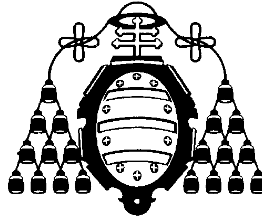


UNIVERSIDAD DE OVIEDO

DESCRIPCIÓN DE CIRCUITOS DIGITALES MEDIANTE VHDL

Francisco Javier Ferrero Martín

21 de Mayo de 1999

INDICE

CAPÍTULO I: INTRODUCCIÓN

1.1 Evolución del diseño electrónico...1

1.2 Lenguajes de descripción de hardware...4

1.2.1 Reseña histórica de los HDLs...4

1.2.2 Modelado con HDLs: Niveles de abstracción y estilos descriptivos...........5

1.2.3 Aportaciones de los HDLs al proceso de diseño...6

1.3 Metodología típica de diseño...8

1.3.1 Flujo de diseño descendente...8

CAPÍTULO II: EL LENGUAJE VHDL

2.1 Características principales...12

2.1.1 Modelo de estructura...12

2.1.2 Modelo de concurrencia...14

2.1.3 Modelo de tiempo...15

2.2 Unidades básicas de diseño...18

2.2.1 Declaración de entidad...18

2.2.2 Arquitectura...19

2.2.3 Configuración...23

2.2.4 Paquetes...24

2.2.5 Bibliotecas...25

2.3 Objetos, tipos de datos y operandos...26

2.3.1 Objetos del VHDL...26

2.3.1.1 Constantes...27

2.3.1.2 Variables.....27
2.3.1.3 Señales.....28
2.3.1.4 Ficheros.....29
2.3.2 Tipos de datos.....29
2.3.2.1 Declaración de tipos de datos.....30
2.3.2.1.1 Tipos de datos escalares.....31
2.3.2.1.2 Tipos enteros y tipos reales.....31
2.3.2.1.3 Tipos físicos.....32
2.3.2.1.4 Tipos enumerados.....33
2.3.2.1.5 Expresiones y operadores.....34

2.4 Sentencias secuenciales.....35
2.4.1 La sentencia *wait*.....35
2.4.2 Asignación a señal.....37
2.4.3 Asignación a variable.....39
2.4.4 La sentencia *if*.....39
2.4.5 La sentencia *case*.....42
2.4.6 La sentencia *loop*.....43
2.4.7 La sentencia *exit*.....45
2.4.8 La sentencia *next*.....45
2.4.9 Llamada secuencial a subprogramas.....46

2.5 Sentencias Concurrentes.....47
2.5.1 La sentencia *process*.....48
2.5.2 Asignación a señal concurrente.....49
2.5.3 Asignación concurrente condicional.....49
2.5.4 Asignación concurrente con selección.....50
2.5.5 Sentencias estructurales.....51
2.5.5.1 Componentes.....51
2.5.5.2 Configuración de un diseño.....54

2.6 Subprogramas.....56
2.6.1 Funciones.....56
2.6.2 Procedimientos.....57

CAPÍTULO III:
DESCRIPCIÓN DE CIRCUITOS DIGITALES

3.1 Descripción del sistema..59

3.2 Lógica combinacional……………………………………………………..60

3.3 Lógica secuencial………………………………………………………..64

3.3.1 Descripción de *"latches*"………………………………………………...64

3.3.2 La señal de reloj…………………………………………………………64

3.3.3 Registros……………………………………………………..…......65

3.3.4 Contadores……………………………………………………………..67

3.3.5 Descripción de MSMs………………………………………….......71

3.4 Recomendaciones generales...74

3.4.1 Recomendaciones para síntesis...74

3.4.1.1 Descripción *"hardware"*...74

3.4.1.2 Limpieza del código..75

IV. BIBLIOGRAFÍA..77

CAPÍTULO I: INTRODUCCIÓN

El objetivo de esta introducción es mostrar que son los lenguajes de descripción de hardware (HDL, Hardware Description Languages), cuando y como aparecen dentro de la evolución del diseño electrónico y cuales son sus principales aportaciones de tales lenguajes, así como su incidencia en el propio proceso de diseño.

Además de presentar brevemente la evolución del desarrollo electrónico, en este capítulo se trata de ubicar el VHDL (VHSIC Hardware Description Language; donde VHSIC: Very High Speed Integrated Circuits) en el contexto del diseño electrónico, su origen, evolución (impacto sobre las técnicas EDA, Electronic Design Automation) y ámbitos de aplicación (modelado, simulación, síntesis).

1.1 EVOLUCIÓN DEL DISEÑO ELECTRÓNICO

El desarrollo electrónico de los últimos tiempos se ha visto fuertemente dominado y conducido por la impresionante evolución de la microelectrónica desde su nacimiento en 1959-60. Durante los años setenta, junto con la revolución que suponen las memorias RAM y procesadores en forma de chip monolítico, se preparan las condiciones para el gran salto que el diseño microelectrónico dará en los años ochenta.

El desarrollo de nuevas tecnologías, alternativas de fabricación y diseño de circuitos integrados, junto con la evolución de las metodologías y herramientas de diseño asistido por ordenador, han sido las innovaciones más importantes de la década de los ochenta. Estas se han reflejado tanto en el continuo incremento de la complejidad y prestaciones de los chips, y por ende de los sistemas electrónicos, como en la gran difusión de las técnicas, metodología, y

herramientas de diseño de circuitos integrados que, junto con las nuevas alternativas de fabricación, han ampliado el rango de posibilidades de los ingenieros de aplicación, permitiéndoles diseñar chips específicos (*Application Specific Integrated Circuits, ASICs*) para los productos que desarrollan.

Todos estos factores han contribuido directamente a la evolución de los recursos de cálculo (procesadores, estaciones de trabajo, etc.) quienes a su vez tienen una incidencia decisiva en el desarrollo de nuevas herramientas y entornos integrados de diseño de diseño de sistemas electrónicos. Con el desarrollo y uso de tales herramientas para crear nuevos componentes se cierra el ciclo del soporte mutuo entre ambas tecnologías: microelectrónica e informática.

La Figura 1-1 esquematiza como a partir de las especificaciones de un circuito y a fin de poder proceder a su fabricación, el proceso de diseño de CIs atraviesa tres etapas o dominios diferentes: **funcional o comportamental** (algoritmos y funciones que indican la relación o comportamiento entrada/salida, E/S, pero no su implementación), **arquitectual** (componentes funcionales interconectados que definen la arquitectura/estructura de la implementación sin detallar la realización física final) y **físico** (se detalla una materialización concreta a nivel eléctrico y geométrico para una determinada tecnología). Obsérvese que la complejidad del diseño en términos de volumen de información a manejar aumenta drásticamente a medida que avanzamos por estas fases, al incrementar la precisión y disminuir el nivel de abstracción.

A finales de los ochenta se consolidan los niveles estructural y físico, a la vez que los desarrollos sobre dispositivos programables complejos (FPGAs, CPLDs, FPLDs) empiezan a crecer en importancia frente a los ASICscustom. Con el nacimiento del VHDL (1987) se empiezan a desarrollar métodos y herramientas para abordar el diseño a nivel funcional o comportamental, cuya implantación definitiva se está produciendo durante esta década de los noventa.

En cuanto tecnologías, la CMOS sigue siendo la más utilizada (75 por 100 del mercado) incluso creciendo. Las bipolares/ECL se mantienen alrededor del 15 por 100. Las BICMOS crecen hasta un 5 por 100. Las NMOS TTL decrecen considerablemente, junto con las nuevas tecnologías del tipo AsGa y similares se reparten el 5 por 100 restante.

Al nivel de ASIs los desarrollos *full* y *semi-custom* han perdido relevancia frente a las considerables prestaciones y complejidades que los dispositivos programables son capaces de asumir, de forma que solo producciones elevadas o requisitos muy específicos (velocidad, área, consumo, confidencialidad) hacen necesarios los *ASIC-custom*.

