuficiente para simulaciones realistas. Por ejemplo, no es posible simular con este tipo la salida de una puerta triestado. Tampoco se modela bien el comportamiento cuando se producen contenciones en un bus, etc.

Para modelar con mayor detalle lo que ocurre físicamente en una línea, que se corresponde naturalmente con una señal, existe un modelo basado en fuerzas y valores lógicos. De esta manera se puede decir que el estado de una línea en un momento dado es una combinación de una fuerza y un nivel lógico.

Los niveles lógicos básicos son tres: *1*, *0* y *X*, siendo este último un valor desconocido, es decir, indica que la línea puede estar a nivel alto, o bajo, pero no se sabe cuál de los dos.

Las fuerzas pueden ser *S*, *R*, *Z* e *I*. La fuerza *S*, o *Strong*, equivale a la salida que se obtiene de una conexión a la alimentación o a tierra a través de un transistor; es la salida típica de un *buffer totem-pole*. La fuerza *R*, o *Resistiva*, equivale a la salida que se obtiene de una conexión a tierra o alimentación a través de una resistencia; es la salida típica de un *buffer* en colector abierto. La fuerza *Z*, o *alta impedancia*, equivale a la salida que se obtiene de una conexión a tierra o alimentación a través de una alta impedancia. Es la salida típica de un *buffer* triestado. La fuerza *I*, o *indeterminada*, sirve para indicar que no se sabe qué fuerza hay en la línea.

En el caso de contenciones se mira primero el valor de la fuerza. La *S* es la más fuerte y luego le siguen la *R*, que es una fuerza débil, y la *Z* que es una fuerza prácticamente nula. Una fuerza fuerte siempre gana a otra más débil, de manera que el resultado de la contención toma la fuerza y valor lógico de la señal ganadora. Si las fuerzas son iguales, entonces hay que mirar el valor lógico; si el valor también coincide, la línea toma el valor y fuerza de cualquiera de las dos; si los valores son diferentes entonces hay varias opciones dependiendo del tipo de lógica que implemente el bus (ver el capítulo de conceptos avanzados en la sección 9.1 de buses y funciones de resolución).

En el caso de que una de las fuerzas que contienden sea indeterminada (*I*), la línea tendrá también la fuerza indeterminada, a no ser que alguna de las otras fuerzas sea *S*, en cuyo caso la línea resultante será también fuerte (*S*).

En algunos simuladores existe también otra fuerza llamada *F (fija)* que indica una conexión directa a tierra o alimentación. Esta fuerza gana a cualquier otra.

Con estas fuerzas y valores lógicos se puede modelar cualquier familia lógica por complicada que sea. Dependiendo de la familia serán necesarias más o menos fuerzas. Por ejemplo, en una puerta TTL no es necesaria la fuerza indeterminada, en cambio en una familia CMOS sí que puede ser necesaria.

Este modelo no se aplica directamente al lenguaje, ya que implica que cada línea esté compuesta por dos cantidades, resultando más complicado que tener una sola cantidad que define el contenido de esa línea. Por esta razón, lo que se hace en VHDL es tener una única cantidad para cada combinación de los valores de fuerza y valor lógico vistos anteriormente. Estas cantidades vienen definidas en el tipo enumerado `std_logic` del paquete `std_logic_1164` de la biblioteca `ieee`.

En la tabla 10.1 se encuentran los niveles lógicos definidos en el tipo `std_logic`. Este tipo ya permite un modelado bastante preciso de casi cualquier familia lógica. Hay que destacar que no todas las combinaciones de valores y fuerzas están contempladas, y a la inversa, se han definido unos niveles lógicos que no se pueden describir como combinación de fuerza y valor. Estos niveles que introduce VHDL son el `'U'` (*Uninitialized*), y `'-'` (*Don't care*). Estos dos no son niveles lógicos realmente, sino que en realidad se trata de comentarios que sirven para indicar que una señal no está inicializada, o bien, que no importa lo que ponga en ella.

`std_logic`	Fuerza-Valor	Descripción	Puerta típica
`'U'`		No inicializado	
`'X'`	*SX*	Desconocido fuerte	*totem-pole*
`'0'`	*S0*	0 fuerte	*totem-pole*
`'1'`	*S1*	1 fuerte	*totem-pole*
`'Z'`	*ZX*	Alta impedancia	Triestado
`'W'`	*RX*	Desconocido resistivo	Em-Col abierto
`'L'`	*R0*	0 resistivo	Emisor abierto
`'H'`	*R1*	1 resistivo	Colector abierto
`'-'`		No importa	

Tabla 10.1: Niveles lógicos en simulación

Como la fuerza *Z* es prácticamente nula, no hay mucha diferencia entre *Z0*, *Z1*, y *ZX*, por esta razón sólo es necesario uno de los tres en VHDL. Por otro lado, los valores `'W'` y `'X'` vienen a sustituir también al caso en que la fuerza sea indeterminada.

10.3 NOTIFICACIÓN DE SUCESOS

Durante la simulación de un circuito descrito en VHDL es muchas veces interesante la notificación de ciertos sucesos. Por ejemplo, puede ser útil advertir con un mensaje por pantalla que cierta señal se ha activado. Estos mecanismos de notificación son especialmente útiles en la detección de violaciones de los tiempos de *set-up* y *hold* en registros y *latches*.

La forma en que se pueden notificar estos sucesos es mediante la utilización de la palabra clave **ASSERT** que tendrá como elemento de activación una condición:

```
ASSERT condición [ REPORT mensaje ] [ SEVERITY nivel_gravedad ] ;
```

Si *no* se cumple la condición especificada en `condición`, entonces aparece el mensaje especificado por la pantalla, indicándose, además, un nivel de gravedad. Tanto el mensaje como el nivel de gravedad son opcionales. Si no se especifica ningún mensaje aparece la cadena `"Assertion Violation"`. El nivel de gravedad es del tipo predefinido `severity_level`, teniendo este tipo los niveles de gravedad definidos: `note`, `warning`, `error` y `failure`, siendo el valor por defecto `error`. En el momento en que se produce una violación de cualquier tipo, el simulador puede detener la ejecución o no dependiendo del nivel de gravedad del error.

Esta sentencia puede ser usada tanto en entornos concurrentes como serie. En entornos concurrentes se ejecutará cada vez que cambien algunas de las señales que intervienen en la condición. En el entorno serie se ejecuta cuando le corresponda en el proceso normal de ejecución serie. A continuación se dan tres ejemplos de utilización de estos mensajes:

```
-- Violación de set-up:
ASSERT NOT(clk'event AND clk='1' AND NOT(d'stable(20 ns)))
   REPORT "Violación del tiempo de set-up"
   SEVERITY warning;

-- Para depurar código secuencial:
ASSERT false
   REPORT "La ejecución pasó por aquí"
   SEVERITY note;
```

La notificación de sucesos no tiene ningún sentido en síntesis. Los compiladores para síntesis pueden ignorarla o bien dar error de compilación.

10.4 PROCESOS PASIVOS

Un uso interesante de las instrucciones de aviso ASSERT es la inclusión de estas sentencias en lo que se llaman procesos pasivos. Cuando se explicó la declaración de entidad, se dijo que era posible definir sentencias concurrentes e incluso procesos en el propio cuerpo de la entidad. A estos procesos, definidos dentro de una entidad, y que por tanto no describen funcionalidad, se les llama *procesos pasivos*.

Lo único que pueden hacer los procesos pasivos es realizar comprobaciones y, mediante sentencias ASSERT, advertir de violaciones dentro de la ejecución. Resulta interesante colocarlos en la entidad, ya que así sirven para cualquier arquitectura que se pueda definir. A continuación se muestra un ejemplo de utilización de un proceso pasivo para la comprobación del tiempo de establecimiento (*set-up*).

Ejemplo 10.2 *Añadir un proceso pasivo en la entidad del registro del ejemplo 10.1 que detecte la violación del tiempo de establecimiento y emita un mensaje.*

En principio la arquitectura sería la misma, por lo que no habría que modificarla, solamente la entidad incluirá, entre un BEGIN y un END, el ASSERT visto en los tres ejemplos anteriores. En el supuesto de que no se disponga de dichos atributos para la señal de entrada, se da a continuación una posible solución donde se muestra cómo se maneja el tiempo de simulación:

```
ENTITY ff IS
PORT(d,clk: IN std_logic;
     q: OUT std_logic);
BEGIN
  PROCESS(clk,d)
    VARIABLE tiempo_d_cambio: time := 0 ns;
    VARIABLE clk_ultimo, d_ultimo: std_logic := 'X';
  BEGIN
    IF d/=d_ultimo THEN
      tiempo_d_cambio:=now;
      d_ultimo:=d;
    END IF;
    IF clk/=clk_ultimo THEN
      IF clk='1' THEN
        ASSERT (now-tiempo_d_cambio>=5 ns)
          REPORT "Error en el tiempo de establecimiento"
          SEVERITY warning;
      END IF;
      clk_ultimo:=clk;
    END IF;
  END PROCESS;
END ff;
```

Aquí se ha supuesto que no se tiene el atributo `'event`, lo cual se ha considerado por razones pedagógicas. En el caso de haber utilizado este atributo, bastaría conservar el IF más interior añadiendo AND `clk'event` a la condición; con esto, la condición y la variable de reloj no serían necesarias. También se ha introducido la función predefinida `now` que da el tiempo de simulación en el momento en que se ejecuta la instrucción.

10.5 DESCRIPCIÓN DE UN BANCO DE PRUEBAS

Una parte crucial, y muchas veces las más costosa en tiempo en el diseño de sistemas digitales, es la verificación de su correcta funcionalidad. Con las metodologías de diseño tradicionales esto sólo era totalmente realizable tras su implementación física, con el consiguiente riesgo y coste adicional. Con el lenguaje de descripción del hardware VHDL se permite acelerar el proceso, pudiendo realizar una exhaustiva simulación de un modelo que, si es correcto, puede ser directamente sintetizado.

La manera más directa de verificar el correcto funcionamiento de un modelo VHDL consistiría, mediante el uso de una herramienta de simulación, en cambiar las entradas y observar cómo evolucionan las salidas. Esta metodología de simulación interactiva puede ser útil para diseños sencillos, pero en el caso de modelos más complejos es mejor definir lo que se denomina un *banco de pruebas*.

Un banco de pruebas no es más que la definición de un subconjunto de entradas llamadas *patrones de test*, con las que comprobar el circuito o modelo VHDL. Normalmente las herramientas de CAD permiten la definición de estos vectores de test. Con el lenguaje de descripción del hardware VHDL se puede modelar el banco de pruebas independientemente de la herramienta de simulación. Además de esta ventaja del VHDL para la generación del banco de pruebas sobre la simulación interactiva, cabe añadir que el mismo modelo de banco de pruebas puede ser utilizado en cualquier fase del diseño con VHDL, permitiendo un importante ahorro de tiempo y esfuerzo.

El banco de pruebas es una entidad sin entradas ni salidas en su tipo más simple. La arquitectura del banco de pruebas, de tipo estructural, tiene como señales internas las entradas y salidas del circuito. El único componente es el correspondiente a la entidad que se desea simular.

Si se realiza una descripción de un modelo VHDL, se puede simultáneamente describir un banco de pruebas para éste. Si posteriormente se sintetiza el modelo, el mismo banco de pruebas utilizado para verificar la descripción pre-síntesis puede ser utilizado para simular el modelo VHDL post-síntesis generado por la herramienta software.

La construcción de un banco de pruebas para un modelo determinado puede abordarse de distintas maneras. Según la forma de generación de los vectores de test, estas metodologías se pueden clasificar en:

1. Método tabular.
2. Utilización de ficheros con vectores de test.
3. Metodología algorítmica.

A continuación se describen con detalle estas diversas formas de abordar la construcción del banco de pruebas.

10.5.1 Método tabular

Se va a realizar un banco de pruebas para un registro de desplazamiento que se especifica en el ejemplo 10.3.

Ejemplo 10.3 *Realizar el modelo de un registro de desplazamiento síncrono de 4 bits. El registro tendrá como entrada paralela la señal* `e[3..0]` *y como salida* `s[3..0]`. *El modo de funcionamiento se seleccionará con la señal de control* `m[1..0]`, *de manera que cuando la entrada sea 00 el registro no cargará ni desplazará su valor, cuando sea 11 realizará una carga en paralelo, cuando sea 01 se desplazará a la derecha, y con 01 a la izquierda. Adicionalmente existirán 2 entradas serie (*`ed` *y* `ei`*) para realizar los desplazamientos hacia la derecha y la izquierda.*

Utilizando los conceptos de **componente** y **paquete** descritos en el capítulo 8, se puede realizar el modelo del ejemplo 10.3 tal como se describe a continuación.

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

PACKAGE registros IS
  COMPONENT desplazamiento
  PORT(clk, rst, ed, ei : IN std_logic;
       m : IN std_logic_vector(1 DOWNTO 0);
       e : IN std_logic_vector(3 DOWNTO 0);
      sal: OUT std_logic_vector(3 DOWNTO 0));
  END COMPONENT;
END registros;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY desplazamiento IS
PORT(clk, rst, ed, ei : IN std_logic;
       m : IN std_logic_vector(1 DOWNTO 0);
       e : IN std_logic_vector(3 DOWNTO 0);
     sal : OUT std_logic_vector(3 DOWNTO 0));
END desplazamiento;

ARCHITECTURE comportamiento OF desplazamiento IS
-- Señal auxiliar para leer la salida:
  SIGNAL s: std_logic_vector(3 DOWNTO 0));
BEGIN
  sal<=s;     -- Así se puede leer la salida.
  PROCESS (clk,rst)
  BEGIN
    IF rst='1' THEN s<="0000";
    ELSIF (clk'EVENT AND clk='1') THEN
      CASE m IS
        WHEN "01" => s(3)<=ed; s(2)<=s(3);     -- Desplazamiento a
                     s(1)<=s(2); s(0)<=s(1);   -- la derecha
        WHEN "10" => s(3)<=s(2); s(2)<=s(1);   -- Desplazamiento a
                     s(1)<=s(0); s(0)<=ei;     -- la izquierda
        WHEN "11" => s<=e;                     -- Carga en paralelo
        WHEN OTHERS => NULL;                   -- La salida no cambia
      END CASE;
    END IF;
  END PROCESS;
END comportamiento;
```

En este caso se ha creado el componente `desplazamiento`, dentro del paquete `registros` que a su vez formará parte de una biblioteca.

Para verificar la correcta funcionalidad del registro de desplazamiento, con el método tabular, hay que elaborar una tabla con entradas y la respuesta prevista del circuito. Esta tabla formará parte del código VHDL que referencia el componente desde su biblioteca, que aplica los estímulos del patrón de test, y que verifica si la respuesta del circuito coincide con la respuesta prevista. Un ejemplo de banco de pruebas utilizando esta metodología se describe a continuación.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY test_shift IS
END test_shift;

USE work.registros.desplazamiento;
ARCHITECTURE test OF test_shift IS
  SIGNAL clk, rst, ed, ei : std_logic;
  SIGNAL m : std_logic_vector(1 DOWNTO 0);
  SIGNAL e : std_logic_vector(3 DOWNTO 0);
  SIGNAL s : std_logic_vector(3 DOWNTO 0);
  TYPE tipo_vtest IS RECORD
       clk, rst, ed, ei : std_logic;
```

```
      m : std_logic_vector(1 DOWNTO 0);
      e : std_logic_vector(3 DOWNTO 0);
      s : std_logic_vector(3 DOWNTO 0);
  END RECORD;
  TYPE lista_vtest IS ARRAY(0 TO 10) OF tipo_vtest;
  CONSTANT varios_v : lista_vtest :=(
    -- Réset del registro
(clk=>'0', rst=>'1', ed=>'-', ei=>'-', m=>"--", e=>"----", s=>"0000"),
(clk=>'1', rst=>'1', ed=>'-', ei=>'-', m=>"--", e=>"----", s=>"0000"),
(clk=>'0', rst=>'0', ed=>'-', ei=>'-', m=>"00", e=>"----", s=>"0000"),
    -- Se comprueba que no se carga nada en el modo 0
(clk=>'1', rst=>'0', ed=>'-', ei=>'-', m=>"00", e=>"1011", s=>"0000"),
(clk=>'0', rst=>'0', ed=>'-', ei=>'-', m=>"00", e=>"1011", s=>"0000"),
    -- Se carga el código 1010 en paralelo
(clk=>'1', rst=>'0', ed=>'-', ei=>'-', m=>"11", e=>"1011", s=>"1011"),
(clk=>'0', rst=>'0', ed=>'-', ei=>'-', m=>"11", e=>"1011", s=>"1011"),
    -- Se desplaza hacia la derecha 0->101
(clk=>'1', rst=>'0', ed=>'0', ei=>'-', m=>"01", e=>"----", s=>"0101"),
(clk=>'0', rst=>'0', ed=>'0', ei=>'-', m=>"01", e=>"----", s=>"0101"),
    -- Se desplaza hacia la izquierda 101<-1
(clk=>'1', rst=>'0', ed=>'0', ei=>'1', m=>"10", e=>"----", s=>"1011"),
(clk=>'0', rst=>'0', ed=>'0', ei=>'1', m=>"10", e=>"----", s=>"1011"));

BEGIN
  -- Se referencia un componente para testearlo
  registro1: desplazamiento
  PORT MAP(clk=>clk, rst=>rst, ed=>ed, ei=>ei, m=>m, e=>e, sal=>s);
  -- Se aplican los vectores de test y se verifica el resultado
  PROCESS
    VARIABLE vector : tipo_vtest;
    VARIABLE errores : boolean := false;
  BEGIN
    FOR i IN 0 TO 10 LOOP
      vector:=varios_v(i);
      clk<=vector.clk; rst<=vector.rst; ed<=vector.ed; ei<=vector.ei;
      m<=vector.m; e<=vector.e;
      WAIT FOR 20 ns;
      IF s /= vector.s THEN
        ASSERT false REPORT "Salida incorrecta. ";
        errores:=true;
      END IF;
    END LOOP;
    ASSERT NOT errores REPORT "Error en el test. ";
    ASSERT errores     REPORT "Test superado. ";
    WAIT;
  END PROCESS;
END;
```

Tal como se ha comentado con anterioridad, la entidad carece de puertos de entrada y salida, es de tipo estructural, y el único componente que aparece es el que va a ser simulado. En este caso se ha definido el tipo `tipo_vtest` como una estructura RECORD con todas las señales de entrada y salida del componente. Posteriormente se define un nuevo tipo `lista_vtest` como un ARRAY de estos vectores de test. El patrón de test concreto que se desea aplicar al componente será una constante del tipo `lista_vtest`.

El código que se ejecuta secuencialmente dentro del PROCESS, simplemente se limita a acceder a un vector concreto del patrón de test, aplicárselo al componente y verificar si el resultado es correcto. Cabe hacer notar el interesante uso de las cláusulas ASSERT y REPORT para avisar si el funcionamiento del componente no se corresponde con el esperado. La espera de 20 ns. que se especifica en la sentencia `WAIT FOR 20 ns;` se podría reducir a un simple WAIT al no tener retrasos asociados el modelo de simulación del ejemplo 10.3.

Si se desea modificar el vector de test sólo hay que cambiar la definición de la constante `varios_v` y ajustar el número de iteraciones del bucle.

10.5.2 Utilización de ficheros con vectores de test

El mecanismo de generación del banco de pruebas descrito en la sección anterior es independiente del modelo a ser simulado. De hecho, el modelo del componente y el banco de pruebas pueden estar en ficheros y bibliotecas distintas. Sólo es necesario que el modelo del componente sea visible para el banco de pruebas.

Esta estructura aporta modularidad y portabilidad. El mismo banco de pruebas puede utilizarse para simular un modelo post-síntesis de este componente, o bien un modelo distinto del mismo componente.

Para simulaciones más costosas o para tener una organización mayor en el banco de pruebas, puede ser interesante separar los vectores de test del mecanismo de simulación. En el caso anterior se describen en el mismo código los patrones de test y el mecanismo de simulación. Existe la posibilidad de realizar una separación total de estas dos partes, teniendo por un lado un fichero con los patrones de test, y por otro lado el código VHDL que accede a estos patrones y realiza la simulación. Esta separación es posible debido a la existencia en VHDL de paquetes de entrada/salida con procedimientos para leer y escribir en ficheros.

En la biblioteca estándar `std` se encuentra el paquete `textio`, donde se encuentra el procedimiento `readline(fich,linea)`. Este procedimiento accede al fichero `fich`, leyendo una línea y asignándoselo a la variable `linea`. Análogamente el procedimiento `read(linea,ch)` tiene como entrada una línea, asignándole a la variable `ch` del tipo `character` un carácter. En la sección 9.3 se explica el acceso a ficheros con más detalle.

Desgraciadamente, a la escritura del presente capítulo no existe un procedimiento estándar similar que lea de un fichero devolviendo un valor del tipo `std_logic` o

std_logic_vector. Por esto se hace necesario, para simplificar el diseño del banco de pruebas, el diseño de dos procedimientos que realicen esta función. Estos procedimientos tendrán el mismo nombre, tal como permite la sobrecarga de operadores en VHDL, y se llamarán r_std_logic. Definidos a partir del procedimiento read(fich,ch), y formando parte del paquete mi_io, tendrían la forma:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.all;

PACKAGE mi_io IS
  PROCEDURE r_std_logic(l: INOUT line; valor: OUT std_logic);
  PROCEDURE r_std_logic(l: INOUT line; valor: OUT std_logic_vector);
  TYPE caracteres_std_logic IS ARRAY(character) OF std_logic;
  CONSTANT a_stdlogic : caracteres_std_logic:=
        ('U'=>'U', 'X'=>'X', '0'=>'0', '1'=>'1', 'Z'=>'Z', 'W'=>'W',
         'L'=>'L', 'H'=>'H', '-'=>'-', OTHERS=>'X');
END mi_io;

PACKAGE BODY mi_io IS
  PROCEDURE r_std_logic (l: INOUT LINE; valor: OUT std_logic) IS
  VARIABLE caracter: character;
  BEGIN
        read(l, caracter);
        valor:=a_stdlogic(caracter);
  END r_std_logic;

  PROCEDURE r_std_logic(l: INOUT line; valor: OUT std_logic_vector) IS
  VARIABLE caracter: string(valor'range);
  BEGIN
        read(l,caracter);
        FOR i IN caracter'range LOOP
          valor(i):=a_stdlogic(caracter(i));
        END LOOP;
  END r_std_logic;
END mi_io;
```

A partir de estos procedimientos ya se puede escribir el código VHDL que accederá al fichero con los patrones de test. Se va a suponer que el fichero con estímulos sólo contiene los valores de los vectores de test, ordenados por filas, y en el mismo orden en que están definidas las señales en el ejemplo 10.3. Con estas condiciones, el fichero de patrones de test shift_test.est que realizaría la misma simulación que en la aproximación tabular sería:

```
clk     rst     ed      ei       m      e[3..0] s[3..0]
 0       1      -       -       --       ----    0000
 1       1      -       -       --       ----    0000
 0       0      -       -       00       ----    0000
 1       0      -       -       00       1011    0000
 0       0      -       -       00       1011    0000
 1       0      -       -       11       1011    1011
 0       0      -       -       11       1011    1011
```

```
1       0       0       -       01      ----    0101
0       0       0       -       01      ----    0101
1       0       -       1       10      ----    1011
0       0       -       1       10      ----    1011
```

La primera línea con los nombres de las señales se incluye en el fichero para aumentar la legibilidad. Después se leerá sin realizarse ninguna acción con esta primera línea. Las siguientes líneas tienen los caracteres separados por un tabulador también para aumentar la legibilidad del fichero, incluyendo el tabulador al inicio de la línea. El código VHDL'93 que lee estos ficheros tiene en cuenta este formato, tal como se indica a continuación.

```
ENTITY test2_shift IS
END test2_shift;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL;
USE work.mi_io.ALL;
USE work.registros.desplazamiento;

ARCHITECTURE test OF test2_shift IS
  SIGNAL clk, rst, ed, ei : std_logic;
  SIGNAL m : std_logic_vector(1 DOWNTO 0);
  SIGNAL e : std_logic_vector(3 DOWNTO 0);
  SIGNAL s : std_logic_vector(3 DOWNTO 0);

BEGIN
  -- Se referencia un componente para testearlo
  registro1: desplazamiento
  PORT MAP(clk=>clk, rst=>rst, ed=>ed, ei=>ei, m=>m, e=>e, sal=>s);
  -- Se aplican los vectores de test y se verifica el resultado

  PROCESS
    VARIABLE errores : boolean := false;
    FILE fichero : text OPEN read_mode IS "shift_test.est";
    VARIABLE invector : line;
    VARIABLE ok : boolean;
    VARIABLE ch : character;
    VARIABLE vclk, vrst, ved, vei : std_logic;
    VARIABLE vm : std_logic_vector(1 DOWNTO 0);
    VARIABLE ve, vs : std_logic_vector(3 DOWNTO 0);
  BEGIN
    -- Se lee la 1a línea del fichero (nombres)
    readline(fichero,invector);
    WHILE NOT endfile(fichero) LOOP
      readline(fichero,invector);  -- Se lee una línea del fichero
      -- Se leen los TABs y los valores
      read(invector,ch); r_std_logic(invector,vclk);
      read(invector,ch); r_std_logic(invector,vrst);
      read(invector,ch); r_std_logic(invector,ved);
      read(invector,ch); r_std_logic(invector,vei);
      read(invector,ch); r_std_logic(invector,vm);
      read(invector,ch); r_std_logic(invector,ve);
      read(invector,ch); r_std_logic(invector,vs);
      -- Se aplica el vector de test
```

```
      clk<=vclk; rst<=vrst; ed<=ved; ei<=vei; m<=vm; e<=ve;
      WAIT FOR 20 ns;
      IF vs/=s THEN
        ASSERT false REPORT "Salida incorrecta" SEVERITY warning;
        errores:=true;
      END IF;
    END LOOP;
    ASSERT NOT errores REPORT "Error en el test. " SEVERITY error;
    ASSERT errores     REPORT "Test superado. " SEVERITY note;
    WAIT;
  END PROCESS;
END;
```

El equivalente en VHDL'87 sería prácticamente igual salvo la línea en la que se define el fichero de estímulos, que se describiría como:

```
FILE fichero : text IS IN "shift_test.est";
```

La estructura del código que implementa el banco de pruebas es similar al caso anterior. De nuevo la entidad no tiene entradas ni salidas, siendo una descripción estructural en la que se referencia el único componente que se va simular. Tras leer el vector de test, y aplicarlo, se espera a la estabilización del circuito durante 20 ns., comprobando posteriormente si la salida coincide con la prevista.

Este código, que describe un banco de pruebas para el registro de desplazamiento del ejemplo 10.3, es independiente de los patrones de test que se almacenan en el fichero `shift_test.est`. Para cambiar los vectores de simulación sólo se tendrá que editar este fichero de estímulos. Sólo si se cambia el formato de los ficheros, o la descripción del componente a simular, se editará este código VHDL. En este tipo de diseño de banco de pruebas se consigue una gran modularidad y organización en la simulación.

10.5.3 Metodología algorítmica

El método tabular puede ser útil para patrones de test no muy extensos. La posterior organización modular en ficheros separados de los estímulos y el código VHDL que realiza la simulación favorece la realización y organización de simulaciones extensas.

A pesar de las facilidades que aporta la aproximación modular con ficheros, pueden haber simulaciones en las que generar los patrones de test sea muy costoso temporalmente. Otra posibilidad de realizar un banco de pruebas consiste en incluir algoritmos en el mismo código que generen estos vectores de test.

Un ejemplo de un banco de pruebas de estas características puede ser la simulación de circuitería combinacional o aritmética. Puede ser sencillo realizar una bucle que genere todas las combinaciones posibles de entrada, se las aplique al componente a simular, y compruebe si la salida se corresponde con la que se ha calculado en el algoritmo.

Para el caso del registro de desplazamiento que está sirviendo para ejemplificar esta sección, un banco de pruebas algorítmico podría ser el código VHDL que se adjunta.

