


VNIVERSITAT  VALÈNCIA



UNIVERSITAT DE VALÈNCIA

VHDL

Lenguaje para descripción y modelado de circuitos

INGENIERÍA INFORMÁTICA

Fernando Pardo Carpio

©Fernando Pardo Carpio, 14 de octubre de 1997

Prólogo

Los apuntes que se contienen en las próximas páginas corresponden a parte primera de la asignatura de *Tecnología Informática* que se imparte en el segundo curso de la carrera de Ingeniería Informática de la Universidad de Valencia. Se trata de una asignatura optativa y cuatrimestral con un total de 4.5 créditos teóricos y 1.5 prácticos.

El objetivo de la asignatura es familiarizar al alumno con el flujo de diseño de circuitos electrónicos, desde su especificación hasta su realización. Este flujo comienza con la explicación de las principales herramientas y metodologías para la descripción del diseño. Se pasa por explicar algunos conceptos de simulación tanto digital como eléctrica, y se termina por presentar dos formas en que pueden acabar los diseños electrónicos: circuitos integrados y circuitos impresos. Para cubrir estos objetivos el curso se ha dividido en cuatro materias si bien las dos últimas vienen unidas en una única parte que es la de realización. Estas cuatro materias son:

Lenguajes de descripción hardware En esta materia, que corresponde a la parte de descripción de circuitos, se analizan las diferentes formas de definir y describir circuitos. El tema principal de esta materia es el lenguaje VHDL.

Simulación Esta materia cubre los conceptos básicos de simulación y comprobación de circuitos tanto digitales como analógicos.

Microelectrónica Ya en la parte de realización la primera materia es la de microelectrónica donde se explican los procesos de fabricación de circuitos integrados prestando especial atención al proceso CMOS.

Circuitos Impresos Por último se explica el proceso de fabricación de circuitos impresos o PCBs (Printed Circuit Boards) revisando las diferentes posibilidades tecnológicas tanto de encapsulados como de tolerancia al ruido, etc.

Los objetivos del curso, es decir, recorrer todo el flujo de diseño desde la definición del problema hasta su realización práctica, son extremadamente extensos por lo que en el curso se da prioridad a unos temas dejando otros para ser explicados en otras asignaturas dentro del programa general de la carrera de Ingeniería Informática, y más particularmente de la línea de optatividad del área de arquitectura y tecnología de los computadores.

Considerando los contenidos de otras asignaturas dentro de la carrera, y también las actuales tendencias y demandas de la industria y el diseño hardware, se ha optado por hacer hincapié en los lenguajes de descripción hardware. Es por estas razones que una gran parte del curso está dedicada al lenguaje VHDL como lenguaje de especificación de circuitos tanto para síntesis como para la realización de modelos de simulación, siendo esta parte la que se recoge en estos apuntes.

Fernando Pardo, en Valencia, Octubre de 1997

Índice General

1	Metodología de diseño	1
1.1	Concepto de herramientas CAD-EDA	1
1.2	Diseño <i>Bottom-Up</i>	3
1.3	Diseño <i>Top-Down</i>	4
1.3.1	Ventajas del diseño <i>Top-Down</i>	5
1.4	Ingeniería concurrente	6
2	Descripción del diseño	9
2.1	Captura de esquemas	10
2.2	Generación de símbolos	11
2.3	Diseño modular	12
2.4	Diseño jerárquico	12
2.5	El netlist	13
2.5.1	El formato EDIF	13
2.5.2	Otros formatos de Netlist	14
2.5.3	Ejemplo de diferentes Netlist	15
3	Introducción al lenguaje VHDL	21
3.1	El lenguaje VHDL	22
3.1.1	VHDL describe estructura y comportamiento	23
3.2	Ejemplo básico de descripción VHDL	23
4	Elementos sintácticos del VHDL	27
4.1	Operadores y expresiones	27
4.2	Tipos de datos	29
4.2.1	Tipos escalares	29
4.2.2	Tipos compuestos	30
4.2.3	Subtipos de datos	31
4.3	Atributos	32
4.4	Declaración de constantes, variables y señales	32
4.5	Declaración de entidad y arquitectura	34
5	Ejecución concurrente	39
5.1	Ejecución concurrente y ejecución serie	39
5.2	Descripción comportamental RTL	40
5.3	Estructuras de la ejecución concurrente RTL	41
6	Descripción serie comportamental abstracta	45
6.1	Diferencias entre variable y señal	47
6.2	Estructuras de la ejecución serie	49
7	Poniendo orden: subprogramas, paquetes y librerías	57
7.1	Subprogramas	57
7.1.1	Declaración de procedimientos y funciones	58

7.1.2	Llamadas a subprogramas	59
7.1.3	Sobrecarga de operadores	60
7.2	Librerías, paquetes y unidades	61
7.2.1	Paquetes: PACKAGE y PACKAGE BODY	64
7.2.2	Configuración: CONFIGURATION	65
8	VHDL para simulación	67
8.1	Los retrasos y la simulación	67
8.1.1	Retrasos inerciales y transportados	70
8.2	Descripción de un banco de pruebas	71
8.3	Notificación de sucesos	72
8.3.1	Procesos pasivos	73
9	VHDL para síntesis	75
9.1	Restricciones en la descripción	76
9.2	Construcciones básicas	77
9.2.1	Descripción de lógica combinacional	78
9.2.2	Descripción de lógica secuencial	79
10	Conceptos avanzados en VHDL	81
10.1	Buses y resolución de señales	81
10.2	Descripción de máquinas de estados	84
11	Utilización del lenguaje VHDL	89
11.1	Errores más comunes usando VHDL	89
11.2	Ejemplos para simulación y síntesis	91
11.2.1	El botón	91
11.2.2	Los semáforos	93
11.2.3	El ascensor	95
11.2.4	La memoria ROM	97
11.2.5	El microprocesador	98
11.2.6	La lavadora	100
11.2.7	El concurso	106
11.2.8	El pin-ball	109
11.3	Ejercicios propuestos	111
	Bibliografía	115
	Índice de Materias	117

Índice de Figuras

1.1	<i>Flujo de diseño para sistemas electrónicos y digitales</i>	2
1.2	<i>Metodología de diseño Bottom-Up</i>	4
1.3	<i>Metodología de diseño Top-Down</i>	5
2.1	<i>Ejemplo de esquema para su descripción Netlist</i>	15
3.1	<i>Esquema del ejemplo básico en VHDL</i>	24
7.1	Las librerías y las unidades que la componen	62
8.1	<i>Flujo de simulación por eventos en VHDL</i>	69
8.2	<i>Retrasos inerciales y transportados</i>	70
11.1	<i>Figura del ejercicio de la lavadora</i>	101
11.2	<i>Figura del ejercicio del microondas</i>	112
11.3	<i>Figura del ejercicio de la máquina de café</i>	113

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 como para el dibujo, o diseño gráfico, como para el resto de herramientas que ayudan al diseño (como 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) para la ayuda al diseño de sistemas electrónicos. Dentro del EDA, las herramientas de CAD juegan un importante papel, sin embargo, no sólo el software es importante, workstations cada día más veloces, elementos de entrada de diseño cada vez más sofisticados, etc. son también elementos 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 en detalle en la figura 1.1.

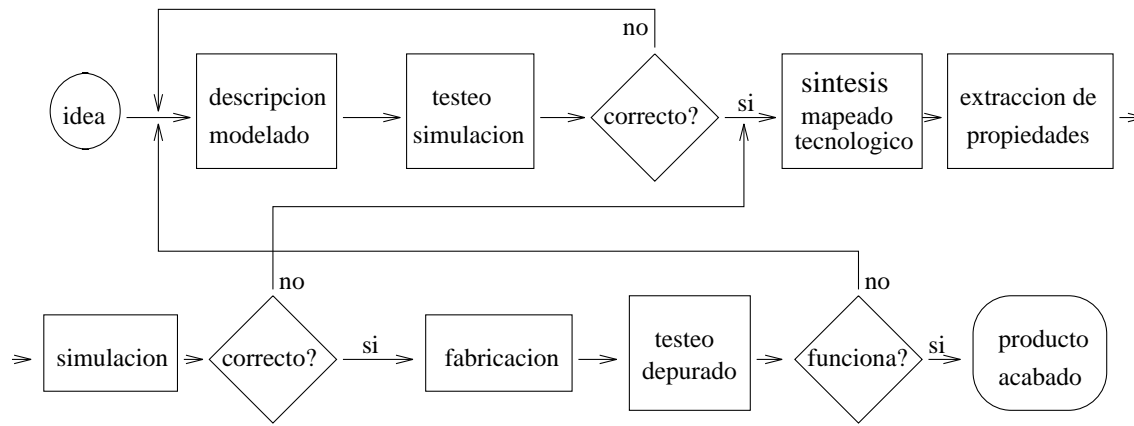


Figura 1.1: *Flujo de diseño para sistemas electrónicos y digitales*

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 simulaciones de 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 (Application Specific Integrated Circuits) herramientas éstas que nos 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 de bloques, donde se muestra la arquitectura del diseño, o 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.

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 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 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 la realización 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 sistemas para, más tarde, agruparlos en diferentes módulos, y estos 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.

Esta metodología de diseño no implica una estructuración jerárquica de los elementos del sistema. Esta estructuración, al contrario de lo que ocurre en el diseño top-down que se verá después, se realiza una vez realizada 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 librería que contiene chips, resistencias, condensadores, y otros elementos que representan unidades funcionales con significado propio dentro del diseño. 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 si 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. Para poder encontrar errores de diseño, o saber si el circuito realizará la

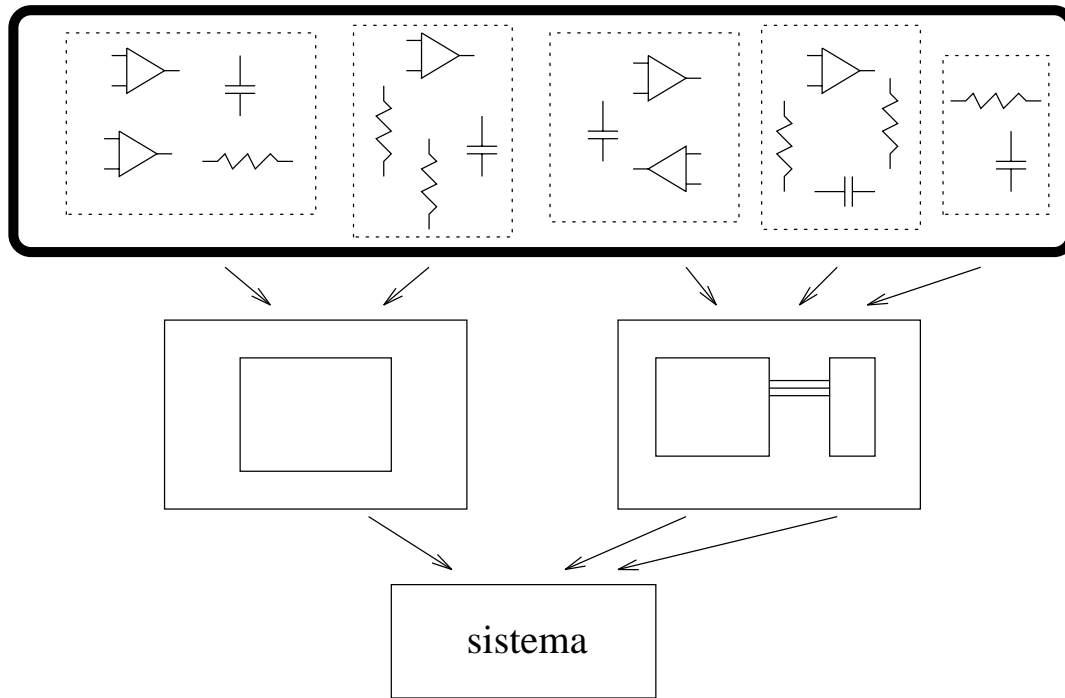


Figura 1.2: Metodología de diseño Bottom-Up

función para la que ha sido diseñado, es necesario perder mucho más tiempo en lo que es la definición, diseño y análisis en alto nivel para ver entonces si funciona como deseamos.

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 implementar esa idea primero en un muy alto nivel, 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.

Los años 80 trajeron una revolución en 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, nos encontramos en un marco en que 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

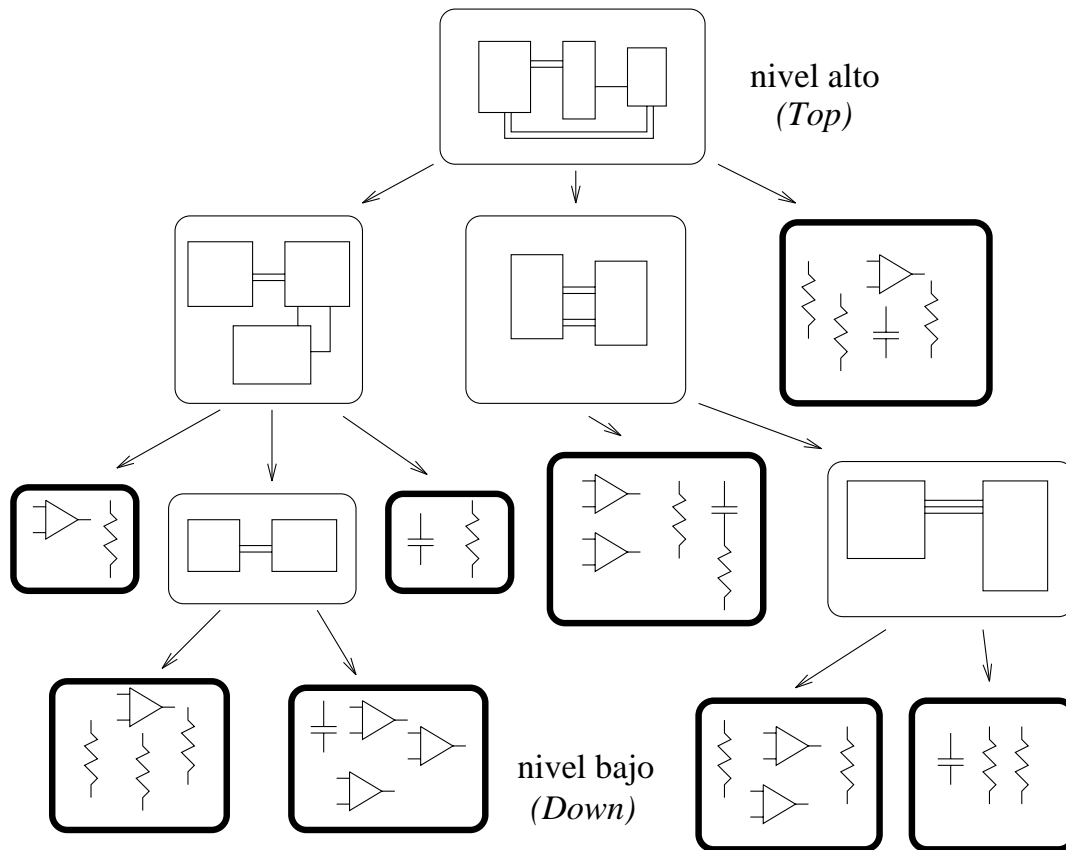


Figura 1.3: Metodología de diseño Top-Down

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.

