UNIVERSITAT DE VALÈNCIA

TECNOLOGÍA INFORMÁTICA
(Parte Primera)

Fernando Pardo Carpio

Ingeniería Informática
Universidad de Valencia

Indice General

Indice de Figuras

Prólogo

Los apuntes que se contienen en las próximas páginas corresponden a parte 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.

Resulta evidente que 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, Abril de 1996

Parte I

Descripción

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.

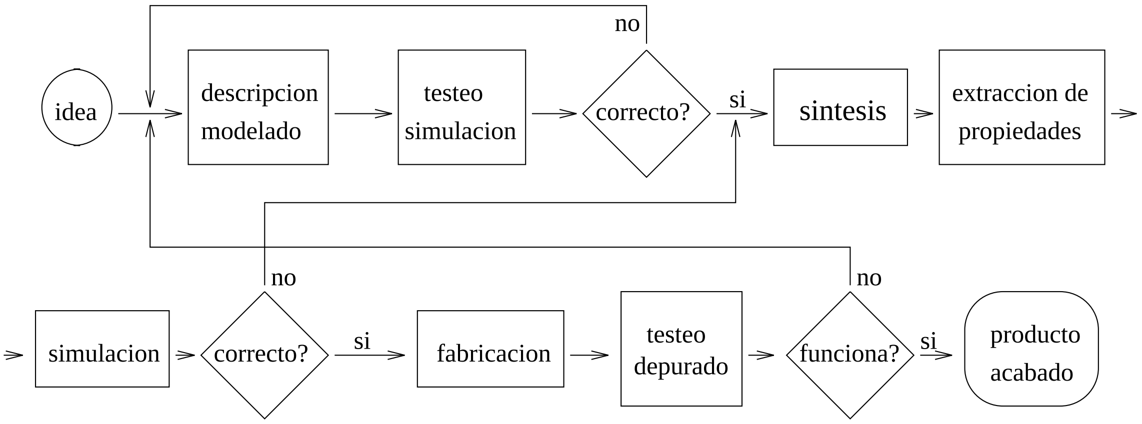


Figura 1.1: *Flujo de diseño para sistemas electrónicos y digitales*

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 eventos. 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.

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.

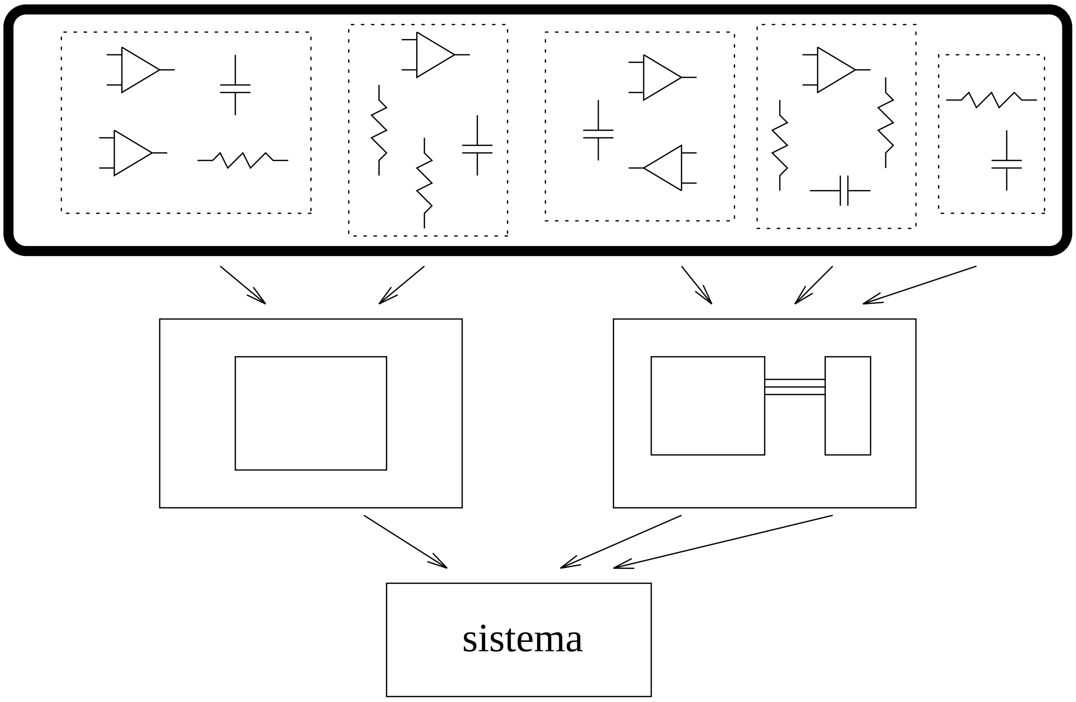


Figura 1.2: *Metodología de diseño Bottom-Up*

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 funcionales, 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 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.