```
ENTITY test3_shift IS
END test3_shift;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.registros.desplazamiento;

ARCHITECTURE test OF test3_shift IS
  SIGNAL clk, rst, ed, ei : std_logic;
  SIGNAL m : std_logic_vector(1 DOWNTO 0);
  SIGNAL e : std_logic_vector(3 DOWNTO 0);
  SIGNAL s : std_logic_vector(3 DOWNTO 0);
  CONSTANT valor : std_logic_vector(3 DOWNTO 0):= "1011";
BEGIN
    -- Se referencia el componente para testearlo
    registro1 : desplazamiento
    PORT MAP(clk=>clk, rst=>rst, ed=>ed, ei=>ei, m=>m, e=>e, sal=>s);

    -- Se genera el test y se verifica el resultado
    PROCESS
      VARIABLE reg_aux : std_logic_vector(3 DOWNTO 0):="0000";
    BEGIN
      -- Inicialmente se hace un réset
      rst<='1'; WAIT FOR 20 ns;
      ASSERT (s=reg_aux) REPORT "Réset incorrecto" SEVERITY error;
      WAIT FOR 20 ns;

      -- Se desplaza un '1' hacia la izquierda
      FOR i IN 0 TO 3 LOOP
        clk<='0'; rst<='0'; ei<='1'; m<="10";
        WAIT FOR 10 ns;
        clk<='1'; rst<='0'; ei<='1'; m<="10";
        reg_aux:=reg_aux(2 DOWNTO 0) & '1';
        WAIT FOR 10 ns;
        ASSERT (s=reg_aux)
            REPORT "Desplazamiento a la izquierda incorrecto"
            SEVERITY error;
      END LOOP;

      -- Se desplaza un '0' hacia la derecha
      FOR i IN 0 TO 3 LOOP
        clk<='0'; rst<='0'; ed<='0'; m<="01";
        WAIT FOR 10 ns;
        clk<='1'; rst<='0'; ed<='0'; m<="01";
        reg_aux:='0' & reg_aux(3 DOWNTO 1);
        WAIT FOR 10 ns;
        ASSERT (s=reg_aux)
            REPORT "Desplazamiento a la izquierda incorrecto"
```

```
            SEVERITY error;
        END LOOP;

        -- Se prueba la carga en paralelo
        clk<='0'; rst<='0'; e<=valor; m<="11";
        WAIT FOR 10 ns;
        clk<='1'; rst<='0'; e<=valor; m<="11";
        WAIT FOR 10 ns;
        ASSERT (s=valor)
            REPORT "Carga incorrecta"
            SEVERITY error;
        ASSERT FALSE
            REPORT "Test finalizado"
            SEVERITY note;
    END PROCESS;
END;
```

Este código VHDL de nuevo tiene una entidad sin puertos de entrada salida, con las señales en la arquitectura necesarias para referenciar el componente que se va a simular.

Como novedad se observa cómo no hay acceso a unos vectores de test predefinidos, sino que éstos se van generando dentro del mismo código que realiza la simulación. En el ejemplo se simula el réset del circuito, el desplazamiento a la izquierda, y el desplazamiento a la derecha. Estas operaciones se realizan externamente al circuito a simular para comprobar que la respuesta del circuito es correcta.

CAPÍTULO 11

VHDL PARA SÍNTESIS

La síntesis de un circuito, a partir de una descripción VHDL, consiste en reducir el nivel de abstracción de la descripción del circuito hasta convertirlo en una definición puramente estructural cuyos componentes son los elementos de una determinada biblioteca. Esta biblioteca dependerá del circuito que se quiera realizar, la herramienta de síntesis, etc. Al final del proceso de síntesis se debe obtener un circuito que *funcionalmente* se comporte igual que la descripción que de él se ha hecho.

En un principio *cualquier descripción en VHDL es sintetizable*, no importa el nivel de abstracción que la descripción pueda tener. Esto, que en principio puede parecer sorprendente no lo es en absoluto, ya que cualquier descripción en VHDL se puede simular, y si se puede simular, el propio simulador (en general un ordenador ejecutando un programa) es un circuito que funcionalmente se comporta tal y como se ha descrito, por lo tanto, desde este punto de vista es una síntesis del circuito que se ha diseñado. Es evidente que no será el circuito más optimizado para realizar la tarea que se pretende, ni lo hará a la velocidad que se requiere, pero seguro que funcionalmente se comporta tal y como se ha descrito.

La complejidad del circuito resultante, y también incluso la posibilidad o no de realizar el circuito, va a depender sobre todo del nivel de abstracción inicial que tenga la descripción. Como primera solución se puede utilizar un ordenador que ejecute la simulación y ya se tiene la síntesis. A partir de este primer intento, hay que ir optimizando el circuito. En realidad las herramientas de síntesis siguen una aproximación distinta, ya que de otra manera el circuito acabaría siendo algo parecido a un microprocesador cuando quizá en realidad sólo se pretende, eventualmente, implementar una puerta lógica.

La aproximación de las herramientas de síntesis consiste en, partiendo de la descripción original, reducir el nivel de abstracción hasta llegar a un nivel de descripción estructural. La síntesis es por tanto una *tarea vertical* entre los niveles de abstracción de un circuito. Así, una herramienta de síntesis comenzaría por la descripción comportamental abstracta algorítmica e intentaría traducirla a un nivel de transferencia entre registros descrita con ecuaciones de conmutación. A partir de esta descripción se intenta transformarla a una descripción estructural donde se realiza, además, lo que se llama el *mapeado tecnológico*, es decir, la descripción del circuito utilizando los componentes de una biblioteca concreta que depende de la tecnología con la cual se quiera implementar el diseño.

Las herramientas de síntesis actuales cubren a la perfección la síntesis a partir de descripciones RTL y estructurales, pero no están tan avanzadas en el manejo de diseños descritos en un nivel de abstracción más alto. No es que no se pueda sintetizar a partir de un nivel alto de abstracción, lo que ocurre es que la síntesis obtenida no es quizá la más óptima para el circuito que se pretende realizar.

11.1 RESTRICCIONES Y CONSEJOS EN LA DESCRIPCIÓN

No se va a explicar en esta sección el funcionamiento interno de las herramientas de síntesis, pero sí que conviene dar algunas nociones de cómo una herramienta de síntesis interpreta algunas de las instrucciones en VHDL. Esto es interesante porque muchas veces es más sencillo para el diseñador simplificar ciertas cosas que dejar esta tarea a una máquina que lo puede hacer mal. Además, puede ocurrir que un sintetizador dé una interpretación algo diferente de cierta estructura, por lo tanto conviene aclarar también si un circuito es combinacional, secuencial, síncrono, etc.

Por lo general, existe una definición exacta de cómo una descripción VHDL se va a simular, y cuál va a ser el resultado de esa simulación. En cambio, no existe este consenso en cuanto a la síntesis. La síntesis de una descripción VHDL es muy posible que no se comporte igual que cuando se simuló el código, no sólo porque en el circuito real existen los retrasos, sino que incluso funcionalmente puede comportarse de forma diferente. Incluso, entre herramientas de síntesis, el resultado puede variar de unas a otras.

Todo esto hace que sean necesarias unas restricciones a lo que es el lenguaje puro; conocer estas restricciones, y especialmente las interpretaciones que de él se hacen, es necesario para cualquiera que pretenda usar el VHDL para síntesis de circuitos. Estas restricciones dependen de cada herramienta de síntesis, ya que dependiendo de la

calidad de la herramienta pueden interpretar más estructuras del lenguaje o menos. El fabricante de estas herramientas suele dar el subconjunto del lenguaje que el sintetizador es capaz de interpretar, así como las interpretaciones que hace de determinadas estructuras, que dejan de ser estándar, pero que facilitan el diseño y su posterior síntesis. No obstante, hay determinadas recomendaciones que suelen ser comunes a la mayoría de las herramientas de síntesis. Veamos a continuación unas cuantas:

Evitar las cláusulas temporales Normalmente los simuladores prohíben expresamente el uso de asignaciones con retraso en las señales, en otras simplemente los ignoran, pero lo que está claro es que el sintetizador intentará implementar el circuito *funcionalmente*, por lo que estos retrasos no tienen sentido para el sintetizador. Por la misma razón, no se permiten las asignaciones múltiples a una señal en una única sentencia

Identificar cada puerta con claridad Las puertas lógicas y otros elementos tienen generalmente una estructura clara e incluso se pueden utilizar comandos directos que realizan estas funciones. No es nada conveniente describir una puerta lógica tal y como se hizo en el ejemplo 6.2.

Utilizar funciones Normalmente el uso de funciones ayuda a la herramienta de síntesis a particionar de forma óptima. El particionado es bastante importante, ya que permite la reutilización de lógica por parte de la herramienta.

Evitar las sentencias de espera En algunos sintetizadores quizá sea posible utilizar sentencias de espera `WAIT` dentro de los procesos, pero no es nada aconsejable puesto que la herramienta puede tener dificultades en interpretar estas sentencias. Es aconsejable en cambio el uso de listas sensibles, y en algunos sintetizadores es casi la única posibilidad. El uso del `WAIT` está bastante restringido, así, si se usa, algunas herramientas exigen que sea la primera instrucción del `PROCESS`, y sólo se permite una sentencia de espera por proceso.

Cuidado con las listas sensibles La mayoría de sintetizadores admiten la lista sensible o una sentencia `WAIT` al principio, pero no siempre la interpretan como lo haría un simulador, ya que en determinadas ocasiones el resultado de la síntesis hace que se active el proceso cuando cambie algune señal que se encuentra en el propio proceso pero que no se había puesto en la lista sensible o en el `WAIT`. Es decir, el sintetizador a veces *amplía* la lista sensible según le parece y sin avisar, normalmente con el objetivo de que la lógica que describe el proceso sea puramente combinacional.

Permitir discrepancia Normalmente es fácil sintetizar algo simple como `s<=NOT s`, ya que no es más que una puerta inversora conectada a sí misma que puede servir muy bien para generar una señal de reloj, con periodo el doble que el retraso

que la puerta presente. Si se intenta simular algo como la instrucción anterior, se comprobará que la simulación se queda colgada en esa instrucción puesto que no hay retrasos y se llama a sí misma una y otra vez. Por lo tanto, en estos casos, aunque la simulación es incorrecta, la síntesis no lo es.

La discrepancia no es sólo temporal, también es funcional. Tal y como se comentaba anteriormente, es fácil que una misma descripción tenga un comportamiento diferente en simulación que en síntesis.

Cuidado con la inicialización de variables y señales En simulación esto funciona, pero no tiene por qué ser así en síntesis. De hecho, lo más normal es que el sintetizador no tome en cuenta ninguna inicialización. Sólo a veces, si el dispositivo dispone de réset o algún mecanismo de inicialización, el valor inicial será tomado en cuenta para inicializar la señal, siempre, claro está, que dicha señal describa un registro. En cualquier caso es preferible hacer el réset explícitamente, o, en su caso, leerse el manual de la herramienta de síntesis.

Señales de reloj Normalmente sólo se permite una señal de reloj por proceso y, además, debe especificarse claramente el flanco de subida del reloj mediante la condición `clk='1' AND clk'event`. En general sólo puede ponerse esta condición una vez por proceso y en ningún caso se puede poner `ELSE` en el `IF` en el que se usó la condición.

Asignaciones únicas Aunque en simulación es bastante corriente que a una señal se le asignen varios valores a lo largo de un mismo proceso, en síntesis esto resulta difícil de interpretar y no debe usarse (normalmente la herramienta no lo permite).

Niveles lógicos No todos los niveles lógicos del `std_logic` se pueden interpretar, y no todas las herramientas lo hacen igual. Especial cuidado requiere el nivel `'-'` (*don't care*) que puede parecer un mecanismo para indicarle al sintetizador que elija el mejor valor posible (para minimizar lógica) y en cambio no lo es; es más, es muy probable que la herramienta lo interprete como un nivel lógico más, sintetizando lógica que no es la que se espera.

Evitar `IF`s anidados Normalmente las herramientas tienden a no sintetizar de manera óptima varios condicionales anidados entre sí. Los condicionales es mejor utilizarlos a solas.

Utilizar `CASE` mejor que varios `IF`s Las estructuras `CASE` tienen para los sintetizadores un modelo optimizado de síntesis, generalmente mejor que lo mismo descrito mediante `IF`s.

Utilizar el estilo indicado para las máquinas de estados Muchos de los problemas digitales se pueden resolver de forma sencilla mediante una máquina de estados. En

VHDL hay muchos estilos diferentes para poder describir máquinas de estados. Puede ocurrir entonces que el sintetizador no se dé cuenta de que lo que tiene delante es una máquina de estados, y entonces no optimiza bien el circuito. En los manuales de los sintetizadores suelen venir ejemplos de lo que la herramienta entenderá que es una máquina de estados, entonces es mejor utilizar ese estilo aunque no resulte cómodo, el resultado final será bastante más óptimo.

Especificar la arquitectura Es posible que se creen varias descripciones para un mismo circuito. Normalmente el sintetizador cogerá la primera que le parezca, generalmente la última compilada, por lo que resulta conviene especificar cuál de todas las arquitecturas de la entidad se desea sintetizar mediante un bloque de configuración `CONFIGURATION`, o bien mediante el mecanismo de que disponga la herramienta para ello.

Con estas restricciones ahora expuestas, y hay algunas más que dependerán del sintetizador, es fácil darse cuenta de que no basta con describir algo en VHDL y ver que funciona para poderlo sintetizar, hay que además conocer bien la herramienta de síntesis, saber qué cosas no se pueden describir, y además, hacer la descripción lo más optimizada posible. Para ello es bueno que se conozcan cómo se sintetizan algunas de las estructuras básicas del VHDL, o por lo menos, conocer si lo que se está describiendo es lógica combinacional o secuencial.

11.2 CONSTRUCCIONES BÁSICAS

El primer paso es ver si un circuito describe lógica combinacional o secuencial. Un circuito describe lógica combinacional si la salida depende únicamente de la entrada en ese instante, y no de la entrada que hubiera tenido en un pasado, es decir, ante una entrada dada la salida es siempre la misma. Un circuito describe lógica secuencial cuando la salida depende de la entrada actual y de las entradas anteriores, o dicho de otra forma, la salida depende de la entrada y del estado del sistema. Esto introduce un nuevo elemento dentro del sistema que será la memoria. Normalmente el elemento de memoria será una señal que frente a unos estímulos captura otra señal y en caso contrario permanece igual. Esto nos da una pista de si un circuito es secuencial y si se realizará por tanto a partir de elementos de memoria como puedan ser cerrojos o registros.

11.2.1 Descripción de lógica combinacional

La idea básica es que si en la estructura del lenguaje no se introducen "elementos de memoria" entonces se está delante de una descripción combinacional. Se va a mostrar entonces cómo evitar que aparezcan elementos de memoria para que el circuito se realice sólo con puertas lógicas. Los requisitos para conseguir esto serán entonces:

- Si la ejecución es concurrente se define lógica combinacional cuando:
  - La señal que está siendo asignada no interviene en la asignación. Ejemplos:

```
a<=b WHEN h='1' ELSE c;   -- combinacional
a<=b WHEN h='1' ELSE a;   -- secuencial
a<=b WHEN a='1' ELSE c;   -- secuencial
```

    Si la señal interviniera en la asignación entonces habría casos para los cuales se conserva su valor y, por tanto, sería un elemento de memoria.

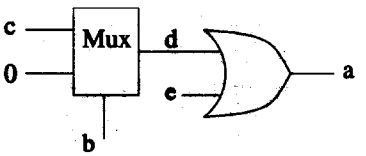
  - No hay lazos combinacionales (en realidad ésta es una extensión de la anterior). Ejemplos:

```
-- Secuencial
d<=b AND a;
a<=d OR e;
```

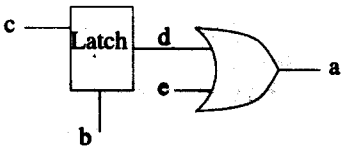
```
-- Combinacional
d<=b AND c;
a<=d OR e;
```

- Si la ejecución es serie (proceso) se sintetiza lógica combinacional cuando:
  - La lista sensible de un proceso incluye todas las señales implicadas en las asignaciones. Es claro que si alguna señal no está en la lista sensible, cuando se produzca un cambio en esta señal el proceso no se ejecutará y no habrá cambios en las señales internas del proceso que por tanto conservan su valor, por lo que se tratará de un circuito secuencial.
  - Se asignan todas las variables y señales que intervienen, y se contemplan todos los casos de las condiciones. Normalmente esto se aplica a instrucciones condicionales. Si hay una condición para la cual la señal no se asigna, es decir, se queda igual, esto indica la presencia de un *latch*. La explicación es la misma, si para una determinada condición no se realiza la asignación, entonces la señal no asignada conserva su valor y por tanto es un elemento de memoria. Ejemplo:

```
-- Combinacional
PROCESS(b,c,e,d)
BEGIN
   IF b='1' THEN d<=c;
   ELSE d<=0;
   END IF;
   a<=d OR e;
END PROCESS;
```




```
-- Secuencial
PROCESS(b,c,e,d)
BEGIN
   IF b='1' THEN d<=c;
   END IF;
   a<=d OR e;
END PROCESS;
```



11.2.2 Descripción de lógica secuencial

Un circuito es secuencial si su salida puede depender del estado del sistema. Desde el punto de vista del VHDL, las reglas que se han visto para la descripción de lógica combinacional sirven aquí para decir que, si cualquiera de las reglas anteriores no se cumple, entonces el circuito describe lógica secuencial. En este sentido, en vez de repetir las mismas reglas pero negándolas, se verá a continuación cómo describir algunos de los elementos básicos de un circuito secuencial como *latches*, registros, relojes, etc.

Descripción de *latches* Un *latch* o cerrojo es un circuito que mantiene la salida a un valor cuando una señal de control está inactiva, y la salida es igual a la entrada cuando dicha señal de control está activa. Normalmente la señal está activa si está a nivel alto o '1' e inactiva si está a nivel bajo o '0', pero puede ser al revés, por eso mejor usar activo-inactivo. Veamos cómo se describe esto de forma serie (proceso) y concurrente:

Serie Hay varias posibilidades:

- Cuando en un proceso no se consideran todas las posibles asignaciones se pone un latch en esa señal:

```
-- biestable tipo D:
PROCESS(d,en)
BEGIN
   IF (en='1') THEN q<=d;
   END IF;
END PROCESS;
```

- Cuando no se especifican todas las señales en la lista sensible:

```
PROCESS(b)
BEGIN
   a<=d OR b; -- d esta 'latcheado' por b
END PROCESS;
```

Esta segunda forma no es la más adecuada en un caso real de síntesis, ya que muchos sintetizadores no lo interpretan bien, de hecho, supondrían

en la mayoría de los casos que `d` forma parte de la lista sensible y se realizaría como si fuera lógica combinacional.

Concurrente Suele ponerse un *latch* cuando la señal que está siendo asignada interviene en la asignación:

```
a<=b AND c WHEN h='1' ELSE a;
```

En este caso la señal `a` se encuentra *latcheada* por la señal `h`. En realidad esta expresión puede interpretarse también como lógica combinacional donde se da una realimentación, pero precisamente esta realimentación es la base para la realización de cerrojos y lógica secuencial (por ejemplo, los *flip-flops* están realizados internamente mediante puertas). Lo que sí que puede ocurrir es que dependiendo de la herramienta que se esté utilizando, lo anterior se realice mediante puertas o mediante un elemento de la biblioteca que sea un cerrojo o *latch*, siendo esta segunda opción la más común y óptima en general.

Descripción de señales de reloj La señal de reloj se define mediante la detección del flanco de subida o bajada de una señal. Normalmente se utiliza una condicional de manera que se detecte la siguiente condición:

```
clk'event AND clk='1'    -- Flanco de subida
clk'event AND clk='0'    -- Flanco de bajada
```

En VHDL no hay problema en definir un reloj que sea activo en ambos flancos, pero eso no es sencillo de sintetizar, por lo que si hay un reloj se debe utilizar uno sólo de los flancos. Además, y como se dijo al principio de este capítulo, en la mayoría de los casos sólo se permite una única señal de reloj por proceso.

Descripción de registros Los registros son como los *latches*, pero la entrada pasa a la salida cuando se produce un flanco de la señal de reloj:

```
PROCESS(clk,reset)
BEGIN
  IF reset='1' THEN q<='0';
  ELSIF clk'event AND clk='1'
    THEN q<=d;
  END IF;
END PROCESS;
```

```
PROCESS(clk)
BEGIN
  IF clk='1' THEN q<=d;
  END IF;
END PROCESS;
```

En el ejemplo de la derecha se observa que en realidad no hace falta especificar la condición de evento, ya que al estar la señal de reloj sola en la lista sensible, sólo se ejecuta el proceso si se produjo un evento en la señal. Ambos ejemplos son equivalentes y sintetizan casi lo mismo, un biestable maestro-esclavo (*master-slave*) tipo D, pero el de la izquierda incorpora una señal de réset que pone a cero el registro. En el de la izquierda es necesaria la condición completa de detección

de flanco puesto que, al haber dos señales en la lista sensible, no se sabe cuál es la que provocó la ejecución del proceso.

Consideraciones sobre la señal de reloj Para que un circuito secuencial sea sintetizado con éxito, se deben tener en cuenta algunas directrices que atañen sobre todo a la señal de reloj. Algunas ya se vieron en las recomendaciones iniciales, pero no viene mal recordarlas aquí:

- Sólo debe permitirse una detección de flanco por cada proceso, es decir, debe haber un único reloj por proceso. En realidad el problema viene de que un mismo circuito es difícil de sintetizar si está sincronizado mediante dos relojes diferentes. Normalmente cada proceso en una descripción en VHDL corresponde con una salida o señal interna del sistema, si se pusieran dos relojes en un mismo proceso significaría que esa señal viene sincronizada por dos señales diferentes, lo que implica realizar lógica sobre la señal de reloj que aunque es posible no es nada aconsejable. De esta manera se deja en manos del diseñador generar una única señal de reloj que pueda ser función de otras señales.
- Cuando en un `IF` se comprueba el flanco del reloj, no debe seguir un `ELSE`. Se podría poner pero desde un punto de vista de realización física del circuito no tendrá ningún sentido.
- El reloj, cuando se especifica con flanco, no debe ser usado como operando. Así la instrucción `IF NOT (clk'event AND clk='1') THEN...` sería incorrecta.

11.3 DESCRIPCIÓN DE MÁQUINAS DE ESTADOS

Es muy normal, a la hora de definir hardware, realizar la descripción siguiendo la definición de una máquina de estados. Una máquina de estados está definida por dos funciones, una calcula el estado siguiente en que se encontrará el sistema, y la otra calcula la salida. El estado siguiente se calcula, en general, en función de las entradas y del estado presente. La salida se calcula como una función del estado presente y las entradas.

Hay dos tipos de máquinas de estados, unas son las de *Mealy* y las otras son las de *Moore*. Las de Mealy son más generales y se caracterizan porque la salida depende del estado presente y la entrada. Las máquinas de Moore son un caso particular de las anteriores y se caracterizan porque la salida sólo depende del estado en que se encuentra el sistema.

En VHDL se pueden describir tanto máquinas de Mealy como de Moore y la estructura en ambas es bastante simple. Se muestra a continuación la forma general que tendría una posible, aunque no la única, descripción de una máquina de Moore donde la salida sólo depende del estado del sistema. Para ello se supondrá que `entrada` y `salida` son las entradas y salidas, y que (est1,est2,...,estN) son los estados del sistema. Suponiendo una máquina de estados que no venga sincronizada por un reloj, es decir, una máquina de estado asíncrona, una primera descripción sería:

```
ENTITY maquina IS
PORT (entrada: IN tipoin; salida: OUT tipout);
END maquina;