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 no se fija la tecnología a utilizar 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 muchos errores pronto 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 workstations 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 lo rápido que cada paso de la cadena de diseño podía realizarse.

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 tiene efectos inmediatos sobre la ejecución de otra herramienta.

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 **forwardannotation** 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 **backannotation**. 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 backannotation 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 podemos 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 *backanotar* la información del encapsulado a la herramienta de captura de esquemas.

Capítulo 2

Descripción del diseño

La primera tarea a realizar 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 al computador 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 jugar un papel más importante ya que servía como entrada de información a las herramientas posteriores en el flujo de diseño. Ésto, unido a la metodología Top-down de diseño de circuitos, llevó a la aparición de herramientas de descripción que permitieran al diseñador definir el problema de una forma abstracta de manera que fuera el ordenador quien se ocupara 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 que la electrónica existe. 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

Con 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, instancias, cables, conectores, notas, y muchas otras propiedades importantes y valores necesarios por el resto de aplicaciones para la interpretación del mismo.

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 unas con otras 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 serían las resistencias, condensadores, transistores, puertas lógicas, procesadores, chips de memoria, etc.

Un segundo tipo de símbolos son aquellos que especifican, no un elemento simple, sino otro circuito completo, compuesto a su vez por símbolos, etc. Es decir, este segundo tipo de símbolos son 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 tenemos 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, juntar 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 **nets**. Un hilo en el esquema tiene una correspondiente 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 ser tanto un hilo de cobre, como una pista en un circuito impreso, como un conjunto de hilos, como un cable de una interface 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 interface 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 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 interconectar con otros elementos, no se necesita nada más.

Los símbolos más simples, que corresponden a un elemento físico, se encuentran normalmente agrupados en librerías de símbolos. Para ser usados únicamente se necesita copiarlos de la librería y meterlos en el diseño. Hay otros elementos que no representan primitivas, sino que representan otros esquemas en un nivel más bajo de la jerarquía; en estos casos, los símbolos no suelen estar agrupados en librerías 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 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 el encapsulado con sus diferentes patillas, y las partes de cobre asociadas. Si por el contrario, queremos realizar una simulación para ver el comportamiento del contador, en realidad el símbolo estará haciendo referencia a una descripción del comportamiento del circuito. Y aun pueden 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 muchas cosas 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 realizando 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. Al partir de los elementos básicos de los que se compone el sistema, no resulta tan sencillo agruparlos de forma coherente. Esta 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 que sean organizados de una forma que sea fácil su comprensión. Una forma de organizar el diseño es la creación de un diseño modular jerárquico tal y como se ha venido viendo 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 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 embarcados 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 "Mentor Graphics Corporation"))
```

```

    (program "ENWRITE" (version "v8.4_2.1"))
  )
)
(library (rename &_fasst_pardo_mentor "/fasst/pardo/mentor")
  (edifLevel 0)
  (technology
    (numberDefinition
      (scale 1 (e 1 -6) (unit distance)))
  )
  (cell prutro (cellType generic)
    (view prutro (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 prutro (cellRef prutro (libraryRef &_fasst_pardo_mentor)))
)

```

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 librerías, mientras que sólo las últimas sirven para describir el símbolo. Esta descripción empieza con la sentencia (**cell prutro** (**cellType generic**)) donde se indica que se va a describir una célula llamada internamente **prutro**. A continuación vendría la sección de interface 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 del port. 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 por ejemplo.

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 es sencillo tener una interface. Lo que suelen hacer los fabricantes es utilizar un lenguaje propio y proveer los programas necesarios para pasar de su lenguaje al EDIF y viceversa, de esta manera se aseguran la compatibilidad con el resto de herramientas del mundo, y las suyas propias son más sencillas de realizar.

Un ejemplo de lenguaje de descripción lo tenemos en el Tango, cuyo lenguaje de netlist es muy simple y contempla muchas posibles descripciones, incluida la inclusión 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, este 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 de circuitos digitales. Este lenguaje es utilizado por el simulador para saber exactamente como 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 se trata de un esquema que ha sido generado a partir de la herramienta de captura de esquemas de Tango.

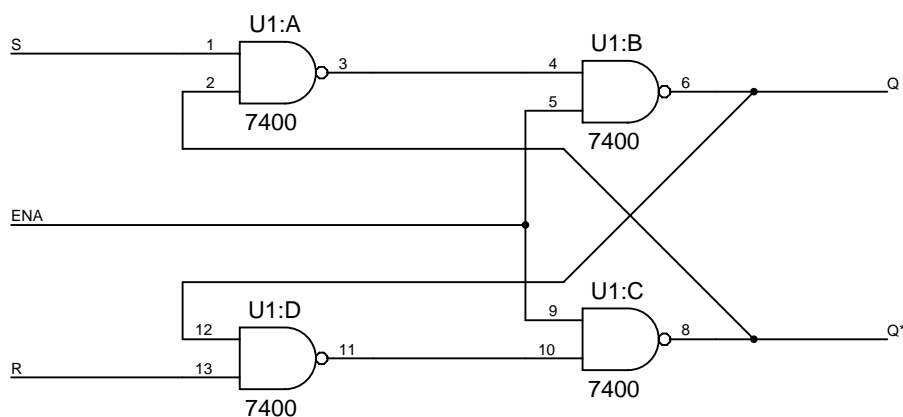


Figura 2.1: *Ejemplo de esquema para su descripción Netlist*

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 "TANGO Schematic" (Version "1.30"))
      (comment "Copyright (C) 1990 ACCEL Technologies Inc.")
    )
  )
  (Design R00T
    (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)))
        )
      )
    )
  )
)

```


[GND)	R
U1	U1-7	(U1-13
DIP14)	Q)
7400	(U1-6	(
]	NET_002	U1-12	S
	U1-10)	U1-1
(U1-11	()
ENA)	Q*	(
U1-5	(U1-2	VCC
U1-9	NET_004	U1-8	U1-14
)	U1-3))
(U1-4	(

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, las primeras líneas encerradas entre corchetes, se encuentra la parte de definición de los elementos. Simplemente 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 es interesante saber de antemano el encapsulado.

Después de la definición de los elementos que componen el esquema vienen las interconexiones. Éstas están agrupadas entre paréntesis. La primera conexión, net, cable, etc, 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. Como vamos a ver es la descripción más simple de todas ya que sólo tiene un objetivo, y es la 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 que se acabó 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 a nivel 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. Otra cosa que se ha visto es 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 determinado tipo de herramientas.

Capítulo 3

Introducción al lenguaje VHDL

Se vio en el capítulo anterior, que la forma más común de describir un circuito era mediante la utilización de esquemas que son una representación gráfica de lo que se pretende realizar. Con la aparición de herramientas de EDA cada vez más complejas, que integran en el mismo marco de trabajo tanto las herramientas de descripción, 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 la herramienta de trabajo.

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 interconexión 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 circuitos con 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 cómo funcionaba.

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 de ese circuito, o componente. Estos lenguajes estaban sobre todo orientados a la simulación, por lo que poco importaba que el nivel de abstracción fuera tan alto que no fuera sencillo 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 como lenguajes de descripción precisamente estos lenguajes de simulación, que si bien alcanzan un altísimo nivel de abstracción, su orientación es básicamente la de simular, por lo que 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 de abstracción cuya orientación o finalidad sea la de la síntesis automática de circuitos, por lo que todavía, de hecho se empieza ahora, se utilizan estos lenguajes orientados a la simulación también para la síntesis de circuitos.

3.1 El lenguaje VHDL

VHDL, viene de VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. VHDL es un lenguaje de descripción y modelado diseñado para describir (en una forma 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 lo 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 puro pero 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 se va 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 herramientas de síntesis para crear implementaciones de diseños a nivel de puertas.

- Al estar basado en un estándar (IEEE Std 1076-1987) 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, permite describir (modelado) el comportamiento de los bloques de alto nivel, analizándolos (simulación), y refinar la funcionalidad de 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 tenemos especificado un circuito y sabemos como funciona; esta es la forma habitual en que se han venido describiendo circuitos y las herramientas utilizadas para ello han sido las de captura de esquemas y las descripciones 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 lo que interesa es el funcionamiento del circuito más que sus componentes. Por otro lado, al encontrarse lejos de lo que un circuito es realmente puede plantear algunos problemas a la hora de realizar un circuito 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, pero 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.

3.2 Ejemplo básico de descripción VHDL

Ejemplo 3.1 *Describir en VHDL un circuito que multiplixe dos líneas (a y b) de un bit, a una sola línea (salida) también de un bit; la señal `selec` sirve para indicar que a la salida se tiene 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.

Lo que se va a realizar a continuación es la descripción comportamental del circuito de la figura 3.1, y luego se realizará la descripción estructural para ver las diferen-

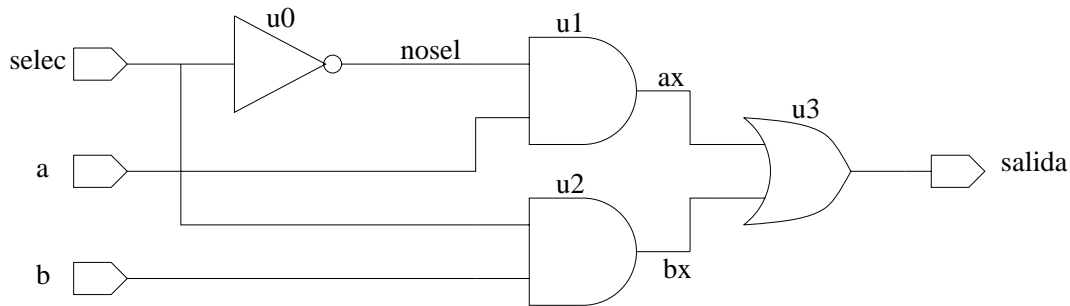


Figura 3.1: Esquema del ejemplo básico en VHDL

cias. Más adelante se verá que hay dos tipos de descripción comportamental, pero de momento, el presente ejemplo únicamente pretende introducir el lenguaje VHDL y su estructura.

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 intentará poner siempre las palabras claves del lenguaje en mayúsculas para distinguirlas de las variables y otros elementos.

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 por 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' y '1'.

La entidad de un circuito es única, sin embargo, se mostró que un mismo símbolo, en este caso entidad, podía tener varias *vistas* o en el caso de VHDL **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 descripciones corresponden al mismo circuito, símbolo, o entidad. Veamos entonces la descripción comportamental:

```
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 algunas 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. 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 un circuito a partir de una descripción comportamental como esta. 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.

La descripción anterior era puramente comportamental, de manera que con una secuencia sencilla de instrucciones podíamos definir 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 instrucciones¹ y que se encuentra más cercana a una descripción estructural del mismo. A continuación se muestran dos ejemplos de una descripción concurrente o también llamada de *transferencia entre registros*:

```

ARCHITECTURE transferencia 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 transferencia;

```

```

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

```

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 definiendo las señales (cables) y los componentes que la definen, aunque no es comportamental ya que en realidad se trata de asignaciones a señales y no una descripción de componentes y conexiones. El segundo caso (derecha) es también una descripción de transferencia aunque basta una única instrucción de asignación para definir el circuito.

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

```

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

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

COMPONENT or2
  PORT(e1,e2: IN bit; y: OUT bit);
END COMPONENT;

COMPONENT inv
  PORT(e: IN bit; y: OUT bit);
END COMPONENT;

SIGNAL ax,bx,nosel: bit;

BEGIN
  u0: inv PORT MAP(e=>selec,y=>nosel);
  u1: and2 PORT MAP(e1=>a,e2=>nosel,y=>ax);
  u2: and2 PORT MAP(e1=>b,e2=>sel,y=>bx);
  u3: or2 PORT MAP(e1=>ax,e2=>bx,y=>salida);
END estructura;
```

Se observa fácilmente que esta descripción es más larga y encima menos clara que las anteriores. Dentro de la arquitectura se definen en primer lugar los componentes que se van a utilizar. Esto se realiza mediante la palabra clave **COMPONENT**, donde se indican además las entradas y salidas mediante la clausula **PORT**. Estos componentes deben tener una entidad y arquitectura propia indicando su comportamiento. Normalmente estas entidades se suelen poner en una librería separada. De momento declararemos estos componentes de esta manera y supondremos que la entidad se encuentra en algún sitio que por ahora no nos preocupa mucho.

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 declarados los componentes y las señales que intervienen se procede a conectarlos entre si. Para ello la sintaxis es muy simple. Lo primero es identificar cada componente, es lo que comúnmente se conoce como *instanciación*, 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 cosa y la única condición es que no haya dos nombres iguales. A continuación del nombre viene el tipo de componente que es, en nuestro 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.

Capítulo 4

Elementos sintácticos del VHDL

El lenguaje 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*.

Algunos de estos elementos sintácticos se muestran a continuación:

Comentarios: Cualquier línea que empieza por dos guiones “--” es un comentario.

Identificadores: Son cualquier cosa que sirve para identificar variables, señales, nombres de rutina, etc. Puede ser cualquier nombre compuesto por letras incluyendo el símbolo de subrayado “_”. Las mayúsculas y minúsculas son consideradas iguales, así que JOSE y jose representan el mismo elemento. No puede haber ningún identificador que coincida con alguna de las palabras clave del VHDL.

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".

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.

Operadores varios

& (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:
punto<=x&y mete en la matriz **punto** la matriz **x** en las primeras posiciones, y la matriz **y** en las últimas.

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 (los tipos **bit** o **bit_vector** no son numéricos).

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

MOD (módulo) Calcula en módulo de dos números. Exactamente se define el módulo como la operación que cumple: $a = b * N + (a \text{ MOD } b)$ 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: $a = (a/b) * b + (a \text{ REM } b)$, 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.

Operadores de desplazamiento

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)

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

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 sean ambos 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.

Operadores lógicos

Son NOT, AND, NAND, OR, NOR y XOR. 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.

Precedencia de operadores y sobrecarga

La precedencia de operadores se presenta en la siguiente tabla:

**	ABS	NOT					Maxima precedencia
*	/	MOD	REM				
+(signo)	-(signo)						
+	-	&					
=	/=	<	<=	>	>=		
AND	OR	NAND	NOR	XOR			Minima 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

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. Veamos a continuación los tipos escalares presentes en VHDL:

Enteros: Son datos cuyo contenido es un valor numérico entero. La forma es 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 se dice que un dato está comprendido en cierto intervalo especificando los límites del intervalo con valores enteros.

Ejemplos:

```
TYPE byte IS RANGE 0 TO 255;
TYPE index IS RANGE 7 DOWNT0 1;
TYPE integer IS -2147483647 TO 2147483647; -- Predefinido en el lenguaje
```

Este último tipo viene ya predefinido en el lenguaje aunque no es muy conveniente su utilización, especialmente pensando en la posterior síntesis del circuito.

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;
    in=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 in, 23 ns.

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 -1e38 TO 1e38; -- Predefinido en el lenguaje
```

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 (nase,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 los tipos de datos escalares 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 DOWNT0 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; -- Predefinido en VHDL
TYPE bit_vector IS ARRAY(natural RANGE <>) OF bit; -- Predefinido en VHDL
TYPE vector IS ARRAY(integer RANGE <>) OF real;
```

Este último ejemplo, y los dos anteriores, muestran una matriz cuyo índice no tiene rango sino que sirve cualquier entero. Más tarde, en la declaración del dato,

se podrá poner los límites de la matriz: `SIGNAL cosa: vector(1 TO 20);`
 Los elementos de una matriz se acceden mediante el índice. Así `dato(3)` es el elemento 3 del dato. De la misma manera se puede acceder a un rango: `datobyte<=datoword(2 TO 9)`. También 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. Veamos algunos ejemplos:

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

Quizá el que se entiende menos es el último donde se ha empleado la palabra `OTHERS` para poner todos los bits del bus a 'Z', de esta forma ponemos un valor a un valor sin necesidad de saber cuantos 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="Jose"`

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 son 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 en VHDL
SUBTYPE positive IS integer RANGE 1 TO entero_mas_alto; -- Predefinido en VHDL
```

El segundo tipo de subtipos son aquellos que restringen el rango de una matriz:

```
SUBTYPE id IS string(1 TO 20);
SUBTYPE word IS bit_vector(31 DOWNT0 0);
```

Los subtipos sirven además para crear *tipos resueltos* que es es una clase especial de tipos que se explicará en detalle en la sección 10.1.