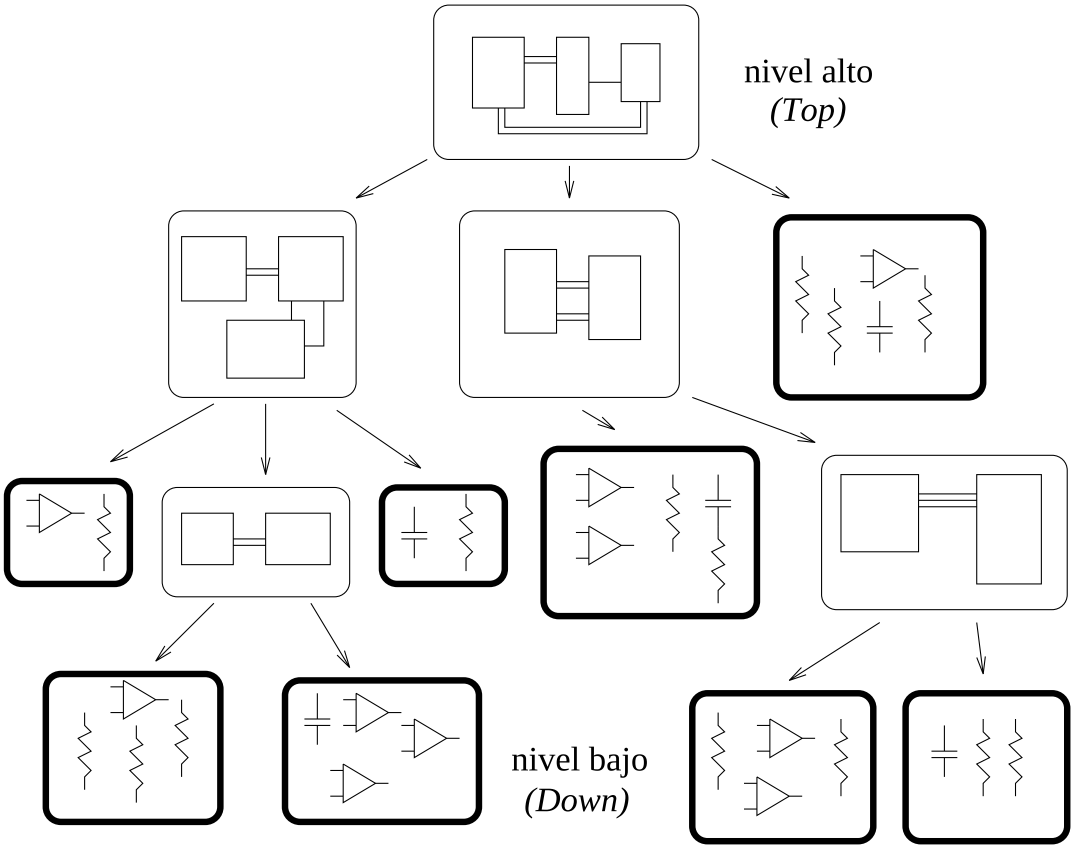


Figura 1.3: *Metodología de diseño Top-Down*

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 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 tipos 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 esquemático 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 esquemático 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 interconexionar 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 son 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 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. 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 descripción mediante un lenguaje que se empezó a usar. 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 como se conectan entre si.

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 pruotro (cellType generic)
      (view pruotro (viewType netlist)
        (interface
          (port in (direction INPUT)
            (property pin (string "in"))
            (property pintype (string "in"))
          )
          (port out (direction OUTPUT)
            (property pin (string "out"))
            (property pintype (string "out"))
          )
```

```
        )
      )
    )
  )
  (design pruotro (cellRef pruotro (libraryRef &_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 pruotro (cellType generic)` donde se indica que se va a describir una célula llamada internamente `pruotro`. 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 sólamente 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.

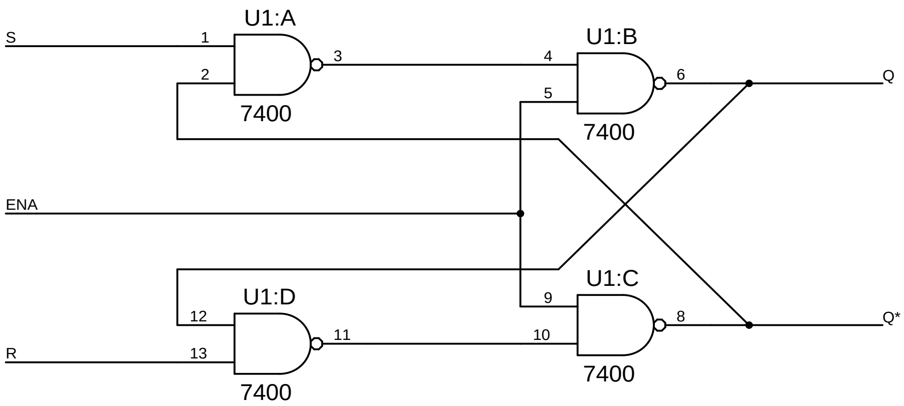


Figura 2.1: *Ejemplo de esquema para su descripción Netlist*

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 ROOT
  (CellRef TI
   (LibraryRef TI_LIB))
 )
 (Library TI_LIB (EdifLevel 0)
  (technology (numberDefinition (scale 1 (E 254 -7) (unit DISTANCE))))
  (cell U1 (cellType GENERIC)
   (property Type (string "7400"))
   (view S (viewType SCHEMATIC)
    (interface
     (Port A (Designator "1")
      (Direction INPUT )
     )
     (Port B (Designator "2")
```

```
      (Direction INPUT )
     )
     (Port Y (Designator "3")
      (Direction OUTPUT )
     )
     (Port A (Designator "4")
      (Direction INPUT )
     )
     (Port B (Designator "5")
      (Direction INPUT )
     )
     (Port Y (Designator "6")
      (Direction OUTPUT )
     )
     (Port GND (Designator "7")
      (property ElecType (string "Power") )
     )
     (Port Y (Designator "8")
      (Direction OUTPUT )
     )
     (Port A (Designator "9")
      (Direction INPUT )
     )
     (Port B (Designator "10")
      (Direction INPUT )
     )
     (Port Y (Designator "11")
      (Direction OUTPUT )
     )
     (Port A (Designator "12")
      (Direction INPUT )
     )
     (Port B (Designator "13")
      (Direction INPUT )
     )
     (Port VCC (Designator "14")
      (property ElecType (string "Power") )
     )
    )
   )
  )
  (cell TI (cellType GENERIC)
   (view N
    (viewType NETLIST)
    (interface)
    (Contents
     (net ENA
      (joined (portRef &B (viewRef S (cellRef U1)))
              (portRef &A (viewRef S (cellRef U1)))
      )
     )
     (net GND
      (joined (portRef &GND (viewRef S (cellRef U1)))
      )
     )
```