ARCHITECTURE moore OF maquina IS
TYPE estado IS (est1,est2,...,estN);
SIGNAL presente: estado:=est1;  -- especifica un estado inicial
SIGNAL siguiente: estado;
BEGIN
PROCESS(entrada,presente)
BEGIN
   CASE presente IS
      WHEN est1=>
          salida<=valor1;
          siguiente<=f1(entrada);
      WHEN est2=>
          salida<=valor2;
          siguiente<=f2(entrada);
      .
      .
      WHEN estN=>
          salida<=valorN;
          siguiente<=fN(entrada);
   END CASE;
END PROCESS;
presente<=siguiente;
END moore;
```

Esta descripción no es en realidad una máquina de estados aunque lo parece. De hecho, se puede observar que ni tan siquiera se trata de un circuito secuencial puro, ya que todas las señales que intervienen en asignaciones están en la lista sensible. En realidad tiene cierta secuencialidad que se obtiene de conservar el estado de `presente` frente a determinados cambios de la entrada. La secuencialidad se puede hacer explícita, si se elimina `presente` de la lista sensible, claro que en este caso la descripción de lo que se pretende hacer es algo confusa.

Aunque no se aconseja el uso de estas máquinas, se ha incluido aquí para mostrar que también es posible su descripción en VHDL. Volviendo a la descripción concreta de la máquina, se puede ver que los valores `valor1, valor2`, etc., son valores concretos que se le asignan a la salida. Las funciones `f1, f2`, etc., son en realidad expresiones que dependen de la entrada, y no significa que existan como tales funciones, simplemente

significa que existe un mecanismo mediante el cual el estado cambia frente a unas entradas dadas.

Para llevar el estado del sistema se han definido dos señales, por un lado `presente` para indicar el estado actual, y por otro la señal `siguiente` para indicar el estado siguiente. En realidad basta una sola señal para indicar el estado, lo cual es bastante sencillo de entender puesto que cuando se sintetice sólo va a existir un *latch* o registro que indique el estado, y no dos como aparentemente aparece en la descripción. Si se ha añadido la señal `siguiente` es por claridad, pero es evidente que es lo mismo que `presente` debido a la instrucción concurrente `presente<=siguiente`. De ahora en adelante se utilizará la señal de `presente` para indicar tanto el siguiente como el actual.

La máquina descrita anteriormente es un ejemplo de descripción de máquina de estados. Sin embargo en la práctica tiene un problema. Para empezar, a pesar de ser un circuito secuencial, sería implementado con puertas lógicas o cerrojos activos por nivel, lo cual puede presentar problemas de metaestabilidad. En efecto, aquí las transiciones entre estados vienen provocadas por los cambios en las entradas y en los estados. Si el cambio de estado no ocurre de forma instantánea en todos los bits que lo definan (lo cual no es difícil) o se producen picos o transiciones en las entradas (que tampoco es raro) el sistema puede acabar en un estado incorrecto. Es decir, se produce una *carrera crítica.*

Por esta razón, las máquinas de estados en circuitos reales vienen sincronizadas por una señal de reloj, de manera que la transición entre estados se da en uno de los flancos de la señal de reloj. Si se le añade una señal de reloj, a la que se le llamará `clk`, a la máquina anterior, la descripción quedaría:

```
ENTITY maquina IS
PORT (entrada: IN tipoin; clk: IN bit; salida: OUT tipout);
END maquina;

ARCHITECTURE moore_sincrono OF maquina IS
TYPE estado IS (est1,est2,...,estN);
SIGNAL presente: estado:=est1;        -- el inicial
BEGIN

estados:
PROCESS(clk)
BEGIN
   IF clk='1' THEN
     CASE presente IS
       WHEN est1=>
          presente<=f1(entrada);
       WHEN est2=>
          presente<=f2(entrada);
       .
       .
```

```
        WHEN estN=>
            presente<=fN(entrada);
      END CASE;
   END IF;
END PROCESS estados;

salida:
PROCESS(presente)
BEGIN
      CASE presente IS
        WHEN est1=>
            salida<=valor1;
        WHEN est2=>
            salida<=valor2;
          .
          .
        WHEN estN=>
            salida<=valorN;
      END CASE;
END PROCESS salida;
END moore_sincrono;
```

En este caso se ha separado la parte secuencial (proceso `estados`) de la combinacional (proceso `salida`). La parte secuencial se encarga de calcular el estado siguiente, y la parte combinacional pura calcula la salida en función del estado y la entrada.

Se ha mostrado que cada una de estas partes (secuencial y combinacional) se puede describir mediante dos procesos separados. Esto aumenta la claridad de la descripción, pero hay que tener cuidado de que el proceso combinacional sea realmente combinacional. En el caso de la máquina de Moore no hay problema, ya que la salida no depende de la entrada, entonces basta con poner `presente` en la lista sensible, pero con las máquinas de Mealy hay que tener cuidado, ya que la salida también depende de la entrada y hay que indicarla.

Si no se desea dividir la máquina en un proceso secuencial y otro combinacional, se puede incluir toda en un único proceso. Si se decide hacerlo así hay que tener mucho cuidado de separar la parte combinacional de la secuencial. De hecho, es muy fácil cometer el error de utilizar el mismo **CASE** de la asignación de estados para poner ahí la salida. ¡¡¡Esto es un error, ya que las salidas inferirán registros, serán síncronas, y tendrán un retraso de un ciclo de reloj respecto del cambio de estado!!! La descripción correcta sería:

```
fsm:
PROCESS(clk,presente)
BEGIN
   CASE presente IS
        WHEN est1=>
            salida<=valor1;
        WHEN est2=>
```

```
            salida<=valor2;
         .
         .
         WHEN estN=>
            salida<=valorN;
      END CASE;
      IF clk='1' THEN
        CASE presente IS
         WHEN est1=>
            presente<=f1(entrada);
         WHEN est2=>
            presente<=f2(entrada);
         .
         .
         WHEN estN=>
            presente<=fN(entrada);
        END CASE;
      END IF;
END PROCESS fsm;
```

Normalmente, las máquinas de estados suelen necesitar una señal de `reset` que lleve a la máquina a un estado conocido de partida. De hecho, es buena práctica, y debería ser obligatorio, poner un réset a cualquier descripción de un circuito que fuera secuencial.

Se muestra a continuación cómo sería la estructura de una máquina genérica que incorporara réset, reloj, y que fuera de tipo Mealy (la salida depende también de la entrada):

```
ENTITY maquina IS
PORT (entrada: IN tipoin; clk,reset: IN bit; salida: OUT tipout);
END maquina;

ARCHITECTURE mealy OF maquina IS
TYPE estado IS (est1,est2,...,estN);
SIGNAL presente: estado:=est1;          -- el inicial
BEGIN

--Bloque secuencial:
estados:
PROCESS(clk,reset)       -- réset asíncrono
BEGIN
   IF reset='1' THEN
     presente<=est1;     -- estado inicial
   ELSIF clk='1' AND clk'event THEN
     CASE presente IS
       WHEN est1=>
          presente<=f1(entrada);
       WHEN est2=>
          presente<=f2(entrada);
       .
       .
       WHEN estN=>
          presente<=fN(entrada);
     END CASE;
```

```
   END IF;
END PROCESS estados;

-- Bloque combinacional:
salida:
PROCESS(entrada,presente)
   CASE presente IS
     WHEN est1=>
        salida<=g1(entrada);
     WHEN est2=>
        salida<=g2(entrada);
     .
     .
     WHEN estN=>
        salida<=gN(entrada);
   END CASE;
END PROCESS salida;

END mealy;
```

En la descripción anterior el proceso `salida` no depende del réset puesto que, fijando el estado cuando se activa la señal de réset, se fija también la salida.

El réset anterior era asíncrono por encontrarse dentro de la lista sensible. Si se desea un réset síncrono, hay que quitarlo de la lista sensible e incorporarlo a la máquina como si se tratara de una entrada más.

La máquina de Mealy también se puede describir en un mismo proceso. Al igual que ocurría con Moore, aquí también hay que tener cuidado de separar la parte secuencial de la combinacional. La descripción en un único proceso sería:

```
fsm:
PROCESS(reset,clk,entrada,presente)
-- Combinacional:
   CASE presente IS
     WHEN est1=>
        salida<=g1(entrada);
     WHEN est2=>
        salida<=g2(entrada);
     .
     .
     WHEN estN=>
        salida<=gN(entrada);
   END CASE;

-- Secuencial:
   IF reset='1' THEN
     presente<=est1;     -- estado inicial
   ELSIF clk='1' AND clk'event THEN
     CASE presente IS
       WHEN est1=>
          presente<=f1(entrada);
       WHEN est2=>
          presente<=f2(entrada);
```

```
        .
        .
        WHEN estN=>
           presente<=fN(entrada);
     END CASE;
   END IF;
END PROCESS fsm;
```

Aunque estas descripciones en un único proceso son perfectamente válidas, se aconseja separar la parte combinacional de la secuencial en procesos diferentes. No sólo por claridad, sino porque las herramientas de síntesis suelen inferir mejor máquinas de estados cuando se describen separadas.

Hay que destacar, además, que cuando se separa en parte secuencial y combinacional, la puramente combinacional debe serlo de verdad. Tanto es así que algunas herramientas de síntesis añaden a la lista sensible todas las señales que necesiten para convertir ese proceso en combinacional puro.

Como complemento a esta parte del capítulo resulta muy interesante el dedicado a ejemplos y utilización del VHDL, donde se han incluido numerosas descripciones de funciones de la vida cotidiana, algunas para síntesis y otras para simulación. En los ejemplos de síntesis se han explicado las diferencias, si las había, entre la descripción simulada y la sintetizada.

11.4 LÓGICA PROGRAMABLE

Se ha visto que el proceso de síntesis consiste en partir de una descripción del circuito para luego sintetizarlo. El lenguaje VHDL es muy interesante como lenguaje de descripción para síntesis. Como complemento a lo que se ha visto hasta ahora, resulta interesante explicar las posibilidades de realización física de la descripción VHDL. Precisamente la utilización de un lenguaje le permite al diseñador de circuitos elegir entre una amplia variedad de posibles implementaciones sin necesidad de cambiar la descripción; por ello se ha incluido este último punto en este capítulo dedicado a la síntesis.

11.4.1 Tecnologías de programación

La aparición en los años ochenta de los primeros dispositivos lógicos programables, ha hecho posible optimizar el silicio utilizado en un sistema digital. Hasta la aparición de los primeros dispositivos lógicos programables, las únicas opciones para diseñar

Tecnología	Ventajas	Desventajas
Antifusible	No volatilidad, Alta integración	No reprogramabilidad
EEPROM	No volatilidad	No ISP
EPROM	No volatilidad	No ISP
SRAM	Reprogramabilidad, ISP	Volatilidad,
FLASH	Reprogramabilidad, No volatilidad, ISP	Baja integración

Tabla 11.1: Tecnologías de programación ventajas y desventajas

hardware digital eran, o bien el diseño de circuitos integrados hechos a medida, o bien la utilización de componentes estándar LSI-MSI conectados entre sí.

La primera opción es la preferible en cuanto a optimización de silicio, velocidad, tamaño y consumo de potencia. Los problemas de esta opción son la falta de flexibilidad y el alto coste de la fabricación de los circuitos *fullcustom* (a medida). La segunda opción tiene también el problema de la falta de flexibilidad para un diseño hardware, y el excesivo silicio no utilizado que ocupa espacio y consume potencia.

Los dispositivos lógicos programables lo son en el sentido de que en el circuito se tienen difundidos de forma fija los dispositivos lógicos, puertas lógicas y registros. Lo que se *programa* son las interconexiones entre estos dispositivos lógicos. De esta manera se puede tener la velocidad del hardware sin haber hecho un costoso circuito a medida. Basta con programar las conexiones internas de manera adecuada para que la PLD realice la función deseada.

Hay diversas formas de programar un dispositivo programable. Las diversas tecnologías de programación están íntimamente relacionadas con la arquitectura del dispositivo lógico programable. En la tabla 11.1 se resumen las principales características, ventajas y desventajas de cada tecnología. Cabe hacer notar que una misma característica, por ejemplo, la no reprogramabilidad, puede ser una ventaja o desventaja dependiendo de la aplicación. La característica ISP es la programabilidad en el sistema o *In System Programmability*, característica fundamental si se desea que el sistema sea reconfigurable de forma automática.

Se debe realizar un análisis de las diferentes clases de dispositivos para tener un criterio de selección del más adecuado.

11.4.2 PALs, PLDs y FPGAs clásicas

Programmable And Logic: PAL

Los dispositivos programables más simples son las matrices lógicas programables, conocidas como PALs (*Programmable And Logic*). La arquitectura de una PAL genérica se muestra en la figura 11.1.

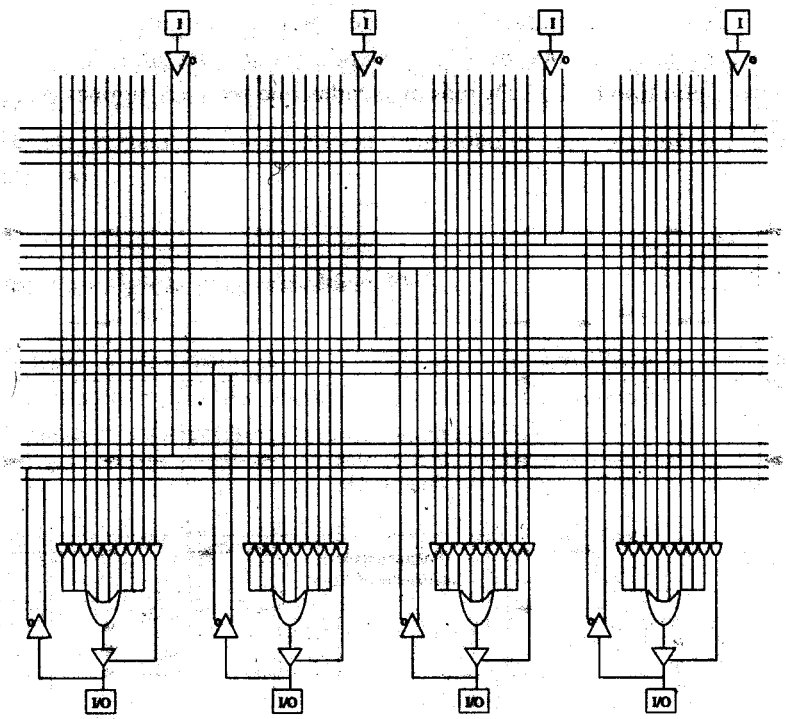


Figura 11.1: Arquitectura de una PAL genérica

Las salidas no son más que una función OR de varias líneas AND cableadas. Las lineas horizontales cruzan completamente la PAL posibilitando una conexión en cada cruce con una línea vertical. Las salidas a su vez se realimentan para hacer posible funciones más complejas que el número de minitérminos que caben en la función OR. En las salidas puede haber biestables, útiles para realizar sistemas secuenciales sencillos.

A partir de esta simple arquitectura se definen las PALs universales. La arquitectura de la matriz de ANDs cableadas y interconexiones con realimentación apenas cambia, la única diferencia estriba que en las salidas en vez de tener registros simples se tienen macroceldas. Las macroceldas a su vez incluyen un registro, de tipo D generalmente,

y lógica, para multiplexar la salida y la entrada a él. El ejemplo más popular de PAL universal es la PAL 22V10, en la que se pueden implementar máquinas de estados y sistemas secuenciales sencillos.

Dispositivos programables lógicos complejos: CPLDs

A partir de las PALs, que son los dispositivos más simples, se construyen las PLDs complejas o CPLDs. En la figura 11.2 se puede observar la arquitectura genérica de las CPLDs de la familia MAX7000 de Altera, siendo las arquitecturas de CPLDs muy similares (salvo algún tipo de CPLDs más avanzadas que recogen algunas de las ventajas de las FPGAs).

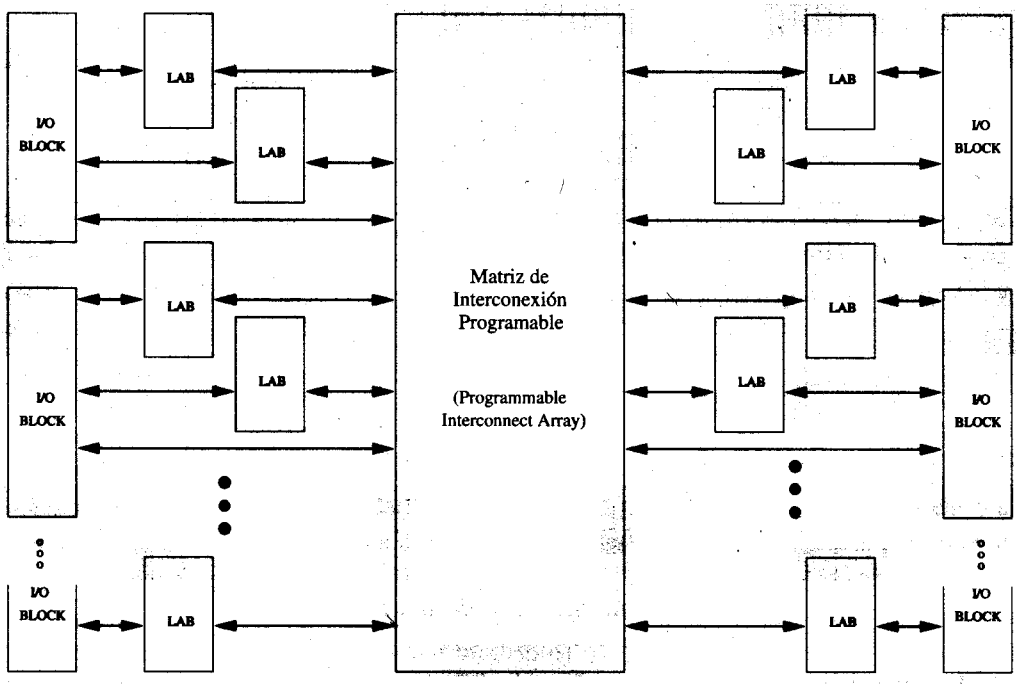


Figura 11.2: Arquitectura de una CPLD

Las CPLDs tienen unos bloques internos, llamados bloques de matrices lógicas o LABs (*Logic Array Blocks*), con una estructura similar a la arquitectura PAL mostrada en la figura 11.1. Además de estas pequeñas PALs, las CPLDs tienen una matriz de interconexión programable que permite el conexionado de los LABs entre sí y, por tanto, permite también la realización de sistemas más complejos que con las PALs. Junto con los LABs, las CPLDs tienen unos bloques especiales de entrada/salida que incorporan salidas triestado y sirven también para suministrar corriente.

Las CPLDs son dispositivos realmente complejos que pueden llegar a incorporar más de 20.000 puertas lógicas equivalentes con más de 50 LABs. El mercado de CPLDs cambia muy rápidamente, ofreciendo los fabricantes cada vez CPLDs más complejas y con mayores prestaciones de velocidad, integración y menor consumo de potencia. Cualquier análisis del estado de la tecnología a la escritura de este capítulo será superado unos pocos meses después.

La principal característica de las CPLDs son sus prestaciones. La arquitectura de las CPLDs hace que se puedan implementar funciones lógicas complejas con pocos niveles de realimentación y, por tanto, pocos retrasos. La arquitectura de las CPLDs también hace que se puedan predecir los retrasos de una manera sencilla, y así, el diseñador puede prever las prestaciones de su diseño (velocidad de funcionamiento) incluso antes de implementarlo.

Redes de puertas lógicas programables: FPGAs

Paralelamente al desarrollo de las CPLDs se han desarrollado las redes de puertas lógicas programables o FPGAs (*Field Programmable Gate Arrays*). En la figura 11.3 puede observarse la arquitectura genérica de una FPGA clásica.

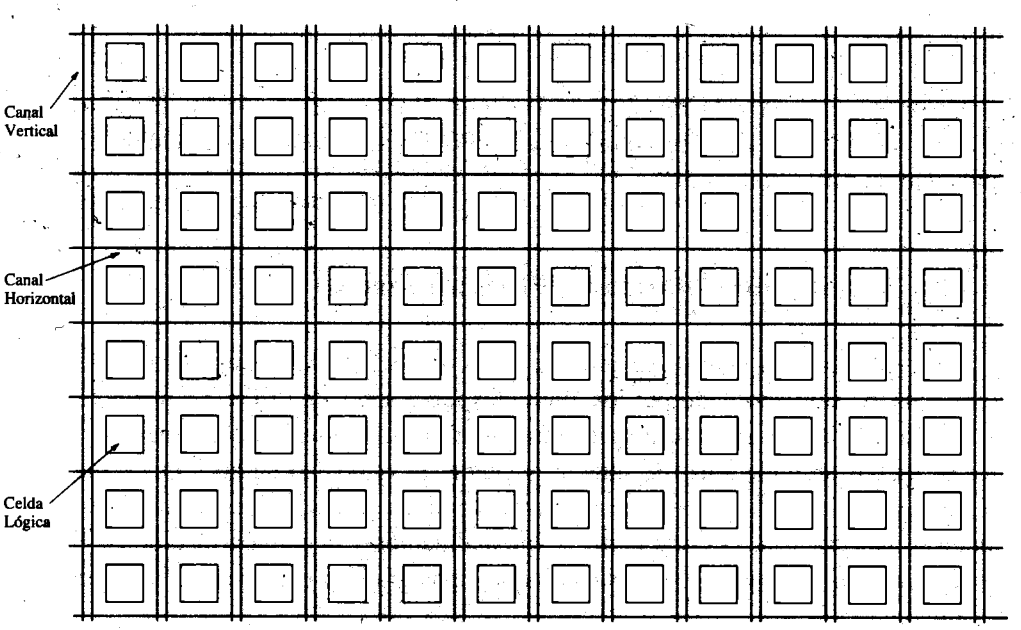


Figura 11.3: Arquitectura de una FPGA genérica

Al contrario de las CPLDs, aquí no se tienen varios bloques complejos como los LABs, ni una matriz de interconexión localizada; en las FPGAs se tienen muchas pequeñas celdas lógicas distribuidas regularmente por su superficie. Cada celda lógica incluye uno o dos registros y unas pocas pequeñas LUTs (*lookup tables*) que pueden implementar funciones combinacionales sencillas. Las celdas lógicas suelen incorporar también algún multiplexor para interconectar las LUTs con los registros.

El conexionado entre las diversas celdas lógicas no se centraliza en una matriz de interconexión, sino que se realiza mediante canales de interconexión. Estos canales están distribuidos uniformemente entre las celdas lógicas y cruzan la superficie de la FPGA horizontal y verticalmente.

La ventaja de esta arquitectura regular es que, aprovechando la alta escala de integración de la tecnología VLSI, se pueden realizar dispositivos muy complejos, bastante más que con las PLDs sencillas. Actualmente las FPGAs más grandes pueden tener una complejidad de hasta 250.000 puertas equivalentes con más de 10.000 registros y celdas lógicas, eso sí, el número de puertas lógicas se dobla cada dos o tres años, por lo que la afirmación anterior posiblemente está anticuada.

La desventaja de la arquitectura FPGA parece clara a la vista de la figura 11.3; para realizar un sistema complejo se deben realizar muchas conexiones entre las celdas lógicas, generando caminos que pueden ser largos y tener realimentaciones profundas, lo que provoca que puedan haber grandes retrasos impredecibles a priori.

Como conclusión a la revisión de la arquitectura genérica de las FPGAs, y las CPLDs, se tiene que a mayor complejidad (FPGAs) se obtiene una mayor perdida de velocidad de funcionamiento.

11.4.3 Arquitecturas híbridas (CPLDs avanzadas)

El problema de la pérdida de prestaciones con las FPGAs al implementar sistemas complejos, ha sido abordado por los diferentes fabricantes de PLDs. La nueva filosofía de diseño consiste en realizar arquitecturas híbridas que intentan combinar los beneficios de ambas arquitecturas. Se intenta mantener la velocidad de las CPLDs mediante una nueva arquitectura de conexionado, agrupando las celdas lógicas elementales en estructuras más grandes. Por otra parte, se intenta mantener la granularidad de las FPGAs difundiendo un gran número de estos bloques más grandes entre las líneas de interconexión.

En la figura 11.4 se muestra, como ejemplo, la arquitectura híbrida de una CPLD de la familia FLEX 8000 de Altera.

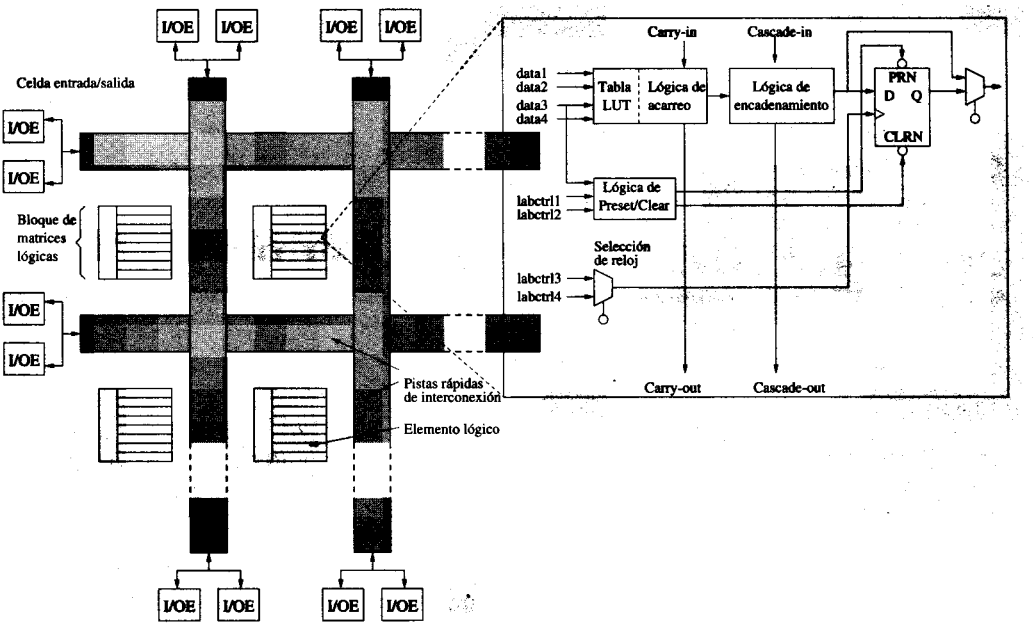


Figura 11.4: Arquitectura de un dispositivo programable híbrido

Se puede observar que en esta arquitectura las celdas lógicas están agrupadas en bloques, que en el caso de la figura son de 8 elementos lógicos. Esta agrupación por bloques facilita el agrupamiento de las señales y la aparición de buses durante la síntesis. En el caso de que se traten señales de 8 bits, un solo bloque cubriría una señal, evitando la aparición de retrasos distintos en los diferentes bits de la señal.

Además, la estructura de los elementos lógicos incluye líneas especiales de acarreo adelantado y de encadenamiento. De esta manera se mejora la velocidad de contadores y otros dispositivos lógicos con realimentaciones y acarreo. La interconexión de los bloques de matrices lógicas se realiza mediante pistas rápidas de interconexión similares a las matrices de interconexión de las CPLDs.

Se puede concluir que las arquitecturas híbridas (CPLDs avanzadas) combinan cierta granularidad y flexibilidad, al igual que las FPGAs, con la agrupación por bloques, y por tanto rapidez, de las CPLDs.

CAPÍTULO 12

UTILIZACIÓN DEL LENGUAJE VHDL

En esta sección se muestran unos cuantos ejemplos de utilización del VHDL para simulación y síntesis. Cuando se explica un lenguaje de programación, y con los de descripción de circuitos pasa igual, resulta difícil explicar cómo resolver un problema mediante el lenguaje. Parece que la mejor forma sigue siendo dar unos cuantos ejemplos de cómo resolver determinados problemas y a partir de ahí poder coger soltura con el lenguaje.

12.1 ERRORES MÁS COMUNES USANDO VHDL

Antes de empezar con los ejemplos resulta interesante dar algunas recomendaciones, aparte de las que se han dado ya, para así evitar el tropezar varias veces en el mismo problema o error de diseño. A continuación se recogen una serie de errores que se suelen cometer al empezar a programar o describir circuitos con VHDL.

- El error más común es probablemente la asignación de una misma señal en procesos diferentes, o lo que es lo mismo, **asignación de una misma señal en instrucciones concurrentes diferentes**. Este error viene de que a veces se divide el problema atendiendo a las entradas en vez de a las salidas, con lo que se pone un proceso para cada entrada o grupo de entradas, con lo que las salidas, que dependerán en general de varias entradas, aparecen en varios procesos al mismo tiempo. La solución consiste en dividir el problema por las salidas y no por las entradas, de manera que en cada proceso se integre toda la lógica referida a una salida o a un grupo de salidas relacionadas. Naturalmente, la utilización de tipos resueltos no es una solución, ya que el problema, que hasta ahora era únicamente

de compilación, se convierte en un problema de diseño mucho más difícil de depurar. El tipo resuelto debe utilizarse únicamente en buses donde se conectan varios dispositivos.

Otra causa de este problema se da en los contadores. Normalmente se tiene un proceso que se encarga de incrementar el contador y, en otro proceso, normalmente el de la máquina de estados, es preciso ponerlo a cero en determinados estados. En este caso no se puede poner a cero en el otro proceso, sino que hay que utilizar una señal auxiliar para poder comunicar ambos procesos entre sí, de manera que si un proceso quiere poner a cero el contador, lo que tiene que hacer es activar esta señal para que el proceso que tiene el contador se dé por enterado y ponga a cero la cuenta.

- Otro error bastante común es pensar que las señales se comportan como variables dentro de los procesos. Así, si se tiene un contador en un proceso, y poco después de incrementarlo se compara a ver si ha llegado a cierto valor en el mismo proceso, probablemente no funcionará bien puesto que la señal no habrá sido actualizada. Colocando la señal de contador en la lista sensible probablemente solucionaría el problema en algunos casos, pero lo normal es que la funcionalidad del proceso cambiara, especialmente si viene sincronizado por una señal de reloj que es el caso más común.

- En el caso de estar describiendo botones, y en general, pulsos de entrada con duración impredecible, a veces se olvida que el botón se mantiene pulsado durante algún tiempo que normalmente es mucho mayor que la frecuencia del reloj y, por supuesto, mucho mayor que los tiempos de respuesta de los circuitos. Esto significa que cuando un botón produce un cambio de estado, sigue estando pulsado en ese estado nuevo que entra. Por lo tanto, se debe tener esto en cuenta, bien añadiendo estados auxiliares o bien, y esto funcionará bien en cualquier caso, definiendo una señal que se activará al pulsarse el botón y que se desactivará al entrar en el estado siguiente. Sincronizar con un reloj también ayuda a solucionar el problema.

- Del mismo estilo del anterior es el problema que surge cuando se pulsa un botón y se suelta enseguida. Hay máquinas que si se describen mal están suponiendo que el botón está continuamente pulsado, y esto no tiene por qué ser así, sería el caso contrario al anterior. En estas situaciones lo que hay que hacer es capturar la pulsación del botón a través de un registro y volverlo a desactivar cuando se llegue a un estado inicial.

- Ya menos frecuentemente, a veces ocurre que se olvidan cosas por asignar. En el caso de las máquinas de estados es importante que al menos el estado inicial, o de réset, contenga todas las señales de salida con unos valores fijos.

- Sólo muy al principio, cuando no se tiene muy clara la diferencia entre el entorno concurrente y el serie, se suele considerar que en el entorno concurrente las instrucciones también se ejecutan una detrás de otra, y no es raro ver cómo se "inicializan" unas señales a cero, y luego, se les da otro valor, etc.

- A veces crea confusión la diferencia entre variable y señal hasta el punto que se declaran variables o señales en lugares que no les corresponden. Como norma general, que sirve para 99% de los casos, y que casi conviene para no armarse mucho lío sobre todo al principio, se puede decir que las señales sólo se pueden declarar en la parte declarativa de la arquitectura, y que las variables sólo se pueden declarar en las partes declarativas de procesos, funciones y procedimientos. Las señales también se pueden declarar en los bloques concurrentes, pero como esta estructura se usa poco al principio, casi conviene no saberlo hasta que de verdad se empiezan a usar.

- Como corolario de lo anterior también se da el problema de usar variables en entornos concurrentes, lo cual no es posible ya que ni siquiera se pueden declarar ahí.

De los errores más comunes, los que realmente se dan con frecuencia y son más fáciles de cometer son los dos primeros. A continuación se muestra cómo resolver algunos de los problemas de diseño que se pueden plantear en VHDL.

12.2 EJEMPLOS PARA SIMULACIÓN Y SÍNTESIS

El estilo de realización de modelos (simulación) es bastante diferente del estilo empleado para la síntesis de circuitos. Para empezar, en el modelado no hay restricciones de ningún tipo y además los modelos suelen incluir información referente a los retrasos. A continuación se muestran las diferencias, cuando las haya, entre lo que se sintetizaría y lo que se simularía, y se verá que muchas veces no coincide.

12.2.1 El botón

Ejemplo 12.1 *Un motor eléctrico viene controlado por un único botón. Cuando se pulsa el botón, el motor cambia de encendido a apagado. Sintetizar el circuito que controla el motor mediante una máquina de estados en VHDL.*

La solución a este problema es bastante simple tal y como ya se mostró en el ejemplo 6.3 donde con un simple biestable se solucionaba el problema. Desde un punto de vista algo más abstracto se puede pensar en una máquina de estados con dos estados, de manera que se pasa de uno a otro cada vez que se pulsa el botón. Esto en principio se puede hacer, pero tiene un problema y es que cuando se pasa de un estado a otro el botón sigue pulsado, por lo que en realidad se produce una transición muy rápida entre estados. Sólo cuando se suelte el botón se parará, pero es imposible predecir si se parará en el estado encendido o en el apagado. Para evitar esto lo normal es pensar en dos estados más que detengan esta transición rápida entre estados. La salida sólo dependerá del estado del sistema, por tanto no es más que una máquina de Moore:

```
ENTITY conmutador IS
PORT (boton: IN bit; motor: OUT bit);
END conmutador;

ARCHITECTURE moore OF conmutador IS
  TYPE estado IS (apagado1,apagado2,encendido1,encendido2);
  SIGNAL presente: estado:=apagado1;
BEGIN
  PROCESS(boton,presente)
  BEGIN
     CASE presente IS
       WHEN apagado1 =>
         motor<='0';
         IF boton='1' THEN presente<=encendido2;
         END IF;
       WHEN encendido2 =>
         motor<='1';
         IF boton='0' THEN presente<=encendido1;
         END IF;
       WHEN encendido1 =>
         motor<='1';
         IF boton='1' THEN presente<=apagado2;
         END IF;
       WHEN apagado2 =>
         motor<='0';
         IF boton='0' THEN presente<=apagado1;
         END IF;
     END CASE;
  END PROCESS;
END moore;
```

También se puede resolver utilizando menos estados tal y como se muestra a continuación, pero no es cierto que hayan menos estados, aparentemente hay menos porque el sintetizador del circuito introducirá *latches* extra. Además, la descripción que sigue, aunque pudiera parecer que tiene la estructura de una máquina de estados, no lo es exactamente porque en la lista sensible no se ha introducido la señal que contiene el estado. Esto significará que se realiza lógica secuencial que no aparece explícitamente en la descripción. En realidad el *truco* está en que la máquina anterior se realizaría con biestables activos por nivel, mientras que la que viene a continuación, como no tiene

presente en la lista sensible se activaría por flanco, por lo que se utilizaría un único biestable maestro-esclavo, pero para obtener un biestable maestro-esclavo hacen falta precisamente dos biestables activos por nivel:

```
ARCHITECTURE pseudomaquina OF conmutador IS
  TYPE estado IS (apagado,encendido);
  SIGNAL presente: estado:=apagado;
BEGIN
  PROCESS(boton)
  BEGIN
     CASE presente IS
       WHEN apagado =>
         motor<='0';
         IF boton='1' THEN
             presente<=encendido;
             motor<='1';  -- Esto es salida futura, por tanto, opuesta.
        END IF;
       WHEN encendido =>
         motor<='1';
         IF boton='1' THEN
             presente<=apagado;
             motor<='0';  -- Lo mismo, salida futura.
      END IF;
     END CASE;
  END PROCESS;
END pseudomaquina;
```

Este segundo caso no se sintetizaría bien puesto que a las herramientas de diseño hay que especificarles qué partes son activas por flanco de forma explícita, generalmente con el atributo 'event. Si se introdujera esta descripción en un sintetizador y luego se simulara lo sintetizado, se vería que efectivamente tiene dos estados, pero al pulsar el botón cambia entre estados de forma rápida tal y como se predijo al principio. En cambio, si se simula la descripción tal y como está, funcionaría bien. Aparte de todo esto, el ejemplo anterior no es precisamente un buen modelo de máquina de estados, ya que la señal de sincronización, en este caso el botón, se encuentra en cada uno de los estados, y por otro lado, hay algo que no se debería hacer nunca, y es cambiar la salida al tiempo que cambia el estado para que así el estado siguiente tenga la salida que se le ha especificado. En general, cada estado debería tener especificadas sus salidas. Para que un sintetizador hubiera interpretado la descripción anterior como lo que realmente se indica, habría que haberlo hecho así:

```
ARCHITECTURE para_sintesis OF conmutador IS
  TYPE estado IS (apagado,encendido);
  SIGNAL presente: estado:=apagado;
BEGIN
  PROCESS(boton)
  BEGIN
     IF boton='1'     -- o boton='1' AND boton'event
       CASE presente IS
```

```
        WHEN apagado =>
          motor<='0';
          presente<=encendido;
         END IF;
        WHEN encendido =>
          motor<='1';
          presente<=apagado;
        END CASE;
      END IF;
  END PROCESS;
END para_sintesis;
```

Si se repasa la descripción anterior y se intenta simular a mano con todo lo que sabe de VHDL, se puede observar que cuando el estado es `apagado`, el motor vale uno, y viceversa, es decir, lo contrario de lo que parece. Si esto causa mucha confusión, se puede dividir el problema en dos procesos, uno que interpreta el estado y otro el cambio de estado:

```
maquina:
PROCESS(boton)
BEGIN
  IF boton='1' THEN
    CASE presente IS
    WHEN apagado=>
      presente<=encendido;
    WHEN encendido=>
      presente<=apagado;
    END CASE;
  END IF;
END PROCESS maquina;
```

```
salida:
PROCESS(presente)
BEGIN
  CASE presente IS
  WHEN apagado=>
    motor<='0';
  WHEN encendido=>
    motor<='1';
  END CASE;
END PROCESS salida;
```

Esta descripción es más interesante, ya que en este caso está más claro lo que se pretende decir, además, tanto la simulación como la síntesis coinciden. Quizá alguien podría pensar que una posible solución sería poner `presente` en la lista sensible, pero esto, aunque en la simulación estaría bien, sintetizaría otro circuito diferente. Esto es, que es aconsejable seguir un único modelo para la máquina de estados que funcione bien para síntesis.

Por último comentar que no es aconsejable utilizar el modelo de máquina de estados para este tipo de problemas que no vienen sincronizados con una señal de reloj. Ya cuando se vio el ejemplo del motor en el ejemplo 6.3, se mostró que bastaba con un biestable para resolver el problema. Por otro lado, las herramientas de síntesis no resuelven bien las máquinas de estados que no vienen sincronizadas por una señal de reloj. Así que, aunque se ha comentado aquí esta máquina y se ha puesto un ejemplo, no es una buena práctica de descripción.

12.2.2 Los semáforos

Ejemplo 12.2 *Realizar el circuito de control de unos semáforos que se encuentran en un cruce entre un camino rural y una carretera. En principio, el semáforo del camino rural siempre está en rojo y el de la carretera en verde. Una célula en el camino rural detecta la presencia de un coche, momento en el cual el semáforo de la carretera pasa de verde a rojo pasando por el ámbar, al tiempo que el semáforo del camino se pone en verde. El semáforo del camino permanece en verde unos 10 segundos, momento en el cual empieza la secuencia de puesta a rojo, al tiempo que el semáforo de la carretera empieza la secuencia de cambio hacia el verde. El semáforo del camino no debe ponerse en verde otra vez hasta transcurridos 30 segundos por lo menos. El circuito tiene una entrada de reloj de 1 segundo de periodo y las señales de entrada y salida suficientes para el control del semáforo.*

Como se da una señal de reloj como entrada, es interesante realizar la máquina de estados de manera que sea síncrona con este reloj, de esta manera se evitan problemas de metaestabilidad con las entradas, además de que las herramientas de síntesis interpretarán mejor que el circuito es una máquina de estados y el resultado será más óptimo. Junto con la máquina de estados habrá otros procesos que controlen los tiempos de espera mediante contadores.

Las entradas al sistema serán una señal de réset asíncrona, que es lo habitual, y las fotocélulas del camino, que indicarán un `'1'` cuando detecten un coche. Las salidas serán un total de 6: 3 para cada semáforo, indicando cada una de estas tres el color rojo, ámbar y verde.

La máquina de estados de la descripción se muestra en la figura 12.1. En cada estado se ha puesto lo que hace el contador; no se han puesto las salidas de los semáforos porque resultan evidentes dado el nombre del estado. Se han puesto también las señales que cambian el estado; si no se pone nada es que el cambio se realiza siempre. Esto funciona ya que hay estados donde hay que esperar un segundo, por tanto, basta esperar al siguiente ciclo de reloj.

La figura 12.2 muestra el diagrama de bloques de la descripción. Se muestra que el circuito está compuesto por la máquina de estados, el contador, y dos comparadores.

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

ENTITY semaforo IS
PORT (sensor,reset,clk: IN std_logic;
```

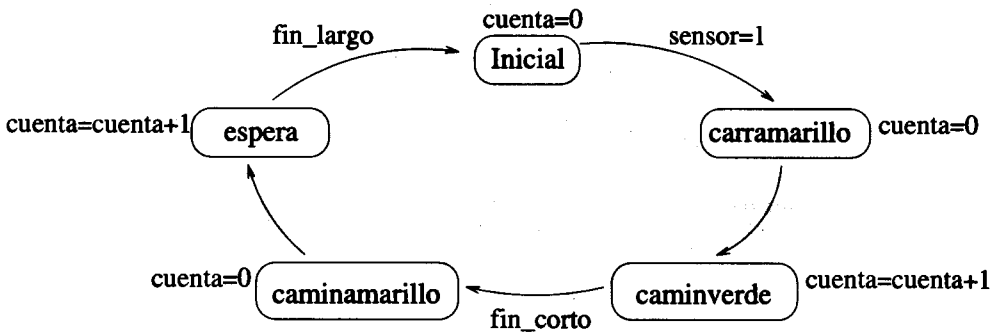


Figura 12.1: Máquina de estados del problema del semáforo

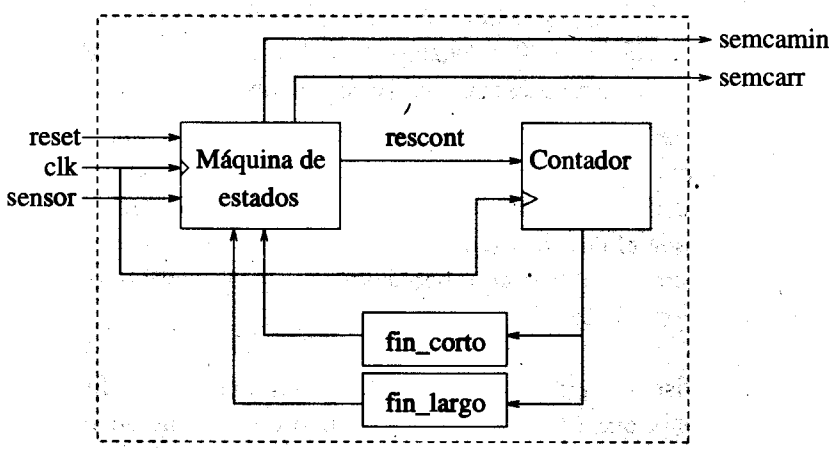


Figura 12.2: Diagrama de bloques del problema del semáforo

```
      semcamin,semcarr: OUT std_logic_vector(0 TO 2));
END semaforo;

ARCHITECTURE descripcion OF semaforo IS
  TYPE estado IS
     (inicial,carramarillo,caminverde,caminamarillo,espera);
  CONSTANT verde:     std_logic_vector(0 TO 2):="001";
  CONSTANT amarillo:  std_logic_vector(0 TO 2):="010";
  CONSTANT rojo:      std_logic_vector(0 TO 2):="100";
  SIGNAL presente: estado:=inicial;
  SIGNAL rescont: boolean:=false;       -- Pone a cero la cuenta
  SIGNAL fin_largo,fin_corto: boolean; -- Indica fin de  cuenta
  SIGNAL cuenta: integer RANGE 0 TO 63;
BEGIN

-- Lo primero es definir la máquina de estados:
maquina:
PROCESS(clk,reset)
BEGIN
  IF reset='1' THEN
```

```
    presente<=inicial;
  ELSIF clk='1' AND clk'event THEN
     CASE presente IS
       WHEN inicial=>
          IF sensor='1' THEN
             presente<=carramarillo;
          END IF;
       WHEN carramarillo=>
             presente<=caminverde;
       WHEN caminverde=>
          IF fin_corto THEN
             presente<=caminamarillo;
          END IF;
       WHEN caminamarillo=>
             presente<=espera;
       WHEN espera=>
          IF fin_largo THEN
             presente<=inicial;
          END IF;
     END CASE;
  END IF;
END PROCESS maquina;

salida:
PROCESS(presente)  -- No depende de las entradas
BEGIN
     CASE presente IS
       WHEN inicial=>
          semcarr<=verde;
          semcamin<=rojo;
          rescont<=true;
       WHEN carramarillo=>
          semcarr<=amarillo;
          semcamin<=rojo;
          rescont<=true;
       WHEN caminverde=>
          semcarr<=rojo;
          semcamin<=verde;
          rescont<=false;
       WHEN caminamarillo=>
          semcarr<=rojo;
          semcamin<=amarillo;
          rescont<=true;
       WHEN espera=>
          semcarr<=verde;
          semcamin<=rojo;
          rescont<=false;
     END CASE;
END PROCESS salida;

-- El siguiente proceso define el contador:
contador:
PROCESS(clk)
BEGIN
   IF clk='1' THEN
     IF rescont THEN cuenta<=0;
     ELSE cuenta<=cuenta+1;
     END IF;
   END IF;
END PROCESS contador;

-- Queda la detección de los tiempos largos y cortos:
```

```
fin_largo<=true WHEN cuenta=29 ELSE false;
fin_corto<=true WHEN cuenta=9  ELSE false;

END descripcion;
```

12.2.3 El ascensor

Ejemplo 12.3 *Describir el controlador de un ascensor único en una vivienda de 4 pisos. Las entradas al circuito serán, por un lado, el piso al que el usuario desea ir mediante 4 botones, y el piso en el que se encuentra el ascensor en un momento dado. Por otro, habrá una célula que detecte la presencia de algún obstáculo en la puerta, si hay un obstáculo la puerta no debe cerrarse. Hay dos salidas: por un lado la del motor (2 bits), y por otro la de la puerta (1 bit). El funcionamiento es bien simple: el ascensor debe ir al piso indicado por los botones, cuando llegue abrirá las puertas que permanecerán así hasta que se reciba otra llamada. El ascensor no tiene memoria, por lo que si se pulsan los botones mientras el ascensor se mueve no se debe hacer caso.*

La forma más sencilla es realizar la descripción a partir de una máquina de estados. A continuación se presenta una primera posibilidad que, aunque se trata de una máquina de estados, lleva un registro oculto:

```
ENTITY ascensor IS
PORT(boton: IN bit_vector(0 TO 3);
     piso:  IN bit_vector(1 DOWNTO 0);
     clk,reset,celula: IN bit;
     motor:  OUT bit_vector(0 TO 1);
     puerta: OUT bit);
END ascensor;

ARCHITECTURE mover OF ascensor IS
   TYPE estado IS (inicial,cerrar,voy);
   SIGNAL presente: estado:=inicial;
   SIGNAL bot: bit_vector(1 DOWNTO 0); -- Almacena el botón pulsado

FUNCTION codifica(pulso: bit_vector(0 TO 3)) RETURN bit_vector IS
BEGIN
   CASE pulso IS
     WHEN "0001"=>RETURN "00";
     WHEN "0010"=>RETURN "01";
     WHEN "0100"=>RETURN "10";
     WHEN "1000"=>RETURN "11";
     WHEN OTHERS=>RETURN "00";
   END CASE;
END codifica;

BEGIN

fsm:
PROCESS(reset,clk)
```

```
BEGIN
   IF reset='1' THEN presente<=inicial;
   ELSIF clk='1' AND clk'event THEN
     CASE presente IS
       WHEN inicial=>
          IF boton/="0000" AND bot/=piso THEN presente<=cerrar;
          END IF;
       WHEN cerrar=>
          IF celula='0' THEN presente<=voy;  -- Sin obstáculos
          END IF;
       WHEN voy=>
          IF bot=piso THEN presente<=inicial;
          END IF;
     END CASE;
   END IF;
END PROCESS fsm;

salida:
PROCESS(presente,boton)
BEGIN
   CASE presente IS
     WHEN inicial=>
        motor<="00";              -- Parado
        puerta<='1';              -- Abierta
        bot<=codifica(boton);     -- Captura el botón
     WHEN cerrar=>
        motor<="00";
        puerta<='1';
     WHEN voy=>
        puerta<='0';              -- Cerrada
        IF bot>piso THEN
             motor<="10";         -- Subir
        ELSE motor<="01";         -- Bajar
        END IF;
   END CASE;
END PROCESS salida;

END mover;
```

El funcionamiento no es muy complejo. Si no se pulsa nada, se mantiene en el estado inicial; si se pulsa, y el botón que se pulsó corresponde a un piso diferente, entonces se cierran las puertas y el motor se pone en marcha en dirección al piso que se llamó. Cuando llega se abren las puertas y se queda a esperar una nueva llamada.

La función `codifica` se ha puesto para mostrar la inclusión de una función en una descripción. Realmente el programa funciona exactamente igual de bien, con pequeñas modificaciones, si se utiliza `bot` como una señal de 4 bits. En cualquier caso, `bot` sigue siendo necesaria puesto que se encarga de capturar la pulsación del botón.

Como se dijo al inicio, esta máquina de estados tiene un registro oculto. De hecho, el proceso de salida, que debería ser combinacional puro, no lo es. El registro es precisamente la señal `bot` que realiza la captura del botón pulsado. En efecto, en el estado inicial se carga el botón, según lo que haya pulsado, y cuando se sale del estado inicial

no cambia, por lo que queda almacenado. Normalmente una herramienta de síntesis entenderá efectivamente que esto es así. En caso de duda conviene poner el cerrojo fuera para que la síntesis de la máquina de estados sea lo más precisa posible. En ese caso, se quitaría el `bot<=codifica(boton)` del proceso `salida`, del estado `inicial` y se pondría en un proceso aparte:

```
cerrojo:
PROCESS(presente)
BEGIN
  IF presente=inicial THEN bot<=codifica(boton);
  END IF;
END PROCESS cerrojo;
```

Pero esta solución, aunque buena, todavía puede tener problemas. Por ejemplo, normalmente la señal presente estará codificada con varios bits, de los cuales no todos tienen el mismo retraso, y eso quiere decir que entre un estado y otro puede pasar por estados intermedios. Este ejemplo es muy simple y difícilmente va a ocurrir algo anormal, pero es una buena práctica sincronizar con una única señal lo más limpia posible. Por tanto, la propuesta buena sería:

```
registro:
PROCESS(clk)
BEGIN
  IF clk='1' THEN     -- Flanco de subida del reloj
    IF presente=inicial THEN bot<=codifica(boton);
    END IF;
  END IF;
END PROCESS registro;
```

El ejemplo del ascensor que se acaba de mostrar no es demasiado realista, por un lado las puertas se cierran de golpe y, por otro, la parada y puesta en marcha del ascensor es también muy brusca. De todas formas pone de manifiesto la capacidad de funcionamiento del VHDL para la descripción de hardware. Como ejercicio adicional se puede hacer el ejemplo anterior pero añadiéndole características más realistas como la detección de obstáculos durante el cierre de puertas, o la posibilidad de gestionar más de un botón pulsado.