La ventaja de utilizar un subtipo es que las mismas operaciones que servían para el tipo sirven igual de bien para el subtipo. Esto tiene especial importancia por ejemplo cuando se describe un circuito para ser sintetizado, ya que si utilizamos `integer` sin más, esto se interpretará como un bus de 32 líneas (puede cambiar dependiendo de la plataforma) y lo más probable es que en realidad necesitemos muchas menos. Otro caso se da cuando tenemos una lista de cosas y les queremos asignar un entero a cada una, dependiendo de las operaciones que queramos hacer puede resultar más conveniente definirse 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 debe ser 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 escalar tenemos 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`.

Para tipos `t`, `x` miembro de este tipo, y `N` un entero, 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`.

Para `a` siendo un tipo u elemento de tipo matriz, y `N` un entero de 1 a al 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 cogido los más interesantes):

`s'EVENT` Indica si se ha producido un cambio en la señal.

`s'STABLE(t)` Indica si la señal estuvo estable durante el último periodo `t`.

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. Es por esto que es probablemente el atributo más utilizado en VHDL.

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, están las variables, las señales y las constantes. Las variables y

constantes son una cosa muy parecida 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.

Constantes

Una constante es un elemento que se inicializa a un determinado valor y no puede ser cambiado una vez inicializado, conservando para siempre su valor. 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. Esto se hace así para las declaraciones en *packages* que se verán más adelante.

Variables

Una variable es lo mismo que una constante con la diferencia de 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 resulta muchas veces muy útil. Ejemplo:

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

Señales

Las señales se declaran igual que las constantes y variables con la diferencia de que las señales pueden además ser de varios tipos que son normal, register y bus. Por defecto son de tipo normal. Al igual que en variables y constantes, a las señales 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";
```

Constantes, señales y variables

Constantes, señales y variables son cosas diferentes. Las variables, por ejemplo, sólo tienen sentido dentro de un proceso (**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.

Mientras que las variables son elementos abstractos con poco significado físico, las señales tienen 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** dentro de un proceso 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.

4.5 Declaración de entidad y arquitectura

Ya se ha visto, en anteriores ejemplos, cómo se declaran tanto las entidades como las arquitecturas. Veremos 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 estructura se declaran de las formas vistas hasta el momento.

Declaración de entidad

La entidad es la parte del programa que define el módulo. Es decir, define las entradas y salidas del circuito. Además, la entidad es la estructura que permite en VHDL realizar diseños jerárquicos, 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**:

```
id_instr:
ENTITY nombre IS
    GENERIC(lista de propiedades);
    PORT(lista de puertos);
    declaraciones
BEGIN
    sentencias
END nombre;
```

Se observa fácilmente que la declaración de entidad es una cosa algo más compleja de lo que se había visto. La primera cosa que destaca es la palabra **id_instr**, seguida por dos puntos, delante de **ENTITY**. Esto es algo común a todas las instrucciones en VHDL, siempre se puede poner un nombre para identificar unas instrucciones de otras. Este nombre es opcional, se puede poner casi en cualquier instrucción, y permite realizar un mejor seguimiento de la ejecución de un programa durante la simulación. Esta información extra está especialmente indicada para estructuras de tipo **PROCESS** que de por sí no tienen ningún nombre asignado, pero se puede usar en casi cualquier otro tipo de estructura.

Las partes **GENERIC** y **PORT** son las más usadas en la entidad. 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 sino se le pasa ningún valor, la constante tomará el valor que se asigne en **GENERIC**.

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 tipo de conexión, y seguido por el tipo de datos de la línea. Habíamos visto dos tipos 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 pueden 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 tipos **IN** y **OUT** existen otros que también pueden ser usados. Estos otros tipos son 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 que tener un poco de cuidado con este tipo, ya que su significado hardware nunca hay que olvidarlo de manera que pueden producirse contenciones en la misma línea, cuestión que a nivel de programa importa poco, pero que a nivel de hardware puede destruir un chip. Otro tipo que existe es el **BUFFER** que es equivalente al **INOUT** visto con anterioridad, con la diferencia de que sólo una fuente puede escribir sobre él. El último tipo, muy poco usado, es el **LINKAGE** que es como el **INOUT** también pero que sólo puede ser usado con elementos de tipo **LINKAGE**. 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 no se suele usar casi nunca. El tipo de sentencias que se pueden usar 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 DOWNT0 0);
        data: OUT bit_vector(ancho-1 DOWNT0 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;
```

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 mediante la palabra clave **ARCHITECTURE** y su sintaxis completa es:

```
id_instr:
ARCHITECTURE nombre OF la_entidad IS
  declaraciones
BEGIN
  Instrucciones
END 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 las señales sólo pueden ser declaradas dentro de la parte declarativa de una arquitectura.

A continuación vienen, después del **BEGIN**, 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 instanciación que es en realidad también una construcción concurrente. En el caso de definir estructura las instrucciones serán de instanciación, 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 RTL que se verán a continuación. Hay que reseñar que una de estas instrucciones concurrentes es el bloque `PROCESS` dentro del cual la ejecución puede ser secuencial.

Capítulo 5

Ejecución concurrente

Se vio en el ejemplo 3.1, en el capítulo 3, 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. A esta posibilidad de descripción de un circuito se le llama *descripción de 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*). Existe otra forma de describir circuitos en un nivel de abstracción todavía más elevado. A este nivel se le conoce como *descripción comportamental* propiamente dicha. 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 el nivel RTL es concurrente, es decir, las sentencias, más que mandatos o comandos, indican conexiones o leyes que se cumplen, por tanto, es como si se ejecutaran continuamente. En la descripción comportamental abstracta es posible describir partes con instrucciones que se ejecutan en serie de la misma manera que se ejecutan los comandos en un lenguaje como el C o Pascal.

A nivel de síntesis siempre es más sencillo sintetizar un circuito descrito al nivel RTL que otro descrito en un nivel más abstracto. Esto es debido a que la mayoría de estructuras de la descripción RTL 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 más difuso.

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. Naturalmente este es un caso particular de sistema, muy utilizado por otra parte, pero que no se trata del caso general. En general, cualquier sistema hardware está compuesto por numerosas unidades procesadoras con

unidades de almacenamiento distribuidos. 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, 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. Es por esto que cualquier lenguaje que pretenda describir hardware debe ser como mínimo concurrente, es decir, sus sentencias no se ejecutan cuando les llega el turno sino que son como aseveraciones que se deben cumplir siempre.

La arquitectura típica monoprocesadora es un circuito hardware y por tanto internamente tendrá muchas unidades funcionales que funcionan 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. Es por esto que 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. Esto es debido en parte al nivel de abstracción que se consigue con una descripción algorítmica serie al encontrarse cercano 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. Es por esto que en un alto nivel de abstracción de un circuito deba 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 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. La descripción de transferencia de registros se encontraría por tanto a mitad de camino entre una descripción puramente estructural y la puramente comportamental.

5.2 Descripción comportamental RTL

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 obje-

tos 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 también 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 más cercano al hardware. Es fácil comprobar que la descripción con una única instrucción es incluso más simple que la vista en aquel ejemplo donde habían, 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 abstracto.

Se observa que la ejecución en estas descripciones RTL no es serie, es decir, no se ejecuta instrucción por instrucción, sino que la única instrucción que hay se ejecuta continuamente, o sea, 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. Veremos 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 DOWNT0 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 concurrente RTL

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`. Veremos a continuación otras estructuras propias de la ejecución concurrente o descripción RTL.

Asignación condicional: `WHEN...ELSE`

Ya se ha visto anteriormente y su utilización es muy sencilla. Es importante, en toda expresión condicional que describa hardware de forma concurrente, incluir todas las opciones posibles y contemplar todos los casos posibles de variación de una variable. En este sentido es obligatorio siempre acabar esta expresión condicional con un `ELSE`.

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

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

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 contenido de cierto objeto o resultado de cierta expresión. Ejemplo:

```
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`. Esto es así ya que de lo contrario la expresión podría tomar valores frente a los cuales no se sabe qué respuesta dar.

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, 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 puede 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:

```
block_id:
BLOCK(expresion de guardia)
```

```

cabecera
declaraciones
BEGIN
    sentencias concurrentes
END BLOCK block_id;

```

El nombre `block_id` es opcional y su uso sirve para nombrar a diferentes bloques en un mismo diseño y así ayudar en la depuración, simulación y sobre todo en la legibilidad del programa.

La cabecera puede tener clausulas de tipo genérico, declaraciones de puertos de entrada salida, etc, es decir, es equivalente a la declaración de entidad y su alcance es el del bloque. El hecho de poder declararse 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 teníamos en otro diseño, e incorporarlo al nuevo sin necesidad de cambiar todas las señales internas del bloque. Por ejemplo, supongamos que tenemos una memoria ROM definida como bloque con entradas `direccion` y `enable`, y salida `dato`; entonces estas señales se definen con PORT como la entidad, y la conexión entre estas señales y todo lo de fuera, que vamos a suponer que son las señales `rom_dir`, `rom_ena` y `rom_dato`, mediante PORT MAP, es decir:

```

rom: BLOCK
    PORT(direccion: IN bit_vector(15 DOWNT0 0);
          enable: IN bit;
          dato: OUT bit_vector(7 DOWNT0 0));
    PORT MAP(direccion=>rom_dir,enable=>rom_ena,dato=>rom_dato);
BEGIN
    ...
END BLOCK rom;

```

En la parte de declaración se puede incluir desde subprogramas a señales al igual que al principio de la arquitectura, siendo la visibilidad local al bloque lo que permite la modularidad y portabilidad del código.

La expresión de guardia 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 guardia 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 definirse un registro activo por nivel alto del reloj, osea:

```

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

```

Sólo cuando `clk` 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.

Por último, encerrados entre el `BEGIN...END` 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.

Ejemplo 5.2 Realizar la descripción RTL de un circuito que desplace 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 desplazamiento es a izquierdas. Si es “10” el desplazamiento es a derechas, 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;

```

Capítulo 6

Descripción serie comportamental abstracta

La descripción RTL es una descripción 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 comportamental 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 soft, es el modo de ejecución. Mientras el VHDL, hasta ahora, se ejecutaba de forma concurrente, los lenguajes 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 cuando cambian algunos de los argumentos de estas reglas. La ejecución concurrente en VHDL funciona igual que en Prolog, es decir, lo que se hace es declarar *reglas* o *leyes* que en el caso del VHDL son leyes 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. 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 pueden haber múltiples bloques `PROCESS`. En el caso de haber varios de estos bloques, 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, ¿cuando 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 pueden 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 ejecuta 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 abstracta 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 abstracta;

```

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 abstracta 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 abstracta;

```

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 nos dejara usar una descripción así. El problema es que los algoritmos de síntesis tienen problemas cuando a una señal se le asignan varias cosas en un mismo proceso, aunque sea una detrás de otra.

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 y corriente 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 este *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* indicamos que se quiere que, en el próximo paso de simulación (o más tarde 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, pero eso es 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, ya tienen el valor que se les haya asignado durante la ejecución del proceso debido a que se encontrará 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 futuro. Cuando *leemos* una señal estamos echando mano de la sección donde se guarda el valor actual. Cuando se le asigna algo a una señal estamos *escribiendo* en 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, lo llamaremos *driver* de la señal, y a la sección de valor actual lo llamaremos simplemente *señal*. En realidad, y se verá mejor en el apartado 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. Ésto, 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 algo diferente 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 solamente 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 mucho mejor en el siguiente ejemplo:

<pre>-- Uso incorrecto de las senyales ARCHITECTURE ejempl1 OF cosa 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 ejempl1;</pre>	<pre>-- Uso correcto de las variables ARCHITECTURE ejempl1 OF cosa 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 ejempl1;</pre>
---	---

En el ejemplo de la izquierda sólo se utilizan señales y la ejecución tiene lugar de la siguiente manera: En primer lugar se hace el *driver* 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 *driver* el valor 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 del *driver* 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 *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**, tenemos 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 está en la lista sensible se volverá a ejecutar. De manera que se repiten de nuevo las operaciones, es decir, primero se hace el *driver* de *c* igual a *a*, luego se hace *x<=c+2*, con lo que *x=8*, luego se mete *b* en *c*, y por último se hace *y<=c+2* con lo que *y=8*. Vemos que *c* conserva su valor puesto que sigue siendo *c=b=6*, por lo tanto la ejecución se detiene.

Se ha visto que la instrucción segunda (*c<=a*) se ha ignorado completamente en toda la ejecución. Esto es debido a que la instrucción tercera (*c<=b*), al venir después, sobrescribe el *driver* 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 sólo dentro de la ejecución serie las variables tienen sentido. 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 bien 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 el *driver* 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 final del **PROCESS** por tanto, ocurrirá que *x=6* y *y=8*. 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 y se verán algunas características típicas de este tipo de ejecución que es sin duda la más utilizada en VHDL por permitir un alto grado de abstracción y encontrarse más cerca del lenguaje natural.

El bloque de ejecución serie: PROCESS

La forma de entrar en la ejecución 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 declaración es como sigue:

```
proc_id:
PROCESS(lista sensible)
    declaraciones
BEGIN
    instrucciones serie
END PROCESS proc_id;
```

El `proc_id` es simplemente una etiqueta opcional y que puede servir 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. 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 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 puede salir.

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

Sentencia de espera: WAIT

La ejecución de un bloque **PROCESS**, sino 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 no tiene mucho sentido 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 que detenga la ejecución del **PROCESS** hasta que alguna de las señales de la lista sensible cambie.

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

```
WAIT ON lista_sensible UNTIL condicion FOR timeout;
```

La `lista_sensible` es simplemente una lista de señales separadas por comas. La `condicion` es una condición que cuando se cumple sigue la ejecución. El `timeout` 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 timeout, son opcionales y pueden especificarse una, dos o las tres en una misma sentencia `WAIT`. Cualquiera que ocurra primero provocará que se continúe con la ejecución del proceso. Ejemplos:

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

Hay varias cosas 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, el proceso se ejecuta indefinidamente una y otra vez. Si el proceso ya tiene una lista sensible entonces no se puede utilizar la sentencia `WAIT` en el interior de ese proceso, 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:

<pre>-- Con lista sensible p1: PROCESS(b,a) BEGIN a<=b+a+2; END PROCESS p1;</pre>	<pre>-- Con WAIT p2: PROCESS BEGIN a<=b+a+2; WAIT ON a,b; END PROCESS p1;</pre>
--	--

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;
```

Sentencia condicional: IF..THEN..ELSE

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

```
IF condicion THEN
    sentencias
ELSIF condicion THEN
    sentencias
...
ELSE
    sentencias
END IF;
```

Se observa 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, aunque conviene siempre poner un **ELSE** al final de manera que todos los posibles casos de la bifurcación estén contemplados. En programación software esto puede tener poca importancia, pero pensando en la síntesis posterior del circuito, es mucho mejor poner todos los casos posibles para ayudar a las herramientas de síntesis en su labor, no es que no pudieran ser capaces de sintetizar algo sin que estén todos los casos contemplados, pero es posible que el resultado de la síntesis esté más optimizado.