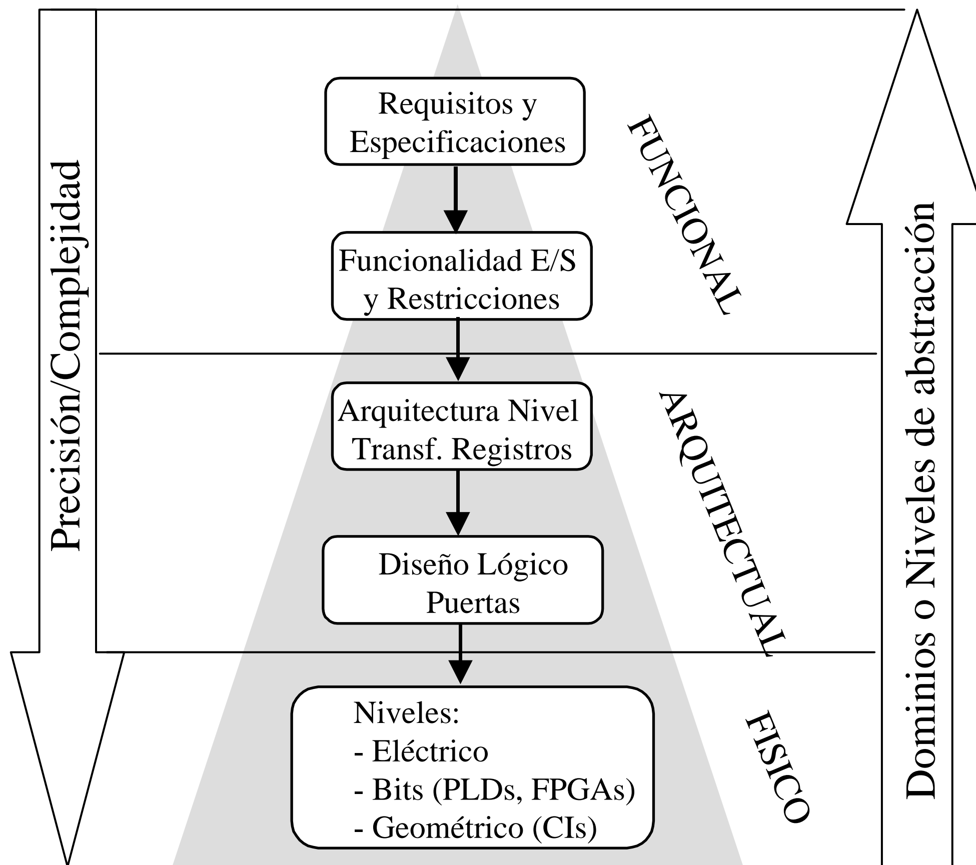


Figura 1-1. Pirámide de complejidad y niveles de abstracción de las distintas fases del diseño electrónico

Con el avance que están siguiendo las tecnologías submicrónicas (se están desarrollando procesos CMOS con transistores de longitud de canal inferior a 0,3 μm y capaces de albergar varios millones de dispositivos), una vez más el cuello de botella del desarrollo microelectrónico va a estar en el diseño más que en la tecnología.

En cuanto a metodologías de diseño, los años noventa se han caracterizado por una implantación progresiva, en fase de consolidación, de los lenguajes de descripción de hardware (Verilog y VHDL) que, junto con las herramientas de simulación y síntesis, promueven el uso de las llamadas metodologías de diseño descendentes *(to-down)*. Se trata de concentrar el esfuerzo en la concepción a nivel funcional-arquitectual, facilitando la

evaluación de soluciones alternativas antes de abordar el diseño detallado y la implementación física, dando lugar así al también llamado diseño de alto nivel.

1.2 LOS LENGUAJES DE DESCRIPCIÓN DE HARDWARE

Los lenguajes de descripción de hardware (HDL) son lenguajes de alto nivel, similares a los de programación (C, PASCAL, ADA, etc.), con una sintaxis y semántica definidas para facilitar el modelado y descripción de circuitos electrónicos, desde las celdas base de un ASIC hasta sistemas completos, pudiéndose realizar estas descripciones a distintos niveles de abstracción y estilos de modelado.

Los HDLs nacen para modelar el comportamiento de un componente de cara a su simulación, aunque también se utilizan para describir el diseño de un circuito para su implentación a través de etapas de síntesis válidas vía simulación

1.2.1 Reseña histórica de los HDLs

Durante los años ochenta, tras detectarse la necesidad de un lenguaje para dar soporte a las distintas etapas y niveles de abstracción del proceso de diseño, se desarrollan y consolidan dos de ellos: *Verilog* y VHDL.

El **VHDL** (*VHSIC* ***H****ardware* ***D****escription* ***L****anguage*) aparece como un proyecto del Departamento de Defensa de EEUU (1982), con el fin de disponer de una herramienta estándar e independiente para la especificación y documentación de los sistemas electrónicos a lo largo de todo su ciclo de vida. Tras las primeras versiones del lenguaje, el IEEE lo adopta y desarrolla como el HDL estándar (1ª versión en 1987 y 2ª en 1993).

El **Verilog** nace como un lenguaje de modelado ligado a un entorno de simulación de la firma Gateway, pasando posteriormente a ser el simulador digital de *Cadence Design Systems Inc.,* llegando a convertirse en un estándar "de facto" a nivel industrial. Al aparecer el VHDL como estándar IEEE, Verilog se encontró con un fuerte competidor, y en 1990 Cadence decide ofrecerlo como lenguaje de dominio público e inicia las gestiones para su estandarización formal, que se logra en 1995.

El desarrollo, difusión y estandarización de los lenguajes Verilog y VHDL, aunque sólo sean herramientas básicas para la descripción de circuitos y sistemas electrónicos fue, y sigue siendo, un hecho determinante en el desarrollo de las nuevas metodologías y herramientas de diseño electrónico.

1.2.2 Modelado con HDLs: niveles de abstracción y estilos descriptivos

Una de las características más importantes de estos lenguajes es su capacidad para abordar descripciones a distintos **niveles de abstracción** (funcional o comportamental, rquitectual o transferencia de registros, lógico o de puertas) y **estilos de modelado** (algorítmico, flujo de datos, estructural)

Los niveles de abstracción hacen referencia al grado de detalle en que se encuentra una determinada descripción HDL respecto a la implementación física de la misma. Puesto que el nivel más bajo que trataremos directamente con los HDLs serán listas de componentes y conexiones a nivel de puertas podemos considerar que desde los HDL no abordamos el nivel físico mostrado en la Figura 1-1, que quedaría como el nivel más bajo de abstracción (donde se fijan todos los detalles para la implementación real del circuito), por encima del cual se sitúa el nivel lógico o de puertas. Así pues, desde la perspectiva de la simulación y síntesis con HDLs, los niveles de abstracción pueden quedar reajustados a los tres siguientes:

- **Funcional o comportamental,** donde se indica el comportamiento del circuito o sistema como una relación funcional entre las entradas y las salidas, pero sin hacer ninguna referencia a su implementación.

- **Arquitectual o de transferencia de registros (RTL).** A este nivel se desarrolla una partición en bloques funcionales y se planifican en el tiempo (ciclos de reloj) las acciones a realizar. Todavía no se conocen los detalles de la realización final de cada bloque.

- **Lógico o de puertas**. Los componentes del circuito están expresados en términos de ecuaciones lógicas o puertas y elementos de una biblioteca, pudiendo ser esta genérica (independiente de la tecnología) o específica de una tecnología.

Otro aspecto o criterio de caracterización de los modelos HDL es el estilo de descripción que, de forma simplificada, podemos distinguir entre los tres siguientes:

- **Algorítmico**: hace referencia a descripciones similares a los programas de software, que deben reflejar la funcionalidad del módulo, componente o circuito, en forma de uno o más procesos concurrentes que contienen descripciones secuenciales del algoritmo correspondiente.

- **Flujo de datos**: descripciones basadas en ecuaciones o expresiones que reflejan el flujo de información y las dependencias entre datos y operaciones.

- **Estructural:** en este estilo se reflejan directamente componentes por referencia y conexiones entre ellos a través de sus puertos de entrada/salida.

Como se puede intuir, normalmente hay una cierta correlación entre los niveles de abstracción y estilos descriptivos. Así pues, las descripciones algorítmicas se usan más a nivel comportamental, el flujo de datos se relaciona más con el nivel RTL y al nivel de puertas el estilo descriptivo siempre es de tipo estructural.