12.2.4 La memoria ROM

Ejemplo 12.4 *Realizar el modelo de simulación de una memoria ROM simple. La ROM tiene una entrada de selección activa a nivel bajo, de manera que, cuando está*

activa, la salida es el contenido de la posición indicada por la dirección de entrada, si no está activa, la salida es alta impedancia. El tiempo que pasa entre que cambia la habilitación y la salida se habilita es de 60 ns. El tiempo que pasa entre que la dirección cambia y cambia la salida es de 100 ns. En el caso de cambio en la dirección, la salida mantiene su valor anterior durante 10 ns. y luego pasa a desconocido.

Como es una descripción para modelar la memoria, existe plena libertad para manejar el lenguaje como se desee. De todas formas, el ejemplo es tan simple que no sirve de mucho tanta libertad. La descripción se haría de la siguiente manera:

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

ENTITY rom IS
PORT( cen: IN std_logic;
      direcc: IN std_logic_vector(1 DOWNTO 0);
      dato: OUT std_logic_vector(7 DOWNTO 0));
END rom;

ARCHITECTURE modelo OF rom IS
   SIGNAL salida: std_logic_vector(7 DOWNTO 0);
   SIGNAL cenr: std_logic;
BEGIN

PROCESS(direcc)
BEGIN
-- Sea cual sea la dirección, a los 10 ns. de cambiar, debe
-- pasar a desconocido:
   salida<="XXXXXXXX" AFTER 10 ns;
   CASE direcc IS
     WHEN "00"=>salida<=TRANSPORT "00000000" AFTER 100 ns;
     WHEN "01"=>salida<=TRANSPORT "00000001" AFTER 100 ns;
     WHEN "10"=>salida<=TRANSPORT "01010101" AFTER 100 ns;
     WHEN "11"=>salida<=TRANSPORT "10101010" AFTER 100 ns;
     WHEN OTHERS=> NULL;
   END CASE;
END PROCESS;

-- Éste es el buffer triestado:
dato<=salida          WHEN cenr='0' ELSE
      (OTHERS => 'Z') WHEN cenr='1' ELSE
      (OTHERS => 'X');

-- Éste es el retraso del chip enable
cenr<=cen AFTER 60 ns;

END modelo;
```

El modelo es bastante simple. Quizá sea interesante resaltar que para el caso del retraso de 100 ns. de la salida se ha empleado el retraso de tipo transportado en vez del inercial, la razón es que este evento se asigna al mismo tiempo que el de 10 ns., de

manera que si no fuese transportado quitaría el otro evento de la lista de eventos y no se ejecutaría nunca.

En la figura 12.3 se muestra la simulación de un par de accesos a memoria, incluyendo la habilitación y deshabilitación de ésta.

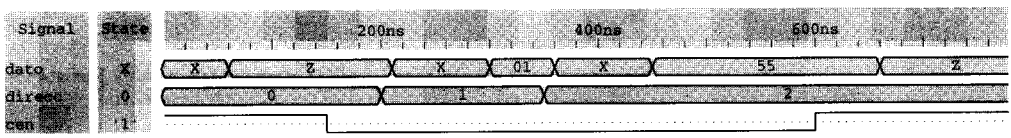


Figura 12.3: Cronograma del ejemplo de la ROM

12.2.5 El microprocesador

Ejemplo 12.5 *Realizar un microprocesador sencillo. El procesador tiene un bus de datos bidireccional de 8 bits. Un bus de direcciones de salida de 8 bits. Una señal lectura_escritura que a uno indica lectura y a cero escritura. El procesador se completa con una señal de reloj y otra de réset. Internamente debe haber un acumulador de 8 bits, el registro de instrucciones de 3 bits, y el contador de programa* program counter*) de 8 bits. El micro cuenta con 8 instrucciones. Las instrucciones están formadas por dos bytes, en el primero se indica el código de operación, y en el segundo el operando, salvo en la última que sólo tiene un byte. A continuación se muestran las instrucciones junto con su codificación:*

ld a,(xx) *Carga el acumulador con lo que haya en la posición de memoria indicada por el operando. (000)*

ld (xx),a *Carga en la posición xx el contenido del acumulador. (001)*

and a,(xx) *Realiza la operación* `and` *entre el acumulador y lo que haya en la posición xx. El resultado se guarda en el acumulador. (010)*

add a,(xx) *Lo mismo que la anterior pero la operación es la suma. (011)*

sub a,(xx) *Al acumulador se le resta lo que haya en la posición xx. El resultado se guarda en el acumulador. (100)*

jz xx *Salta a la posición xx si el acumulador es cero. (101)*

jmp xx *Salta a la posición xx. (110)*

nop *No hace nada. (111)*

Realizar un procesador no encauzado es relativamente sencillo en VHDL. Además, todos tienen una estructura parecida por lo que resulta fácil añadir instrucciones y hacer el procesador tan complicado como se desee. En el caso simple del procesador propuesto, se puede abordar el problema con una simple máquina de estados, en la cual hay un estado inicial de réset al que le sigue el de búsqueda de instrucción. Dependiendo de la instrucción se lee el siguiente operando y se actúa en consecuencia.

La máquina de estados del procesador se muestra en la figura 12.4. Se parte de un estado inicial de réset de manera que, en este estado, todo está inicializado a cero, esto significa que la primera instrucción que se ejecuta está en la dirección cero. Al desactivar el réset se pasa al estado de búsqueda, en el cual se pone la dirección del comienzo de la instrucción, obteniéndose su código. Si la operación es la de no hacer nada, vuelve de nuevo al estado de búsqueda, si no, pasa al estado de ejecución. En el estado de ejecución pueden ocurrir dos cosas: que la operación sea de salto, en cuyo caso actualiza el PC y luego vuelve al estado de búsqueda, o bien, que se trate de cualquier otra operación, en cuyo caso el operando de la instrucción se encuentra en la posición de memoria del segundo argumento, por lo que se tiene que poner esta dirección en el bus de direcciones y leer el contenido, pasándose al siguiente estado que es el de la ejecución particular de cada instrucción. Esto quiere decir que cada instrucción tarda como mucho tres ciclos en completarse.

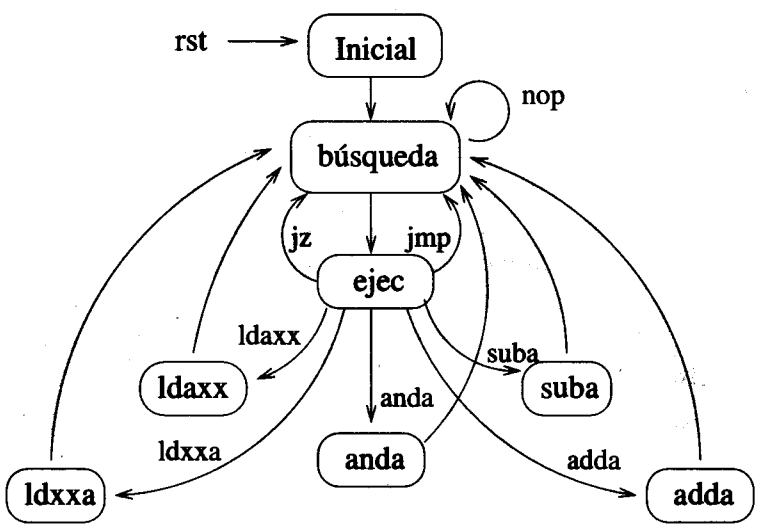


Figura 12.4: Máquina de estados del microprocesador

La figura 12.5 muestra un posible diagrama de bloques que se corresponde, con bastante aproximación, a la descripción del procesador en VHDL.

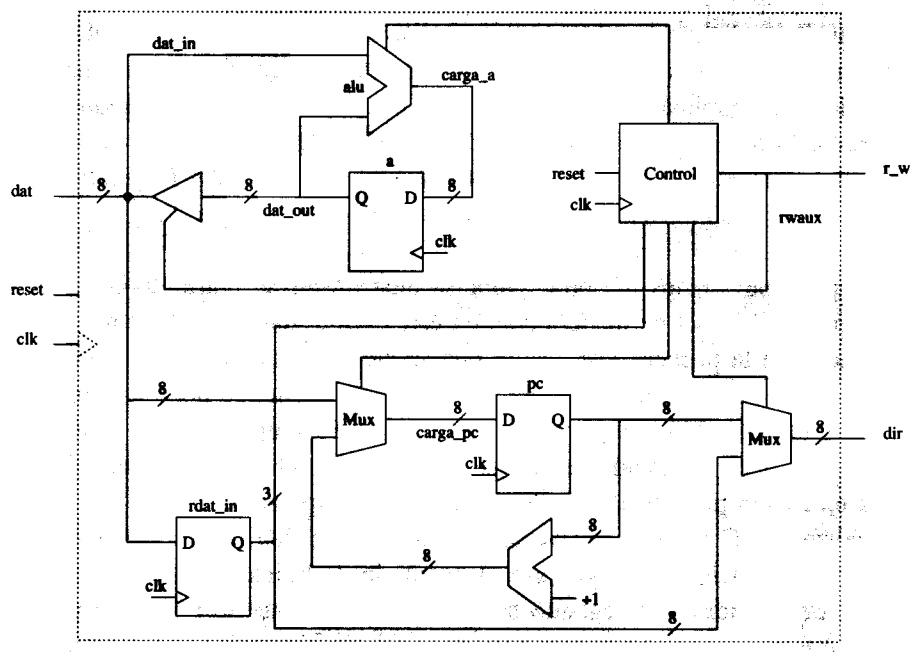


Figura 12.5: Esquema del microprocesador ejemplo

La descripción que se presenta del microprocesador es para síntesis. Se ha intentado en todo momento que la descripción de la máquina de estados sea lo más parecida posible a los modelos dados en el capítulo de síntesis. La descripción utiliza la suma y resta del tipo `std_logic_vector`, lo cual supone que, o bien se han sobrecargado a mano estos operadores, o bien se ha utilizado un paquete que los sobrecarga. Por ejemplo, uno de los paquetes de la biblioteca `ieee` es el `std_arith` que sobrecarga los operadores aritméticos para que se puedan usar con el tipo `std_logic_vector`. A continuación se muestra la descripción de microprocesador:

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

ENTITY procesador IS
PORT(clk,rst: IN std_logic;
     r_w: OUT std_logic;
     dir: OUT std_logic_vector(7 DOWNTO 0);
     dat: INOUT std_logic_vector(7 DOWNTO 0));
END procesador;

ARCHITECTURE descripcion OF procesador IS
  TYPE estado IS (inicial,busqueda,ejec,ldxxa,ldaxx,anda,adda,suba);
  SIGNAL a,pc,carga_a,carga_pc: std_logic_vector(7 DOWNTO 0);
  SIGNAL rdat_in,dat_in,dat_out: std_logic_vector(7 DOWNTO 0);
  SIGNAL rwaux: std_logic;
  SIGNAL presente: estado:=inicial;
```

```
BEGIN

  fsm:
  PROCESS(clk,rst)
  BEGIN
   IF rst='1' THEN
    presente<=inicial;
   ELSIF clk='1' AND clk'event THEN
    CASE presente IS
    WHEN inicial =>
       presente<=busqueda;
    WHEN busqueda=>
       IF dat_in(2 DOWNTO 0)="111" THEN presente<=busqueda;
       ELSE presente<=ejec;
       END IF;
    WHEN ejec =>
       CASE rdat_in(2 DOWNTO 0) IS
       WHEN "000" =>
          presente<=ldaxx;
       WHEN "001" =>
          presente<=ldxxa;
       WHEN "010" =>
          presente<=anda;
       WHEN "011" =>
          presente<=adda;
       WHEN "100" =>
          presente<=suba;
       WHEN OTHERS =>       -- jz,jmp
          presente<=busqueda;
       END CASE;
    WHEN OTHERS=>
       presente<=busqueda;
    END CASE;
   END IF;
  END PROCESS fsm;

  salida:
  PROCESS(presente,pc,a,rdat_in,dat_in)
  BEGIN
    CASE presente IS
    WHEN inicial =>
      carga_pc<="00000000";      -- PC a 0
      carga_a<="00000000";       -- Acumulador a 0
      dir<=pc;                   -- Dirección a 0;
      rwaux<='1';                -- Lectura
    WHEN busqueda=>
      carga_pc<=pc+"00000001";   -- Incrementa PC
      carga_a<=a;                -- Deja A como estaba
      dir<=pc;
      rwaux<='1';
    WHEN ejec =>
      CASE rdat_in(2 DOWNTO 0) IS
      WHEN "101" =>  -- jz
         dir<=pc;
         IF a="00000000" THEN
           carga_pc<=dat_in;     -- Salta si A=0
         END IF;
         rwaux<='1';
         carga_a<=a;
      WHEN "110" =>  --jmp
         dir<=pc;
         carga_pc<=dat_in;       -- Salta siempre
```

```
        rwaux<='1';
        carga_a<=a;
      WHEN OTHERS =>              -- Por defecto para el resto:
        rwaux<='1';               -- Leer
        dir<=pc;                  -- Dirección del operando
        carga_pc<=pc;
        carga_a<=a;
      END CASE;
    WHEN ldaxx =>
      rwaux<='1';
      carga_a<=dat_in;
      carga_pc<=pc+"00000001";
      dir<=rdat_in;
    WHEN ldxxa =>
      rwaux<='0';                 -- Único que escribe
      carga_a<=a;
      carga_pc<=pc+"00000001";
      dir<=rdat_in;
    WHEN anda =>
      rwaux<='1';
      carga_a<=a AND dat_in;
      carga_pc<=pc+"00000001";
      dir<=rdat_in;
    WHEN adda =>
      rwaux<='1';
      carga_a<=a+dat_in;
      carga_pc<=pc+"00000001";
      dir<=rdat_in;
    WHEN suba =>
      rwaux<='1';
      carga_a<=a-dat_in;
      carga_pc<=pc+"00000001";
      dir<=rdat_in;
    END CASE;
END PROCESS salida;

registro_de_entrada:
PROCESS(clk,rst)
BEGIN
  IF rst='1' THEN rdat_in<="00000000";
  ELSIF clk='1' AND clk'event THEN rdat_in<=dat_in;
  END IF;
END PROCESS registro_de_entrada;

regs_con_carga:
PROCESS(clk,rst)
BEGIN
  IF rst='1' THEN
    a <="00000000";
    pc<="00000000";
  ELSIF clk='1' AND clk'event THEN
    a <=carga_a;
    pc<=carga_pc;
  END IF;
END PROCESS regs_con_carga;

r_w<=rwaux;                       -- Para leer la salida.

-- Triestado de salida:
dat_in<=dat;
dat_out<=a;
```

```
  dat<=dat_out WHEN rwaux='0' ELSE "ZZZZZZZZ";
END descripcion;
```

Se pueden pensar descripciones más sencillas, pero en esos casos no es tan fácil separar la parte secuencial de la combinacional en la máquina de estados, lo cual no es ningún problema desde el punto de vista del modelado, pero sí lo es desde el punto de vista de la síntesis. Para poder tener el proceso de salida totalmente combinacional, se han sacado fuera los registros correspondientes al contador del programa (pc) y el acumulador (a). Un error hubiera sido incluir estas señales directamente en el proceso de la salida ya que, en ese caso, el proceso ya no sería combinacional. Alguna herramienta de síntesis hubiera entendido esto bien, pero quizá alguna otra no hubiera sido capaz de sintetizarlo correctamente.

Para entender mejor el funcionamiento del circuito se ha realizado una simulación en VHDL. Para esta simulación se han puesto unos valores en el bus de datos simulando el acceso a una RAM. A continuación se presenta la entidad de test y la arquitectura con los estímulos:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY test IS
END test;

ARCHITECTURE estimulos OF test IS
 SIGNAL clk: std_logic:='1';
 SIGNAL rst: std_logic:='1';
 SIGNAL r_w: std_logic;
 SIGNAL dir: std_logic_vector(7 DOWNTO 0);
COMPONENT procesador
PORT(clk,rst: IN std_logic;
     r_w: OUT std_logic;
     dir: OUT std_logic_vector(7 DOWNTO 0);
     dat: INOUT std_logic_vector(7 DOWNTO 0));
END COMPONENT;
 SIGNAL dat: std_logic_vector(7 DOWNTO 0);

 FOR proc: procesador USE ENTITY work.procesador;
BEGIN
 proc: procesador PORT MAP(clk,rst,r_w,dir,dat);
 clk<=NOT clk AFTER 50 ns;
 rst<='0' AFTER 70 ns;
 dat<="00000000",
      "00010100" AFTER 210 ns,
      "00000100" AFTER 310 ns,
      "00000001" AFTER 410 ns,
      "00010101" AFTER 510 ns,
      "ZZZZZZZZ" AFTER 610 ns,
      "00000110" AFTER 710 ns,
      "00000010" AFTER 810 ns,
      "00000001" AFTER 910 ns;
END estimulos;
```

El cronograma correspondiente, como resultado de la simulación, se muestra en la figura 12.6 donde se han incluido todas las señales internas así como los pines externos. Los valores de los buses están dados en decimal.

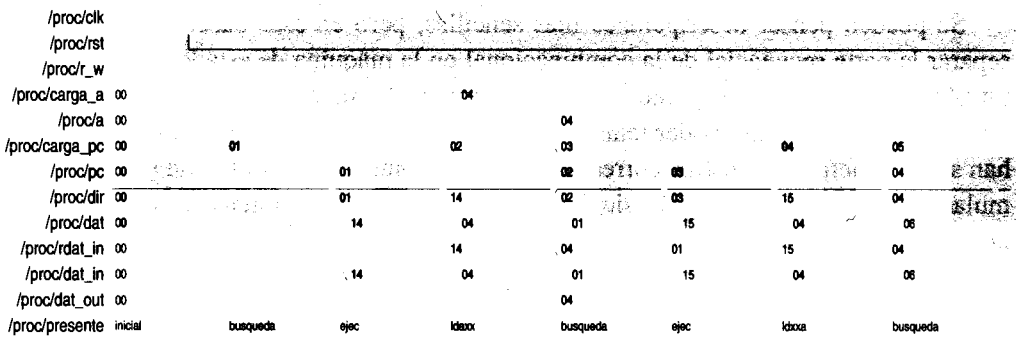


Figura 12.6: Simulación del ejemplo de procesador

12.2.6 La lavadora

Ejemplo 12.6 *Se pretende sintetizar el chip que controla una lavadora doméstica ficticia. La lavadora, junto con las entradas y salidas del chip que la controla, se muestran en la figura 12.7. El funcionamiento se explica a continuación junto con las entradas y salidas:*

Entradas:

color: *Al pulsarse esta tecla se cambia un estado interno de la máquina que indica si el ciclo de lavado es de ropa de color o no. Inicialmente se supone que no es de color (estado a cero).*

centrifuga: *Cada vez que se pulsa cambia un estado interno que indica si se debe centrifugar o no. Inicialmente se supone que no (estado a cero).*

start: *Cuando se pulsa se inicia el lavado, una vez en marcha este botón no hace nada.*

jabon_listo: *Indica que el jabón ya se ha introducido en el lavado.*

vacio: *Indica que el tambor está vacío de agua.*

lleno: *Indica que el tambor ya está lleno de agua.*

clk: *Reloj de sincronización de frecuencia 100 Hz.*

Salidas:

jabon: *Al ponerse a uno se toma el jabón del cajetín y se deposita en el tambor durante el ciclo de lavado.*

llenar: *A uno abre las válvulas del agua para llenar el tambor, se debe monitorizar la señal* **lleno** *para saber cuándo debe ponerse a cero para que no entre más agua.*

vaciar: *A uno abre las válvulas de salida del agua para desaguar el tambor. La señal de entrada* **vacio** *indicará el momento en que no hay más agua en el tambor.*

lento: *Un uno en esta señal hace que el motor gire, en la dirección indicada por la señal* **direccion**, *con un ritmo lento. Esta velocidad se usa en todos los ciclos menos en el centrifugado.*

rapido: *Lo mismo que* **lento** *pero la velocidad es la de centrifugado, es decir, más rápida. Si las dos señales anteriores están a cero entonces el motor está parado, si ambas están a uno entonces se quema la máquina de lavar.*

direccion: *A uno indica que el tambor se moverá a izquierdas y a cero a derechas. El tambor debe moverse alternativamente a derecha e izquierda en todos los ciclos menos en el de centrifugado que se mueve siempre en la misma dirección.*

Ciclos de lavado:

Inicial: *Es el estado inicial de la máquina y está esperando a que se pulse* **start**.

Lavado: *En este ciclo se coge el jabón, se llena de agua el tambor y se pone en marcha el motor alternativamente a izquierda y derecha. La duración es de 10 minutos si la ropa es de color y 20 minutos si la ropa es blanca o resistente. Cuando acaba se vacía el agua del tambor.*

Aclarado: *Se llena el tambor de agua otra vez pero sin jabón. El tambor también se mueve. Dura 5 minutos y hay que vaciar el agua al acabar.*

Centrifugado: *Si la opción de centrifugado está seleccionada entonces entrará en este ciclo, si no volverá al inicial. Este ciclo consiste en mover el tambor a velocidad rápida en un único sentido de giro durante 10 minutos. Al acabar se vuelve al estado inicial.*

Si se pretende sintetizar el circuito es siempre preferible sincronizar la máquina de estados con la señal de reloj. Se van a presentar dos posibles descripciones para la

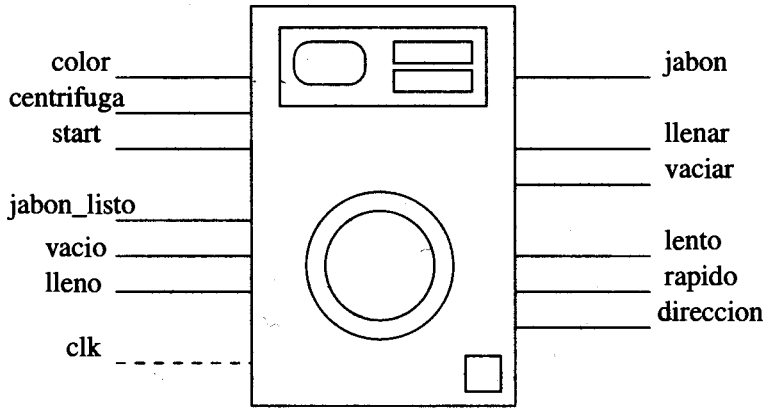


Figura 12.7: Figura del ejercicio de la lavadora

máquina de estados, y se podrán de manifiesto las diferencias que se pueden dar a la hora de sintetizar según el tipo de máquina de estados que se realice. Hay que destacar que tanto la simulación de una como la de la otra prácticamente coinciden.

En la primera descripción se ha realizado la máquina de estados sin separar la parte secuencial de la combinacional, de manera que se han colocado las salidas en el mismo proceso donde se encuentran las transiciones de estados, por lo que la salida cambiará un ciclo de reloj después de que cambie el estado, pero esto en principio no tiene importancia ya que la frecuencia de reloj es muy alta. De todos modos, no es aconsejable realizar la descripción de esta manera, aunque a veces resulta más sencillo.

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

ENTITY lavadora IS
PORT(color,centrifuga,start,jabon_listo,vacio,lleno,clk: IN std_logic;
     jabon,llenar,vaciar,rapido,lento,direccion: OUT std_logic);
END lavadora;

ARCHITECTURE sincrona OF lavadora IS
  CONSTANT diezsec: integer:=1000;       -- Estos tiempos han sido
  -- Se calcula suponiendo una frecuencia de reloj de 100 Hz.:
  CONSTANT cincomin: integer:=30000;
  CONSTANT diezmin: integer:=60000;
  CONSTANT veintemin: integer:=120000;
  TYPE estados IS
     (inicial,lavado,vacia1,aclarado,vacia2,centrifugado);
  SIGNAL presente: estados:=inicial;
  SIGNAL coloraux,centriaux,diraux: std_logic:='0';
  SIGNAL tiempo: integer RANGE 0 TO 16#1FFFF# :=0;
  SIGNAL subtiempo: integer RANGE 0 TO 1023 :=0;
BEGIN
  maquina:
  PROCESS(clk)
```

```
BEGIN
  IF clk='1' THEN
    CASE presente IS
    WHEN inicial=>
      IF start='1' THEN presente<=lavado; END IF;
      jabon<='0'; llenar<='0'; vaciar<='1';
      lento<='0'; rapido<='0'; diraux<='0';
      tiempo<=0;
      subtiempo<=0;
    WHEN lavado=>
      vaciar<='0';
      IF jabon_listo='0' THEN jabon<='1';
      ELSE jabon<='0';
      END IF;
      IF lleno='0' THEN
        llenar<='1';
      ELSE
        llenar<='0';
        lento<='1';
        IF subtiempo=diezsec THEN
          diraux<=NOT diraux;
          subtiempo<=0;
        ELSE
          subtiempo<=subtiempo+1;
        END IF;
        tiempo<=tiempo+1;
        IF coloraux='1' AND tiempo=diezmin THEN presente<=vacia1;
        ELSIF tiempo=veintemin THEN presente<=vacia1;
        END IF;
      END IF;
    WHEN vacia1=>
      vaciar<='1';
      lento<='0';
      IF vacio='1' THEN presente<=aclarado;
      END IF;
      subtiempo<=0;
      tiempo<=0;
    WHEN aclarado=>
      vaciar<='0';
      IF lleno='0' THEN
        llenar<='1';
      ELSE
        llenar<='0';
        lento<='1';
        IF subtiempo=diezsec THEN
          diraux<=NOT diraux;
          subtiempo<=0;
        ELSE
          subtiempo<=subtiempo+1;
        END IF;
        tiempo<=tiempo+1;
        IF tiempo=cincomin THEN presente<=vacia2;
        END IF;
      END IF;
    WHEN vacia2=>
      vaciar<='1';
      lento<='0';
      IF vacio='1' THEN
        IF centriaux='1' THEN presente<=centrifugado;
        ELSE presente<=inicial;
        END IF;
      END IF;
```

```
        tiempo<=0;
      WHEN centrifugado=>
        rapido<='1';
        tiempo<=tiempo+1;
        IF tiempo=diezmin THEN presente<=inicial; END IF;
      END CASE;
    END IF;
  END PROCESS maquina;

  PROCESS(centrifuga)
  BEGIN
    IF centrifuga='1' THEN centriaux<=NOT centriaux; END IF;
  END PROCESS;

  PROCESS(color)
  BEGIN
    IF color='1' THEN coloraux<=NOT coloraux; END IF;
  END PROCESS;

  direccion<=diraux;

END sincrona;
```

Normalmente es aconsejable poner el contador de tiempo fuera de la descripción de la máquina de estados, especialmente por claridad, pero en este caso se ha mostrado que también es posible incluirla dentro. La síntesis de este circuito requiere unos 100 registros para su realización. El hecho de que se hayan puesto las salidas en la propia descripción de la máquina significa que vienen sincronizadas por la señal de reloj, y esto significa que habrá un registro asociado con cada una de las señales que haya en este proceso.