Esta sentencia, cuando se trata de asignaciones, tiene su equivalente en la ejecución concurrente. De hecho, cualquier instrucción concurrente se puede poner de forma serie mediante un proceso. Ejemplo:

<pre>-- Ejecucion 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;</pre>	<pre>-- Ejecucion concurrente p<=2 WHEN a>b ELSE 3 WHEN a>c ELSE 4 WHEN (a=c AND c=b) ELSE 5;</pre>
--	--

Para estos casos particulares resulta más simple la ejecución concurrente, pero es evidente que esta ejecución está limitada a ser usada con asignaciones. Es importante, a fin de que ambas estructuras sean equivalentes, que el **PROCESS** contenga todas las señales que intervienen en las asignaciones en su lista sensible, de otra manera, ambas instrucciones vistas anteriormente no serían equivalentes.

Sentencia de selección: CASE

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

```

CASE expression IS
  WHEN caso1 =>
    instrucciones
  WHEN caso2 =>
    instrucciones
  ...
  WHEN OTHERS =>
    instrucciones
END CASE;

```

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 instrucciones u otras. No pueden haber dos casos duplicados ya que daría error. También, todas 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.

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 poner **OTHERS**. Veamos a continuación un ejemplo simple de esta estructura:

```

CASE puntuacion OF
  WHEN 9 TO 10 => acta<="Sobresaliente";
  WHEN 7 TO 8 => 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.

Sentencias de 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 especificado por la palabra clave **LOOP...END LOOP** y será lo que vaya delante de este bucle lo que indique si es de tipo *for* o *while*. La sintaxis general se da a continuación:

```

bucle_id:
tipo_de_iteracion LOOP
  instrucciones
END LOOP:

```

El `tipo_de_iteracion` indicará si es un **WHILE** o **FOR**. Es opcional, por lo que se podría definir un bucle sin salida, de manera que las sentencias en ese bucle se repetirían siempre.

Las construcciones de los bucles **FOR** y **WHILE** se dan a continuación:

```
bucle_id:
FOR identificador IN rango LOOP
    instrucciones
END LOOP;
```

```
bucle_id:
WHILE condicion LOOP
    instrucciones
END LOOP;
```

En el caso de la sentencia **FOR**, el bucle se repite para cada valor del identificador especificado en el rango. 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.

Junto a estas instrucciones hay 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 se sale del bucle. En el caso de tener varios bucles anidados, la salida se realiza del bucle donde se encuentre la instrucción, o 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:

```
NEXT bucle_id WHEN condicion;
EXIT bucle_id WHEN condicion;
```

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

```
-- Lazo FOR
FOR cuenta IN 5 DOWNT0 0 LOOP
    tabla(cuenta)<=cuenta*2;
END LOOP;
```

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

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;
```

Para terminar este capítulo se ha introducido un pequeño ejemplo. Este mismo ejemplo se encuentra también en el de ejemplos (Ej. 11.1) aunque en esa ocasión se resolverá de forma diferente. Se verá entonces que la descripción que se da en este momento es mucho 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 queremos es que la salida cambie justo cuando pulsamos 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** pues es la única que se puesto en la lista sensible. Como es la única, 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 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, aquí se ha inicializado a cero para que el motor esté parado cuando empieza 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 poner qué hacer para todas las posibilidades de la condición, como si el botón está a cero no queremos que haga nada, pues le asignamos a la señal el valor que ya tiene y así se queda como está. Conviene destacar que el uso de **boton'EVENT** que detecta que la señal del botón ha cambiado, es necesario ya que de lo contrario la instrucción podría entrar en un bucle infinito en este caso concreto; 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 pues entonces se vuelve a ejecutar otra vez la instrucción, cambiando de nuevo y así sucesivamente. Si se sintetiza el circuito entonces 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).

Capítulo 7

Poniendo orden: subprogramas, paquetes y librerías

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, que llamaremos *subprogramas*, que hacen más legibles los programas. 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; esto es lo que formarían los paquetes que a su vez, junto con otros elementos de configuración, etc. formarían las librerías. De todos estos elementos es de los que trata este capítulo.

7.1 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 pequeñas diferencias. Estas diferencias se sumarian 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 con lo que se pueden modificar.
- 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 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.

7.1.1 Declaración de procedimientos y funciones

Las declaraciones de procedimiento y función se encuentran a continuación:

<pre>-- Procedimientos: PROCEDURE nombre(parametros) IS declaraciones BEGIN instrucciones END nombre;</pre>	<pre>-- Funciones: FUNCTION nombre(parametros) RETURN tipo IS declaraciones BEGIN instrucciones -- incluye RETURN END nombre;</pre>
---	---

La lista de **parametros** es opcional y 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 el tipo de objeto que es, 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 pone el tipo de objeto que es.

En la declaración de los parámetros hay cosas que se pueden omitir. Si por ejemplo el puerto es de tipo **IN** entonces no hace falta poner la palabra **CONSTANT** delante puesto que se sobreentenderá que es una constante, de hecho, y si no se pone nada más, se entenderá que los objetos son de tipo **IN**. Para el resto de tipos, y si no se especifica otra cosa, se entenderá que son de tipo **VARIABLE**.

En los procedimientos se pueden utilizar los tres tipos de objetos (constantes, variables y señales), y los tres tipos de puerto (**IN**, **OUT** e **INOUT**). Sin embargo, en las funciones sólo se admiten los objetos de clase constante o señal, y como tipo de puerto sólo se admite el **IN** ya que los parámetros no pueden ser modificados en una función. En la función no se admite la variable puesto que la clase constante ya juega el mismo papel. 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 y hay que tener especial cuidado en el uso de los atributos ya que algunos atributos no están permitidos en el interior de funciones; este 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 además, 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 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.

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`. 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 muchas ocasiones puede resultar útil declarar la función antes de especificar su cuerpo por motivos de visibilidad, etc. En estos casos la declaración se hace igual que se ha visto pero al llegar a la palabra `IS`, se pone un punto y coma y se termina la declaración. Es decir:

```
PROCEDURE nombre(parametros);
FUNCTION nombre(parametros) 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 una matriz cuyo rango ha sido definido en algún sitio como tipo `matriz`.

```
-- mas corto: PROCEDURE extremos(conjunto: matriz; min,max: INOUT integer) IS
PROCEDURE extremos(CONSTANT conjunto: IN matriz; VARIABLE min,max: OUT integer) IS
VARIABLE ind: integer;
BEGIN
  min:=conjunto(conjunto'left);  -- valores iniciales de min y max
  max:=conjunto(conjunto'right);
  FOR ind IN conjunto'range LOOP
    IF min>conjunto(ind) THEN min:=conjunto(ind); END IF;
    IF max<conjunto(ind) THEN max:=conjunto(ind); END IF;
  END LOOP;
END extremos;
```

En este ejemplo se muestra el encabezamiento completo y, comentado, otro encabezamiento que también es válido por tomar los valores por defecto. El funcionamiento es muy simple. Por un lado utiliza la variable de entrada `conjunto` y no la modifica puesto que es de entrada. Como devuelve dos resultados, `min` y `max`, no se puede utilizar una función ya que la función sólo puede devolver un resultado, por lo que se debe utilizar un procedimiento de manera que se le pasen dos parámetros que se puedan modificar.

7.1.2 Llamadas a subprogramas

La forma de invocar un subprograma es bien simple, se pone el nombre seguido por los argumentos entre paréntesis, si los tiene, y eso es todo. A las funciones sólo se las puede invocar como parte de una expresión, mientras que los procedimientos se ejecutan 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. Esta 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 dejarse parámetros por especificar de manera que se cogen 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. Diferentes posibles llamadas al procedimiento del ejemplo anterior, junto con otro procedimiento que no tiene parámetros, sería:

```
extremos(conj(4 TO 20),valmin,valmax);
reset;      -- llamada a un procedimiento sin argumentos
extremos(min=>valmin,max=>valmax,conjunto=>conj(4 TO 20));
```

Los procedimientos se pueden llamar, bien desde entornos concurrentes o bien desde entornos secuenciales. Llamar a un procedimiento desde un entorno secuencial es bien simple, se llama como se ha visto anteriormente y ya está. Sólo hay que tener cuidado de no incluir ninguna instrucción `WAIT` si es que en el proceso que llamó al procedimiento existe una lista sensible.

La ejecución de un procedimiento en entornos concurrentes también es posible. En este sentido, un procedimiento 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 algo que permita suspender la ejecución del procedimiento, de otra manera se ejecutaría indefinidamente. Por defecto se consideran todas los argumentos de tipo `IN` o `INOUT` que se le pasan al procedimiento como la lista sensible. *Si no hay argumentos de este tipo, o sencillamente no hay argumentos, entonces el procedimiento debe incluir al menos una sentencia de espera `WAIT`.*

7.1.3 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 o el tipo de datos de sus argumentos. Esto es especialmente útil cuando se desea ampliar la cobertura de algunos operadores predefinidos. En principio no es necesario indicar que se está sobrecargando una función, en tiempo de ejecución, el intérprete del programa elegirá una función u otra dependiendo de los tipos de datos que se estén utilizando en ese momento.

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 de paréntesis. En VHDL se pueden definir también este tipo de operadores de la misma manera que una función normal, pero poniendo el nombre de la función entre comillas dobles. A continuación se verá un ejemplo donde se sobrecarga la operación suma para que funcione también sobre `bit_vector` de 8 bits definidos como un tipo `byte`. (se supondrá 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 ha redefinido la operación suma y 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 `real`, es en realidad "+" por lo que en realidad se puede usar como una función normal utilizando su nombre completo. De estas manera, las expresiones `X"FF"+X"80"` y `+(X"FF",X"80")` son equivalentes.

Ejemplo 7.1 Se define el tipo enumerado lógico 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.

```
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";
```

7.2 Librerías, 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 juntan 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 interface 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 veremos que hay algunas más que las que se han visto hasta ahora.

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 con 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 *librería de diseño*. Es decir, lo que inicialmente es uno o varios *ficheros de diseño* con la descripción hardware, pasa a ser una única *librería de diseño* después de la compilación, de manera que esta librería contiene todas las descripciones de todos los elementos que componen el circuito. Posteriormente la simulación o síntesis del circuito se realizará sobre esta librería de diseño. A esta librería, donde se guardan los elementos de la descripción después de la compilación se le llama **work**.

Los elementos que componen una librería es lo que se llaman *unidades*. Ya se han visto dos unidades hasta ahora, la entidad y la arquitectura, pero veremos que hay tres más que son los paquetes, los cuerpos de los paquetes, y las configuraciones. A las unidades de tipo declarativo, esto incluiría a la entidad, paquete y configuración, se las conoce como *unidades primarias*. Al resto de unidades que son de tipo ejecutivo, que son las arquitecturas y cuerpo de los paquetes, se las llama *unidades secundarias*.

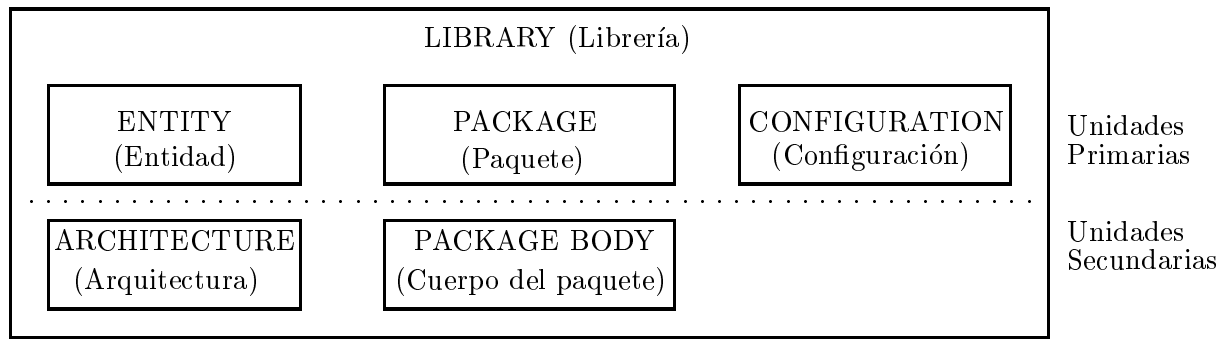


Figura 7.1: Las librerías y las unidades que la componen

Una unidad secundaria tiene siempre asociada una unidad primaria, por lo que deben ser evaluadas las primarias 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 a las arquitecturas. En la figura 7.1 se muestra de forma esquemática cómo se ordenan las unidades dentro de la librería.

Se ha visto que la librería es donde se guardan las unidades de una descripción de un circuito a partir de un fichero. 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 librería correspondiente puede ser muy diversa dependiendo de la herramienta de compilación utilizada y del sistema operativo. Esto quiere decir que no existe un mecanismo estándar en VHDL para la creación de librerías, siendo ésta una tarea de la herramienta que se esté utilizando.

Sí que existe, sin embargo, un mecanismo para incorporar elementos de otras librerías a nuestro propio diseño. Este mecanismo es mediante la inclusión al inicio del fichero de diseño de una cláusula **LIBRARY**. A continuación de esta sentencia se pone la lista de librerías que se desea que sean visibles. Estas librerías se referencian mediante un nombre lógico, de manera que la herramienta traduce este nombre lógico al correspondiente sistema de almacenamiento que la herramienta tenga; puede ser que la librería sea un fichero, o un directorio, o que todas las librerías estén en un único fichero, etc. ésto es algo que depende de la herramienta.

Junto a la cláusula de librería pueden 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 una libreria con 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 librerías que no necesitan ser invocadas puesto que son cargadas por defecto. Una de estas librerías es **work**, es decir, la que contiene las unidades del diseño que se está compilando. La otra librería es la **std** que contiene dos paquetes, el **standard** y el **textio**. El paquete **standard** dentro de esta librería contiene todas las definiciones de tipos y constantes vistos 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 librerías suele venir en las herramientas de simulación y síntesis, otra librería que se usan tanto que prácticamente también es estándar. Esta librería se llama **IEEE** y contiene algunos tipos y funciones que completan los que vienen incorporados por defecto. Dentro de esta librería 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 10.1). Junto con 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.

A 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 librería del 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 function
-----
TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
-----
-- resolution function
-----
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
-----
-- *** industry standard logic type ***
-----
SUBTYPE std_logic IS resolved std_ulogic;
-----
-- unconstrained array of std_logic for use in declaring signal arrays
-----
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
```

Posteriormente a estas definiciones vendría la sobrecarga de operadores, y otras definiciones.