Uno de los valores añadidos de estos lenguajes, y en especial del VHDL, es su capacidad para hacer convivir de forma natural descripciones mixtas-multinivel: distintos estilos y niveles de abstracción para las distintas partes de un diseño. Una cierta descripción estructural puede tener componentes desarrollados a nivel de puertas, otros a nivel RT en forma de flujo de datos y algunos todavía en forma de algoritmos a nivel funcional. Esta es una situación normal que va evolucionando a lo largo de todo el proceso de diseño hasta que, tras los correspondientes procesos de síntesis manual o automática, todos los componentes del circuito a implementar están detallados mediante una descripción estructural a nivel de puertas

1.2.3 Aportaciones de los HDLs al proceso de diseño

Tanto el Verilog como el VHDL nacen con una sintaxis y una semántica definidas y dirigidas hacia el **modelado** para la **simulación** de hardware. Sin embargo, rápidamente se abordó su uso como soporte para todas las fases del proceso de diseño, y muy especialmente para las

etapas de **síntesis** y **verificación**. Pasemos, pues, a comentar las ventajas más relevantes que pueden suponer el uso de estos lenguajes:

- Estos HDLs son interpretables tanto por las personas como por los ordenadores, pudiendo proporcionar soporte tanto a las áreas estrictas de diseño (modelado, simulación, síntesis, verificación) como a las de comunicación e intercambio de modelos entre los distintos equipos de trabajo.
- Son lenguajes de disponibilidad pública, no sometidos a ninguna firma ni patente. Están definidos, documentados y mantenidos por el IEEE, quien garantiza su estabilidad y soporte.
- Soportar descripciones con múltiples niveles de abstracción, pudiéndose mezclar desde módulos descritos a nivel comportamental hasta módulos descritos a nivel de puertas. Con el mismo lenguaje se describe el circuito y el entorno de verificación (banco de pruebas) para su simulación/validación
- La utilización de un lenguaje único a lo largo de todo el proceso de diseño simplifica la gestión del mismo (reducción de herramientas y formatos).
- Proporcionan independencia de la metodología, de herramientas y de tecnología.
- En el caso de las herramientas CAD, los HDLs son lenguajes estándar que definen una sintaxis y una semántica de modelado para simulación. Así pues cualquier herramienta que cumpla con el estándar ha de aceptar y ejecutar (simular) de la misma forma cualquier modelo. Esta potabilidad no es tan aplicable a la síntesis ni a la verificación formal, ya que la semántica del VHDL y del Verilog en estas áreas no está definida desde el IEEE, y son los proveedores de CAD quienes hacen su propia interpretación, dando lugar a pequeñas divergencias.
- Los HDLs, tanto por su definición y diseño como por los niveles de abstracción donde habitualmente se usan, son o pueden ser, totalmente independientes de la tecnología final de implementación de los circuitos. La descripción VHDL de un circuito puede ser totalmente indiferente a la tecnología de fabricación, pero si se quiere también puede incluir información específica de la implementación final (retrasos, consumo, etc.)
- La reutilización del código HDL desarrollado en los proyectos anteriores es un hecho posible gracias a algunas de las características enunciadas anteriormente como ser lenguajes estándar, estables e independientes y su independencia

metodológica, tecnológica y del CAD. Una descripción VHDL de un diseño, inicialmente desarrollado para una tecnología (CMOS, BICMOS, etc.) e implementación (ASIC, FPGA, etc.), puede fácilmente ser reutilizada en diseños o materializaciones posteriores donde la tecnología, la alternativa de implementación y/o el CAD pueden ser distintas

Como vemos, estos lenguajes pueden aportar ventajas importantes pero no debemos ignorar que también están sujetos a algunas limitaciones:

- Al ser lenguajes definidos por consenso en el seno de una comisión (especialmente el VHDL) tienden a ser complejos para contemplar la diversidad de opiniones. Así mismo la evolución del lenguaje mediante revisiones vía comisión es lenta (5-6 años para el VHDL) y con importantes cambios en cada nueva versión
- La falta de una semántica formal para síntesis dificulta la portabilidad de los diseños entre los distintos entornos de síntesis (no todas las herramientas interpretan de las misma forma las mismas construcciones del lenguaje).
- El VDHL, por sus características sintáctico-semánticas, está mejor dotado para las descripciones a nivel funcional/algorítmico, mientras que sus prestaciones se ven más limitadas al trabajar a nivel de puertas.

1.3 METODOLOGÍA TÍPICA DE DISEÑO

Metodolgías, flujos y herramientas CAD de diseño electrónico son conceptos muy interrelacionados y no siempre fáciles de distinguir. La metodología es un concepto más abstracto que hace referencia a procesos de diseño genéricos que relacionan entre sí los distintos niveles de complejidad y abstracción por los que atraviesa el diseño de un circuito o sistema electrónico.

Por su parte los flujos de diseño son una personalización concreta de una cierta metodología, para un tipo de circuitos o área de aplicación específico, y contando con el soporte de unas determinadas herramientas de CAD. Distintos flujos de diseño pueden responder a una misma metodología y un mismo flujo se puede implantar con distintas herramientas de CAD.

1.3.1 Flujo de diseño descendente (top-down)

Las características y ventajas de los HDL en general, y en especial de VHDL, están facilitando y potenciando el desarrollo e implementación de las **metodologías de diseño descendente** *(top-down)* que se basan en un uso intensivo de tales lenguajes (modelo/descripción del circuito y de los bancos de pruebas), convenientemente soportados durante todo el proceso de diseño por herramientas de simulación, síntesis y procesos de análisis-verificación.

La Figura 1-2 muestra de forma esquemática y genérica esta metodología que se basa en un proceso de construcción o diseño descendente.

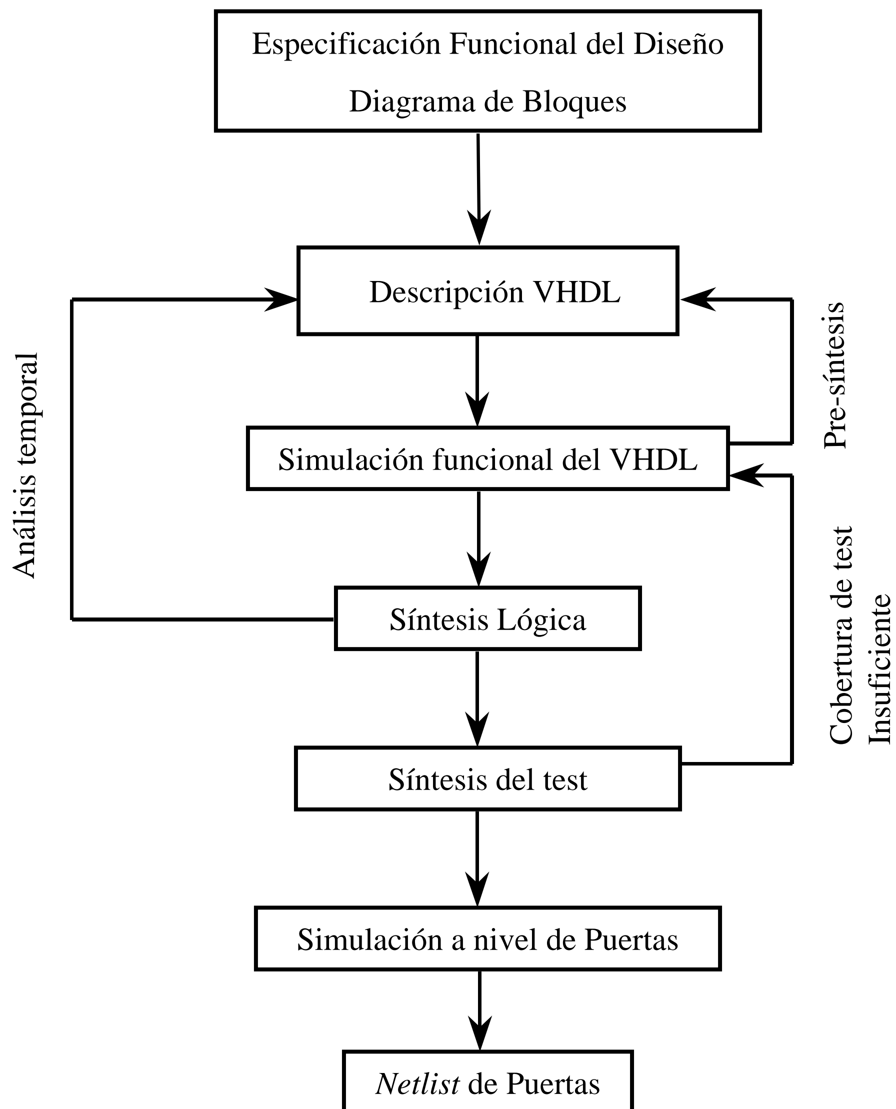


Figura 1-2. Esquema genérico del flujo de diseño descendente

A partir de la idea o concepto que se desea implementar en forma de sistema o circuito electrónico se ha de proceder a una primera fase de **definición de especificaciones**. Hasta ahora esta ha sido la etapa menos formalizada y a la que no se presta la atención que requiere, a pesar de que los resultados de la misma guían o dirigen muchas de las decisiones posteriores durante el proceso de diseño. Con la llegada de los HDLs se estableció la tecnología de base para dar el soporte necesario a la actividad de concepción al nivel funcional.