A continuación se muestra la otra posibilidad que consiste en poner las señales de salida en un proceso aparte que será completamente combinacional, y que, por lo tanto, no necesitará registros adicionales, y que, además, hará que las salidas cambien a la vez que el estado.

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

ENTITY lavadora2 IS
PORT(color,centrifuga,start,jabon_listo,vacio,lleno,clk: IN std_logic;
     jabon,llenar,vaciar,rapido,lento,direccion: OUT std_logic);
END lavadora2;

ARCHITECTURE sincrona2 OF lavadora2 IS
  CONSTANT diezsec: integer:=1000;       -- Estos tiempos han sido
  -- Se calcula suponiendo una frecuencia de reloj de 100 Hz:
  CONSTANT cincomin: integer:=30000;
  CONSTANT diezmin: integer:=60000;
  CONSTANT veintemin: integer:=120000;
  TYPE estados IS
     (inicial,lavado,vacia1,aclarado,vacia2,centrifugado);
  SIGNAL presente: estados:=inicial;
```

```
  SIGNAL coloraux,centriaux,dirauxd,diraux: std_logic:='0';
  SIGNAL tiempo: integer RANGE 0 TO 16#1FFFF# :=0;
  SIGNAL subtiempo: integer RANGE 0 TO 1023 :=0;
  SIGNAL subtiempores, tiempores: boolean :=TRUE;
BEGIN
  maquina:
  PROCESS(clk)
  BEGIN
    IF clk='1' THEN
      CASE presente IS
      WHEN inicial=>
        IF start='1' THEN presente<=lavado; END IF;
      WHEN lavado=>
          IF coloraux='1' AND tiempo=diezmin THEN presente<=vacia1;
          ELSIF tiempo=veintemin THEN presente<=vacia1;
          END IF;
      WHEN vacia1=>
        IF vacio='1' THEN presente<=aclarado;
        END IF;
      WHEN aclarado=>
          IF tiempo=cincomin THEN presente<=vacia2;
          END IF;
      WHEN vacia2=>
        IF vacio='1' THEN
          IF centriaux='1' THEN presente<=centrifugado;
          ELSE presente<=inicial;
          END IF;
        END IF;
      WHEN centrifugado=>
        IF tiempo=diezmin THEN presente<=inicial; END IF;
      END CASE;
    END IF;
  END PROCESS maquina;

salida:
  PROCESS(presente)
  BEGIN
      CASE presente IS
      WHEN inicial=>
        jabon<='0'; llenar<='0'; vaciar<='1';
        lento<='0'; rapido<='0';  dirauxd<='0';
        tiempores<=TRUE; subtiempores<=TRUE;
      WHEN lavado=>
        vaciar<='0';
        rapido<='0';
        IF jabon_listo='0' THEN jabon<='1';
        ELSE jabon<='0';
        END IF;
        IF lleno='0' THEN
          llenar<='1';
          lento<='0';
          tiempores<=TRUE;
        ELSE
          llenar<='0';
          lento<='1';
          jabon<='0';
          tiempores<=FALSE;
        END IF;
        IF subtiempo=diezsec THEN
          dirauxd<=NOT diraux;
          subtiempores<=TRUE;
        ELSE
```

```
        subtiempores<=FALSE;
      END IF;
    WHEN vacia1=>
      jabon<='0';
      vaciar<='1';
      lento<='0';
      rapido<='0';
      subtiempores<=TRUE;
      tiempores<=TRUE;
      llenar<='0';
      dirauxd<='0';
    WHEN aclarado=>
      jabon<='0';
      vaciar<='0';
      rapido<='0';
      IF lleno='0' THEN
        llenar<='1';
        lento<='0';
        tiempores<=TRUE;
      ELSE
        llenar<='0';
        lento<='1';
        tiempores<=FALSE;
      END IF;
      IF subtiempo=diezsec THEN
        dirauxd<=NOT diraux;
        subtiempores<=TRUE;
      ELSE
        subtiempores<=FALSE;
      END IF;
    WHEN vacia2=>
      jabon<='0';
      dirauxd<='0';
      vaciar<='1';
      lento<='0';
      rapido<='0';
      llenar<='0';
      subtiempores<=TRUE;
      tiempores<=TRUE;
    WHEN centrifugado=>
      jabon<='0';
      dirauxd<='0';
      llenar<='0';
      vaciar<='1';
      rapido<='1';
      lento<='0';
      subtiempores<=TRUE;
      tiempores<=FALSE;
    END CASE;
END PROCESS salida;

contador:
PROCESS(clk)
BEGIN
  IF clk='1' THEN
    IF subtiempores THEN
      subtiempo<=0;
    ELSE
      subtiempo<=subtiempo+1;
    END IF;
    IF tiempores THEN
      tiempo<=0;
```

```
      ELSE
        tiempo<=tiempo+1;
      END IF;
    END IF;
  END PROCESS contador;

  PROCESS(centrifuga)
  BEGIN
    IF centrifuga='1' THEN centriaux<=NOT centriaux; END IF;
  END PROCESS;

  PROCESS(color)
  BEGIN
    IF color='1' THEN coloraux<=NOT coloraux; END IF;
  END PROCESS;

  PROCESS(clk)
  BEGIN
    IF clk='1' THEN diraux<=dirauxd;
    END IF;
  END PROCESS;

  direccion<=diraux;

END sincrona2;
```

Hay que apreciar que no sólo se han sacado las señales de salida sino que además han sido necesarios más cambios. Por un lado se ha creado un nuevo proceso para el contador de tiempo, con lo que ha sido necesario añadir unas señales para comunicar la máquina de estados con este proceso. Luego se han puesto todas las señales de salida en cada una de las posibilidades del CASE, de esta manera ese proceso es totalmente combinacional y se ahorran registros. Con todo esto, esta descripción, que es equivalente a la anterior, ocupa unos 70 registros, que es un número sensiblemente inferior al anterior.

Este ejemplo ha demostrado que dos descripciones que resuelven aparentemente el mismo problema se pueden sintetizar de dos formas muy diferentes, dando como resultado circuitos más o menos óptimos. En general, si se siguen las recomendaciones dadas para máquinas de estados, el circuito resultante está más optimizado, pero su descripción es más complicada.

12.2.7 El concurso

Ejemplo 12.7 *Se pretende realizar el modelo de un chip que controla el funcionamiento de un programa concurso de televisión entre tres concursantes. La prueba que tienen que pasar los tres concursantes es la de contestar a unas preguntas eligiendo una de las tres respuestas que se le dan, para ello dispone de tres pulsadores cada uno. Hay un operador humano detrás del escenario que controla la máquina. Tiene tres interrupto-*

res donde programa la respuesta correcta (`correcto`*), un pulsador que le sirve tanto para iniciar el juego como para empezar una nueva pregunta (*`pregunta`*), y un botón de réset para inicializarlo todo. Una vez pulsado* `pregunta`*, los concursantes deben pulsar el botón correspondiente a la respuesta que crean correcta. En el momento en que alguien pulse se pasa a evaluar su respuesta (como los circuitos van a tener un retraso muy pequeño, se supone que es imposible que dos jugadores pulsen a la vez). Si el jugador acierta la respuesta se le sumarán 10 puntos en su marcador, pero si falla entonces se le restarán 5 puntos. Si ningún jugador contesta en 5 segundos entonces se le restarán 5 puntos al que tenga mayor puntuación en ese momento (si hay varios compartiendo la máxima nota, se les resta a todos ellos). El circuito sabrá si la respuesta ha sido correcta comparándola con la que el operador haya programado en los interruptores* `correcto` *antes de iniciar cada pregunta, y que cambiará entre pregunta y pregunta antes de pulsar* `pregunta`*. Cuando algún jugador llegue a 100 puntos, o más, entonces habrá ganado y el juego se parará activándose la salida correspondiente al jugador que haya ganado. Los marcadores del resto de jugadores se ponen a cero salvo el del que ganó que conserva su valor. Así se queda todo en este estado hasta que el operador pulse el botón réset.*

La frecuencia de reloj es fija y vale 1024 Hz. En caso de pregunta acertada, fallada, o bien si pasan 5 segundos, el operador siempre deberá pulsar `pregunta` *para hacer otra pregunta. Los interruptores de* `correcto` *se ponen y se quedan a uno o a cero hasta que se los cambie otra vez. Los botones de los pulsadores están a uno mientras se pulsen, el resto del tiempo están a cero.*

Al igual que en los casos anteriores se va a intentar realizar una máquina de estados separando bien la parte secuencial de la combinacional. Cuando existen marcadores, contadores, etc., que una veces suman una cantidad, otras otra, etc., la descripción de la máquina se complica, ya que se tratan de señales que describen registros. En el caso del procesador se había mostrado que una forma de solucionar esto consiste en poner estos registros de forma explícita fuera de la máquina. En el procesador se utilizaba el valor de carga del registro para controlarlo, en esta ocasión se va más lejos y se crea todo un proceso que es la salida síncrona de la máquina de estados, es decir, en esta máquina de estados hay tres procesos, uno es el secuencial del próximo estado, el otro contiene las salidas combinacionales y el tercero las salidas secuenciales (los marcadores). A continuación se muestra la descripción:

```
ENTITY ajugar IS
PORT (
-- Reloj de frecuencia fija 1024 Hz;
        clk: IN bit;
-- Diferentes pulsadores o botones del operador:
```

```
        reset,pregunta: IN bit;
-- Contiene la respuesta correcta:
        correcto: IN bit_vector(1 TO 3);
-- Pulsadores de los jugadores A,B,C respectivamente:
        pulsaA, pulsaB, pulsaC: IN bit_vector(1 TO 3);
-- Marcadores de cada jugador A, B y C:
        marcaA, marcaB, marcaC: OUT integer RANGE 0 TO 255;
-- Líneas para indicar el ganador:
        ganaA, ganaB, ganaC: OUT bit);
END ajugar;

ARCHITECTURE una_solucion OF ajugar IS
 TYPE estado IS (inicial,responde,evalua,tiempo,espera,final);
 SIGNAL cuenta: integer RANGE 0 TO 8191;
 SIGNAL marcauxA,marcauxB,marcauxC: integer RANGE 0 TO 255;
 SIGNAL timeout: boolean;  -- Para indicar paso de 5 segundos.
 SIGNAL pulsaron: boolean; -- Para saber si alguien pulsó.
 SIGNAL fin: boolean;      -- Para saber cuándo se llega al final.
 SIGNAL rescont: boolean;  -- Pone a cero la cuenta.
 SIGNAL presente: estado;

BEGIN
 marcaA<=marcauxA;    -- Señales auxiliares para poder
 marcaB<=marcauxB;    -- leer la salida
 marcaC<=marcauxC;

 contador:
 PROCESS(clk)
 BEGIN
   IF clk='1' THEN
     IF rescont THEN cuenta<=0; -- Para inicializar la cuenta
     ELSE cuenta<=cuenta+1;
     END IF;
   END IF;
 END PROCESS contador;

 timeout<=true WHEN cuenta=5120 ELSE false; -- pasaron 5 segundos
 pulsaron<=true WHEN (pulsaA/="000" OR pulsaB/="000" OR pulsaC/="000")
      ELSE false;
 fin<=true WHEN (marcauxA>=100 OR marcauxB>=100 OR marcauxC>=100)
      ELSE false;
 ganaA<='1' WHEN marcauxA>=100 ELSE '0';
 ganaB<='1' WHEN marcauxB>=100 ELSE '0';
 ganaC<='1' WHEN marcauxC>=100 ELSE '0';

 maquina:
 PROCESS(reset,clk)
 BEGIN
   IF reset='1' THEN presente<=inicial;
   ELSIF clk='1' AND clk'event THEN
     CASE presente IS
       WHEN inicial=>
         IF pregunta='1' THEN presente<=responde; END IF;
       WHEN responde=>
         IF pulsaron THEN presente<=evalua;
         ELSIF timeout THEN presente<=tiempo;
         END IF;
       WHEN evalua=>
         presente<=espera;
       WHEN tiempo=>
         presente<=espera;
       WHEN espera=>
```

```
        IF fin THEN presente<=final;
        ELSIF pregunta='1' THEN presente<=responde;
        END IF;
      WHEN final=>
        NULL;
    END CASE;
  END IF;
END PROCESS maquina;

salida:
PROCESS(presente)
BEGIN
  CASE presente IS
    WHEN inicial=>
      rescont<=true;
    WHEN responde=>
      rescont<=false;
    WHEN evalua=>
      rescont<=true;
    WHEN tiempo=>
      rescont<=true;
    WHEN espera=>
      rescont<=true;
    WHEN final=>
      rescont<=true;
  END CASE;
END PROCESS salida;

marcadores:
PROCESS(clk,reset)
BEGIN
  IF reset='1' THEN
    marcauxA<=0;
    marcauxB<=0;
    marcauxC<=0;
  ELSIF clk='1' AND clk'event THEN
    CASE presente IS
    WHEN evalua=>
      IF pulsaA/="000" THEN
        IF pulsaA=correcto THEN marcauxA<=marcauxA+10;
        ELSIF marcauxA>=5 THEN marcauxA<=marcauxA-5;
        END IF;
      END IF;
      IF pulsaB/="000" THEN
        IF pulsaB=correcto THEN marcauxB<=marcauxB+10;
        ELSIF marcauxB>=5 THEN marcauxB<=marcauxB-5;
        END IF;
      END IF;
      IF pulsaC/="000" THEN
        IF pulsaC=correcto THEN marcauxC<=marcauxC+10;
        ELSIF marcauxC>=5 THEN marcauxC<=marcauxC-5;
        END IF;
      END IF;
    WHEN tiempo=>
      IF marcauxA>=5 AND marcauxA>=marcauxB AND marcauxA>=marcauxC
         THEN marcauxA<=marcauxA-5;
      END IF;
      IF marcauxB>=5 AND marcauxB>=marcauxA AND marcauxB>=marcauxC
         THEN marcauxB<=marcauxB-5;
      END IF;
      IF marcauxC>=5 AND marcauxC>=marcauxB AND marcauxC>=marcauxA
         THEN marcauxC<=marcauxC-5;
```

```
      END IF;
    WHEN final=>
      IF marcauxA>=100 THEN
        marcauxB<=0;
        marcauxC<=0;
      END IF;
      IF marcauxB>=100 THEN
        marcauxA<=0;
        marcauxC<=0;
      END IF;
      IF marcauxC>=100 THEN
        marcauxB<=0;
        marcauxA<=0;
      END IF;
    WHEN OTHERS=> NULL;
    END CASE;
  END IF;
END PROCESS marcadores;

END una_solucion;
```

Esta descripción puede tener un inconveniente, y es que para discernir bien entre dos pulsaciones muy seguidas, el reloj debería ser bastante más rápido. Lo único que habría que cambiar es el tamaño del contador y el fin de cuenta. Hay que destacar que se ha usado el tipo entero en vez del vector de bits; en principio no hay ningún problema ya que muchas de las herramientas de síntesis entenderán que se trata de un bus de bits y lo sintetizarán bien. Por otro lado, utilizar enteros tiene ventajas, ya que van a funcionar bien con las operaciones aritméticas. Eso sí, hay que especificar siempre un rango para que el sintetizador pueda saber de cuántos bits se trata la señal.

12.2.8 El pin-ball

Ejemplo 12.8 *Realizar la descripción en VHDL del controlador de una máquina de pin-ball. La máquina tiene un marcador que se incrementa según donde toque la bola. La bola puede pegar en cualquiera de dos pivotes de manera que según dé la bola en uno u otro se suman 5 ó 10 puntos respectivamente. Cada 100 puntos se obtiene una bola nueva. También tiene dos tacos que son los que impulsan la bola durante el juego, y otro que sirve para lanzar la bola al principio. Además está la ranura por donde se mete la moneda.*

Para controlar la máquina se necesitan las siguientes entradas:

p_uno, p_dos: *Se ponen a '1' cuando la bola choca contra el pivote uno o dos. Van a servir para saber qué valor hay que sumar al marcador; si se da al pivote* **p_uno** *se suman 5 y si se da al otro 10.*

falta: *Esta señal se pone a '1' cuando se empuja bruscamente la máquina para hacer*

trampa. Cuando esta señal se pone a uno, los tacos deben paralizarse y quedarse así hasta que la bola se pierda por el agujero.

nueva: *Sirve para indicar que se ha introducido una moneda y que empieza la partida. (Por simplicidad no se considera el caso en el que se introducen varias monedas para tener más partidas.)*

pierde: *Sirve para indicar que se ha perdido la bola por el agujero y que por tanto hay que restar una bola a las que quedan. Cuando no quedan más bolas se detiene el juego.*

clk: *Señal de reloj para sincronizar el circuito. Su frecuencia se supone mucho más alta que el tiempo que están activas las señales de* **p_uno, p_dos, nueva** *y* **pierde**.

A partir de dichas entradas el circuito debe producir las siguientes salidas:

marcador: *Es un bus de 12 líneas que se conecta al marcador electrónico de la máquina y que contiene la cuenta de puntos.*

bloqueo: *Mientras está a '1' los tacos no funcionan. Debe estar a '1' mientras no se juega o desde que se movió la máquina bruscamente* (falta) *hasta que sale una nueva bola.*

fin: *Cuando se acaban las bolas esta señal se pone a '1' para encender un gran panel luminoso que pone 'Fin de juego. Inserte moneda'.*

Tener en cuenta que el marcador no debe ponerse a cero al acabar el juego, sino que debe hacerlo en el momento de empezar a jugar después de insertar la moneda.

Como es habitual, el problema se puede solucionar mediante una máquina de estados que se hace síncrona para que resulte sencilla la síntesis del circuito.

Por una vez se pone como ejemplo la utilización de una máquina de estados con un único proceso, esto significa que la máquina requerirá algunos registros más, y las señales de salida estarán un ciclo de reloj retrasadas respecto del cambio de estado. La ventaja es que probablemente es algo más sencilla de describir.

El problema de este ejemplo, aparte de incrementar los marcadores, radica en el tratamiento de las señales de los pivotes y de pérdida de bola, que son de una duración impredecible. Esto se ha solucionado en la máquina poniendo un estado de espera que, como su nombre indica, se espera a que estas señales vuelvan a cero. Si el tiempo que

están activas estas señales es corto no hay problema, pero si es algo largo, puede ocurrir que suceda una después de la otra, y la segunda se solape con la primera de manera que no se reconozca. Por ejemplo, puede ocurrir que pegue en un pivote y enseguida en el otro, y que el segundo choque no sume puntos. Después de la descripción se da otra solución que resuelve este problema pero tiene otros.

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

ENTITY pinball IS
PORT (p_uno,p_dos,falta,nueva,pierde,clk,reset : IN std_logic;
      marcador: OUT integer RANGE 0 TO 4095;    -- 12 bits
      bloqueo,fin: OUT std_logic);
END pinball;

ARCHITECTURE sincrono OF pinball IS
  TYPE estado IS
     (insertar,inicia,juega,suma_cinco,suma_diez,trampa,resta,espera);
  SIGNAL marcaux: integer RANGE 0 TO 4095 :=0; -- Leer la salida
  SIGNAL cien: integer RANGE 0 TO 127 :=0;     -- Para bola extra
  SIGNAL bolas: integer RANGE 0 TO 31 :=0;     -- Número de bolas
  SIGNAL presente: estado:=insertar;
BEGIN
  maquina:
  PROCESS(clk,reset)
  BEGIN
    IF reset='1' THEN
      presente<=insertar;
      marcaux<=0;
      fin<='1';
      bloqueo<='1';
      cien<=0;
    ELSIF clk='1' AND clk'event THEN
      CASE presente IS
      WHEN insertar=>
        IF nueva='1' THEN presente<=inicia;
        END IF;
        fin<='1';
        bloqueo<='1';
        cien<=0;
      WHEN inicia=>
        presente<=juega;
        fin<='0';
        marcaux<=0;
        cien<=0;
        bloqueo<='0';
        bolas<=5;
      WHEN juega=>
        IF pierde='1' THEN presente<=resta; END IF;
        IF falta='1' THEN presente<=trampa; END IF;
        IF bolas=0 THEN presente<=insertar; END IF;
        IF p_uno='1' THEN presente<=suma_cinco; END IF;
        IF p_dos='1' THEN presente<=suma_diez; END IF;
        IF cien>=100 THEN
          cien<=0;
          bolas<=bolas+1;
        END IF;
        IF bolas=0 THEN presente<=insertar; END IF;
        bloqueo<='0';
```

```
      WHEN suma_cinco=>
        presente<=espera;
        marcaux<=marcaux+5;
        cien<=cien+5;
      WHEN suma_diez=>
        presente<=espera;
        marcaux<=marcaux+10;
        cien<=cien+10;
      WHEN resta=>
        bolas<=bolas-1;
        presente<=espera;
      WHEN espera=>
        IF p_uno='0' AND p_dos='0' AND pierde='0' THEN
          presente<=juega;
        END IF;
      WHEN trampa=>
        IF pierde='1' THEN presente<=resta; END IF;
        bloqueo<='1';
      END CASE;
    END IF;
  END PROCESS maquina;

-- Finalmente se pone el marcador igual a su auxiliar.
  marcador<=marcaux;

END sincrono;
```

Otra forma de resolver el problema de la duración impredecible de los pulsos producidos por los pivotes, y el producido cuando se pierde la bola (`pierde`), es la introducción de registros en estas señales. Este registro crea un pulso auxiliar que tiene la duración de un pulso de reloj y que, por tanto, sólo se procesa una vez. Estos pulsos se han creado al final en los procesos `d_pierde`, `d_p_uno` y `d_p_dos`. Para ello se ha conectado cada una de estas señales a la entrada de reloj de un registro, de manera que cuando se produce el flanco de subida se activa, desactivándose al cambiar la máquina de estado. El inconveniente de esta técnica es que la señal proveniente de los pivotes debe ser limpia para evitar falsos disparos, y posiblemente esto no es tan sencillo dada la mecánica del pivote y demás resortes. Naturalmente, con esta técnica ya no es necesario el estado de espera en la máquina.

```
-- Como estos pulsos de entrada son mucho más largos que el periodo
-- del reloj se crean estas señales que son lo mismo pero duran un
-- único pulso:
  d_pierde:
  PROCESS(pierde,presente)
  BEGIN
    IF presente=resta OR presente=inicia THEN d_pierde<='0';
    ELSIF pierde='1' AND pierde'event THEN d_pierde<='1';
    END IF;
  END PROCESS;

  d_p_uno:
  PROCESS(p_uno,presente)
  BEGIN
```

```
    IF presente=suma_cinco OR presente=inicia THEN d_p_uno<='0';
    ELSIF p_uno='1' AND p_uno'event THEN d_p_uno<='1';
    END IF;
  END PROCESS;

  d_p_dos:
  PROCESS(p_dos,presente)
  BEGIN
    IF presente=suma_diez OR presente=inicia THEN d_p_dos<='0';
    ELSIF p_dos='1' AND p_dos'event THEN d_p_dos<='1';
    END IF;
  END PROCESS;
```

12.3 EJERCICIOS PROPUESTOS

Ejemplo 12.9 *Realizar la descripción del circuito de control de un microondas. Las entradas al sistema son:*

Minuto *Es un botón que incrementa el contador del tiempo de cocción en 60 segundos.*

Marcha *Cuando se pulsa se inicia la marcha del horno, y no se parará hasta que se abra la puerta o la cuenta llegue al final o bien se pulse la tecla de stop_reset.*

Stop_Reset *Es un botón que si se pulsa con el horno en marcha lo detiene, pero la cuenta conserva su valor. Si se pulsa con el horno parado la cuenta se pone a cero.*

Puerta *Es una entrada que cuando está a uno indica que la puerta está abierta, y a cero indica que está cerrada.*

clk *Es el reloj de entrada con un periodo de 125 ms.*

Las salidas del circuito a realizar serán:

Segundos(9..0) *Estas 10 líneas le indican a una pantalla el número de segundos de la cuenta. La codificación es binaria, por lo que se pueden programar hasta 1023 segundos.*

Calentar *Cuando está a uno el horno calienta, y si está a cero no hace nada.*

Luz *A uno enciende la luz interna del horno. Esta luz debe estar encendida mientras la puerta esté abierta y mientras el horno esté calentando.*

Alarma *A uno suena. Debe sonar durante 3 segundos cuando la cuenta ha llegado a cero después de que el horno ha estado calentando.*

En la figura 12.8 se muestra el chip del cual se quiere hacer la descripción junto con sus entradas y salidas.