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

7.2.1 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 estando ocultos 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 estarían 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 paquetes y de cuerpos de los paquetes:

<pre>-- Declaracion de paquete PACKAGE nombre IS declaraciones END nombre;</pre>	<pre>-- Declaracion del cuerpo PACKAGE BODY nombre IS declaraciones, instrucciones, etc. END nombre;</pre>
--	--

Naturalmente el nombre del **PACKAGE** y del cuerpo deben coincidir. 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 DOWNT0 1);
  SUBTYPE dato IS bit_vector(15 DOWNT0 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 está compuesto se les puede referenciar 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: tipos_mios.direcc;
pila:=tipos_mios.inicio+X"FF";
desp:=tipos_mios.datotoint(registro);
```

Aunque esto era una forma posible de referenciar los elementos de un paquete, no es la forma usual de referenciarlos. Lo que se suele hacer 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 tipos_mios.ALL
VARIABLE pc: direcc;
pila:=inicio+X"FF";
desp:=datotoint(registro);
```

7.2.2 Configuración: CONFIGURATION

En el ejemplo 3.1, al principio de este capítulo, se mostró un ejemplo de descripción estructural de un circuito utilizando VHDL. En aquella descripción estructural se declararon unos componentes mediante la declaración **COMPONENT**, y luego se instanciaron en el interior de la descripción. Un componente, tal y como estaba definido en aquel ejemplo, no es más que una referencia a una arquitectura y una entidad. Es evidente que este lazo entre el componente y su entidad y correspondiente arquitectura o arquitecturas debe existir ya que de otra forma no se podría saber a qué objeto corresponde el componente y, por tanto, no se podría simular el circuito.

La forma en que a cada componente se le asocia una entidad se especifica en una unidad especial del lenguaje que se llama *configuración*. La forma en la cual se define este bloque es mediante la palabra clave **CONFIGURATION**:

```
CONFIGURATION nombre OF la_entidad IS
    declaraciones
    configuracion
END nombre;
```

En las *declaraciones* lo normal es utilizar clausulas de tipo **USE** para definir tipos y demás, aunque se pueden definir directamente.

En la parte de *configuracion* se especifican las constantes genéricas para bloques y componentes, aparte de otros elementos pertenecientes a bloques y elementos. La forma es que se especifican las características de estos elementos en una configuración es mediante el uso de **FOR**, y es un poquito diferente según sea bloque o componente:

<pre>-- bloques FOR nombre_bloque clausula use elementos END FOR;</pre>	<pre>-- bloques FOR nombre_componente USE objeto union definicion bloque END FOR;</pre>
---	---

El *nombre_bloque* es el nombre de una arquitectura o el nombre de la etiqueta de un bloque. En el caso de componentes, el *nombre_componente* el nombre de la instancia concreta seguido por el nombre del componente y separados por dos puntos. Como instancias se pueden usar las palabras clave **ALL**, todas las instancias, u **OTHERS**, para indicar el resto de instancias. Las instancias se pueden separar por comas.

La clausula **USE** en el caso del componente sirve para indicar la entidad o la configuración que se desea asociar. Con *objeto* decimos si hacemos referencia a una **ENTITY**

o **CONFIGURATION**. En el caso de la entidad se pone el nombre, y opcionalmente la arquitectura entre paréntesis. En el caso de la configuración se pone el nombre de ésta sin más. Es interesante asociar una arquitectura a un componente puesto que entidad sólo hay una, pero arquitecturas puede haber varias, y dependiendo de lo que se esté haciendo en cada momento puede interesar una arquitectura u otra.

Ejemplo 7.2 *Añadir las unidades de entidad, arquitectura y configuración necesarias para completar el ejemplo 3.1.*

```
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;

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

Capítulo 8

VHDL para simulación

El lenguaje VHDL sirve tanto para síntesis automática de circuitos como para descripción de modelos para simulación. Es evidente que la filosofía de descripción en uno y otro caso son diferentes. Por un lado la simulación de un programa en VHDL no tiene demasiadas restricciones, lo único que se necesita es un intérprete de los comandos e 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 lo que viene descrito en el programa. Si el nivel de abstracción es muy alto, la síntesis será muy difícil llegando a la posibilidad de que sea 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, etc. en síntesis, estos elementos, especialmente los retrasos, no tienen ningún sentido y se deben evitar.

8.1 Los retrasos y la simulación

En simulación existe un elemento importantísimo que no se ha visto hasta ahora, y es el retraso de las líneas. En todo circuito digital, aparte de la funcionalidad que se pueda implementar, existe siempre un retraso entre que se producen los estímulos de entrada y la salida cambia.

Para poder poner en práctica este funcionamiento a base de retrasos, los lenguajes de descripción de modelos, suelen utilizar lo que se llaman *drivers*. El concepto de *driver* es algo que ya apareció cuando se explicaron las asignaciones a señales y las diferencias entre señal y variable en la ejecución secuencial.

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 a la señal, sino que se le asigna a su driver. La información del driver pasa a la señal cuando se llega al tiempo especificado en la asignación. Hasta ahora nunca se ha especificado ningún tiempo en las asignaciones, en estos casos se considera que el tiempo, o retraso, es nulo y por tanto la asignación debe producirse de forma inmediata. Esta asignación inmediata significa en realidad que se realice la asignación al final del presente paso de simulación, siendo un paso de la simulación la ejecución de una instrucción concurrente.

Vamos a ver a continuación que en la asignación de una señal se puede especificar además un retraso. Este retraso va a indicar que se le asigne el valor dado a una señal pero cuando haya transcurrido el tiempo especificado en el retraso. Esto quiere decir que la información va a permanecer en el driver hasta que haya pasado el tiempo especificado 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 interesante hacer notar que cuando se simula un circuito descrito en VHDL aparece el concepto de *tiempo de simulación*. Este tiempo transcurre gracias a la sucesión de eventos. En la instrucción anterior, donde se asignaba un '0' a `senal` después de 15 ns, en realidad estábamos produciendo un evento que tendrá lugar dentro de 15 ns. El simulador de VHDL guarda una lista de todos los eventos que se generen y los ordena según el momento en que tengan que procesarse. La simulación tiene lugar por el procesado en serie de 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.

Una descripción del flujo que suele seguir un simulador en VHDL se muestra en la figura 8.1 donde $\Delta\delta$ es un paso de simulación donde el tiempo no corre, y ΔT es un paso donde el tiempo corre realmente. Este flujo corresponde a lo que se conoce como *simulación guiada por eventos*. Los simuladores digitales suelen emplear otra que prácticamente es parecida aunque el tiempo, en vez de incrementarse por el próximo evento que vaya a ocurrir, se incrementa un tiempo fijo cada vez de manera que si hay algún evento en ese intervalo de tiempo se procesa.

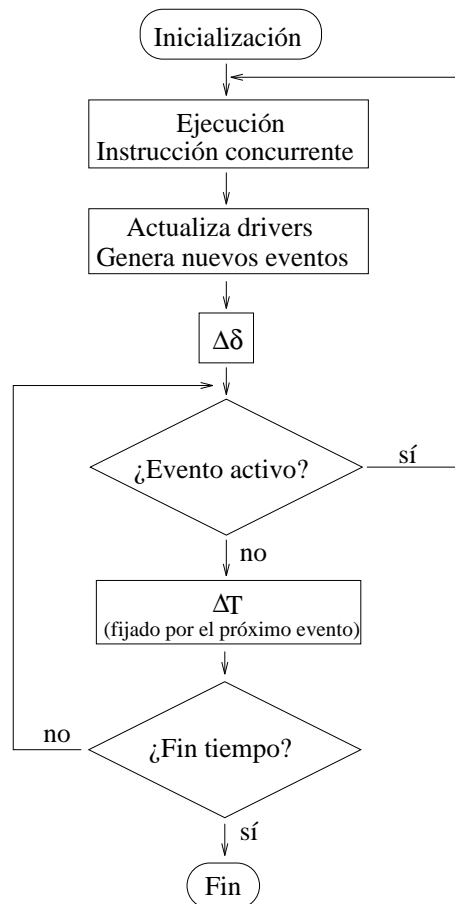
Gracias al concepto de *evento* es fácil entender que en una asignación se pueden programar varios eventos o sucesos que tendrán lugar en el futuro. En el ejemplo siguiente se muestra cómo realizar varias asignaciones a una misma señal:

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

Cuando se produce la ejecución de esta instrucción, se están utilizando dos drivers en realidad, en uno se mete un '1' y en 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 nanosegundos de su ejecución, y 16 nanosegundos después tomará el valor '0'.

Ejemplo 8.1 Realizar el modelo de simulación de un registro tipo *D*, activo por flanco de subida, que tiene un retraso de 10 nanosegundos, 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 supondremos que si se produce una violación se coge el valor anterior de la señal de entrada y no el que haya en el momento del flanco).

Este es un ejemplo típico donde se especifica el retraso en una señal de salida, para

Figura 8.1: *Flujo de simulación por eventos en VHDL*

el retraso respecto del reloj, y un retraso en una señal de entrada, para realizar el tiempo de establecimiento que se pide. 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. Veamos la descripción cómo quedaría:

```

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;

```

8.1.1 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 hacen los eventos se van añadiendo a la lista ordenándolos según les vaya a tocar, 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 juega un papel importante de manera que se pueden definir dos formas de introducir los eventos en la lista, cada forma con un significado físico concreto tal y como se muestra a continuación.

Supongamos una puerta lógica. Esta puerta tendrá un retraso asociado, de manera que la salida cambiará un momento después de que haya cambiado la entrada, por ejemplo, 50 ns después. Esta puerta lógica, un inversor por ejemplo, se realizaría de la siguiente forma:

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

Al principio de la simulación, y cada vez que la entrada `ent` cambia, 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. Supongamos ahora que la salida cambia en menos de 50 ns (imaginar por ejemplo un pulso en la entrada de 30 ns). Como la salida ha cambiado antes de que el cambio anterior se haya producido, ese evento anterior se pierde. Para verlo un poco más claro seguiremos la simulación que aparece en la figura 8.2, 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. Supongamos que al inicio está todo estabilizado y que por tanto para la entrada cero la salida es uno. Después de 10 ns ponemos la entrada a uno, es el comienzo del pulso. Esto quiere decir que se introduce el evento “poner a cero la salida después de 50 ns”, o lo que es lo mismo “poner a cero la salida a los 60 ns de tiempo absoluto”. Este evento se introduce en la lista de drivers de la señal de salida. Supongamos que esperamos otros 30 ns más (duración del pulso) y cambiamos la entrada pasándola a cero otra vez. Entonces se está produciendo un nuevo evento que es “poner a uno la salida después de 50 ns”. Lo que uno espera normalmente es que se ponga en la lista, detrás del evento anterior, y conforme vayan llegando el momento de ejecutarse los eventos se ejecutan y ya está. Esto no ocurre así, al menos definiendo los retrasos como se ha hecho hasta ahora. Lo que va a ocurrir 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.

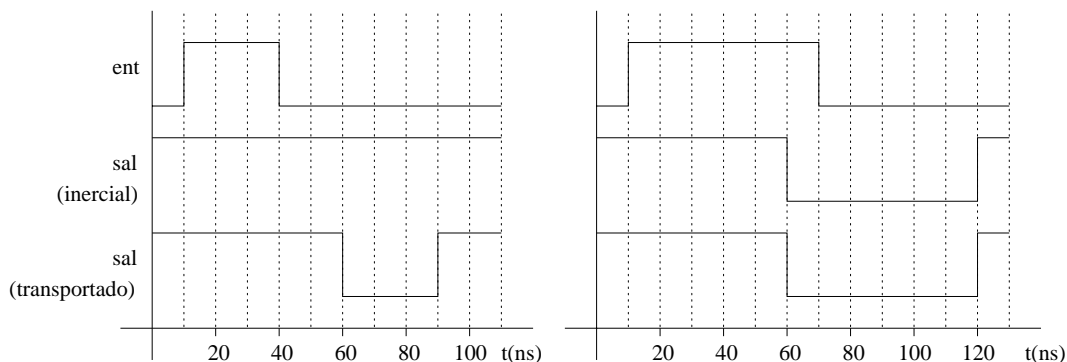


Figura 8.2: Retrasos inerciales y transportados

A esta forma de gestionar los eventos en la lista y de 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. Esto tiene un gran significado físico puesto que es lo que suele pasar en algunas puertas lógicas que cuando se les mete un pulso, si no tiene la duración adecuada, nunca llega a modificarse la salida.

Naturalmente hay sistemas donde este filtraje no es adecuado. Si pensamos por ejemplo 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. En este caso al retraso se le llama **retraso transportado**, puesto que en realidad la señal de entrada se transporta a la salida sin modificarla para nada. En cuanto al tratamiento de la lista de eventos o drivers, lo que se hace es que simplemente se introduce el evento en la lista, en el lugar que le corresponde según el retraso, y cuando le toca procesarse se procesa.

En VHDL hay que indicar explícitamente este tipo de retraso transportado puesto que no es el que hay por defecto. Esta indicación se realiza mediante la palabra clave **TRANSPORT**. Así, si en el ejemplo anterior quisiéramos que los pulsos menores que el retraso pasaran al otro lado pondríamos:

```
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.

8.2 Descripción de un banco de pruebas

El objetivo del lenguaje VHDL es la descripción de circuitos digitales, esta descripción podrá ser usada como un modelo para simulación o como descripción de un circuito para ser sintetizado. En ambos casos interesa simular la descripción que se ha hecho para ver si realmente funciona como se pretende. El problema de la simulación de este modelo se puede abordar de varias formas, probablemente la más rápida sea la de coger la herramienta de simulación y empezar a introducir cambios en las entradas para ver cómo varían las salidas. Este procedimiento puede servir para simular cosas simples, pero para simulaciones más complejas y exhaustivas es mejor definirse lo que se denomina un banco de pruebas (*test bench*).

Este banco de pruebas no es más que la definición de unas cuantas entradas llamadas *patrones de test* con las que comprobar el circuito o modelo. Normalmente las herramientas de simulación ofrecen alguna manera de definirse estos vectores de entrada, bien mediante gráficos, o ficheros de vectores, etc. En el caso del VHDL estos vectores se pueden definir con el propio lenguaje, de manera que se puede crear un banco de pruebas que es independiente del simulador.