Una vez fijadas las especificaciones, comienza la **descripción HDL** de las mismas mediante un proceso de refinamiento gradual de la descripción del circuito hasta alcanzar un modelo arquitectual-RT que sea sintetizable mediante procesos automáticos guiados por el diseñador.

A continuación viene la etapa de **síntesis lógica** que tiene por objeto la obtención de un esquema o lista de componentes y sus interconexiones basado en una determinada biblioteca de celdas y módulos. El diseño resultante debe realizar la funcionalidad especificada, respetar las prestaciones requeridas (área, velocidad, consumo) y garantizar la testeabilidad final del circuito.

Estas herramientas de síntesis RT-lógica automáticas (Tabla 1-1) se implementan mediante distintos procesos de síntesis encadenados de forma transparente para el usuario. Estos procesos son:

1. Síntesis RT que determina los elementos de memoria y el conjunto de ecuaciones lógicas u operadores necesarios, en base a fases de partición, distribución *(scheduling)* y asignación *(allocation)* de recursos, bajo las restricciones impuestas por el diseñador.

2. Síntesis lógica que se encarga de optimizar ecuaciones lógicas para minimizar el hardware necesario al nivel de puertas y registros utilizando algoritmos de reducción y asignación de estados, minimización lógica, eliminación de redundancias, etc.

3. Mapeo tecnológico que es una fase, muchas veces indistinguible en la síntesis lógica, en la que las puertas y registros se mapean de forma optimizada sobre los elementos

disponibles en la biblioteca de celdas y módulos correspondientes a la tecnología escogida para la implementación del diseño.

A lo largo de estas fases se ha de incorporar al diseño las estrategias y el hardware necesario para asegurar la posterior testeabilidad del circuito. Estos son procesos semiautomáticos, que además de definir la estrategia de test y añadir las estructuras necesarias (*scan-paths*, modos *ad-hoc, boundary-scan*) se complementan con procesos de generación y composición de vectores de test que ayudan a obtener un conjunto reducido y óptimo de tales vectores con una buena cobertura de fallos (>96%) del circuito.

Después de esta síntesis RT-lógica se obtienen los dos elementos básicos para poder abordar las fases o etapas de diseño físico: la lista de componentes y conexiones, las restricciones a cumplir y el conjunto de vectores de test. Con estas descripciones ya podemos iniciar los procesos de ubicación y conexionado para generar la topografía y descripciones necesarias para la implementación física del circuito.

Compañía	Nombre del Producto	Nivel de Síntesis	Tipo de Circuitos
Altera	Max+Plus (AHDL, VHDL)	RTL, MEF	CPLD
Cadence	Synergy (VHDL, Verilog)	RTL, MEF Test	ASIC FPGA
Cypress	Warp II, III (VHDL)	RTL, MEF	PLD FPGA
Mentor Graphics	Autologic (VHDL)	RTL, MEF Test, Datapath	ASIC FPGA
Synopsys	Design Compiler (VHDL, Verilog)	RTL, MEF Test	ASIC FPGA
Viewlogic	ViewSynthesis (VHDL, Verilog)	RTL, MEF Test	ASIC FPGA

Tabla 1-1. Principales herramientas de síntesis lógica

CAPÍTULO II:
EL LENGUAJE VHDL

El objetivo de este capítulo es realizar una presentación de la sintaxis del lenguaje VHDL. No se pretende cubrir de forma exhaustiva todas las posibilidades del lenguaje, sino que se intenta cubrir los conceptos del lenguaje, con objeto de poder entender la materia contenida en el capítulo de descripción de circuitos digitales. Los contenidos del capítulo se enfocan desde la perspectiva del VHDL-87.

El capitulo contiene numerosos ejemplos, destinados a clarificar cada una de las características del lenguaje.

2.1 CARACTERÍSTICA PRINCIPALES DE VHDL

Tres son las características principales que incorpora VHDL enfocadas a facilitar o permitir la descripción de hardware: un **modelo de estructura**, un **modelo de concurrencia** y un **modelo de tiempo**. Estas características junto con la capacidad de describir funcionalidad que le confieren las propiedades descritas en el capítulo anterior, hacen de VHDL un lenguaje flexible y potente, que se adapta perfectamente a la descripción de sistemas electrónicos a cualquier nivel de abstracción.

2.1.1. Modelo de estructura

De forma natural cualquier sistema electrónico puede dividirse en subsistemas más pequeños. Por ello VHDL incorpora el concepto de estructura. Esta característica nos permite realizar el modelo de un sistema digital cualquiera a partir de la referencia a las distintas partes que lo forman y especifican la conexión entre estas. Cada una de las partes, a su vez, pueden estar

modelas de forma estructural a partir de sus componentes, o bien estar descritas de forma funcional, usando los recursos de descripción algorítmica del lenguaje.

Al describir cualquier dispositivo en VHDL (desde una puerta hasta un sistema completo) el diseñador debe definir dos elementos principales: la interfaz del dispositivo con el exterior (la entidad o ***entity***) y la descripción de la funcionalidad que realiza el dispositivo (la arquitectura o ***architecture***). La interfaz de un dispositivo tiene por objeto definir que señales del dispositivo son visibles o accesibles desde el exterior, lo que se llaman los puertos (***ports***) del dispositivo. En la arquitectura se definirá la funcionalidad que implementa dicho dispositivo, o sea, que transformaciones se realizarán sobre los datos que entren en los puertos de entrada, para producir nuevos valores sobre los puertos de salida.

Para poder utilizar elementos ya definidos en VHDL en descripciones estructurales de un nuevo diseño, VHDL incorpora el concepto de componente *(**component**)* y de referencia a un componente. Cualquier elemento modelado con VHDL puede ser usado como un componente de otro diseño. Para ello solamente es necesario hacer referencia al elemento a utilizar y conectar los puertos de su interfaz a los puntos necesarios para realizar el nuevo diseño. La Figura 2-1 ilustra esta idea, el sistema bajo desarrollo se forma a partir de dos subsistemas que se habrán definido con anterioridad. El diseñador solo debe preocuparse de las entradas y las salidas de los subsistemas (su interfaz) y de la forma adecuada en que debe conectarlas para formar el nuevo sistema, pero no es necesario conocer cómo está descrito cada uno de los subsistemas.

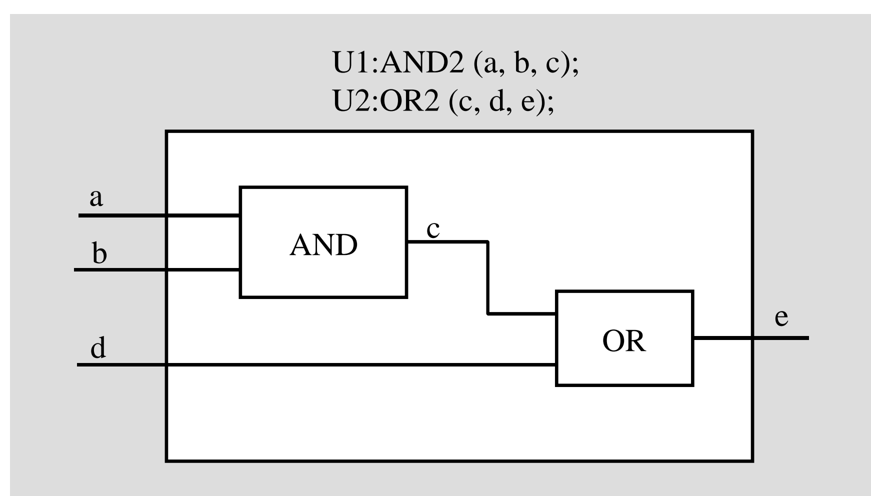


Figura 2-1. Modelo de estructura en VHDL

2.1.2 Modelo de concurrencia

El hardware es por definición concurrente, en ultima instancia cualquier dispositivo digital está formado de un mar de puertas lógicas, todas ellas funcionando en paralelo. El elemento básico que ofrece VHDL para modelar paralelismo es el proceso *(**process**)*.