```
    (net NET_002
     (joined (portRef &B (viewRef S (cellRef U1)))
             (portRef &Y (viewRef S (cellRef U1)))
     )
    )
    (net NET_004
     (joined (portRef &Y (viewRef S (cellRef U1)))
             (portRef &A (viewRef S (cellRef U1)))
     )
    )
    (net Q
     (joined (portRef &Y (viewRef S (cellRef U1)))
             (portRef &A (viewRef S (cellRef U1)))
     )
    )
    (net Q*
     (joined (portRef &B (viewRef S (cellRef U1)))
             (portRef &Y (viewRef S (cellRef U1)))
     )
    )
    (net R
     (joined (portRef &B (viewRef S (cellRef U1)))
     )
    )
    (net S
     (joined (portRef &A (viewRef S (cellRef U1)))
     )
    )
    (net VCC
     (joined (portRef &VCC (viewRef S (cellRef U1)))
     )
    )
   )
  )
 )
)
)
```

Se ve que es una descripción bastante larga para lo pequeño del circuito, pero esto es necesario si se pretende una completa compatibilidad con el resto de herramientas que existen. La primera parte es el encabezamiento y contiene básicamente comentarios. A continuación viene la parte de librería donde se definen los compomentes que aparecen. En este caso hay un único componente, representado por cuatro símbolos, que corresponde al 7400 de Texas Instruments y que contiene 4 puertas NAND. El componente viene definido por sus puertos, cada uno con un nombre diferente y una dirección de entrada o salida. Además, en este caso, cada puerto contiene un `designator` que sirve para indicar la posición del pin dentro del encapsulado que es un DIP14 en este caso. Hay que destacar que aunque en el esquema no aparecen las tensiones de alimentación, sí que aparecen en cambio en el Netlist. Esto es debido a que muchas veces estas tensiones, tanto VCC como GND, aparecen ocultas en los símbolos del esquema, pero son necesarias en el Netlist ya que éste debe contener absolutamente toda la información, incluida la alimentación del circuito.

Después de la definición del componente viene la definición del interconexionado. El emplazamiento del componente se realiza mediante la instrucción `cell`. Las interconexiones se realizan en el bloque indicado por la palabra clave `Contents`. Cada conexión se indica en un bloque `net` donde se indica el nombre y qué cosas conecta mediante el bloque `joined`. Por ejemplo, la primera conexión (ENA) conecta los puertos A y B del chip U1 que es el 7400 especificado en la librería. Y así se van conectando los diferentes elementos. Hay conexiones que no tienen nombre en el esquema, pero todas las conexiones en un Netlist deben tener un nombre, así que lo que hace la herramienta en estos casos es inventarse nombres. Este es precisamente el caso de las conexiones `NET_002` y `NET_004` que son los nombres que la herramienta se ha inventado.

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

```
[
U1
DIP14
7400
]

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

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

En la cabecera, las primeras líneas encerradas entre corchetes, se encuentra la parte de definición de los elementos. Símplemente 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 símbolos; 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

Lenguajes de descripción hardware: VHDL

3.1 Introducción a los lenguajes de descripción del hardware

Se vio en el capítulo anterior, que la forma más común de describir un circuito es 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 interconexionado entre los componentes de un diseño, es decir, se trataba de una *lista de conexiones*.

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

Con una mayor sofisticación de las herramientas de diseño, y con la puesta al alcance de todos de la posibilidad de fabricación de circuitos integrados y de 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.

3.2 Introducción al VHDL

VHDL, significa 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 enteder) 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. Se trata de 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 jeraquía 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.

VHDL describe estructura y comportamiento

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.

3.2.1 Ejemplo básico de descripción VHDL

Ejemplo 3.1 *Describir en VHDL un circuito que multiplexe 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.

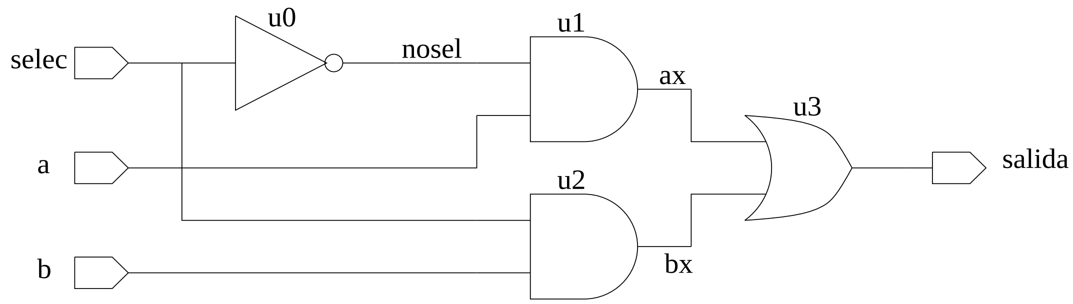


Figura 3.1: *Esquema del ejemplo básico en VHDL*

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 diferencias. 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.

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

  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 la anterior comportamental. Lo primero es definir los componentes. 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 claúsula `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 comunmente 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`. Depués 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.

3.3 Elementos sintácticos

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.

Indentificadores: 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 objeto. 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: `'l'`,`'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"`.

3.3.1 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 MOD b)` donde **N** es un entero. Los operandos sólo pueden ser enteros. El resultado toma el signo de `b`.

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

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

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

Operadores de desplazamiento

SLL, SRL (desplazamiento lógico a izquierda y a derecha)

SLA, SRA

ROL, ROR (rotación a izquierda y a derecha)

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 lista:

```
**        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.

3.3.2 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*.

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

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

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

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"`

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 4 TO 7;
SUBTYPE digitos IS character '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 DOWNTO 0);
```

3.3.3 Declaración de objetos

Un objeto en VHDL es un elemento que contiene un valor de un tipo especificado. Hay tres tipos de objetos 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 objeto 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 son cosas diferentes. Las variables, por ejemplo, sólo tienen sentido dentro de PROCESS, es decir, sólo tienen sentido en entornos de programación donde las sentencias son ejecutadas secuencialmente. Las constantes pueden ser declaradas o dentro de un PROCESS o dentro de un PACKAGE. Las variables sólo pueden ser declaradas y usadas dentro de un PROCESS y son locales a éste.

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. La 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 PROCESS y bloques dentro de una arquitectura, por lo que en realidad representan interconexiones entre bloques dentro de la arquitectura.

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 mientras 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. Esta forma extraña en que se les asignan valores a las señales se entenderá mejor cuando se explique el significado de los PROCESS, la ejecución concurrente y secuencial, y los pasos de simulación.

3.3.4 Atributos

Los objetos en VHDL, como señales, variables, etc, pueden tener información adicional llamada *atributos*. Estos atributos están asociados a estos objetos 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 objeto 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.

3.3.5 Declaración de entidad y arquitectura

Ya se han 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 objeto 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 objetos 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 DOWNTO 0);
        data: OUT bit_vector(ancho-1 DOWNTO 0));
END rom;

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

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.