Este banco de pruebas no es más que una entidad, sin entradas ni salidas en su caso más simple, cuya arquitectura, de tipo estructural, tiene como señales internas las entradas y salidas del circuito y como único componente el correspondiente a la entidad que se desea simular. Esto se ve mucho más claro en el siguiente ejemplo donde se va a realizar el banco de pruebas para una puerta **and2** de dos entradas y una salida. Se

supone que tanto la entidad como la arquitectura están definidas en algún sitio de la librería de trabajo o son visibles:

```
ENTITY test IS    -- no tiene entradas ni salidas
END test;

ARCHITECTURE estimulos OF test IS
SIGNAL a,b,s: bit;

COMPONENT and2 PORT(a,b: IN bit; s: OUT bit); END COMPONENT;

FOR puerta: and2 USE ENTITY work.and2;

BEGIN
    puerta: and2 PORT MAP(a,b,s);
    a<='0', '1' AFTER 200 ns;
    b<='0', '1' AFTER 100 ns, '0' AFTER 200 ns, '1' AFTER 300 ns;
END estimulos;
```

8.3 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 sacar un mensaje por pantalla indicando que cierta señal se ha activado. Estos mecanismos de notificación son especialmente útiles en la detección de violaciones de los tiempos de *setup* 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 condicion REPORT mensaje SEVERITY nivel_gravedad;
```

Si no se cumple la condición especificada en **condicion** entonces se saca el mensaje especificado por pantalla y se da 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 el tipo predefinido **severity_level**. Los niveles de gravedad que hay son: **NOTE**, **WARNING**, **ERROR** y **FAILURE**, y si no se especifica nada el valor por defecto es **ERROR**. En el momento se produce una violación de cualquier tipo, el simulador puede detener la ejecución o no dependiendo del nivel de gravedad del error.

Esta instrucción puede ser usada tanto en entornos concurrentes como en 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 toque el turno en el proceso normal de ejecución. A continuación se dan tres ejemplos de utilización de estos mensajes:

```
-- Violacion de setup:
ASSERT NOT(clk'EVENT AND clk='1' AND NOT(d'stable(20 ns)))
    REPORT "Violacion del tiempo de setup"
    SEVERITY WARNING;

-- Violacion de la anchura de un pulso:
ASSERT (preset'delayed='1' AND preset='0' AND preset'delayed'last_event>=25 ns)
    REPORT "Anchura de pulso demasiado pequenya";
```



```
-- Para depurar programas:
ASSERT FALSE
REPORT "La ejecucion paso por aqui"
SEVERITY NOTE;
```

8.3.1 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 definirse 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** avisar 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 veremos un ejemplo de utilización de un proceso pasivo para la comprobación del tiempo de establecimiento (set-up).

Ejemplo 8.2 *Añadir un proceso pasivo en la entidad del registro del ejemplo 8.1 que detecte la violación del tiempo de establecimiento y emita un mensaje.*

En principio la arquitectura quedaría igual 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. Pero supongamos que no se dispone de dichos atributos para la señal de entrada y queremos hacerlo “a pelo”, entonces una posible solución, donde además se muestra cómo se maneja el tiempo de simulación, se da a continuación:

```
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;
```

Naturalmente aquí se ha supuesto que ni siquiera se tiene el atributo **'EVENT**, lo cual sólo se justifica considerando que este ejemplo es más pedagógico que otra cosa. En el caso de haber utilizado este atributo, bastaría conservar el **IF** más interior añadiéndole **AND clk'EVENT** en la condición; con esto, la condición y la variable de reloj sobran. También se ha introducido la función predefinida **NOW** que da el tiempo de simulación en el momento en que se ejecuta la instrucción.

Capítulo 9

VHDL para síntesis

La síntesis de un circuito a partir de VHDL consiste en reducir el nivel de abstracción de la descripción de un circuito hasta convertirlo en una definición puramente estructural cuyos componentes son los elementos de una determinada librería de componentes, que 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 comporta 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 es una síntesis del circuito que se ha descrito. 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 la descripción tenga. En primera aproximación se puede coger un ordenador que ejecute la simulación, y ya tengo la síntesis. A partir de esta primera aproximación hay que ir optimizando el circuito. En realidad las herramientas de síntesis siguen una aproximación distinta, ya que de otra manera, el circuito sería algo parecido a un microprocesador cuando quizá sólo se pretende 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 y secuencial e intentaría traducirla a un nivel de transferencia entre registros descrita con ecuaciones de conmutación. A partir de esta descripción se intentaría transformarla a una descripción estructural donde se realiza además lo que se llama el *mapeado tecnológico*, es decir, la descripción con los componentes de una librería especial que depende de la tecnología con la cual se quiera realizar el circuito.

Las herramientas de síntesis actuales cubren a la perfección la síntesis a partir de descripciones RTL y estructurales, pero no están tan avanzadas si el diseño se encuentra descrito 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.

9.1 Restricciones 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. Ésto 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.

Todas estas consideraciones van a imponer unas restricciones a lo que es el lenguaje, por lo tanto, cualquiera que pretenda usar el VHDL para síntesis de circuitos debe conocerlas. 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. Aparte de esto, no se permiten las asignaciones múltiples, en una única sentencia, a una señal por la misma razón.

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. Desde luego no es nada conveniente definirse una puerta tal y como se hizo en el ejemplo 6.2.

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 muchos 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 condición.

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 proceso se ejecutará cuando cambia una señal que se encuentra en el proceso pero no en la lista sensible o en el `WAIT`.

Permitir discrepancia Normalmente es fácil sintetizar algo simple como `s<=NOT s` ya que no es más que una puerta inversora conectada sobre si 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 lo es.

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 no se permite).

Evitar IFs 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 IFs Las estructuras `CASE` tienen para los sintetizadores un modelo optimizado de síntesis, generalmente mejor que lo mismo descrito mediante `IFs`.

Utilizar el estilo indicado para las máquinas de estado 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, entonces a veces ocurre que el sintetizador no se da cuenta de que lo que tiene delante es una máquina de estados y no optimiza bien el circuito resultante. 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 nos 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, por lo que conviene especificar cuál de todas las arquitecturas se desea sintetizar mediante un bloque de configuración `CONFIGURATION`.

Con estas restricciones ahora expuestas, y hay algunas más que dependerán del sintetizador, ya nos damos 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.

9.2 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 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, por tanto, cualquier sistema que tenga al menos una señal que ante el cambio de unas señales cambie, pero que pueda ocurrir que ante el cambio de las mismas señales conserve su valor, entonces se tratará de un sistema

secuencial ya que dicha señal es un elemento de memoria. Esto nos da una pista de si un circuito será secuencial y se realizará por tanto a partir de elementos de memoria como puedan ser cerrojos o registros.

9.2.1 Descripción de lógica combinacional

La idea básica es que si en la estructura del lenguaje no introducimos “elementos de memoria” entonces estaremos delante de una descripción combinacional. Vamos a ver entonces cómo evitar que aparezcan elementos de memoria y que por tanto 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
```

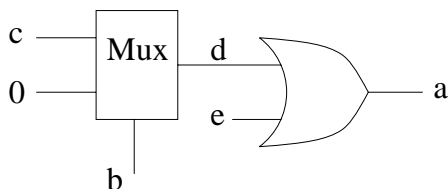
 Si la señal interviniera en la asignación entonces habría casos para los cuales se conserva su valor, y por tanto sería un elemento de memoria.
 - No hay lazos combinacionales (en realidad esta es una extensión de la anterior). Ejemplos:

```
-- Secuencial
d<=b AND a;
a<=d OR e;
```

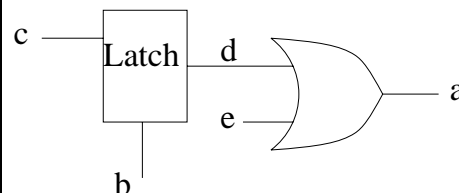
```
-- Combinacional
d<=b AND c;
a<=d OR e;
```

- 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. 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;
```



9.2.2 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 hemos visto para la descripción de lógica combinacional nos 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 poniéndolas en negado, vamos a ver 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 lo anterior 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-flop están realizados internamente mediante puertas). Lo que sí que puede ocurrir es que dependiendo de la herramienta que estemos utilizando, lo anterior se realice mediante puertas o mediante un elemento de la librería 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 definirse un reloj que fuera 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 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:

<pre> PROCESS(clk,reset) BEGIN IF reset='1' THEN q<='0'; ELSIF clk'EVENT AND clk='1' THEN q<=d; END IF; END PROCESS; </pre>	<pre> PROCESS(clk) BEGIN IF clk='1' THEN q<=d; END IF; END PROCESS; </pre>
---	---

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 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 detección de flanco completa puesto que al haber dos señales en la lista sensible no se sabe cual 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, veamos algunas:

- 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 una misma circuitería es difícil de sintetizar si está sincronizada 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.

Como complemento a este capítulo resulta muy interesante el dedicado a conceptos avanzados donde se verá la descripción de máquinas de estados, así como el capítulo de ejemplos donde se han incluido numerosas descripciones de funciones de la vida cotidiana.

Capítulo 10

Conceptos avanzados en VHDL

Hasta este momento, se ha intentado dar una visión general de lo que es el lenguaje VHDL y se han dado algunas notas sobre su uso para síntesis y modelado de circuitos. Las explicaciones vistas cubren buena parte de las posibilidades del lenguaje, pero todavía quedan muchas cosas por contar. No es que en esta sección se vayan a cubrir el resto de características del lenguaje, pero sí que se expondrán algunas cosas importantes que quedaron por explicar. Como parte del lenguaje en sí se explicará la utilización de buses y funciones de resolución, y como ejemplos de descripciones se verá la forma en que se pueden describir máquinas de estados mediante VHDL.

10.1 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, en VHDL. 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, se da la circunstancia de que hay varios elementos conectados a la misma línea. Pensemos por ejemplo en el bus de datos de un ordenador. 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, a eso se le llama *contención*, o lo que es lo mismo, que hay una lucha o contienda por el bus.

Para evitar que las contenciones, o luchas, en el bus acaben con los dispositivos que escriben sobre él, éstos escriben en el bus a través de buffers que, dependiendo del tipo que sean (tristado, colector abierto, etc), administrarán la contención de una forma u otra. En el caso de procesadores y memorias, se suele utilizar el *buffer tristado* que tiene una señal de habilitación de manera que cuando esta señal está activa el buffer puede escribir sobre el bus, y cuando está inactiva también escribe pero lo que se llama un valor de 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 están 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 un driver puede escribir sobre una señal (en el caso de poner retrasos, en una multiasignación, son varios driver consecutivos, pero sólo hay uno que escribe sobre la señal). Como corolario de este principio que acabamos de dar, se puede decir que **no se puede asignar dos veces un valor a una señal en procesos diferentes**, y conviene recalcarlo tanto ya que es un error muy común a la hora de describir circuitos usando VHDL.

Para solucionar el problema de los buses en VHDL se han creado 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 varios drivers 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 ver esto más claro veamos 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'. Supongamos que se tienen señales sobre las que se pueden dar accesos múltiples a un tiempo (ej. un bus de datos). Con estas consideraciones vamos a ver cómo se describiría un bus de este tipo para que soportara múltiples drivers 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 definirse una función de resolución que calcule el valor del driver en función de todas las asignaciones que se están haciendo. En nuestro ejemplo cualquier señal que contenga 'Z' no interviene, y para el resto, será cero si al menos hay uno que es cero (*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, aplicaremos el tipo anterior (`logico_resuelto`) para solucionar el problema de la contención en el bus. Supongamos que tenemos 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 triestado que tienen como señales de habilitación `micro_ena` y `mem_ena`. Por último vamos a suponer 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 senyal 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 tendríamos dos instrucciones concurrentes que escriben al mismo tiempo sobre la misma señal.

Como anécdota final, aunque ya se vio en las asignaciones a matrices (capítulo 4), 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 suponemos 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` de la librería `std` se definían los tipos `bit` y `bit_vector` como no resueltos, es decir, no tienen función de resolución y por tanto no se los 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 7), posee funciones de resolución. Estos tipos venían definidos en el paquete `std_logic_1164` de la librería `ieee` que incluye además los tipos `std_ulogic` y `std_ulogic_vector` que son los tipos no resueltos equivalentes.

10.2 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.

Normalmente 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 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 es ambas es bastante simple. Veamos a continuación la forma general que tendría una posible, que no la única, descripción de una máquina de Moore donde la salida sólo depende del estado del sistema. Para ello supondremos que **entrada** y **salida** son las entradas y salidas, y que (est1,est2,...,estK) son los estados del sistema. Con esto, la descripción será:

```
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; -- especificar un estado inicial
SIGNAL siguiente: estado;      -- en realidad basta con una se'nal
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;
```

En este caso 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 una expresión en la que pueden intervenir las entradas.

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 la misma cosa 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 típico de descripción de sistema secuencial. 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 por lo cambios 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 que no es el que le toca.

Por esta razón, las máquinas de estados en circuitos reales, suelen venir 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 le ponemos una señal de reloj, que llamaremos **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 IF;
END PROCESS salida;
END moore_sincrono;
```

Hemos visto que se ha separado la parte secuencial (proceso `estados`) de la combinatorial (proceso `salida`). Esto es especialmente útil cuando se trabaja con máquinas de estados síncronas. La parte secuencial se encarga de calcular el estado siguiente, y la parte combinatorial pura calcula la salida en función del estado y la entrada. Cada una de estas partes se puede describir mediante dos procesos separados. Normalmente, además, las máquinas de estados suelen necesitar una señal de `reset` que lleve a la máquina a un estado conocido de partida.

Veamos a continuación cómo sería la estructura de una máquina genérica que incorpora `reset`, `reloj`, y es 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)    -- reset asincrono
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 combinatorial:
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 `reset` puesto que fijando un estado cuando está activa la señal de `reset`, fijamos también la salida. En el caso de necesitar una salida especial para cuando el `reset` está activo se especificaría en este proceso, pero esto en general no es necesario ya que se suele elegir el estado correspondiente al `reset` para que su salida sea la que se desea.

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 fuera una entrada más.

Capítulo 11

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 confianza y soltura con el lenguaje.

11.1 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 variable 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, utilizar 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 tenemos un proceso que se encarga de incrementar el contador, y en otro proceso, normalmente el de la máquina de estados, queremos 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 de 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 tenemos un contador por ejemplo, en un proceso, y poco después de incrementarlo, comparamos 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 cosas así, es decir, 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 muchísimo 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 y que por lo tanto se debe tener en cuenta, bien añadiendo estados auxiliares o bien, y esto funcionará bien en cualquier caso, definiendo una señal que se activará al activarse 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 de un 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 nos 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 variable 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, podemos 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, lo que realmente se dan con frecuencia y son más fáciles de cometer son los dos primeros. Veamos a continuación cómo resolver algunos de los problemas de diseño que se pueden plantear en VHDL.

11.2 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. Aquí veremos algunos ejemplos en los cuales son modelos puesto que incluyen retrasos, y otros, que por la forma de estar descritos no son sintetizados correctamente. Veremos las diferencias, cuando las haya entre lo que se sintetiza y lo que se simularía, y veremos que muchas veces no coincide.