Un proceso puede entenderse como un programa, se compone de sentencias, puede llamar a subprogramas, puede definir datos locales, etc. En general, un proceso describe un comportamiento y el código que contiene se ejecuta de forma secuencial. Pero todos los procesos contenidos en una descripción VHDL se ejecutan de forma paralela. Desde este punto de vista un modelo VHDL puede entenderse como un mar de programas secuenciales ejecutándose de forma paralela. De hecho cualquier descripción VHDL es transformada en un conjunto de procesos concurrentes equivalentes, y este mar de procesos concurrentes es la información de entrada del simulador.

Estos procesos que se ejecutan concurrentemente deben poder comunicarse (sincronizarse) entre ellos. El elemento necesario para comunicar dos procesos es la señal *(**signal**)*. Cada proceso tiene un conjunto de señales a las que es sensible. Ser sensible a una señal significa que en cuanto se produzca un cambio en el valor de dicha señal (un evento en la señal), el proceso se ejecutará hasta que encuentre una sentencia de suspensión del proceso *(**wait**)*. Al llegar a esta sentencia, el proceso quedará suspendido, esta suspensión será por un período determinado de tiempo, o bien hasta que se produzca un nuevo evento en alguna de las señales a las que sea sensible dicho proceso. Aparte de poder suspender la ejecución de un proceso (sentencia wait), este es un bucle infinito, o sea, al llegar a su final vuelve a ejecutarse desde el principio.

Para ilustrar mejor este concepto, la Figura 2-2 define los procesos equivalentes a una puerta *and* y una puerta *or* de dos entradas cada una. Notar que en este ejemplo se utiliza la seña *c* para sincronizar los dos procesos, siempre que se produzca un evento en la señal *c*, se ejecutará el proceso *OR2*.

Por supuesto, y dado el paralelismo en la ejecución de los procesos, si en un momento de la simulación se producen eventos sobre las señales de la lista de sensibilidad de ambos procesos (por ejemplo, en a y en d), los dos se ejecutan en ese tiempo de simulación.

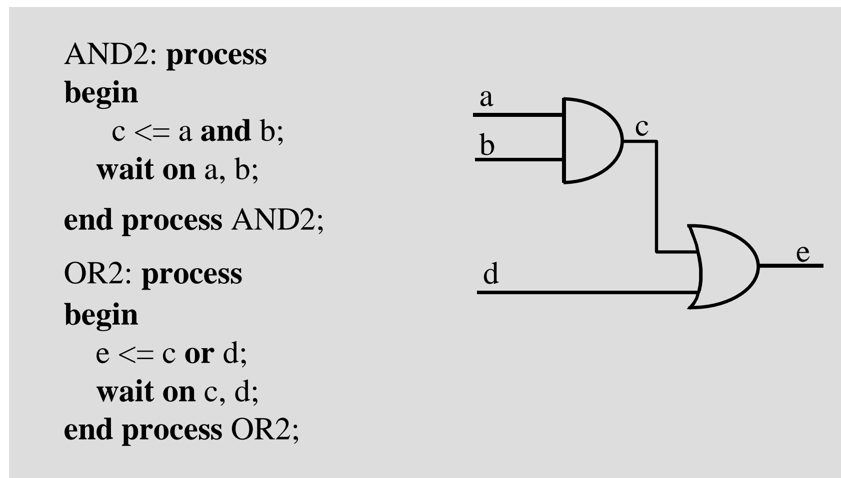


Figura 2-2. Modelo de concurrencia en VHDL

Sobre las señales sólo diremos de momento que son objetos que pueden ir variando su valor a lo largo de la simulación (en este aspecto son parecidas a las variables). Su característica principal es que tienen asociada una o más colas de eventos *(drivers)* que define su comportamiento a lo largo del tiempo. La cola de eventos está formada por conjuntos de pares tiempo/valor, y en las asignaciones a señal es esta cola de eventos la que recibe los valores asignados.

2.1.3 Modelo de tiempo

Una de las finalidades del modelado en VHDL del *hardware* es poder observar su comportamiento a lo largo del tiempo (simulación). El concepto de tiempo es fundamental para definir cómo se desarrolla la simulación de una descripción VHDL.

La simulación de un modelo VHDL es una simulación dirigida por eventos. Esto significa que el simulador mantiene unas listas de eventos (cambios en las señales internas del modelo y también de las entradas y salidas) que se han de producir a lo largo del tiempo de simulación. Como el comportamiento del modelo es estable mientras no se produzca un evento, la tarea del simulador consiste en avanzar el tiempo de simulación hasta el siguiente evento y calcular sus consecuencias sobre la lista de eventos futuros

La simulación VHDL abstrae el comportamiento real del *hardware*, implementando el mecanismo de estímulo respuesta (componentes funcionales reaccionan a la actividad en sus entradas produciendo cambios en sus salidas) implementando un ciclo de simulación de dos

etapas (Figura 2-3), basado en los procesos (elementos funcionales) y las señales (entradas y salidas de estos elementos funcionales; conexiones entre elementos).

En la primera etapa las señales actualizan su valor. Esta etapa finaliza cuando todas las señales que debían obtener un nuevo valor en el tiempo actual de simulación (tenían un evento programado en su cola de eventos) han sido actualizadas. En la segunda etapa, los procesos que se activan (aquellos que tengan en su lista de sensibilidad una señal en la que se haya producido un evento) se ejecutan hasta que se suspenden (con la ejecución de una sentencia ***wait***). Esta etapa finaliza cuando todos los procesos que se habían activado se hayan suspendido. Entonces el tiempo de simulación avanza hasta el siguiente instante de tiempo en el que haya un evento programado, y se repiten los dos pasos del ciclo de simulación. La simulación termina cuando no haya más eventos programados o cuando se llegue al tiempo de simulación especificado.

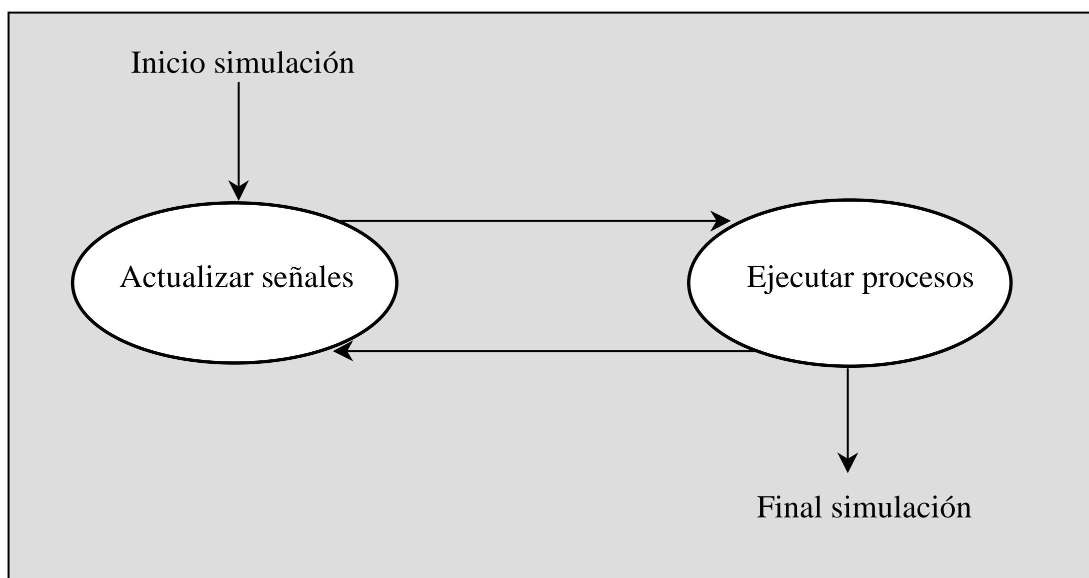


Figura 2-3. Ciclo de simulación VHDL

Es importante notar que el modelo de tiempo implementado por el ciclo de simulación VHDL implica que siempre hay un cierto retardo entre el momento en que un proceso coloca un nuevo valor en la cola de eventos de una señal (el proceso ejecuta la asignación sobre la señal) y el momento en que esta señal toma el valor programado en la cola de eventos. Incluso en el caso de que no se especifique un retardo concreto, se utilizará un retardo delta *(delta delay).* Un retardo delta no implica actualizar el tiempo de simulación, pero sí que implica ejecutar un nuevo ciclo de simulación.