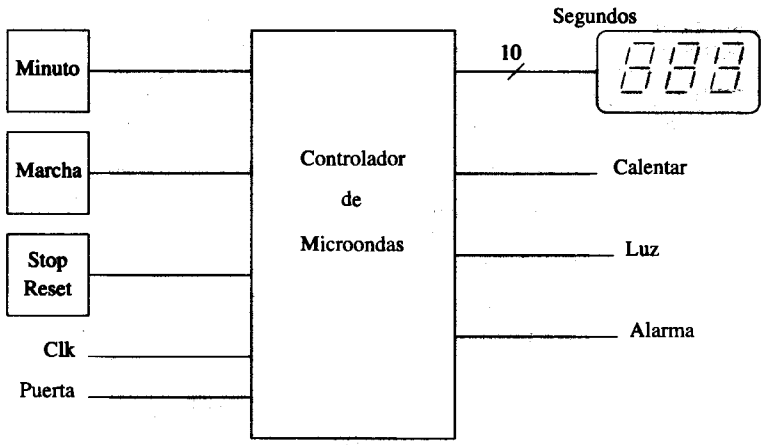


Figura 12.8: Figura del ejercicio del microondas

Aunque el ejemplo parece sencillo no lo es debido a los diferentes elementos que deben sincronizarse. Es un ejercicio bastante completo y su realización correcta implica que se está en condiciones de abordar casi cualquier sistema que se quiera describir mediante VHDL. Como recomendación cabe destacar que hay que prestar especial atención a los botones.

Ejemplo 12.10 *Describir con VHDL el circuito que controla una máquina de café. Las entradas y salidas del circuito se muestran en la figura 12.9. Las entradas son:*

moneda_clk *El flanco de subida de esta señal indica que se ha introducido una moneda en la máquina.*

moneda(7..0) *Indica el valor de la moneda introducida.*

tecla(6..1) *Son los seis botones que permiten elegir entre los seis diferentes cafés que prepara la máquina.*

no_azucar *Pulsando esta tecla al mismo tiempo que la de selección, la máquina no le pondrá azúcar al café.*

listo *La parte de la máquina que hace el café pone a uno esta señal durante unos instantes para indicar que el café está listo y puede preparar otro.*

Las salidas del circuito deberán ser:

error *Es una luz que se enciende cuando se realiza una selección y no hay dinero suficiente.*

cambio(7..0) *Es la diferencia entre el dinero introducido y lo que cuesta el café. Le sirve al bloque de cambio para devolver el cambio.*

cambio_clk *Cuando esta señal pasa de cero a uno el bloque de cambio debe leer la información de cambio(7..0) y devolver esa cantidad. Como no es éste el circuito que se debe sintetizar se supondrá que funciona correctamente.*

tipo(2..0) *Estos tres bits indican el tipo de café, el 1, el 2, etc. Si no se ha seleccionado nada el tipo es el cero. La parte que hace café empieza en el momento en que detecte un cambio en estas líneas.*

azúcar *A uno le indicará a la parte de la máquina que hace el café que le ponga azúcar.*

Debe tenerse en cuenta que hay precios diferentes según el café. Así, el café tipo 1 vale 40 pts, el tipo 2 vale 50, y el resto vale 60 pts.

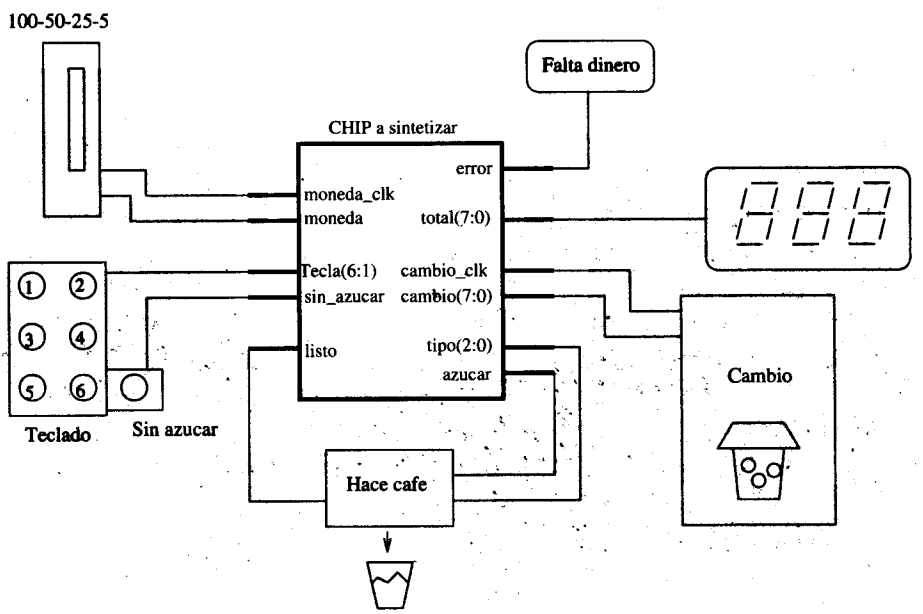


Figura 12.9: Figura del ejercicio de la máquina de café

Este ejercicio no es tan complejo como el anterior pero también tiene sus pegas.

APÉNDICE A

NOTACIÓN BNF DEL VHDL'93

A.1 NOTACIÓN BNF

BNF es el acrónimo de "Backus Naur Form". John Backus y Peter Naur introdujeron por primera vez una notación formal para describir la sintaxis de un lenguaje dado, en concreto fue el lenguaje ALGOL 60. La mayor parte de la notación BNF fue introducida por Backus en un informe de una conferencia de la UNESCO sobre ALGOL 58. Poca gente leyó este informe en aquel momento. Cuando Peter Naur lo leyó, vio que era diferente a lo que él pensaba que sería la descripción de este lenguaje. De esta manera, introdujo unos cambios a la definición propuesta por Backus y escribió un informe para el ALGOL 60 con las nuevas modificaciones. Esta segunda propuesta es la mundialmente aceptada y usada. La discusión sobre quién merece el mérito está en el aire, uno fue el precursor, pero la notación finalmente usada es la del otro.

Desde aquel momento, prácticamente todos los autores de libros sobre algún nuevo lenguaje de programación han utilizado esta notación para especificar las reglas sintácticas del lenguaje.

La notación BNF es en realidad muy simple pero potente, ya que permite fácilmente utilizar una notación para definir la sintaxis de un lenguaje. Los símbolos de la notación BNF son los siguientes:

`::=` significa "se define como".

`|` significa "o".

<> los paréntesis angulares se utilizan alrededor de los símbolos no-terminales. Sirven para distinguir entre los nombres de las reglas de sintaxis (símbolos no-terminales) y los símbolos terminales que se escriben exactamente como van a ser representados.

La forma en que se define una regla sintáctica BNF que define un símbolo no-terminal es como sigue:

```
noterminal ::= secuencia_de_alternativas
```

La `secuencia_de_alternativas` consiste en cadenas de símbolos terminales y no-terminales separados por el símbolo `|`.

Por ejemplo, la descripción BNF de la sintaxis de un lenguaje podría empezar así:

```
<programa> ::= program
                  <declaraciones>
               begin
                  <sentencias>
               end ;
```

Este ejemplo muestra que el lenguaje tendría la palabra clave `program` que da comienzo, luego seguiría una parte declarativa, a especificar en otra regla, y luego entre las otras palabras clave, que son `begin` y `end`, vendrían todas las sentencias. El programa acaba con un `;` al final.

Éste es el BNF original. A lo largo del tiempo se han ido introduciendo algunas modificaciones y extensiones que permiten una descripción más rica y clara:

- Los elementos que sean opcionales se ponen entre corchetes `[]`. Ejemplo:

```
<if_sentencia> ::= if <expresion_booleana> then
                      <sentencias>
                   [ else
                      <sentencias> ]
                   end if ;
```

- Los elementos que se repitan ninguna, una, o más veces, se encierran entre llaves {}. Ejemplo:

```
<identificador> ::= <letra> | { <letra> | <digito> }
```

Sin esta extensión, la definición hubiera sido recursiva:

```
<identificador> ::= <letra> |
                    <identificador> [ <letra> | <digito> ]
```

- Los elementos terminales de un solo carácter suelen ir encerrados entre comillas dobles " ", aunque algunas veces no se ponen las comillas.
- En publicaciones recientes se distinguen los terminales de los no-terminales por el tipo de letra. Por ejemplo, las palabras clave se ponen en negrita mientras que los no-terminales se dejan con la letra normal eliminándose los <> que de esta manera ya no son necesarios. Con esto se gana en legibilidad.
- Como último ejemplo, aunque no el más simple, se da la definición de la notación BNF en BNF:

```
sintaxis    ::= { regla }
regla       ::= identificador  "::="  expresion
expresion   ::= term { "|" term }
term        ::= factor { factor }
factor      ::= identificador |
                simbolo_comillas |
                "(" expresion ")" |
                "[" expresion "]" |
                "{" expresion "}"
identificador ::= letra { letra | digito }
simbolo_comillas ::= """ { any_character } """
```

A.2 VHDL'93

A lo largo del libro se ha utilizado la notación BNF para mostrar la sintaxis de la mayoría de instrucciones y construcciones sintácticas, sin embargo, a fin de aclarar la exposición, se han dado las descripciones parciales o simplificadas de cada una de las estructuras vistas. En ocasiones la descripción ha sido exacta, pero en otras no, debido básicamente a que el VHDL es un lenguaje con una sintaxis compleja.

Para aquellos que deseen explorar las diferentes posibilidades de cada una de las instrucciones, se da a continuación la descripción completa de la sintaxis en notación BNF.

Nota: Es posible que algunas reglas no se encuentren en la descripción. No es que falten, es que simplemente hay sinónimos. Por ejemplo, la regla `entity_header` depende de dos reglas más, `formal_generic_clause` y `formal_port_clause`, en cambio, estas dos reglas no se encuentran en ningún otro sitio. Cuando ocurre esto hay que quitar la primera palabra, y lo que queda es la regla que hay que aplicar.

```
abstract_literal ::= decimal_literal | based_literal
access_type_definition ::= access subtype_indication
actual_designator ::=
    expression
    | signal_name
    | variable_name
    | file_name
    | open
actual_parameter_part ::= parameter_association_list
actual_part ::=
    actual_designator
    | function_name ( actual_designator )
    | type_mark ( actual_designator )
adding_operator ::= + | - | &
aggregate ::=
    ( element_association { , element_association } )
alias_declaration ::=
    alias alias_designator [ : subtype_indication ] is name [ signature ] ;
alias_designator ::= identifier | character_literal | operator_symbol
allocator ::=
    new subtype_indication
    | new qualified_expression
architecture_body ::=
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture ] [ architecture_simple_name ] ;
architecture_declarative_part ::=
    { block_declarative_item }
architecture_statement_part ::=
    { concurrent_statement }
array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition
assertion ::=
    assert condition
        [ report expression ]
        [ severity expression ]
assertion_statement ::= [ tag : ] assertion ;
association_element ::=
    [ formal_part => ] actual_part
association_list ::=
    association_element { , association_element }
attribute_declaration ::=
    attribute identifier : type_mark ;
attribute_designator ::= attribute_simple_name
attribute_name ::=
    prefix [ signature ] ' attribute_designator [ ( expression ) ]
attribute_specification ::=
    attribute attribute_designator of entity_specification is expression ;
base ::= integer
base_specifier ::= B | O | X
base_unit_declaration ::= identifier ;
based_integer ::=
    extended_digit { [ underline ] extended_digit }
based_literal ::=
    base # based_integer [ . based_integer ] # [ exponent ]
basic_character ::=
    basic_graphic_character | format_effector
basic_graphic_character ::=
    upper_case_letter | digit | special_character| space_character
basic_identifier ::=
    letter { [ underline ] letter_or_digit }
binding_indication ::=
    [ use entity_aspect ]
```

```
    [ generic_map_aspect ]
    [ port_map_aspect ]
bit_string_literal ::=  base_specifier " bit_value "
bit_value ::= extended_digit { [ underline ] extended_digit }
block_configuration ::=
    for block_specification
        { use_clause }
        { configuration_item }
    end for ;
block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
block_declarative_part ::=
    { block_declarative_item }
block_header ::=
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]
block_specification ::=
    architecture_name
    | block_statement_tag
    | generate_statement_tag [ ( index_specification ) ]
block_statement ::=
    block_tag :
        block [ ( guard_expression ) ] [ is ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_tag ] ;
block_statement_part ::=
    { concurrent_statement }
case_statement ::=
    [ case_tag : ]
        case expression is
            case_statement_alternative
            { case_statement_alternative }
        end case [ case_tag ] ;
case_statement_alternative ::=
    when choices =>
        sequence_of_statements
character_literal ::= ' graphic_character
choice ::=
    simple_expression
    | discrete_range
    | element_simple_name
    | others
choices ::= choice { | choice }
component_configuration ::=
    for component_specification
```

```
        [ binding_indication ; ]
        [ block_configuration ]
    end for ;
component_declaration ::=
    component identifier [ is ]
        [ local_generic_clause ]
        [ local_port_clause ]
    end component [ component_simple_name ] ;
component_instantiation_statement ::=
    instantiation_tag :
        instantiated_unit
            [ generic_map_aspect ]
            [ port_map_aspect ] ;
component_specification ::=
    instantiation_list : component_name
composite_type_definition ::=
    array_type_definition
    | record_type_definition
concurrent_assertion_statement ::=
    [ tag : ] [ postponed ] assertion ;
concurrent_procedure_call_statement ::=
    [ tag : ] [ postponed ] procedure_call ;
concurrent_signal_assignment_statement ::=
      [ tag : ] [ postponed ] conditional_signal_assignment
    | [ tag : ] [ postponed ] selected_signal_assignment
concurrent_statement ::=
    block_statement
    | process_statement
    | concurrent_procedure_call_statement
    | concurrent_assertion_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement
condition ::= boolean_expression
condition_clause ::= until condition
conditional_signal_assignment ::=
    target <= options conditional_waveforms ;
conditional_waveforms ::=
    { waveform when condition else }
    waveform [ when condition ]
configuration_declaration ::=
    configuration identifier of entity_name is
        configuration_declarative_part
        block_configuration
    end [ configuration ] [ configuration_simple_name ] ;
configuration_declarative_item ::=
    use_clause
    | attribute_specification
    | group_declaration
configuration_declarative_part ::=
    { configuration_declarative_item }
configuration_item ::=
    block_configuration
    | component_configuration
configuration_specification ::=
    for component_specification binding_indication ;
constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;
constrained_array_definition ::=
    array index_constraint of element_subtype_indication
constraint ::=
    range_constraint
    | index_constraint
context_clause ::= { context_item }
context_item ::=
    library_clause
    | use_clause
```

```
decimal_literal ::= integer [ . integer ] [ exponent ]
declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | group_template_declaration
    | group_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration
delay_mechanism ::=
    transport
    | [ reject time_expression ] inertial
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
designator ::= identifier | operator_symbol
direction ::= to | downto
disconnection_specification ::=
    disconnect guarded_signal_specification after time_expression ;
discrete_range ::= discrete_subtype_indication | range
element_association ::=
    [ choices => ] expression
element_declaration ::=
    identifier_list : element_subtype_definition ;
element_subtype_definition ::= subtype_indication
entity_aspect ::=
      entity entity_name [ ( architecture_identifier) ]
    | configuration configuration_name
    | open
entity_class ::=
    entity       | architecture  | configuration
    | procedure  | function      | package
    | type       | subtype       | constant
    | signal     | variable      | component
    | label      | literal       | units
    | group      | file
entity_class_entry ::= entity_class [ <> ]
entity_class_entry_list ::=
    entity_class_entry { , entity_class_entry }
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
      [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;
entity_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
```

```
    | use_clause
    | group_template_declaration
    | group_declaration
entity_declarative_part ::=
    { entity_declarative_item }
entity_designator ::= entity_tag [ signature ]
entity_header ::=
    [ formal_generic_clause ]
    [ formal_port_clause ]
entity_name_list ::=
    entity_designator { , entity_designator }
    | others
    | all
entity_specification ::=
    entity_name_list : entity_class
entity_statement ::=
    concurrent_assertion_statement
    | passive_concurrent_procedure_call_statement
    | passive_process_statement
entity_statement_part ::=
    { entity_statement }
entity_tag ::= simple_name | character_literal | operator_symbol
enumeration_literal ::= identifier | character_literal
enumeration_type_definition ::=
    ( enumeration_literal { , enumeration_literal } )
exit_statement ::=
    [ tag : ] exit [ loop_tag ] [ when condition ] ;
exponent ::= E [ + ] integer | E - integer
expression ::=
      relation { and relation }
    | relation { or relation }
    | relation { xor relation }
    | relation [ nand relation ]
    | relation [ nor relation ]
    | relation { xnor relation }
extended_digit ::= digit | letter
extended_identifier ::=
     graphic_character { graphic_character }
factor ::=
    primary [ ** primary ]
    | abs primary
    | not primary
file_declaration ::=
    file identifier_list : subtype_indication file_open_information ;
file_logical_name ::= string_expression
file_open_information ::=
    [ open file_open_kind_expression ] is file_logical_name
file_type_definition ::=
    file of type_mark
floating_type_definition := range_constraint
formal_designator ::=
    generic_name
    | port_name
    | parameter_name
formal_parameter_list ::= parameter_interface_list
formal_part ::=
    formal_designator
    | function_name ( formal_designator )
    | type_mark ( formal_designator )
full_type_declaration ::=
    type identifier is type_definition ;
function_call ::=
    function_name [ ( actual_parameter_part ) ]
generate_statement ::=
```

```
    generate_tag :
        generation_scheme generate
            [ { block_declarative_item }
        begin ]
            { concurrent_statement }
        end generate [ generate_tag ] ;
generation_scheme ::=
    for generate_parameter_specification
    | if condition
generic_clause ::=
    generic ( generic_list ) ;
generic_list ::= generic_interface_list
generic_map_aspect ::=
    generic map ( generic_association_list )
graphic_character ::=
    basic_graphic_character | lower_case_letter | other_special_character
group_constituent ::= name | character_literal
group_constituent_list ::= group_constituent { , group_constituent }
group_template_declaration ::=
    group identifier is ( entity_class_entry_list ) ;
group_declaration ::=
    group identifier : group_template_name ( group_constituent_list ) ;
guarded_signal_specification ::=
    guarded_signal_list : type_mark
identifier ::=
    basic_identifier | extended_identifier
identifier_list ::= identifier { , identifier }
if_statement ::=
    [ if_tag : ]
        if condition then
            sequence_of_statements
        { elsif condition then
            sequence_of_statements }
        [ else
            sequence_of_statements ]
        end if [ if_tag ] ;
incomplete_type_declaration ::=  type identifier ;
index_constraint ::= ( discrete_range { , discrete_range } )
index_specification ::=
    discrete_range
    | static_expression
index_subtype_definition ::= type_mark range <>
indexed_name ::= prefix ( expression { , expression } )
instantiated_unit ::=
    [ component ] component_name
    | entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
instantiation_list ::=
    instantiation_tag { , instantiation_tag }
    | others
    | all
integer ::= digit { [ underline ] digit }
integer_type_definition ::= range_constraint
interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication
    [ := static_expression ]
interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration
    | interface_file_declaration
interface_element ::= interface_declaration
interface_file_declaration ::=
    file identifier_list : subtype_indication
interface_list ::=
    interface_element { ; interface_element }
```

```
interface_signal_declaration ::=
    [signal] identifier_list : [ mode ] subtype_indication [ bus ]
    [ := static_expression ]
interface_variable_declaration ::=
    [variable] identifier_list : [ mode ] subtype_indication
    [ := static_expression ]
iteration_scheme ::=
    while condition
    | for loop_parameter_specification
tag ::= identifier
letter ::= upper_case_letter | lower_case_letter
letter_or_digit ::= letter | digit
library_clause ::= library logical_name_list ;
library_unit ::=
    primary_unit
    | secondary_unit
literal ::=
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | null
logical_name ::= identifier
logical_name_list ::= logical_name { , logical_name }
logical_operator ::= and | or | nand | nor | xor | xnor
loop_statement ::=
    [ loop_tag : ]
        [ iteration_scheme ] loop
            sequence_of_statements
        end loop [ loop_tag ] ;
miscellaneous_operator ::= ** | abs | not
mode ::= in | out | inout | buffer | linkage
multiplying_operator ::= * | / | mod | rem
name ::=
    simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name
next_statement ::=
    [ tag : ] next [ loop_tag ] [ when condition ] ;
null_statement ::= [ tag : ] null ;
numeric_literal ::=
    abstract_literal
    | physical_literal
object_declaration ::=
    constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
operator_symbol ::= string_literal
options ::= [ guarded ] [ delay_mechanism ]
package_body ::=
    package body package_simple_name is
        package_body_declarative_part
    end [ package body ] [ package_simple_name ] ;
package_body_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | shared_variable_declaration
    | file_declaration
```

```
    | alias_declaration
    | use_clause
    | group_template_declaration
    | group_declaration
package_body_declarative_part ::=
    { package_body_declarative_item }
package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package ] [ package_simple_name ] ;
package_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
package_declarative_part ::=
    { package_declarative_item }
parameter_specification ::=
    identifier in discrete_range
physical_literal ::= [ abstract_literal ] unit_name
physical_type_definition ::=
    range_constraint
        units
            base_unit_declaration
            { secondary_unit_declaration }
        end units [ physical_type_simple_name ]
port_clause ::=
    port ( port_list ) ;
port_list ::= port_interface_list
port_map_aspect ::=
    port map ( port_association_list )
prefix ::=
    name
    | function_call
primary ::=
    name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | ( expression )
primary_unit ::=
    entity_declaration
    | configuration_declaration
    | package_declaration
procedure_call ::= procedure_name [ ( actual_parameter_part ) ]
procedure_call_statement ::=
    [ tag : ] procedure_call ;
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
```

```
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration
process_declarative_part ::=
    { process_declarative_item }
process_statement ::=
    [ process_tag : ]
        [ postponed ] process [ ( sensitivity_list ) ] [ is ]
            process_declarative_part
        begin
            process_statement_part
        end [ postponed ] process [ process_tag ] ;
process_statement_part ::=
    { sequential_statement }
qualified_expression ::=
    type_mark ' ( expression )
    | type_mark ' aggregate
range ::=
    range_attribute_name
    | simple_expression direction simple_expression
range_constraint ::= range range
record_type_definition ::=
    record
        element_declaration
        { element_declaration }
    end record [ record_type_simple_name ]
relation ::=
    shift_expression [ relational_operator shift_expression ]
relational_operator ::=   =  |  /=  |  <  |  <=  |  >  |  >=
report_statement ::=
    [ tag : ]
        report expression
            [ severity expression ] ;
return_statement ::=
    [ tag : ] return [ expression ] ;
scalar_type_definition ::=
    enumeration_type_definition   | integer_type_definition
    | floating_type_definition    | physical_type_definition
secondary_unit ::=
    architecture_body
    | package_body
secondary_unit_declaration ::=  identifier = physical_literal ;
selected_name ::= prefix . suffix
selected_signal_assignment ::=
    with expression select
        target  <= options selected_waveforms ;
selected_waveforms ::=
    { waveform when choices , }
    waveform when choices
sensitivity_clause ::=  on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }
sequence_of_statements ::=
    { sequential_statement }
sequential_statement ::=
    wait_statement
    | assertion_statement
    | report_statement
    | signal_assignment_statement
    | variable_assignment_statement
```

```
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
shift_expression ::=
    simple_expression [ shift_operator simple_expression ]
shift_operator ::= sll | srl | sla | sra | rol | ror
sign ::= + | -
signal_assignment_statement ::=
    [ tag : ] target <= [ delay_mechanism ] waveform ;
signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ]
    [ := expression ] ;
signal_kind ::= register | bus
signal_list ::=
    signal_name { , signal_name }
    | others
    | all
signature ::= [ [ type_mark { , type_mark } ] [ return type_mark ] ]
simple_expression ::=
    [ sign ] term { adding_operator term }
simple_name ::= identifier
slice_name ::= prefix ( discrete_range )
string_literal ::= " { graphic_character } "
subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ subprogram_kind ] [ designator ] ;
subprogram_declaration ::=
    subprogram_specification ;
subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration
subprogram_declarative_part ::=
    { subprogram_declarative_item }
subprogram_kind ::= procedure | function
subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
    | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
        return type_mark
subprogram_statement_part ::=
    { sequential_statement }
subtype_declaration ::=
    subtype identifier is subtype_indication ;
subtype_indication ::=
    [ resolution_function_name ] type_mark [ constraint ]
suffix ::=
```

```
    simple_name
    | character_literal
    | operator_symbol
    | all
target ::=
    name
    | aggregate
term ::=
    factor { multiplying_operator factor }
timeout_clause ::= for time_expression
type_conversion ::= type_mark ( expression )
type_declaration ::=
    full_type_declaration
    | incomplete_type_declaration
type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
type_mark ::=
    type_name
    | subtype_name
unconstrained_array_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
        of element_subtype_indication
use_clause ::=
    use selected_name { , selected_name } ;
variable_assignment_statement ::=
    [ tag : ] target  := expression ;
variable_declaration ::=
    [ shared ] variable identifier_list : subtype_indication [ := expression ] ;
wait_statement ::=
    [ tag : ] wait [ sensitivity_clause ] [ condition_clause ]
                  [ timeout_clause ] ;
waveform ::=
    waveform_element { , waveform_element }
    | unaffected
waveform_element ::=
    value_expression [ after time_expression ]
    | null [ after time_expression ]
```

APÉNDICE B

VHDL Y HERRAMIENTAS EN INTERNET

Actualmente Internet se ha convertido en la mayor, y más rápida, fuente de información en todo el mundo. Cualquier tema, por raro que sea, tiene su página en la red. Naturalmente VHDL no es menos, y mucha de la información útil sobre VHDL se puede encontrar directamente navegando por la red.

La proliferación de páginas web sobre VHDL ha sido tan abrumadora en los últimos años que resultaría absurdo dar una lista con todas ellas. En cambio, hay ciertas páginas que contienen un buen porcentaje de información útil, pero sobre todo, poseen numerosos enlaces a otras páginas, convirtiéndose en verdaderas puertas de entrada a un tema tan amplio como el VHDL.