11.2.1 El botón

Ejemplo 11.1 *Un motor eléctrico viene controlado por un único botón. Cuando se pulsa 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;
```

Se puede hacer algo con 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, y 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 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 cosas son activas por flanco de forma explícita, generalmente con el atributo **'EVENT'**. Si se metiera esta descripción en un sintetizador, y luego simuláramos lo sintetizado, descubriríamos 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 pone, 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 repasamos la descripción anterior y la intentamos simular a mano con todo lo que sabemos, veremos que cuando el estado es apagado motor el vale uno, y viceversa, es decir, lo contrario de lo que parece. Si esto nos causa mucha confusión, podemos dividir el problema en dos procesos, uno que interpreta el estado y otro el cambio de estado:

<pre> 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; </pre>	<pre> salida: PROCESS(presente) BEGIN CASE presente IS WHEN apagado=> motor<='0'; WHEN encendido=> motor<='1'; END CASE; END PROCESS salida; </pre>
--	---

Esta descripción es más interesante ya que en este caso está más claro lo que queremos decir y 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 la simulación estaría bien, sintetizaría otro circuito diferente. O sea, que es aconsejable seguir un único modelo para la máquina de estados, que funcione bien para síntesis, y no salirse de ahí.

11.2.2 Los semáforos

Ejemplo 11.2 *Realizar el circuito de control de unos semáforos que controlan 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.

La entrada al sistema será una señal de reset 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. Con estas consideraciones la entidad y arquitectura quedarían:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY semaforo IS
PORT (sensor,reset,clk: IN std_logic;
      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 definirse la maquina 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)
  BEGIN
    CASE presente IS
      WHEN inicial=>
        semcarr<=verde;
        semcamin<=rojo;
        rescont<=true;
      WHEN carramarillo=>
        semcarr<=amarillo;
        semcamin<=rojo;
    END CASE;
  END PROCESS salida;

```

```

        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;
```

11.2.3 El ascensor

Ejemplo 11.3 *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. La salida será por un lado el motor (mediante dos bits), y la puerta (un 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 hará caso.*

```

ENTITY ascensor IS
PORT(boton: IN bit_vector(0 TO 3);
     piso: IN bit_vector(1 DOWNT0 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);
    SUBTYPE vector IS bit_vector(2 DOWNT0 0);
    SIGNAL presente: estado:=inicial;
    SIGNAL bot: bit_vector(2 DOWNT0 0);

    FUNCTION codifica(pulso: bit_vector(0 TO 3)) RETURN vector IS
    BEGIN
        CASE pulso IS
```

```

        WHEN "0001"=>RETURN "000";
        WHEN "0010"=>RETURN "001";
        WHEN "0100"=>RETURN "010";
        WHEN "1000"=>RETURN "011";
        WHEN OTHERS=>RETURN "100";
    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 bot/="100" THEN presente<=cerrar;
                END IF;
            WHEN cerrar=>
                IF celula='0' THEN presente<=voy; -- Sin obtaculos
                END IF;
            WHEN voy=>
                IF bot(1 DOWNT0 0)=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);
        WHEN cerrar=>
            motor<="00";
            puerta<='1';
        WHEN voy=>
            puerta<='0'; -- Cerrada
            IF bot(2 DOWNT0 0)>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 nadie pulsa nada se mantiene en el estado inicial, si alguien pulsa 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. Esta señal sigue siendo necesaria puesto que se encarga de capturar la pulsación del botón.

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.

11.2.4 La memoria ROM

Ejemplo 11.4 *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, sino está activa, la salida es alta impedancia. El tiempo que pasa entre que cambia la selección y la salida 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.*

```

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
        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;

    dato<=salida WHEN cenr='0' ELSE
        (OTHERS 'Z') WHEN cenr='1' ELSE
        (OTHERS 'X');
    cenr<=cen AFTER 60 ns;

END modelo;

```

El modelo no requiere demasiadas explicaciones. 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 de 10 ns, de manera que si no fuese transportado quitaría el otro evento de la lista de eventos y no se ejecutaría nunca.

11.2.5 El microprocesador

Ejemplo 11.5 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 de lectura_escritura (a uno indica lectura y a cero escritura). Una señal de reloj y una de reset. Internamente debe haber un acumulador de 8 bits, el registro de instrucción de 3 bits, y el programa counter de 8 bits. El micro cuenta con 8 instrucciones. Las instrucciones están formadas por dos bytes, en el primero se pone el código, 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 es relativamente sencillo en VHDL. Además tienen todos una estructura parecida por lo que resulta fácil añadir instrucciones y hacer el procesador lo complicado que 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.

Uno de los paquetes de la librería **ieee** es el **std_arith** que sobrecarga los operadores aritméticos para que se pueda, por ejemplo, sumar un entero a un **std_logic_vector**. Veamos la descripción:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_arith.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,ir: std_logic_vector(7 DOWNTO 0);
  SIGNAL rdat_in,dat_in,dat_out: std_logic_vector(7 DOWNTO 0);
  SIGNAL rwaux,seldir: std_logic;
  SIGNAL presente: estado:=inicial;
BEGIN

  fsm:
  PROCESS(clk)
  BEGIN

```

```

IF clk='1' THEN
CASE presente IS
WHEN inicial =>
    seldir<='1'; -- dir<=pc
    pc<=(OTHERS=>'0');
    rwaux<='1';
    ir<=(OTHERS=>'0');
    a<=(OTHERS=>'0');
    presente<=busqueda;
WHEN busqueda=>
    ir<=dat_in;
    pc<=pc+1;
    IF dat_in(2 DOWNT0 0)="111" THEN presente<=busqueda;
    ELSE presente<=ejec;
    END IF;
WHEN ejec =>
    seldir<='0'; -- dir<=dat_in
    pc<=pc+1;
    presente<=busqueda;
    CASE ir(2 DOWNT0 0) IS
    WHEN "000" =>
        presente<=ldaxx;
    WHEN "001" =>
        dat_out<=a;
        rwaux<='0'; -- Escribir
        presente<=ldxxa;
    WHEN "010" =>
        presente<=anda;
    WHEN "011" =>
        presente<=adda;
    WHEN "100" =>
        presente<=suba;
    WHEN "101" =>
        seldir<='1';
        IF a=0 THEN
            pc<=dat_in;
        END IF;
    WHEN "110" =>
        seldir<='1';
        pc<=dat_in;
    WHEN OTHERS => NULL;
    END CASE;
WHEN ldaxx =>
    a<=dat_in;
    seldir<='1';
    presente<=busqueda;
WHEN ldxxa =>
    rwaux<='1';
    seldir<='1';
    presente<=busqueda;
WHEN anda =>
    a<=a AND dat_in;
    seldir<='1';
    presente<=busqueda;
WHEN adda =>
    a<=a+dat_in;
    seldir<='1';
    presente<=busqueda;
WHEN suba =>
    a<=a-dat_in;
    seldir<='1';
    presente<=busqueda;
END CASE;
IF rst='1' THEN presente<=inicial;
END IF;
END IF;
END PROCESS fsm;

```

```

latch_in:    -- Registro en la entrada del bus de datos
PROCESS(clk)
BEGIN
    IF clk='1' THEN rdat_in<=dat_in;
    END IF;
END PROCESS latch_in;

dir<=pc WHEN seldir='1' ELSE rdat_in; -- Multiplexor de las direcciones
r_w<=rwaux;
dat<=dat_out WHEN rwaux='0' ELSE (OTHERS=>'Z'); -- Buffer de Salida
dat_in<=dat;

END descripcion;

```

11.2.6 La lavadora

Ejemplo 11.6 *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 11.1. 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 para sincronizar de frecuencia 100 Hz.*

Salidas:

jabon: *Al ponerla a uno coge el jabón del cajetín y lo mete en el tambor en el ciclo de lavado.*

llenar: *A uno abre las válvulas del agua para llenar el tambor, se debe monitorizar la señal **lleno** para saber cuando ponerla 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, o sea, más rápida. Si las señales anteriores están las dos a cero entonces el motor está parado, si están a uno las dos 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, sino 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.

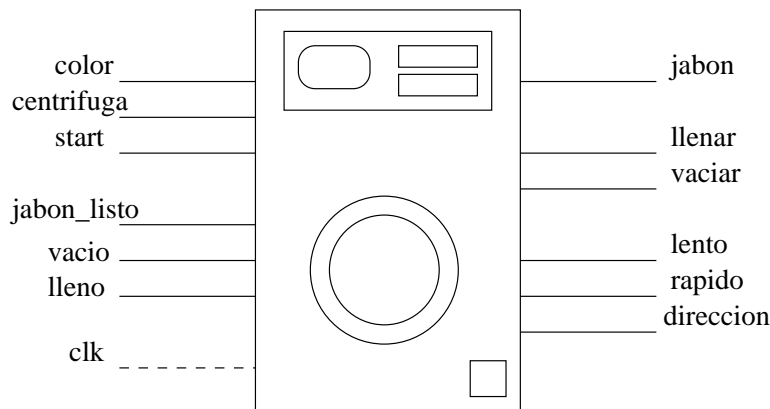


Figura 11.1: Figura del ejercicio de la lavadora

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 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 de otra coinciden.

En la primera descripción ponemos las salidas en el mismo proceso donde colocamos la transición de estados. Lo que producirá esto es que la salida cambiará un ciclo de reloj después de que cambie el estado, pero esto da igual ya que la frecuencia de reloj es muy alta.

```

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
  CONSTANT cincomin: integer:=30000;    -- calculados suponiendo una frecuencia
  CONSTANT diezmin: integer:=60000;     -- de reloj de 100 Hz.
  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<=vacial;
            ELSIF tiempo=veintemin THEN presente<=vacial;
            END IF;
          END IF;
        WHEN vacial=>
          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;
      END CASE;
    END IF;
  END PROCESS;
END maquina;

```

```

        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 buena práctica poner el contador de tiempo fuera de la descripción de la máquina de estados, especialmente por claridad, pero en este caso hemos visto que también es posible incluirla dentro. La síntesis de este circuito requiere unos 105 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 veremos la otra posibilidad que consiste en poner las señales de salida en un proceso aparte que será completamente combinacional, 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
    CONSTANT cincomin: integer:=30000;    -- calculados suponiendo una frecuencia
    CONSTANT diezmin: integer:=60000;    -- de reloj de 100 Hz.
    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 subtiempos, tiempos: 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<=vacial;
        ELSIF tiempo=veintemin THEN presente<=vacial;
        END IF;
    WHEN vacial=>
        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';
        tiempos<=TRUE; subtiempos<=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';
            tiempos<=TRUE;
        ELSE
            llenar<='0';
            lento<='1';
            jabon<='0';
            tiempos<=FALSE;
        END IF;
        IF subtiempo=diezsec THEN
            dirauxd<=NOT diraux;
            subtiempos<=TRUE;
        ELSE
            subtiempos<=FALSE;
        END IF;
    WHEN vacial=>
        jabon<='0';
        vaciar<='1';
        lento<='0';
        rapido<='0';
        subtiempos<=TRUE;
        tiempos<=TRUE;
        llenar<='0';
        dirauxd<='0';
    WHEN aclarado=>
        jabon<='0';
        vaciar<='0';
        rapido<='0';
        IF lleno='0' THEN
            llenar<='1';
            lento<='0';
            tiempos<=TRUE;

```



```

        ELSE
            llenar<='0';
            lento<='1';
            tiempos<=FALSE;
        END IF;
        IF subtiempo=diezsec THEN
            dirauxd<=NOT diraux;
            subtiempos<=TRUE;
        ELSE
            subtiempos<=FALSE;
        END IF;
    WHEN vacia2=>
        jabon<='0';
        dirauxd<='0';
        vaciar<='1';
        lento<='0';
        rapido<='0';
        llenar<='0';
        subtiempos<=TRUE;
        tiempos<=TRUE;
    WHEN centrifugado=>
        jabon<='0';
        dirauxd<='0';
        llenar<='0';
        vaciar<='1';
        rapido<='1';
        lento<='0';
        subtiempos<=TRUE;
        tiempos<=FALSE;
    END CASE;
END PROCESS salida;

contador:
PROCESS(clk)
BEGIN
    IF clk='1' THEN
        IF subtiempos THEN
            subtiempo<=0;
        ELSE
            subtiempo<=subtiempo+1;
        END IF;
        IF tiempos 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 una lado se ha creado un nuevo proceso para el contador de tiempo, con lo que han sido necesarias 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 ahorramos registros. Con todo esto, esa descripción, que era equivalente a la anterior, ocupa unos 70 registros, que es un número sensiblemente inferior al anterior.

Esto ejemplo nos ha demostrado que dos descripciones que resuelven aparentemente el mismo problema, se pueden sintetizar de dos formas muy diferentes.

11.2.7 El concurso

Ejemplo 11.7 *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 interruptores donde programa la respuesta correcta (**correcto**), un pulsador que le sirve para iniciar el juego (**start**), otro pulsador que le sirve para indicar otra pregunta (**nueva**) y un botón de réset para inicializarlo todo. Una vez presionado **start** los concursantes deben pulsar el botón correspondiente a la pregunta que crean correcta. En el momento 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 acertó la respuesta se le sumarán 10 puntos en su marcador, pero si falló 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 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 **nueva**. Cuando algún jugador llegue a 100 puntos o más entonces habrá ganado y el juego se parará activándose la salida correspondiente al jugador que ha 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 le de al réset.*

*La frecuencia de reloj es fija y vale 1024 Hz. En caso de pregunta acertada, fallada, o que pasaron los 5 segundos, el operador siempre deberá pulsar **nueva** para hacer otra pregunta. Los interruptores se ponen y se quedan a uno o a cero hasta que se los cambie otra vez. Los botones están a uno mientras se pulsen, el resto del tiempo están a cero.*

En este caso se da una descripción más para modelado y simulación que para síntesis, ya que si se intenta sintetizar no sale lo que en principio debería ser. La razón es que cuando se describe una máquina de estados sin sincronía con un reloj, el sintetizador no lo optimiza por no reconocerlo como máquina de estados, y por otro lado está el problema de la interpretación que hace el sintetizador de la lista sensible; si nos fijamos en el proceso **salida**, la lista sensible no es más que la señal **presente**, lo cual significa que todas las señales de este proceso vienen sincronizadas por el cambio de estado, esto quiere decir que instrucciones como la de sumas 10 puntos, etc, sólo tienen lugar una vez durante el cambio de estado. Si esta descripción se sintetiza se observa que esto

no ocurre así, sino que lo que realmente sucede es que la instrucción que suma 10, por ejemplo, se repite indefinidamente con el retraso propio de las puertas mientras el pulso está en alto. Esto es así porque el sintetizador supone que todas las señales del proceso están en la lista sensible y lo sintetiza como lógica combinatorial y no funciona bien.