El concepto de retardo delta es importante para entender otra diferencia importante entre variable y señal. Una variable actualiza su contenido en cuanto se ejecuta una asignación sobre ella. En cambio cuando se ejecuta una asignación sobre una señal, se proyecta un nuevo evento sobre su cola de eventos y solo cuando todos los procesos se hayan ejecutado y estén suspendidos, el valor de la señal se actualizará con el valor proyectado en su cola de eventos.

Este mecanismo de retardo delta se introduce para permitir la simulación de *hardware* (paralelo por naturaleza) usando máquinas secuenciales. Consideremos el código VHDL de la Figura 2-4, en el que aparecen dos elementos secuenciales conectados en forma de registro de desplazamiento.

El mecanismo de retardo delta permite que, independientemente del orden en que se ejecuten los dos procesos, el segundo (FF2) siempre reciba el valor correcto de Q1, ya que aunque se haya ejecutado con anterioridad el primer proceso (FF1), la asignación que éste realiza sobre Q1 aún no habrá tenido lugar (en todo caso se habrá proyectado el evento sobre la cola de eventos de Q1). De forma que al realizar la asignación de D1 sobre Q2 se colocará en la cola de eventos de Q2 el valor correcto de D1 (aún sin actualizar). Sólo en el momento en que ambos procesos se hayan suspendido, se actualizarán las señales con los valores que contengan sus colas de eventos.

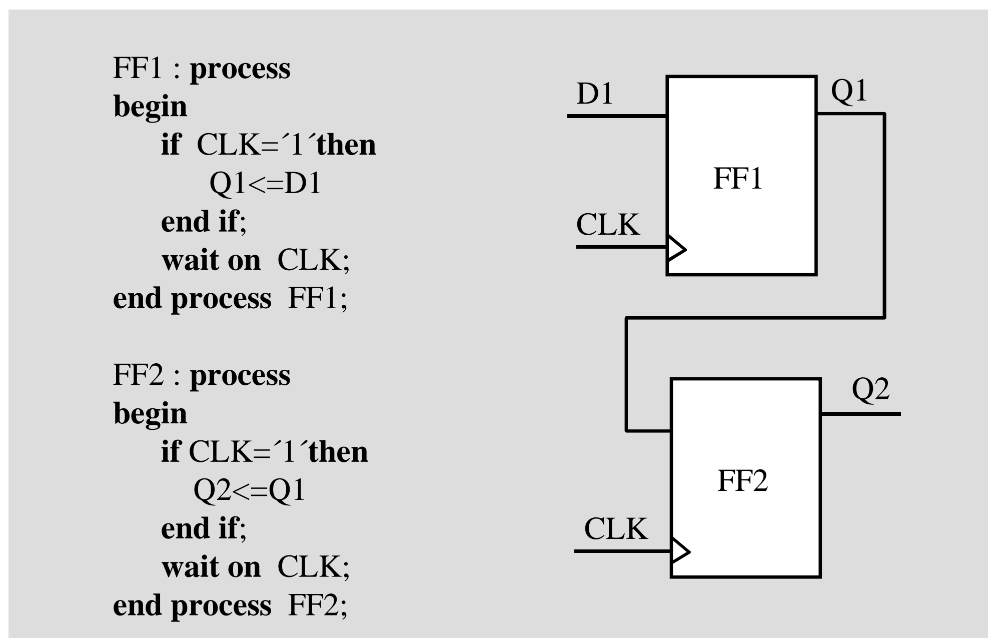


Figura 2-4. Determinismo en la simulación VHDL

2.2 UNIDADES BÁSICAS DE DISEÑO

Una unidad de diseño es una construcción VHDL que puede ser analizada independientemente. Existen cinco tipos diferentes de unidades de diseño: la declaración de entidad *(entity declaration)*, la arquitectura de una entidad *(architecture)*, la configuración *(configuration)*, la declaración de paquete *(package declaration)* y el cuerpo del paquete *(package body)*.

La declaración de entidad, la declaración de paquete y la configuración se llaman unidades primarias, mientras que la arquitectura de entidad y el cuerpo del paquete se consideran unidades secundarias porque dependen de una entidad primaria que debe ser analizada antes de poder ser analizadas ellas mismas.

Un dispositivo se representa en VHDL mediante una entidad, que consta de una declaración de entidad, donde se da una visión externa del dispositivo definiéndose la interfaz con su entorno, y una arquitectura, donde se define su funcionalidad. Para poder probar diferentes opciones a la hora de modelar un dispositivo, VHDL permite definir múltiples arquitecturas asociadas a una única entidad. La configuración es la construcción encargada de seleccionar de seleccionar la arquitectura específica que se va a utilizar para una entidad.

En VHDL cada objeto debe ser declarado antes de utilizarse. En general, las declaraciones se realizan en la unidades de diseño donde estos objetos son necesarios, por lo que no serán visibles en las demás unidades. Para declaraciones útiles para varias unidades de diseño, VHDL proporciona el paquete, que evita la multiplicidad de declaraciones comunes. Normalmente el paquete se divide en dos unidades de diseño VHDL: la declaración y el cuerpo del paquete.

2.2.1 Declaración de entidad

La declaración de una entidad sirve para definir la visión externa del dispositivo que dicha entidad representa, la interfaz con su entorno. VHDL separa esta visión externa de la implementación concreta del dispositivo para dar la posibilidad de que esta quede oculta. De este modo, después de haber analizado la declaración de una entidad y, por tanto, haberla

almacenado en una biblioteca, esta entidad podrá ser utilizada por otros diseños que solo requieran de dicha interfaz para usarla.

La sintaxis VHDL para declarar una entidad es la siguiente:

```
entity  identificador  is
            [genéricos]
            [puertos]
            [declaraciones]
end   [identificador]
```

el identificador es el nombre que va a recibir la entidad y servirá para poder referenciarla más tarde. Excepto la primera y la última línea de la declaración, todas las demás son opcionales. La declaración de una entidad que implemente un semisumador se muestra en la Figura 2-5.

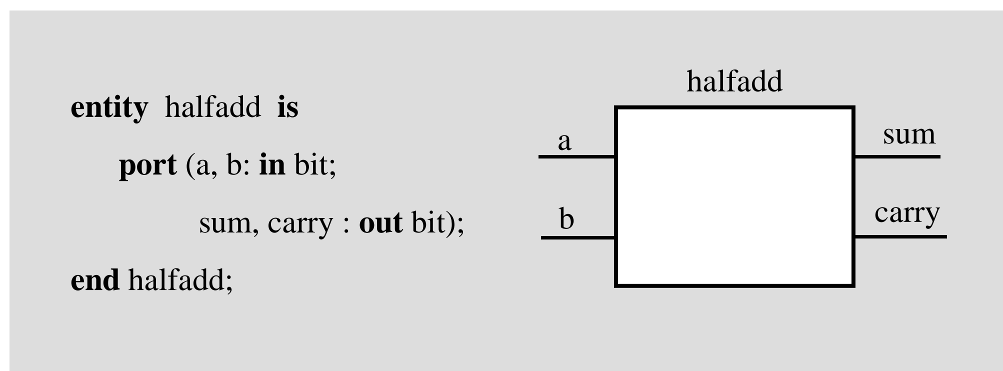


Figura 2-5. Diagrama de la interfaz del semisumador de 2 bits

Los puertos determinan la interfaz del dispositivo con el exterior y para comprender mejor que son se pueden comparar con las patillas de un circuito. Para cada puerto se tendrá que indicar el nombre, tipo y el modo. El nombre se utilizará para poder referenciarlo, el tipo definirá la clase de información que se transmitirá por el puerto mientras que el modo servirá para definir la dirección de la información, en el sentido que los puertos puedan ser de entrada, de salida o bidireccionales.

2.2.2 Arquitectura

Sirve para definir la funcionalidad de la entidad que representa. Describe un conjunto de operaciones sobre las entradas de la entidad que determinan el valor de las salidas en cada momento. Antes de poder ser analizadas es imprescindible haber analizado la declaración de la entidad, de modo que cuando ésta se modifique la arquitectura tendrá que ser reanalizada.