3.4 VHDL como lenguaje de descripción comportamental

Se vio en el ejemplo 3.1, al principio de este capítulo, 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 secuencialmente 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 secuencial de las instrucciones que tienen mucho sentido en la ejecución de programas, pero cuyo significado hardware es algo más difuso.

3.4.1 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 objetos abstractos del lenguaje en vez de entre componentes físicos. Además, se permiten estructuras de tipo condicional que facilitan la abstracción de ciertos comportamientos.

En el ejemplo 3.1 se mostró la descripción de un multiplexor sencillo. Entonces se presentaron dos posibles descripciones, una completamente estructural, y la otra completamente comportamental abstracta. Vamos a ver a continuación cómo sería la descripción RTL a nivel de transferencia de registros de aquel circuito:

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

Es fácil comprobar que esta descripción 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 de esta descripción no es secuencial, 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 3.2 *Realizar una descripción RTL, transferencia entre registros, de un comparador de dos buses (*`a` *y* `b`*) de 11 bits. El comparador tendrá tres salidas activas a nivel alto. Una se activará si* `a=b`*, otra si* `a>b`*, y la última si* `a<b`*.*

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

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

En esta ocasión se tienen tres instrucciones en vez de una. Estas instrucciones no se ejecutan secuencialmente, 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.

3.4.2 Estructuras de la ejecución concurrente RTL

Se ha visto anteriormente en los ejemplo cómo asignar señales y hemos visto también la instrucción condicional básica en la ejecución concurrente que se hacía mediante la palabra clave `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, 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 importante, al igual que ocurría con la asignación condicional, incluir todos los posibles valores que pueda tomar la expresión. Es por esto que en el ejemplo anterior se haya incluido la claúsula `WHEN OTHERS`. Si no se hace así, se corre el peligro de que no se simule bien el circuito, o que no se pueda sintetizar a partir de esta descripción, debido a que 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. De esta manera se puede subdividir un mismo programa en una jerarquía de bloques 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 guarda)
   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 y simulación. La cabecera puede tener claúsulas 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. La parte de declaración puede incluir desde subprogramas a señales al igual que al principio de la arquitectura, siendo la visibilidad local al bloque.

Encerrados entre el `BEGIN...END` se encuentran las instrucciones concurrentes a ejecutar, por tanto se pueden incluir otros bloques dentro de un bloque, dando como resultado un diseño jerárquico.

Ejemplo 3.3 *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 descipción 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;
```

3.4.3 Descripción 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 de forma secuencial, 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 secuenciales, 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 secuencial. En VHDL esta programación secuencial se define dentro de bloques indicados con la palabra clave `PROCESS`. Por tanto, siempre que en VHDL se precise de una descripción secuencial, se deberá utilizar un bloque de tipo `PROCESS`.

En un mismo programa puede 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 secuencial, 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? En la ejecución concurrente se había visto que una instrucción se activaba 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, condicionales, y no se tiene por qué saber cuándo activar el bloque y ejecutarlo. 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 un `WAIT` explícito en algún lugar dentro del bloque `PROCESS`. Más adelante se explicarán con mayor detalle estos mecanismos.

Ejemplo 3.4 *Realizar la descripción comportamental secuencial del comparador del ejemplo 3.2.*

```
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 secuencial, 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;
```

3.4.4 Estructuras de la ejecución secuencial comportamental

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

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 existían 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 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 secuenciales. Como se trata de una única instrucción concurrente, todas las instrucciones secuenciales 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 `PROCESS` 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 secuenciales, por lo que sólamente se las puede definir dentro de bloques `PROCESS` 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:

```
-- Uso incorrecto de las senyales
ARCHITECTURE ejemp1 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 ejemp1;
```

```
-- Uso correcto de las variables
ARCHITECTURE ejemp1 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 ejemp1;
```

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` se le asigna el valor `c+2`, pero cogiendo como `c` el valor de la señal antes de empezar el `PROCESS`.

En defnitiva, 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, sobreescribe 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 secuencial 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`. Depués 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.

El bloque de ejecución secuencial: `PROCESS`

La forma de entrar en la ejecución secuencial 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 secuenciales
END PROCESS proc_id;
```

El `proc_id` es simplemente una etiqueta que se puede poner o no, y que puede servir para ponerle nombre a los diferentes `PROCESS`. 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.

La parte de declaración no resulta ser diferente de otras estructuras, de forma que se pueden definir aquí señales, tipos, subprogramas, etc. Además, se pueden definir también variables, cosa que no se puede hacer en otros tipos de estructura como entidades, bloques concurrentes o arquitectura. A continuación, y entre el `BEGIN...END`, vienen todas las instrucciones secuenciales, 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 lazo 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 bloque `PROCESS` y la simulación no acabaría nunca. Por lo tanto, debe existir un mecanismo que permita detener la ejecución del bloque secuencial. 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 bloque 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 posivilidad más compleja de detener la ejecución del bloque secuencial resultando la lista sensible un caso particular de esta operación más compleja. La forma genérica de detener la ejecución en un bloque `PROCESS` se realiza mediante la palabra clave `WAIT` que suspende la ejecución secuencial 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 continue con la ejecución del bloque secuencial. Ejemplos:

```
WAIT ON pulso;
WAIT UNTIL counter>7;
WAIT FOR 1ns;
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:

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

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

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

Ejemplo 3.5 *Realizar un process 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, 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 legivilidad.

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 secuencia mediante un bloque PROCESS. Ejemplo:

```
-- Ejecucion secuencial
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;
```

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

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 expresion 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 LOOP`s

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 indentificador especificado en el rango. Esto quiere decir que el tipo del indentificador 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 instruciones 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 corriente 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                              -- Lazo WHILE
FOR cuenta IN 5 DOWNTO 0 LOOP            cuenta:=5;
   tabla(cuenta)<=cuenta*2;              WHILE cuenta>=0 LOOP
END 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;
```

3.4.5 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. Por ejemplo, una función siempre devuelve un valor, mientras que un procedimiento puede devolver valores a través de los parámetros que se le pasen. Los parámetros, o 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 de entrada.