```

ENTITY ajugar IS
PORT (
-- Reloj de frecuencia fija 1024 Hz;
    clk: IN BIT;
-- Diferentes pulsadores o botones del operador:
    reset,start,nueva: 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;
-- Lineas para indicar quien de todos gana:
    ganaA, ganaB, ganaC: OUT BIT);
END ajugar;

ARCHITECTURE una_solucion OF ajugar IS
    TYPE estado IS (inicial,responde,evalua,tiempo,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 pulso.
    SIGNAL fin: BOOLEAN; -- Para saber cuando se llega al final.
    SIGNAL rescont: BOOLEAN; -- Pone a cero la cuenta.
    SIGNAL presente: estado;

BEGIN
    marcaA<=marcauxA; -- Senyales 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;

    maquina:
    PROCESS(reset,start,nueva,pulsaron,timeout,fin) -- senyales que cambian
    BEGIN -- el estado presente.
        IF reset='1' THEN presente<=inicial;
        ELSE
            CASE presente IS
                WHEN inicial=>
                    IF start='1' THEN presente<=responde; END IF;
                WHEN responde=>
                    IF pulsaron THEN presente<=evalua;
                    ELSIF timeout THEN presente<=tiempo;
                    END IF;
                WHEN evalua=>
                    IF fin THEN presente<=final;
                    ELSIF nueva='1' THEN presente<=responde;
            END CASE;
        END IF;
    END PROCESS maquina;

```

```

        END IF;
    WHEN tiempo=>
        IF nueva='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=>
            marcauxA<=0;
            marcauxB<=0;
            marcauxC<=0;
            ganaA<='0';
            ganaB<='0';
            ganaC<='0';
            rescont<=true;
        WHEN responde=>
            rescont<=false;
        WHEN evalua=>
            rescont<=true;
            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=>
            rescont<=true;
            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;
                ganaA<='1';
            END IF;
            IF marcauxB>=100 THEN
                marcauxA<=0;
                marcauxC<=0;
                ganaB<='1';
            END IF;
            IF marcauxC>=100 THEN
                marcauxB<=0;
                marcauxA<=0;
                ganaC<='1';
            END IF;
        END CASE;
END PROCESS;

```

```
END PROCESS salida;  
END una_solucion;
```

Aparte de que la máquina de estados no es síncrona con un reloj, hay otra diferencia con otras descripciones que hemos visto, y es que el contador del tiempo está situado en un proceso aparte, que no es raro, y se han utilizado unas señales para indicar los finales de cuenta.

11.2.8 El pin-ball

Ejemplo 11.8 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, tiene 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 de darle a la bola 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 le da al pivote **p_uno** se suman 5 y 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 hacemos síncrona para que resulte sencilla la síntesis del circuito.

Se ha utilizado una máquina de estados con un único proceso, esto en principio no nos supone ningún problema ya que importa poco que las señales de salida estén un

ciclo de reloj retrasadas respecto del cambio de estado, debido a que la señal de reloj se supone de una frecuencia elevada.

Como aspecto novedoso en este ejemplo se puede ver el tratamiento que se ha seguido con los pulsos producidos por los pivotes y el pulso producido cuando se pierde la bola (*pierde*). Ya se comentó al inicio del capítulo que un error común se daba en el tratamiento de pulsos más largos que la señal reloj, que provocaba que una misma operación se repitiera una y otra vez mientras dura el pulso. En este caso se ha evitado creando 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*. Por lo demás el programa sigue las reglas básicas para síntesis vistas en el resto de ejemplos.

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY pinball IS
PORT (p_uno,p_dos,falta,nueva,pierde,clk : 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);
  SIGNAL marcaux: integer RANGE 0 TO 4095 :=0; -- Auxiliar para leer la salida
  SIGNAL cien: integer RANGE 0 TO 127 :=0;    -- Servira para las bolas extras
  SIGNAL bolas: integer RANGE 0 TO 31 :=0;    -- Almacena el numero de bolas
  SIGNAL d_p_uno,d_p_dos,d_pierde: std_logic; -- Auxiliares para esas entradas
  SIGNAL presente: estado:=insertar;
BEGIN
  maquina:
  PROCESS(clk)
  BEGIN
    IF clk='1' THEN
      CASE presente IS
        WHEN insertar=>
          IF nueva='1' THEN presente<=inicia;
          END IF;
          fin<='1';
          bloqueo<='1';
        WHEN inicia=>
          presente<=juega;
          fin<='0';
          marcaux<=0;
          cien<=0;
          bloqueo<='0';
          bolas<=5;
        WHEN juega=>
          IF d_pierde='1' THEN presente<=resta; END IF;
          IF falta='1' THEN presente<=trampa; END IF;
          IF bolas=0 THEN presente<=insertar; END IF;
          IF d_p_uno='1' THEN presente<=suma_cinco; END IF;
          IF d_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<=juega;
          marcaux<=marcaux+5;
          cien<=cien+5;
        WHEN suma_diez=>
          presente<=juega;
      END CASE;
    END IF;
  END PROCESS;

```

```

        marcaux<=marcaux+10;
        cien<=cien+10;
    WHEN resta=>
        bolas<=bolas-1;
        presente<=juega;
    WHEN trampa=>
        IF d_pierde='1' THEN presente<=resta; END IF;
        bloqueo<='1';
    END CASE;
END IF;
END PROCESS maquina;

-- Como estos pulsos de entrada son mucho mas largos que el periodo del reloj
-- se crean estas senyales que son lo mismo pero duran un 'unico 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;

-- Finalmente se pone el marcador igual a su auxiliar.
marcador<=marcaux;

END sincrono;

```

11.3 Ejercicios propuestos

Ejemplo 11.9 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 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 11.2 se muestra el chip del cual se quiere hacer la descripción junto con sus entradas y salidas.

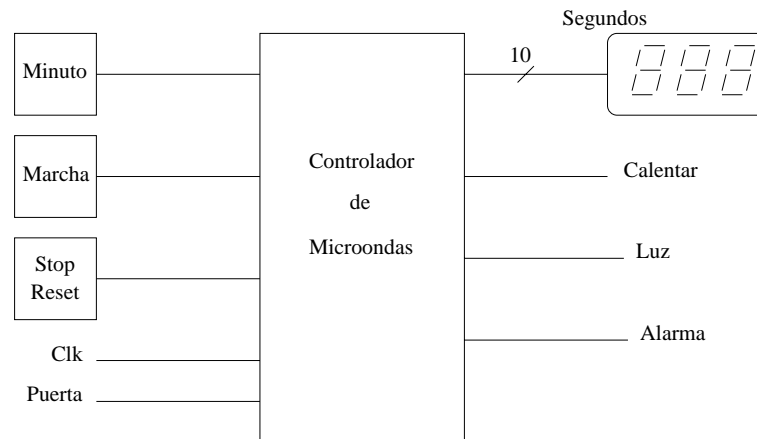


Figura 11.2: Figura del ejercicio del microondas

Ejemplo 11.10 Describir con VHDL el circuito que controla una máquina de café. Las entradas y salidas del circuito se muestran en la figura 11.3. 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 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 valen 60 pts.

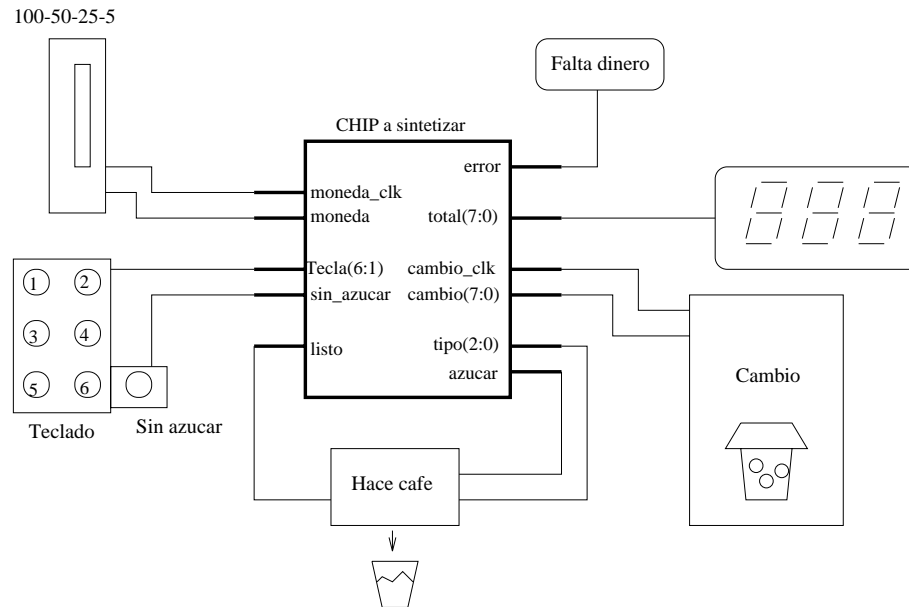


Figura 11.3: Figura del ejercicio de la máquina de café

Bibliografia

- [1] Barry Hawkes. *CAD/CAM*. Paraninfo, 1989.
- [2] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1991.
- [3] Douglas L. Perry. *VHDL*. McGraw-Hill, 2 edition, 1993.
- [4] Kevin Skahill. *VHDL for programmable logic*. Addison-Wesley, 1996.
- [5] Peter J. Ashenden. *The VHDL Cookbook*. 1990.
- [6] *Mentor Graphics Introduction to VHDL*, 1994.
- [7] *Mentor Graphics VHDL Reference Manual*, 1994.
- [8] *Altera VHDL*, 1996.
- [9] *WARP VHDL Synthesis Reference*, 1994.

Índice de Materias

- AFTER, 68
- Aggregate, *véase* Agregado
- Agregado, 31, 83
- ALIAS, 33
- ALL, 62
- ARCHITECTURE, 36
- ARRAY, 30
- ASIC, 2
- ASSERT, 72
- Atributos, 32

- Backannotation, 6
- Banco de pruebas, 71–72
- Bases, 27
- Bit, 63
- Bit_vector, 63
- BLOCK, 34, 42
 - Expresión de guardia, 43
- Bottom-Up, 3, 11, 12
- BUFFER, 35
- BUS, 33

- CAD, 1
 - herramientas, 2
- Cadenas, 27
- CASE, 51–52
- CI
 - herramientas, 3
- Circuitos integrados, *véase* CI
- Comentarios, 27
- COMPONENT, 26, 65
- Concatenación, 28
- CONFIGURATION, 65
- CONSTANT, 33

- DELAYED, 58
- Descripción
 - comportamental, 25, 39, 45
- Diseño
 - Bottom-Up, *véase* Bottom-Up
 - concurrente, 6
 - flujo, 1, 9
 - jerárquico, 12
 - VHDL, 42
 - modular, 12
 - Top-Down, *véase* Top-Down
- Driver, 48, 67

- EDA, 1, 6, 21
- EDIF, 13, 14
 - ejemplo, 13
- Ejecución
 - concurrente, 27, 41, 45
 - serie, 45, 49
- ELSE, 42, 51
- ELSIF, 51
- Entero, 29
- Entidad
 - declaración, 34, 73
- ENTITY, 35
- Esquemas, 2
- EVENT, 32
- Evento, 68, 71
- EXIT, 53
- Expresión de guardia, 43

- FOR, 52–53
 - en configuración, 65
- Forwardannotation, 6
- FPGA, 3
- Función de resolución, 63, 82
- FUNCTION, 57–59
 - de resolución, 82
 - Declaración, 58

- GENERIC, 35

- HDL, 2, 21

- Identificadores, 27
- IF..THEN..ELSE, 51
- IN, 35
- Ingeniería concurrente, 6
 - de grupo, 6
 - personal, 6
- INOUT, 35
- Integer, 29

- LIBRARY, 62
- Librería, *véase* VHDL Librería
 - ieee, 83
- LINKAGE, 35
- Lista sensible, 49
- LOOP, 52–53

- Máquinas de estados, 84–87

- Matrices, 30, *véase* Vectores
- Microprocesador
 - ejemplo, 98
- Modelado, 21, 22, 67
- Números, 27
- Netlist, 13–15, 21, 23, 25
 - ejemplos, 15
- NEXT, 53
- NOW, 73
- Operadores
 - aritméticos, 28
 - desplazamiento, 28
 - lógicos, 29
 - relacionales, 28
- OTHERS, 31, 42, 52, 83
- OUT, 35
- Overloading, 29, 60
- PACKAGE, 64
 - BODY, 64
 - declaraciones en, 34
- Package, 33
- PACKAGE BODY, 64
- PAL, 3
- Paquete, *véase* VHDL Paquete
- PCB, 2
 - herramientas, 3
- PLD, 3
- PORT, 35
- PORT MAP, 26
- PROCEDURE, 57–59
 - Declaración, 58
- Proceso pasivo, 73
- PROCESS, 49
 - Ejecución serie, 45, 46
 - en subprogramas, 58
- QUIET, 58
- RANGE, 29–31
- Real, 30
- RECORD, 31
- Registro, 31
- REPORT, 72
- Retraso
 - inercial, 70–71, 97
 - transportado, 70–71, 97
- Retroanotación, *véase* Backannotation
- RETURN, 57, 58
- ROL, 28
- ROM, 97
- ROR, 28
- RTL, 39, 40
 - Ejemplo, 44
- Síntesis, 22, 75
 - lógica combinatorial, 78
 - lógica secuencial, 79
- Señal
 - diferencias, 47
- SELECT, 42
- SEVERITY, 72
- SIGNAL, 33
- Simulación
 - de sistemas, 2
 - digital, 3
 - eléctrica, 3
 - funcional, 2
- SLL, 28
- Spice, 15, 18
- SRL, 28
- STABLE, 58
- Std_arith, 98
- Std_logic, 63, 83
- Std_logic_1164, 63, 83
- Subprogramas
 - Declaración, 58
- Subtipos, 31
- SUBTYPE, 31
- Tango, 15, 17
- Test bench, *véase* Banco de pruebas
- Time, 30
- Tipos, 29–31
 - compuestos, 30
 - enumerados, 30
 - escalares, 29
 - físicos, 30
 - resueltos, 31, 82
- Top-Down, 4, 9, 11, 12
- Top-down
 - ventajas, 5
- TRANSACTION, 58
- Transferencia entre registros, 25
- TRANSPORT, 71
 - Ejemplo, 97
- TYPE, 29
- USE, 62, 65
 - en configuración, 65
- VARIABLE, 33

- Variable
 - diferencias, 47
- Vectores, 30
- VHDL
 - Arquitectura, 36, 61
 - Atributos, 32
 - Buses, 81
 - Configuración, 65
 - Constantes, 33
 - ejemplos, 23, 89
 - Entidad, 34, 61
 - Introducción, 22
 - Librería
 - de diseño, 61
 - IEEE, 63
 - ieee, 83
 - std, 62
 - work, 61, 62
 - Operadores, 27
 - Paquete, 64
 - standard, 62, 83
 - std_logic_1164, 63
 - textio, 62
 - Señales, 33
 - sintaxis, 27
 - Subprogramas, 57–61
 - llamadas a, 59
 - Tipos de datos, 29
 - Subtipos, 31
 - Unidades, 61
 - Variables, 33
 - ventajas, 22
- WAIT, 34, 49–50
 - en subprogramas, 60
- WHEN, 42, 52
- WHILE, 52–53
- WITH, 42