La sintaxis VHDL para definir la arquitectura de una entidad es la siguiente:

```
architecture identificador of identificador_entidad is
        [declaraciones]
begin
        [sentencias concurrentes]
end [identificador];
```

El identificador es el nombre que va a recibir la arquitectura y servirá para referenciarlo más tarde. Además debe indicarse el nombre de la entidad a la que pertenece. La sección de sentencias concurrentes describe propiamente la funcionalidad del dispositivo. Existen muchos tipos de sentencias concurrentes. Dependiendo del tipo de sentencias utilizadas se puede modelar una arquitectura siguiendo diferentes estilos:

- **Estilo algorítmico**

Define la funcionalidad del dispositivo mediante un algoritmo ejecutado secuencialmente, de forma muy parecida a como lo hace cualquier programa escrito en un lenguaje de programación común. Por tanto, no se hace ninguna referencia a la estructura que se seguirá para implementar el algoritmo en hardware.

La arquitectura de un multiplexor de dos bits utilizando un estilo de modelado algorítmico sería:

```
architecture Algoritmico of Mux21 is
begin
    process (a, b, ctrl)
    begin
        if (ctrl = ´0´) then
            z <= a;
        else
            z <= a;
        end if;
    end process;
end Algoritmico;
```

Más adelante en este capítulo se verán con detalle las sentencias utilizadas en el código. En este momento se puede decir que un proceso, definido mediante la palabra clave ***process***, es una sentencia concurrente, en el sentido que todos los procesos se ejecutan simultáneamente, que está formado por una o más instrucciones secuenciales. Por esta razón, una arquitectura con un solo proceso es equivalente a un algoritmo ejecutado secuencialmente.

- **Estilo flujo de datos**

Una descripción de estilo de flujo de datos refleja la funcionalidad de un dispositivo mediante un conjunto de ecuaciones ejecutadas concurrentemente, que determinan el flujo que van a seguir los datos entre módulos encargados de implementar las operaciones. En este estilo ya existe una correspondencia directa entre el código y su implementación hardware. Suele considerarse que este tipo de descripción es funcional y estructural al mismo tiempo, ya que define tanto el comportamiento de los módulos como su interconexión con los demás módulos.

El multiplexor de dos bits declarado anteriormente siguiendo un estilo de descripción de flujo de datos sería:

```
    architecture  FlujoDatos  of  Mux21  is
        signal  ctrl_n, n1, n2,  :  bit;
        begin
            Ctrl_n <= not  (ctrl) after 1ns;
             n1 <= ctrl_n  and a after 2 ns;
             n2 <= ctrl_n  and b after 2 ns;
             z <= (n1 or n2)  after 2 ns;
end  FlujoDatos;
```

- **Estilo estructural**

Una arquitectura definida utilizando un estilo estructural consiste en un conjunto de componentes interconectados mediante señales. Un ejemplo típico de descripción utilizando este estilo es la representación de un circuito como una lista de componentes interconectados *(netlist)* de una biblioteca de celdas estándar de una tecnología determinada. La descripción es puramente estructural en el sentido que no incluye ningún tipo de funcionalidad, ésta en todo caso está incluida en la definición de la arquitectura de los componentes que forman la descripción.

El multiplexor de dos bits declarado anteriormente podría describirse en estilo estructural como un conjunto de puertas interconectadas de la manera siguiente:

```
    architecture  Estructural  of  Mux21  is
        signal ctrl_n, n1, n2 : bit;
        component  INV
            port ( y : in bit;
                    z : out bit);
        end component;
        component AND2
            port (  x : in bit;
                     y : in bit;
                     z : out bit);
        end component;
        component  OR2
            port (  x : in bit;
```

```
                    y : in bit;
                    z : out bit);
        end component;
        begin
            U0 : INV  port map (ctrl, ctrl_n);
            U1 : AND2  port map (ctrl, ctrl_n);
            U2 : AND2 port map (ctrl, ctrl_n);
            U3 : OR2  port map (ctrl, ctrl_n);
    end Estructural;
```

Hay que dejar claro que aunque se hayan explicado diferentes estilos para describir una arquitectura VHDL y se hayan dado ejemplos de cada uno de ellos, todos estos estilos pueden mezclarse en la implementación de una sola arquitectura.

2.2.3 Configuración

La configuración es la construcción VHDL encargada de seleccionar la arquitectura que se quiere utilizar para una entidad concreta. VHDL permite definir más de una arquitectura por entidad para facilitar el estudio de varias posibilidades a la hora de implementarla. La sintaxis simplificada de una configuración es la siguiente:

```
configuration  identificador of  identificador_entidad is
   for identificador_arquitectura
   end for;
end  [identificador];
```

El identificador es el nombre que va a recibir la configuración y servirá para poder referenciarla más tarde. Aparte de aportar su nombre, es necesario identificar la entidad y la arquitectura relacionados en la configuración mediante sus identificadores respectivos. Cuando el diseño sea jerárquico, también pueden determinares las entidades y arquitecturas que se van utilizar para los componentes de más bajo nivel. En este caso es necesario relacionar las referencias de los componentes con una entidad y una arquitectura o bien indicar la configuración que se quiere usar para cada componente. Como se podría dar el caso de que dos referencias de un mismo componente utilizaran diferentes arquitecturas (o

entidades), se da flexibilidad para configurar todas las referencias de un componente a la vez o por separado.

La configuración del multiplexor de dos bits utilizado en el apartado anterior en el caso que se quiera trabajar con la arquitectura llamada *FlujoDatos* sería:

```
configuration  Mux21_cfg of  Mux21 is
    for FlujoDatos
    end for;
end   Mux21_cfg;
```

2.2.4 Paquetes

Un paquete permite agrupar un conjunto de declaraciones para que puedan ser usadas por varios dispositivos sin ser repetidas en la declaración de cada dispositivo. De esta forma se facilita la reutilización y la actualización del código.

Normalmente en un paquete se suelen declarar constantes, tipo y subtipos de datos, subprogramas y componentes. Más adelante se verá con más detalle el significado y la utilización de cada uno de estos elementos del lenguaje.

Un aspecto importante del paquete es que al igual que pasaba con las entidades, se divide en dos unidades de diseño diferenciadas: la declaración y el cuerpo del paquete. La declaración de paquete aporta la visión externa de los elementos que se declaran mientras que el cuerpo del paquete define su implementación. De este modo se pueden ocultar los detalles de implementación a un diseñador que puede estar interesado en cómo utilizar un elemento pero no necesita saber cómo está implementado.

La sintaxis VHDL para declarar un paquete es la siguiente:

```
package  identificador  is
      [declaraciones]
end  [identificador];
```

Para el cuerpo del paquete la sintaxis VHDL es:

```
package body identificador is
        [declaraciones cuerpo]
end [identificador];
```

Como puede apreciarse, la sintaxis es muy parecida para la declaración y el cuerpo del paquete, la única diferencia reside en la naturaleza de las declaraciones de las dos unidades. Al analizar el cuerpo de un paquete es imprescindible haber analizado la declaración antes, de forma que si ésta varía se tendrá que reanalizar el cuerpo del paquete.

Cuando se analiza un paquete, el resultado del análisis queda almacenado en una biblioteca para poder ser usado más adelante.

2.2.5 Bibliotecas

Una biblioteca sirve para almacenar el resultado del análisis de las unidades de diseño para su uso futuro. Las bibliotecas son beneficiosas porque facilitan la compartición y la reutilización del código en diferentes diseños.

Aunque las unidades de diseño se analicen separadamente, se tiene que respetar un cierto orden ya que algunas unidades dependen de otras. En general, la declaración de una entidad tiene que analizarse antes que su arquitectura y la declaración de un paquete antes que su cuerpo. Además, cuando una entidad utilice algún elemento de un paquete, las unidades que este paquete tienen que analizarse antes que las unidades de la entidad. Por último antes de anlizar una configuarción tienen que haberse analizado las arquitecturas seleccionadas en dicha configuarción.

La biblioteca *work* o de trabajo sirve de biblioteca por defecto y es la que se utiliza siempre que no se especifique otro nombre. De todos modos, el diseñador puede crear el número de bibliotecas que crea necesario y repartir sus diseños entre las bibliotecas de la forma que crea más conveniente.

Desde un modelo almacenado en una biblioteca no puede accederse directamente a las unidades de diseño de otras bibliotecas, ya que solamente se tiene visibilidad de la biblioteca donde está almacenado este modelo. Para dar visibilidad a una biblioteca se utiliza la sentencia ***library*** seguida del nombre de la biblioteca. Por ejemplo, para usar los elementos de un paquete que se llame *PaqueteEjemplo* almacenado en la biblioteca *BibliotecaEjemplo* desde un modelo que se vaya a guardar en otra biblioteca se tendría que empezar el modelo de la forma:

```
library BibliotecaEjemplo;
use BibliotecaEjemplo.PaqueteEjemplo.all;
```

Las bibliotecas *work* y *std* son excepciones en el sentido que siempre son visibles y, por tanto, no requieren la sentencia **library**.