Declaración de procedimientos y funciones

Las declaraciones de procedimiento y función se encuentran a continuación:

```
-- Procedimientos:
PROCEDURE nombre(parametros) IS
   declaraciones
BEGIN
   instrucciones
END nombre;
```

```
-- Funciones:
FUNCTION nombre(parametros) RETURN tipo IS
   declaraciones
BEGIN
   instrucciones -- incluye RETURN
END nombre;
```

La lista de `parametros` es opcional y si no hay parámetros tampoco se ponen 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 sobreentederá que es una constante. Para el resto de tipos, y si no se especifica otra cosa, se entenderá que son de tipo `VARIABLE`. Si tampoco se especifica el tipo de puerto se entenderá que es el de entrada `IN`.

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 y señal, y como tipo de puerto sólo se admite el `IN`, puesto que los parámetros no pueden ser modificados en una función. En la función no se admite la variable puesto que la clase constante se comporta exactamente igual. 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.

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 secuencial. No tiene por tanto mucho sentido declarar señales en una función o procedimiento. Naturalmente, todo lo que se declara en esta parte sólo es visible en el cuerpo de la función.

En 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: OUT 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);
      IF max<conjunto(ind) THEN max:=conjunto(ind);
   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.

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 concurrente de un procedimiento 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 secuencial. Como en el `PROCESS` es necesario incluir algo que permita suspender la ejecución del procedimiento, de otra manera se ejecutaría indefinidamente. Como no existe una manera de meter una lista sensible en el procedimiento, esto significa que *todo procedimiento que se invoque desde entornos concurrentes debe incluir al menos una sentencia de espera* `WAIT`.

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 covertura 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 3.6 *Se define el tipo enumerado* `logico` *como* `('X','0','1','Z')`*, es decir, como el tipo* `bit` *pero añadiendo los valores* `'X'` *(desconocido) y* `'Z'` *(alta impedancia). Sobrecargar el operador* `AND` *para que se pueda usar esta operación con este nuevo tipo.*

```
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";
```

3.5 Organización de los programas: 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*. 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.

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 claúsula `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 claúsula de librería pueden haber también claúsulas 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 referenciadas 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 es también 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 lógicos 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 3.7.1). 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 continuación se van a mostrar las unidades que no se han visto hasta ahora y que son los paquetes, cuerpo de los paquetes y configuraciones.

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:

```
-- Declaracion de paquete
PACKAGE nombre IS
   declaraciones
END nombre;
```

```
-- Declaracion del cuerpo
PACKAGE BODY nombre IS
   declaraciones, instrucciones, etc.
END nombre;
```

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

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

```
PACKAGE BODY tipos_mios IS
```

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

Una vez se han declarado los paquetes de esta manera. los elementos de los que 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);
```

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 claúsulas 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:

```
-- bloques
FOR nombre_bloque
   clausula use
   elementos
END FOR;
```

```
-- bloques
FOR nombre_componente
   USE objeto union
   definicion bloque
END FOR;
```

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 claúsula `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 3.7 *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 aqui, 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;
```

3.6 Síntesis y modelado de circuitos

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 una simulación. Estos elementos son retrasos, señalización de errores, etc. en síntesis, estos elementos, especialmente los retrasos, no tienen ningún sentido y se deben de evitar.

A continuación se explicarán los dos estilos de describir hardware atendiendo a su objetivo, es decir, la simulación o la síntesis. Veremos que cada estilo tiene sus propias estructuras, y que incluso puede haber comandos que sirvan sólamente para uno de los estilos mientras que para el otro no.

3.6.1 VHDL para 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 pueda implementar, existe siempre un retraso entre que se producen los estímulos de entrada y la salida cambia.

Para poder implementar 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 un señal, pero cuando haya trascurrido 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 depués de trascurrido 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 trascurre gracias a la sucesión de eventos. En la instrucción anterior, donde se asignaba un `'0'` a `senal` después de 15 ns, en realidad estamos produciendo un evento que tendrá lugar dentro de 15 ns. El simulador de VHDL guarda una lista de todos los eventos que se van produciendo y los ordena según el momento en que tengan que ocurrir. 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, por lo que el `tiempo de simulación` habrá trascurrido. 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. Una descripción del flujo que suele seguir un simulador en VHDL (casi generalizable a cualquier simulador digital) se muestra en la figura 3.2 donde $\Delta\delta$ es un paso de simulación donde el tiempo no corre, y ΔT es un paso donde el tiempo corre realmente.

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'`.

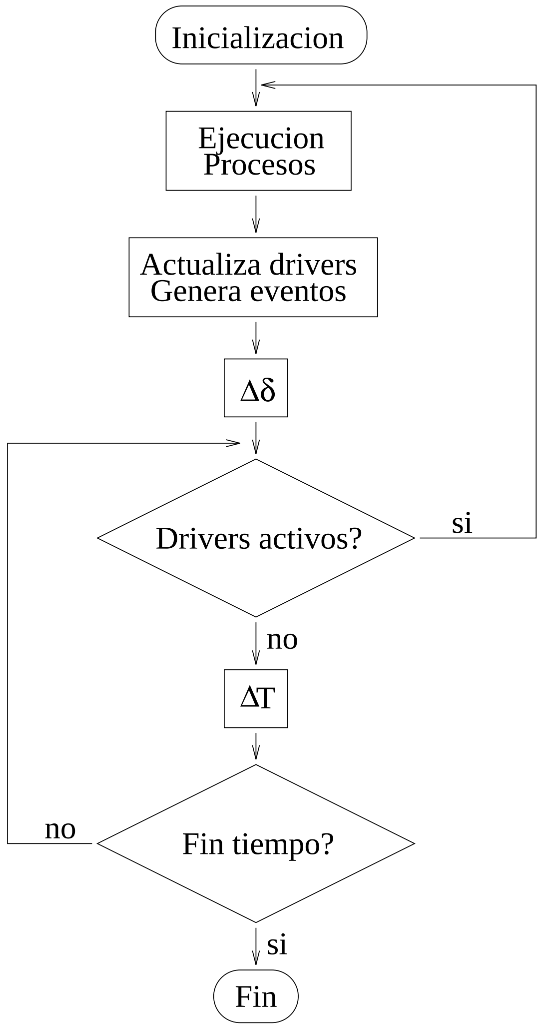


Figura 3.2: *Flujo de simulación en VHDL*

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 definirán 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 25 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 sigamos la simulación nanosegundo a nanosegundo empezando en cero. Supongamos que la entrada es cero al inicio (0 ns). Después de 50 ns la salida se pondrá a uno. Dejemos pasar 50 ns más, entonces el tiempo de simulación será 100 ns. En este momento se produce un cambio en la entrada pasando a uno. Esto quiere decir que se introduce el evento "poner a cero la salida después de 50 ns (150 ns)" en la lista de eventos de la salida. Supongamos que esperamos otros 25 ns más (125 ns) 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 (175 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 no se ejecutará nunca y la señal de salida nunca pasará a cero a pesar de que la entrada ha sido uno por 25 ns.