Una de estas *puertas de entrada* se encuentra en:

`tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html`

En esta página se puede encontrar de todo: modelos, artículos, bibliografía, cursos, bibliotecas matemáticas, otros enlaces, etc. Es sin duda una de las primeras páginas a visitar. Otra interesante es la del VHDL internacional (`www.vhdl.org`).

Una de las primeras características que se observan al navegar por la red es la carencia de herramientas de dominio público. Normalmente, las desarrolladas como dominio público, no son todavía muy buenas, están en periodo de prueba, y no ofrecen una interfaz muy fácil de usar. Hace poco, sin embargo, algunas compañías fabricantes de dispo-

sitivos programables están ofreciendo de forma gratuita programas de síntesis basados en VHDL. A veces el software es completo, y otras veces es una muestra de evaluación con una potencia media de síntesis, pero que es más que suficiente para iniciarse.

Algunas de las compañías que ofrecen software de síntesis completo o en evaluación son las siguientes:

Página web	Compañía	Alcance
www.altera.com	ALTERA	PLDs y CPLDs de Altera
www.cypress.com	Cypress Semiconductors	PLDs y FPGAs de Cypress
www.xilinx.com	Xilinx Inc.	FPGAs y PLDs de Xilinx
www.actel.com	Actel	FPGAs y PLDs de Actel
www.synopsys.com	Synopsys	Dispositivos de varias compañías

Synopsys es una compañía de herramientas, de manera que no se trata de una herramienta específica para una tecnología concreta, sino que permite la síntesis de casi cualquier dispositivo lógico del mercado (incluyendo el resto de los de la tabla). Es una herramienta muy completa y potente desde el punto de vista de la síntesis de alto nivel. Dependiendo también de la versión de Synopsys, es posible no sólo sintetizar, sino también simular el código VHDL. Lo que ocurre con el resto de herramientas de síntesis es que la simulación se realiza teniendo en cuenta el chip sobre el que se ha sintetizado, y como ya se explicó muchas veces, lo sintetizado no siempre se corresponde con lo que se simula.

Las herramientas específicas de cada compañía son interesantes ya que, por un lado, optimizan mejor para las arquitecturas específicas de sus dispositivos, y por otro, se puede realizar la programación de los dispositivos utilizando el mismo software. Lo que también ofrecen estas herramientas es una simulación del chip sintetizado mucho más precisa que la que pudieran ofrecer otras herramientas.

La mayoría de estas herramientas funcionan sobre un PC con Windows'95 o NT. Algunas están disponibles también para workstations (normalmente Sun o HP), aunque no suele haber software de dominio público para estas plataformas.

Hay otras compañías fabricantes de dispositivos programables, como Atmel, Lattice Semiconductor, Lucent Technologies, QuickLogic Corp, etc., pero hasta esta edición no se tiene noticia de que ofrezcan herramientas de evaluación, o completas, de forma gratuita. Naturalmente esto puede haber cambiado.

En `www.prep.org` se pueden encontrar *testbenches* para evaluar las diferentes herramientas de síntesis.

Aunque es relativamente sencillo encontrar herramientas de síntesis que utilicen el lenguaje VHDL, es bastante complicado encontrar herramientas puramente de simulación de VHDL. Sólo actualmente se tiene constancia de una que funciona al completo durante un mes, pero luego sigue funcionando aunque con limitaciones (aun con las limitaciones, la herramienta viene muy bien como introducción). Este software está disponible en `www.veribest.com/vhdl.html`. Existen otras compañías que tienen herramientas de simulación, pero es muy raro que las ofrezcan, aun como muestra para evaluación.

En cuanto a páginas españolas conviene destacar la del GUVE (Grupo de Usuarios de VHDL en España) que se encuentra en
`www.uc3m.es/uc3m/dpto/IN/dpin08/GUVE/`

Por otro lado, este mismo libro tiene su propia página de web en
`tapec.uv.es/VHDL/libro.html`
donde se pueden encontrar los códigos de los ejercicios y ejemplos que aparecen, así como numerosos enlaces a otras páginas relacionadas con el VHDL.

APÉNDICE C

CD-ROM CON HERRAMIENTAS PARA VHDL

Este libro incluye un CD-ROM con herramientas de CAD que contienen un simulador y una herramienta de síntesis para Windows 95/98 o NT. La diferencia entre un simulador de VHDL y una herramienta de síntesis es que el simulador interpreta las sentencias del lenguaje según el estándar, mientras que una herramienta de síntesis puede interpretarlas de formas diferentes. Por otro lado, las herramientas de síntesis sólo entienden un subconjunto del lenguaje, lo que presenta una gran limitación ya que no permiten el modelado de circuitos.

Normalmente es sencillo encontrar herramientas de síntesis en Internet, ya en el apéndice B se dieron algunas direcciones para este tipo de herramientas. Sin embargo resulta algo más complejo encontrar herramientas de simulación completas aunque sea para evaluación. En este sentido el contenido de este CD-ROM es un poco atípico, ya que por un lado se incluye una herramienta de simulación que sí que se encuentra en Internet aunque cuesta bastante descargarla, y por otro una herramienta de síntesis que aunque es de dominio público no está disponible de forma directa por la red.

Ambas herramientas son versiones de evaluación de sus correspondientes distribuciones comerciales, lo cual quiere decir que tienen algunas limitaciones. En el caso del simulador VeriBest la limitación es básicamente la longitud del programa que no puede pasar de 2.000 líneas, aunque existe la posibilidad de probarlo durante 30 días al cien por cien de sus posibilidades pidiendo una licencia de prueba que se entrega gratuitamente. En el caso del sintetizador de Altera, hay que pedir por Internet una licencia para que funcione con limitaciones. Estas limitaciones están en los dispositivos que se pueden seleccionar ya que sólo se dispone de dos. A pesar de todo, y aun con las li-

mitaciones impuestas, se trata de unas herramientas con bastantes posibilidades incluso para el programador más avanzado.

Aparte del fichero *leeme.txt* en el directorio raíz del CD-ROM se encuentran los siguientes directorios:

Ejemplos Incluye todos los ejemplos en VHDL que aparecen en el libro. Muchos de ellos vienen con sus patrones de test para ser simulados. Los ejemplos se han ordenado por capítulos de manera que hay un directorio para cada uno de ellos. Es interesante leer el fichero *leeme.txt* que se encuentra en este directorio, ya que describe el contenido de los diferentes ejemplos.

MaxPlus2 Incluye los ficheros de instalación de la herramienta de Altera MAX+PLUS II Student Edition ver. 7.21. No se trata únicamente de una herramienta de síntesis de VHDL, sino de un software completo de diseño de circuitos de lógica programable (PLDs) que incluye varias entradas en diferentes lenguajes (entre ellos VHDL), captura de esquemas, diferentes tipos de simulación, etc. Su única limitación es que sólo hay unos pocos dispositivos sobre los que se puede realizar la síntesis.

VeriBest Incluye los ficheros de instalación del simulador de VHDL de VeriBest. Se encuentra también una versión de evaluación de un simulador eléctrico de características similares a Spice.

C.1 INSTALACIÓN Y PRUEBA DEL SINTETIZADOR DE VHDL

Como ya se ha dicho anteriormente el software MAX+PLUS II de Altera es en realidad un sistema completo de diseño hardware, esto quiere decir que la herramienta de síntesis a partir de VHDL es una herramienta más dentro de este sistema de diseño completo. Lo que se explica a continuación es la instalación del software de Altera, así como la utilización de la herramienta de síntesis.

La instalación de todo el software se realiza en un momento. Se necesita un PC compatible de prestaciones medias o altas. Funciona casi con cualquier versión de Windows, aunque se aconseja Windows95. Para la instalación del software y la licencia hay que seguir estos pasos:

- Ejecutar el fichero *\MaxPlus2\maxplus2\Install.exe* y seguir las sencillas instrucciones que se dan. (Básicamente sólo pide que se indique el lugar donde se instalará y el directorio de trabajo.)

- Una vez instalado aparecerá el grupo de programas MAX+plus II con un único icono que es el acceso directo al programa. Ejecutarlo.

- Una vez lanzado el programa aparece un mensaje de bienvenida que anuncia que si se tiene el código de autorización se puede pulsar OK y continuar, pero que si no, hay que conectarse al web de Altera e introducir el código que aparece en la parte inferior del mensaje. Como es la primera vez que se lanza el programa aún no se dispone de este código, para conseguirlo hay que apuntarse el número que aparece en el mensaje y seguir los pasos que siguen a continuación.

- Una vez se tiene apuntado el código que aparece por primera vez al entrar, hay que conectarse a la página *http://www.altera.com/maxplus2-student*. En esta página aparece un formulario simple donde habrá que introducir este número que se tiene apuntado. Hay que poner también una dirección de correo electrónico, ya que es a esa dirección a donde se envía el código de autorización, o llave software, que permite la utilización del software de Altera.

- Una vez se tiene el código de autorización (la recepción es inmediata a través del correo electrónico) hay que lanzar de nuevo el programa MAX+Plus II Student Edition 7.21, contestando OK al mensaje de bienvenida (ahora ya sí que se tiene código de autorización).

- Al pulsar OK aparece otra ventana donde se pide que se introduzca el código de autorización. Se introduce con cuidado y luego se pulsa *Validate*. Si el código se ha introducido bien se puede comprobar que todas las *Unavailable features* pasan a la caja de *Available features* indicando que ya se puede utilizar el programa.

- Esta operación sólo hay que realizarla la primera vez que se instala, luego ya se puede lanzar el programa y funcionará perfectamente.

- El programa así instalado funciona perfectamente. Sin embargo Altera realiza periódicamente actualizaciones de este software. Para acceder a la última actualización hay que acceder a *http://www.altera.com/html/univ/univ-download.html* y seguir las sencillas instrucciones para traerse el fichero *univers.exe*. Una vez se tiene este fichero hay que copiarlo al directorio donde se encuentra instalado Max+Plus II y entonces ejecutarlo con la opción *-o*, es decir, poner *univers.exe -o* y pulsar intro. Con esto queda instalada la actualización. En el CD-ROM se incluye el fichero *\MaxPlus2\univers.exe* correspondiente a la última actualización disponible en el momento de la edición.

La herramienta de síntesis de Altera es muy completa y posee numerosas opciones. No se pretende a continuación describir el funcionamiento de la herramienta, sino que

simplemente se exponen los pasos mínimos para cargar, simular y compilar un programa. Es interesante, a fin de aprender cómo funciona el sintetizador, seguir con la mayor precisión posible los pasos que se exponen.

- En primer lugar hay que lanzar la herramienta. (Se supone que ya se han introducido los códigos de autorización, etc.)

- Una vez en el entorno de la herramienta existen varias posibilidades. Lo normal es crear un nuevo proyecto, para esto hay que ir al menú *File* > *Project* > *Name*. Entonces aparece una ventana de diálogo con todos los ficheros que haya en el directorio de trabajo que puedan ser susceptibles de ser entrada al diseño (ficheros VHDL, Verilog, captura de esquemas, AHDL, etc.). Si lo que se quiere es crear un nuevo proyecto se introducirá un nombre sin extensión.

- Una vez hecho esto se ha creado un fichero con la información del proyecto en curso, pero todavía no se ha especificado un fichero de entrada VHDL. Para esto hay que hacer *File* > *New* y en el menú que aparece elegir la opción *Text editor File*. Con esto aparece una ventana de texto para que se introduzca la descripción del circuito en VHDL.

- Para asociar esa entrada de texto con el proyecto con el que se está trabajando hay que salvarlo al disco. Para ello, y con la ventana de texto activa, hay que hacer *File* > *Save*, momento en el cual aparece una ventana de diálogo donde se indica el nombre del fichero. Por defecto aparece el nombre del proyecto con la extensión *.tdf*, entonces hay que cambiarle esta extensión y poner *.vhd*; esto es muy importante, ya que dependiendo de la extensión que se ponga aquí entenderá que la entrada es VHDL u otro de los formatos que maneja.

- Otra forma de empezar un diseño en Max+Plus II es abriendo un fichero VHDL que ya exista. En ese caso se hace también *File* > *Project* > *Name* y se selecciona el fichero VHDL, siempre que tenga la extensión correcta (*.vhd*). Con esto se genera un proyecto con el mismo nombre que tenía el fichero pero sin la extensión. Automáticamente se entiende que ese fichero VHDL es el superior de la jerarquía, es decir, el fichero a partir del cual se compila el diseño.

- Una vez se tiene un fichero raíz (*Top*) de la jerarquía de diseño se puede pasar a compilarlo. Para esto hay que ir al menú *Max+Plus II* > *Compiler* con lo que aparece la ventana del compilador. Existen numerosas opciones con las que experimentar. Si las opciones por defecto se consideran adecuadas, y habitualmente lo son en la mayoría de los casos, bastará con pulsar el botón *Start* en la ventana de compilación. Si la compilación da errores éstos aparecen en una ventana de manera que se pueden seleccionar para obtener ayuda o bien para localizar el error en el texto.

- Una vez compilado se puede simular. La simulación que se realiza es la simulación del dispositivo que se ha programado y no la simulación del código VHDL. Esto quiere decir que la información sobre los retrasos viene dada por el dispositivo elegido para realizar la síntesis. Existe sin embargo la posibilidad de simular funcionalmente de manera que los retrasos no aparecen.

- Antes de ejecutar el simulador hay que crear los estímulos. Esto se puede hacer de varias formas, la más sencilla es la siguiente: Lo primero es abrir un fichero de formas de onda; esto se hace mediante *File > New > Waveform Editor file*. Nada más crearlo se puede salvar para que tenga ya así un nombre. Se salva con *File > Save*, con lo que aparece la ventana de diálogo de guardar en disco, la propia herramienta propone un nombre (el del proyecto) y una extensión (*.scf*), así que sólo hay que pulsar OK.

- Con el fichero de formas de onda abierto se le introducirán las entradas y salidas así como los nodos internos que se quieran ver. Lo normal, y para empezar, es suficiente con poner las entradas y salidas del sistema. Esto se puede hacer fácilmente, siempre que se haya compilado con éxito y se tenga seleccionada la ventana de formas de onda, haciendo *Node > Enter nodes from SNF*, que lo que hace es tomar los nodos de la descripción y listarlos para que se puedan seleccionar los que se quieran. Una vez seleccionados se pulsa OK.

- Una vez se tienen las entradas y salidas al sistema en la ventana de formas de onda éstas se pueden editar. Esto se hace de forma interactiva con el ratón o bien seleccionando el nodo y luego haciendo *Edit > Overwrite*, donde aparecen varias opciones para introducir valores altos, bajos, señal de reloj, etc.

- Una vez introducidos los estímulos se puede simular. Para ello hay que ejecutar *Max+Plus II > Simulator*. Al arrancar el simulador lee el fichero de formas de onda (*.scf*). Si se llama igual que el proyecto lo cargará automáticamente, pero si tiene otro nombre, o si se tienen varios ficheros de estímulos, hay que indicar qué fichero cargar haciendo *File > Inputs/Outputs*. Una vez cargado el fichero de estímulos basta con pulsar el botón de *Start* en la ventana del simulador. Los resultados se mostrarán en el propio fichero de estímulos.

- El último paso en el ciclo de diseño, y una vez que se ha compilado y simulado con éxito, es el de descargar el programa en el dispositivo seleccionado. Para ello hace falta un hardware especial de programación suministrado por Altera.

C.2 INSTALACIÓN Y PRUEBA DEL SIMULADOR DE VHDL

Aparte del fichero *README_First.hlp* en el directorio VeriBest del CD-ROM se pueden encontrar los siguientes sub-directorios:

VHDL_Simulator En este directorio se encuentra la copia de evaluación del *VeriBest VHDL Simulator*, que incluye también *WaveBench* para la edición gráfica de estímulos y la generación de bancos de prueba. El simulador viene completo, pero si no se tiene licencia, la capacidad de simulación está limitada a 2.000 referencias de componentes o 2.000 líneas de código.

Existe la posibilidad de obtener una licencia completa durante 30 días. Lo único que hay que hacer es conectarse a *http://www.veribest.com/vhdl.html* y seguir el enlace *Download VB VHDL Today*. Con esto se llega a un formulario que hay que rellenar. Una vez completado hay que marcar la caja que pone *Download VeriBest VHDL Software* para que se inicie de esta forma el envío de la licencia vía correo electrónico. Como la licencia se recibe por correo electrónico, hay que especificar bien la dirección electrónica.

Además del ejemplo que viene con la instalación del software (*candy*), se incluyen dos diseños adicionales. Estos ejemplos vienen con ficheros de ejecución por lotes para compilarlos y simularlos con el simulador VeriBest. El resultado es un tiempo de simulación que permite comparar este simulador frente a otros del mercado. Estos diseños se encuentran en el subdirectorio *Benchmark_Designs* debajo del directorio *VHDL_Simulator*.

Analog_Simulator En este directorio se encuentra una copia de evaluación del *VeriBest Analog Individual Simulator*. Es un entorno de simulación analógica, compatible con los modelos de Spice, que permite la simulación temporal, en frecuencia y en continua con la posibilidad de realizar simulaciones estadísticas con variación de parámetros y análisis de distorsión y ruido.

Se puede conseguir una licencia completa durante 30 días conectándose a la dirección *http://www.veribest.com/analog.html*.

Product_Web_Information En este directorio se encuentran las páginas web de VeriBest donde se presenta la compañía haciendo especial hincapié en las herramientas de CAD digitales y analógicas. La puerta de entrada se encuentra en el fichero *index.html* de este mismo directorio.

El software del CD-ROM funciona tanto en Windows 95/98 como en NT. Un compatible PC de prestaciones medias suele ser suficiente para la ejecución del simulador.

La instalación del simulador de VHDL es bastante simple y sólo hay que seguir los siguientes pasos:

- En el directorio *VHDL_Simulator* del CD-ROM hay que ejecutar el programa *Setup.exe*.
- A continuación se pregunta el directorio dónde se desea instalarlo, se puede dejar el que viene por defecto, o bien elegir otro.
- Luego se elige lo que se quiere instalar. En total hay dos programas: el simulador de VHDL y el *WaveBench*. Lo único realmente necesario es el simulador, si se desea instalar la otra herramienta puede hacerse en otra ocasión. El simulador ocupa 29.331 KBytes, y el *WaveBench* 6.850 KBytes.
- Una vez elegido esto la instalación continúa hasta copiar todos los archivos en el disco.
- Una vez instalado se habrá creado un grupo de programas llamado *VeriBest VB99.0*, en el cual se incluye el programa de desinstalación y una carpeta que se llama *VeriBest VHDL Simulator*. En esta carpeta se encuentra la ayuda *On Line*, un fichero *README*, un tutorial y el simulador.

Es muy recomendable poner en marcha el tutorial antes de utilizar el simulador para aprender los rudimentos más comunes. En cualquier caso se dan a continuación los pasos básicos para editar y simular una descripción en VHDL.

- En primer lugar hay que lanzar el programa que se encuentra en la carpeta mencionada anteriormente.
- A continuación hay que crear un espacio de trabajo (*Workspace*). Esto se realiza desde los menús *File* > *New* > *Workspace*.
- Si no se tiene ninguna descripción VHDL se puede crear con *File* > *New* > *VHDL Source File*. Una vez creada, o si ya existía, se debe añadir al espacio de trabajo. Esto se realiza mediante *Workspace* > *Add Files into Workspace*. Si una descripción está formada por varios ficheros habrá que añadirlos al espacio de trabajo.
- Una vez realizados los pasos anteriores ya se está en condiciones de compilar el diseño. Para ello hay que ejecutar *Workspace* > *Compile* > *All*.
- Una vez compilado sin errores se puede empezar a simular, pero antes hay que elegir la entidad y arquitectura *raíz* para la simulación. Este paso es importante,

ya que, si no se hace, al lanzar el simulador se produce un error. La raíz se especifica mediante *Workspace* > *Settings*, de esta manera aparece un menú con varias carpetas, hay que elegir la de *Simulate*. A la izquierda del menú aparece *WORK*, que es la biblioteca donde se ha compilado el diseño, y hay que pulsar un par de veces sobre el nombre para que aparezcan las arquitecturas y entidades del diseño. En este momento hay que seleccionar la arquitectura que se desea simular y una vez seleccionada pulsar el botón *Set* a la derecha del menú. Por último hay que pulsar *Aceptar*. De esta manera se ha especificado qué arquitectura se desea simular.

- Para lanzar el simulador hay que seleccionar *Workspace* > *Execute Simulator*, o bien, pulsar Ctrl-F5. Mientras se lanza aparecerá un mensaje diciendo que no hay licencia pero que funcionará con limitaciones.

- Una vez en el simulador hay que abrir una ventana para ver las formas de onda. Esto se realiza con *Tools* > *New WaveForm Window*. A esta ventana hay que añadirle las señales que se desean ver. Para ello hay que pulsar el icono situado más a la izquierda en la misma ventana de formas de onda. A continuación aparece una lista con todas las señales, se pueden seleccionar aquellas que se quieran añadir, o bien directamente se puede pulsar *Add All* que las añadirá todas. Para dejar de añadir señales hay que pulsar *Close* en este menú.

- La definición de los estímulos debe estar incluida como un banco de pruebas en la propia descripción VHDL. Ver la sección 10.5 donde se explica la creación de un banco de pruebas.

- El último paso consiste en simular el circuito. Para ello hay que hacer *Simulate* > *Run* o pulsar F5. Esto simulará durante el periodo especificado que aparece en la barra de utensilios. Se puede repetir esta operación hasta simular el periodo que se desee.

Estos pasos recogen todas las tareas a realizar para simular una descripción. La herramienta permite más funciones, como la ejecución paso a paso, ver por dónde va la simulación, insertar puntos de ruptura, etc. Todas estas características las puede ir descubriendo el lector sólo con navegar un poco por los diferentes menús y la ayuda en línea. Además, existen numerosos botones, barras de herramientas e iconos que agilizan muchas de las operaciones más comunes descritas aquí.

BIBLIOGRAFÍA

[1] IEEE. *IEEE Standard VHDL Languaje Reference Manual, Std 1076-1993*, 1993.

[2] IEEE. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164), Std IEEE 1164-1993, Std 1076-1993*, 1993.

[3] Douglas L. Perry. *VHDL.* McGraw-Hill, segunda edición, 1993.

[4] Peter J. Ashenden. *The designer's guide to VHDL.* Morgan Kaufmann, 1995.

[5] Peter J. Ashenden. *The VHDL Cookbook*, 1990.
ftp://ftp.cs.adelaide.edu.au/pub/VHDL/.

[6] DOULOS. *The VHDL golden reference guide*, 1997.

[7] Stanley Mazor Patricia Langstraat. *A Guide to VHDL.*
Kluwer Academic Publishers, 1992.

[8] D. Coelho. *The VHDL Handbook.* Kluwer Academic Publishers, 1989.

[9] Lluís Terés, Yago Torroja, Serafín Olcoz, Eugenio Villar, *et al. VHDL, lenguaje estándar de diseño electrónico.* McGraw-Hill, 1997.

[10] Zainalabedin Navabi. *VHDL Analysis and Modelling of Digital Systems.*
McGraw-Hill, 1993.

[11] J.M. Schoen. *Performance and Fault Modelling with VHDL.* Prentice Hall, 1992.

[12] Kevin Skahill. *VHDL for programmable logic.* Addison-Wesley, 1996.

[13] Steven Leung y Michael Shanblatt. *ASIC System Design with VHDL: A Paradigm*. Kluwer Academic Publishers, 1989.

[14] Mentor Graphics. *Mentor Graphics Introduction to VHDL*, 1994.

[15] Mentor Graphics. *Mentor Graphics VHDL Reference Manual*, 1994.

[16] Altera. *Altera VHDL*, 1996.

[17] Cypress Semiconductor Corporation. *Programmable logic data book*, 1996.

[18] Cypress Semiconductor Corporation. *WARP VHDL Synthesis Reference*, 1994.

[19] Xilinx Inc. *Porgrammable Logic Data Book*, 1994.

ÍNDICE ALFABÉTICO

M

N

O

P

Q

R

VHDL

Lenguaje para síntesis y modelado de circuitos

Los lenguajes de descripción hardware son los pilares sobre los que se asienta la fuerte evolución que el diseño electrónico digital ha venido sufriendo durante los últimos años, por lo que el VHDL ha emergido como estándar en la industria convirtiéndose en el más utilizado hoy en día.

El objetivo de este libro no es únicamente el de presentar el lenguaje y su sintaxis, sino también el de introducir la metodología de trabajo inherente al lenguaje, ya que se trata del flujo de diseño actual de circuitos digitales. Además, se centra en las dos grandes áreas de aplicación del VHDL: la simulación y la síntesis automática de circuitos. Se compone de 12 capítulos y 3 apéndices, donde se incluyen los temas siguientes:

- Metodología y posibilidades en la descripción del diseño electrónico. Lenguajes de descripción hardware.
- Introducción y sintaxis del lenguaje.
- Estilos de descripción: estructural, flujo de datos y algorítmica.
- Bibliotecas, Paquetes y Unidades. Conceptos avanzados.
- Simulación y modelado.
- Descripción y síntesis automática de circuitos.
- VHDL en la práctica: Ejemplos y ejercicios resueltos.
- VHDL y herramientas CAD.

La estructura y contenido de esta obra están basados en varios años de experiencia en la enseñanza del VHDL y diseño digital, por lo que el principal objetivo perseguido por los autores es su carácter didáctico y pedagógico, sin olvidar que también va dirigido a los ingenieros que actualmente empiezan a incorporar estas técnicas de diseño a su entorno laboral.

Contiene un CD ROM que incluye los ejemplos del libro, software completo de simulación VHDL (Licencia de VeriBest limitada a 2.000 líneas de código), y sintetizador VHDL de ALTERA®; y se requiere un ordenador con sistema operativo Windows 95/98 o NT.

ISBN 84-7897-351-6

ra-ma®