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 trasmisión, 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, 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 actualizarse se actualiza.

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.

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 las 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 o *test bench* en inglés.

Este banco de pruebas no es más que la definición de unas cuantas entradas llamadas *patrones de test* con las que testear 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 VHDL, 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;
```

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 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 tipo del nivel de gravedad es el predefinido `severity_level` y si no se especifica nada, por defecto es `error`. Los niveles de gravedad son: `note, warning, error` y `failure`. 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 secuenciales. En entornos secuenciales se ejecutará cada vez que cambien algunas de las señales que intervienen en la condición. En el entorno secuencial se ejecuta cuando le toque el turno en el proceso normal de ejecución. A continuación se dan dos ejemplos de utilización de estos mensajes:

```
-- Violacion de setup:
ASSERT NOT(clk'EVENT AND clk='1' AND preset='1' AND NOT(preset'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";
```

3.6.2 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, pero seguro que funciona.

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 implementar el circuito.

Las herramientas de síntesis actuales cubren a la perfección la síntesis a partir de descripciones RT 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 implementar.

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 aclara también si un circuito es combinacional, secuencial, síncrono, etc. Algunas de estas líneas a seguir serían:

Evitar las claúsulas temporales Normalmente los simuladores prohiben 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 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 3.5.

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.

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.

Evitar `IF`s anidados Normalmente las herramientas tienden a no sintetizar de manera óptima varios condicionales anidados entre sí. Los condicionales es mejor utilizarlos a solas.

Utilizar `CASE` mejor que varios `IF`s Las estructuras `CASE` tienen para los sintetizadores un modelo optimizado de síntesis, generalmente mejor que lo mismo descrito mediante `IF`s.

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`.

Descripción de lógica combinacional

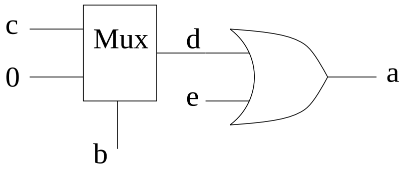
Un circuito, o parte de él, es combinacional si las salidas sólo dependen de las entradas y no del estado en que se encuentre el circuito. En VHDL se puede describir lógica combinacional siempre que se cumplan los siguientes requisitos:

- 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
    ```
  - No hay lazos combinacionales (en realidad esta es una extensión de la anterior). Ejemplos:
    ```
    -- Secuencial                          -- Combinacional
    d<=b AND a;                            d<=b AND c;
    a<=d OR e;                             a<=d OR e;
    ```
- Si la ejecución es secuencial se sintetiza lógica combinacional cuando:

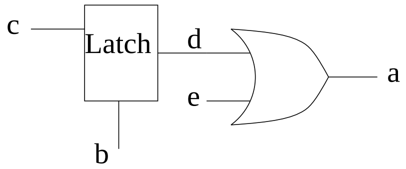
– La lista sensible de un proceso incluye todas las señales implicadas en las asignaciones. Es claro que si alguna señal no está en la lista sensible, cuando se produzca un cambio en esta señal, el proceso no se ejecutará y no habrá cambios en las señales, por lo que se tratará de un circuito secuencial.

– 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. 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;
```



Descripción de lógica secuencial

Un circuito es secuencial si su salida puede depender del estado del sistema. En este sentido 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. Vamos cómo se describe esto de forma secuencial y concurrente:

Secuencial 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;
```

Concurrente Suele implementarse 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 pura lógica combinacional donde se da una realimentación. Dependiendo de una herramienta u otra interpretará lo anterior como combinacional o secuencial.

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.

Descripción de registros Los registros son como los latches pero la entrada pasa a la salida cuando se produce un flanco de la señal de reloj:

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

En el ejemplo de la derecha se observa que en realidad no hace falta especificar la condición de evento ya que al estar la señal de reloj sola en la lista sensible, sólo se ejecuta el proceso si se produjo un evento en la señal. Ambos dos ejemplos son equivalentes y sintetizan lo mismo, un biestable master-slave tipo D,

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. Veamos algunas:

- Sólo debe permitirse un evento 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.
- Cuando en un `IF` se comprueba el flanco del reloj, no debe seguir un `ELSE`. Se podría poner y tendría sentido, pero la síntesis se complicaría innecesariamente.
- 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.

3.7 Conceptos avanzados en VHDL

A lo largo de este capítulo, dedicado íntegramente al lenguaje VHDL, se ha intentado dar una visión general de lo que es el lenguaje 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 si 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.

3.7.1 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 en VHDL. Por lo tanto, no es sobre la creación de buses sobre lo que trata esta sección.

Normalmente en un bus, y se puede dar también en una señal, se da la característica de que hay varios elementos conectados en el mismo bus. 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. Normalmente este tipo de buses se controlan mediante la utilización de buffers, de manera que sólo hay un buffer activo que escribe sobre el bus y el resto se encuentran en alta impedancia. Desde el punto de vista del VHDL, en realidad todos los buffer están escribiendo algo sobre el bus, se necesita por tanto un mecanismo que permita ignorar los buffers que escriben alta impedancia sobre el bus, y quedarse con aquellos que escriben datos reales. A este mecanismo se le llama **resolución de señales**.

Otro ejemplo lo tenemos en algunas líneas de arbitraje donde varios dispositivos escriben, y además escriben valores lógicos como `'0'` o `'1'`, en esos casos la función de resolución decidirá qué valor escoger dependiendo de que se quiera implementar lo que se llama una *OR cableada* o *AND cableada* o cualquier otro tipo de función.

En VHDL normalmente se admite un único driver por señal (en el caso de poner retrasos son varios driver consecutivos, pero sólo hay uno que escribe sobre la señal), lo que significa que sólo puede asignársele un valor a una señal. Para poder escribir múltiples valores en una señal a un mismo tiempo, hay que utilizar un tipo especial de señales llamado *resuelto* porque tiene en cuenta todas las escrituras sobre una señal y a partir de ellas calcula lo que realmente escribe. Un tipo *resuelto* incluye en su definición una función de resolución que es precisamente la que resuelve el conflicto de una escritura múltiple y decide qué valor asignarle a la señal.

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 lógicos 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 sera':

```
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;
```

3.7.2 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 regitro 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.

Normalmente, es más conveniente tener la máquina de estados sincronizada con algún tipo de reloj. Esto se hace así para tener todo un sistema complejo sincronizado con una señal. Además se evitan problemas de metaestabilidad en las transiciones entre estados. 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 (proces estados) de la combinacional (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 implementar 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 réset, 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 combinacional:
salida:
PROCESS(entrada,presente)
   CASE presente IS
     WHEN est1=>
        salida<=g1(entrada);
     WHEN est2=>
        salida<=g2(entrada);
     .
     .
     WHEN estN=>
        salida<=gN(entrada);
   END CASE;
END PROCESS salida;
```

```
END mealy;
```

En la descripción anterior el proceso `salida` no depende del réset puesto que fijando un estado cuando está activa la señal de réset, fijamos también la salida. En el caso de necesitar una salida especial para cuando el réset está activo se especificaría en este proceso, pero esto en general no es necesario ya que se suele elegir el estado correspondiente al réset 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.

3.8 Ejemplos en VHDL

En esta sección se muestran unos cuantos ejemplos de utilización del VHDL para simulación y síntesis.

Ejemplo 3.8 *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 bien sencilla. En principio se puede pensar en dos estados y si el botón está a uno se pasa de uno a otro. Esto en principio se puede hacer, y se verá más adelante, pero lo normal es pensar en cuatro estados que eviten esta transisción 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, 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 implementa lógica secuencial que no aparece explícitamente en la descripción:

```
ARCHITECTURE pseudomaquina OF conmutador IS
  TYPE estado IS (apagado,encendido);
  SIGNAL presente: estado:=encendido;
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;
```

Ejemplo 3.9 *Realizar el circuito 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 ambar, 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 seáforo 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. Junto con la máquina de estados habrá otros procesos que sontrolen los tiempos de espera mediante contadores.

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

```
ENTITY semaforo IS
PORT (sensor,reset,clk: IN bit;
      semcamin,semcarr: OUT bit_vector(0 TO 2));
END semaforo;

ARCHITECTURE descripcion OF semaforo IS
   TYPE estado IS (inicial,carramarillo,caminverde,caminamarillo,espera);
   CONSTANT verde:    bit_vector(0 TO 2):="001";
   CONSTANT amarillo: bit_vector(0 TO 2):="010";
   CONSTANT rojo:     bit_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)
BEGIN
   IF clk='1' THEN
     CASE presente IS
       WHEN inicial=>
          IF sensor='1' AND reset='0' THEN
             presente<=carramarillo;
          END IF;
       WHEN carramarillo=>
          IF reset='1' THEN
             presente<=inicial;
          ELSE
             presente<=caminverde;
          END IF;
       WHEN caminverde=>
          IF reset='1' THEN
             presente<=inicial;
          ELSIF fin_corto THEN
             presente<=caminamarillo;
```

```
          END IF;
       WHEN caminamarillo=>
          IF reset='1' THEN
             presente<=inicial;
          ELSE
             presente<=espera;
          END IF;
       WHEN espera=>
          IF reset='1' OR 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;
          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 deteccion de los tiempos largos y cortos:
fin_largo<=true WHEN cuenta=29 ELSE false;
```

```
fin_corto<=true WHEN cuenta=9  ELSE false;

END descripcion;
```

Ejemplo 3.10 *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 permanerán 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 DOWNTO 0);
     clk,reset,celula: IN bit;
     motor:  OUT bit_vector(0 TO 1);
     puerta: OUT bit);
END ascensor;

ARCHITECTURE mover OF ascensor IS
   TYPE estado IS (inicial,cerrar,voy);
   SUBTYPE vector IS bit_vector(2 DOWNTO 0);
   SIGNAL presente: estado:=inicial;
   SIGNAL bot: bit_vector(2 DOWNTO 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<=voy;
          END IF;
       WHEN cerrar=>
          IF celula='0' THEN presente<=voy;  -- Sin obtaculos
```

```
          END IF;
       WHEN voy=>
          IF bot(1 DOWNTO 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 DOWNTO 0)>piso THEN
             motor<="10";  -- Subir
        ELSE motor<="01";  -- Bajar
        END IF;
   END CASE;
END PROCESS salida;

END mover;
```

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.

Ejemplo 3.11 *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
      "ZZZZZZZZ" WHEN cenr='1' ELSE
      "XXXXXXXX";
cenr<=cen AFTER 60 ns;

END modelo;
```

Ejemplo 3.12 *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 program 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 al final. continuación:*

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.

En la descripción se supone que existe el operador suma y resta que opera sobre el tipo `std_logic_vector`. Esto es razonable de suponer ya que la mayoría de las herramientas de síntesis de circuitos tienen sobrecargados estos operadores. Veamos la descripción:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY procesador IS
PORT(clk,rst: IN std_logic;
     r_w: OUT std_logic;
     dir: OUT std_logic_vector(7 DOWNTO 0);
     dat: INOUT std_logic_vector(7 DOWNTO 0));
END procesador;

ARCHITECTURE descripcion OF procesador IS
  TYPE estado IS (inicial,busqueda,ejec,ldxxa,ldaxx,anda,adda,suba);
  SIGNAL a,pc,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<="00000000";
    rwaux<='1';
    ir<="00000000";
    a<="00000000";
    presente<=busqueda;
  WHEN busqueda=>
    ir<=dat_in;
    pc<=pc+"00000001";
    IF dat_in(2 DOWNTO 0)="111" THEN presente<=busqueda;
    ELSE presente<=ejec;
```

```
    END IF;
  WHEN ejec =>
    seldir<='0';  -- dir<=dat_in
    pc<=pc+"00000001";
    presente<=busqueda;
    CASE ir(2 DOWNTO 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="00000000" 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 "ZZZZZZZZ"; -- Buffer de Salida
dat_in<=dat;

END PROCESS latch_in;

END descripcion;
```

Ejemplo 3.13 *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 3.3 se muestra el chip del cual se quiere hacer la descripción junto con sus entradas y salidas.

Ejemplo 3.14 *Describir con VHDL el circuito que controla una máquina de café. Las entradas y salidas del cicuito se muestran en la figura 3.4. Las entradas son:*

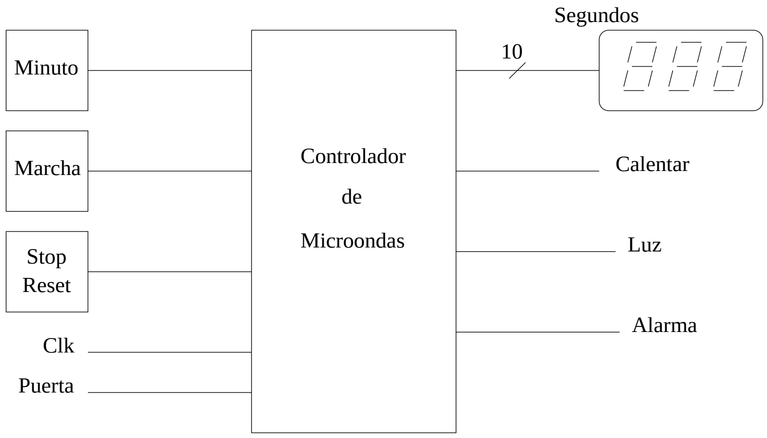


Figura 3.3: *Figura del ejercicio del mircroondas*

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 azucar 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.*

azucar *A uno le indicará a la parte que hace el café que le ponga azucar.*

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.

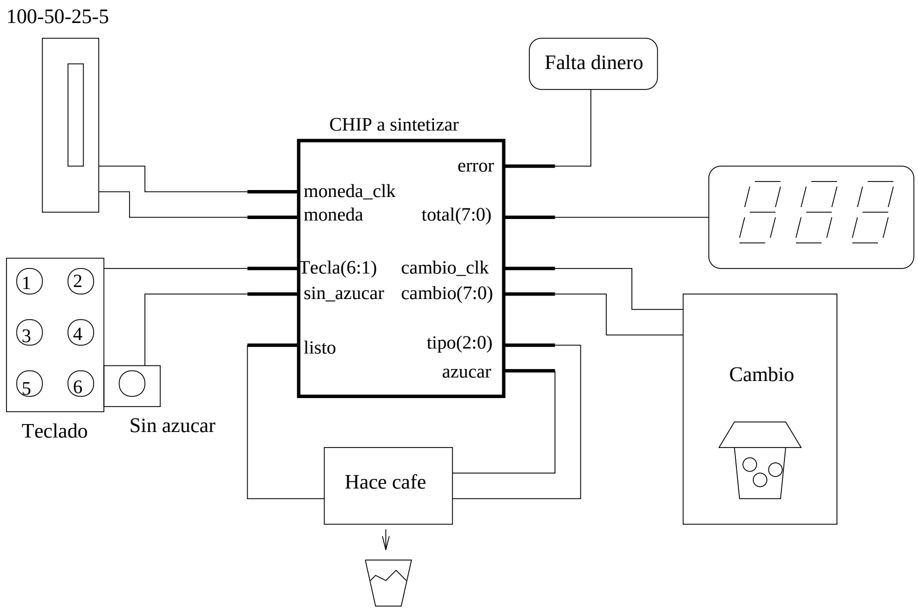


Figura 3.4: *Figura del ejercicio de la máquina de café*

Parte II

Simulación

Capítulo 4

Simulación

4.1 Conceptos básicos de simulación

4.2 Simulación digital

4.3 Simulación eléctrica

4.3.1 Introducción a Spice y sintaxis

4.3.2 Análisis eléctrico con Spice

4.3.3 Análisis del transistor en conmutación

Parte III

Realización

Capítulo 5

Circuitos integrados. VLSI

5.1 Introducción

5.2 Procesos de fabricación

5.3 Librería de celdas

5.4 Síntesis automática de circuitos integrados

Capítulo 6

Diseño de circuitos impresos (PCBs)

6.1 Diversas tecnologías de circuitos impresos. SMT y MCM

6.2 Encapsulados

6.3 Reglas de diseño

6.4 Emplazado óptimo. Estrategias

6.5 Trazado de pistas. Algoritmos de trazado de pistas

6.6 Estrategias de rutado de pistas

6.7 Fabricación y costes

6.8 Interferencias electromagnéticas

Parte IV

Bibliografía

Bibliografía

[1] Barry Hawkes. *CADCAM.* Paraninfo, 1989.

[2] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1991.

[3] Peter J. Ashenden. *The VHDL Cookbook.* 1990.

[4] Averill M. Law and W. David Kelton. *Simulation Modelling & Analisis.*

[5] Paul W. Tuinenga. *A guide to circuit simulation & analysis using Pspice.* Prentice Hall, 1992.

[6] Paolo Antognetti and Giuseppe Massobrio. *Semiconductor Device Modelling with Spice.* McGraw-Hill, 1987.

[7] Meares and Hymowitz. *Simulating with Spice.* Intusoft, 1988.

[8] Karl Heinz Müller. *A Spice Cookbook.* Intusoft, 1991.

[9] S.M. Sze. *Semiconductor Devices, Physics and Technology.* Wiley, 1985.

[10] John P. Uyemura. *Fundamentals of MOS Digital Integrated Circuits.* Addison Wesley, 1988.

[11] Eugene D. Fabricius. *Introduction to VLSI design.* McGraw-Hill, 1990.

[12] Clyde F. Coombs Jr. *Printed Circuits Handbook.* McGraw-Hill, 1988.

[13] Howard W. Johnson and Martin Graham. *High-Speed Digital Design.* Prentice Hall, 1993.

[14] Ven Solberg. *Design Guidelines for Surface Mount Technology.* TAB Books Inc., 1990.

[15] Frank Classon. *Surface Mount Technology for Concurrent Engineering and Manufacturing.* McGraw-Hill, 1993.

Indice de Materias