Finalmente cabe destacar que la definición de biblioteca es una definición lógica, en el sentido de que cada herramienta puede implementarla como quiera sobre el sistema de ficheros. En algunos casos una biblioteca será un fichero, en otros un directorio o una estructura jerárquica de directorios. Por esta razón, cada herramienta debe aportar facilidades para crear bibliotecas y mapaer su estructura lógica a la posición física en el disco.

2.3 OBJETOS, TIPOS DE DATOS Y OPERADORES

Un objeto es un elemento del lenguaje que tiene un valor de un tipo de datos concreto. Este tipo de datos determina el conjunto de valores posibles que el objeto puede contener así como la clase de operaciones que se podrán aplicar. En general, no será posible realizar operaciones entre dos objetos de distinto tipo si no se aplica explícitamente una función de conversión de tipo a los operados. Aunque esta faceta del VHDL implica más atención por parte del diseñador a la hora de escribir un modelo, permite detectar errores durante el análisis del código sin necesidad de simulación.

2.3.1 Objetos del VHDL

Un objeto VHDL es un elemento que tiene asignado un valor de un tipo determinado. Hay cuatro clases distintas de objetos: las constantes, las variables, las señales y los ficheros.

Todos los objetos deben declararse antes de poder ser utilizados. La declaración consiste básicamente en asignar un identificador y un tipo al objeto.

2.3.1.1 Constantes

Una constante es un objeto que mantienen su valor inicial, de modo que no puede ser modificada una vez ha sido creada. La sintaxis VHDL para declarar una constante es la siguiente:

```
constant identificador {, …} : tipo [:=expresión];
```

El identificador dará nombre a la constante y servirá para referenciarla más adelante, el tipo indica la naturaleza del valor que contiene y la expresión sirve para inicializar la constante. Debido a que el valor de una constante no se puede cambiar, en general se incluirá la parte de inicialización en la declaración.

A continuación se ven algunos ejemplos de declaraciones de constantes:

```
contant Pi : real := 3.1415927;
constant BitsPalabra : integer := 8;
constant RetardoAND2, RetardoOR2 : time := 2 ns;
```

Cualquier constante podría ser sustituida directamente por un literal con su valor; no obstante, es aconsejable utilizar constantes para mejorar la legibilidad del código, ya que normalmente el identificador de la constante será más significativo que su valor.

2.3.1.2 Variables

A diferencia de las constantes, las variables pueden cambiar su valor una vez han sido declaradas mediante las sentencias de asignación. Una variable no tiene ninguna analogía directa en *hardware*, normalmente se utiliza en el estilo algorítmico para almacenar valores intermedios dentro de un proceso. La sintaxis VHDL para declarar una variable es la siguiente:

```
variable identificador {, …} : tipo [:=expresión];
```

Como se puede ver, a excepción de la palabra reservada que es diferente, la sintaxis para declarar una variable es idéntica a la que se requería para la declaración de una constante.

A continuación se muestran algunos ejemplos de declaración de variables:

```
variable Indice1, Indice2, Indice3 : integer := 0;
variable Comparación : boolean;
```

Cuando una variable ha sido creada su valor puede ser modificado utilizando una sentencia de asignación de variable. La sintaxis VHDL de estas sentencias es la siguiente:

```
identificador := expresión;
```

El nuevo valor de la variable será el resultado de evaluar la expresión incluida en la sentencia. La asignación será inmediata en el sentido que el nuevo valor sustituirá al antiguo inmediatamente después de haberse evaluado la expresión, de modo que si en la siguiente sentencia se hace referencia a esta variable ya se tendrá en cuenta el nuevo valor. Normalmente las variables se declaran en la parte declarativa de los procesos, de forma que solamente son visibles en el proceso donde se van a utilizar. En caso de que una variable fuera visible e más de un proceso, teniendo en cuenta que la ejecución de los procesos es concurrente, sería impredecible el resultado que se produciría cuando un proceso modificara una variable mientras otro utiliza su valor.

2.3.1.3 Señales

Una señal es un objeto que, al igual que un variable, puede modificar su valor dentro de los posibles valores de su tipo pero, a diferencia de está, tiene una analogía directa con el *hardware*, ya que se puede considerar como una abstracción de una conexión física o bus. Por esta razón no está restringida a un proceso sino que sirve para interconectar componentes de un circuito y para sincronizar la ejecución y suspensión de procesos. La sintaxis VHDL para declarar una señal es muy parecida a la sintaxis requerida para declarar constantes y variables:

```
signal identificador {, …} : tipo [:=expresión];
```

A diferencia de las variables, una señal no se declarará en la parte declaratoria de un proceso sino en la arquitectura del dispositivo.

Los puertos de una entidad son señales que se utilizan para interconectar el dispositivo con otros dispositivos. Su declaración es un poco especial, ya que aparte de determinar un identificador y un tipo de datos es necesario indicar la dirección de la señal respecto de la entidad. La sección de declaración de puertos de una entidad tiene la siguiente sintaxis VHDL:

```
port ({identificador {, …} : dirección tipo [:= expresión];);
```

En este caso la expresión opcional se utiliza en caso de que el puerto esté desconectado. A continuación se muestran algunos ejemplos de declaración de señales:

```
signal Reloj : std_logic := ´0´;
signal Comparacion : bit;
signal Resultado : integer range 0 to 7;
port  (a, b  : in integer range 0 to 7;
         c     : out  integer range 0 to 7;
         d     : inout  std_logic);
```

Una señal puede modificar su valor mediante la sentencia de asignación de señales. A diferencia de las variables, la asignación de una señal no se realiza de inmediato sino que antes se acaban de ejecutar los procesos activos. De esta forma se puede asegurar que, el resultado siempre será el mismo, independientemente del orden en que se ejecuten los procesos.

2.3.1.4 Ficheros

Un fichero es un objeto que permite comunicar un diseño VHDL con un entorno externo, de manera que un modelo puede escribir datos y leer datos que persisten cuando la simulación termina. Un uso bastante común de los ficheros es el almacenar los estímulos de simulación

que se quieren aplicar al modelo en un fichero de entrada y salvar los resultado de simulación en un fichero de salida para su posterior estudio.

Un fichero es de un tipo de datos determinado y sólo puede almacenar datos de ese tipo. La sintaxis para declarar un fichero es la siguiente:

file identificador {, ...} : tipo_fichero [**is** dirección "nombre";]

El tipo de fichero indica el tipo de datos que contendrá el fichero y debe declararse explícitamente antes de declarar el objeto fichero.

2.3.2 Tipos de datos

El tipo de dato es un concepto fundamental en VHDL, ya que cada objeto debe ser de un tipo concreto que determinará el conjunto de valores que puede asumir y las operaciones que se podrán realizar con este objeto. VHDL proporciona sentencias específicas de nuevos tipos de datos además de un conjunto de tipos predefinidos de uso común.

2.3.2.1 Declaración de tipo de datos

La declaración de un tipo de datos es la sentencia VHDL utilizada para introducir un nuevo tipo. La sintaxis será:

type identificador **is** definición_tipo;

La parte de definición de tipo sirve para indicar el conjunto de valores del tipo y puede tener varios formatos, que se verán a medida que se expliquen los diferentes tipos predefinidos del lenguaje. A continuación se dan ejemplos de declaraciones:

Una vez definido un nuevo tipo de datos ya se pueden declarar objetos de este tipo, teniendo en cuenta que los ficheros requieren un tipo especial llamado tipo fichero. Por ejemplo se podría declarar los siguientes objetos:

constant DiasEnero : DiaMes := 31;

```
variable DiaHoy : DiaMes;
signal Dirección : PuntosCardinales;
port (Entrada : in PuntosCardinales;
      Salida : out PuntosCardinales);
```

Cada tipo es diferente e incompatible con los demás tipos, aún cuando las definiciones sean idénticas. Si por ejemplo se declara el tipo

```
type De1a31 is range 1 to 31;
```

no será posible asignar a un objeto de tipo *De1a31* el valor de un objeto de tipo *DiaMes.* Para hacerlo será necesario incluir implícitamente una función de conversión de tipo para que la asignación se realice entre operandos del mismo tipo.

2.3.2.2 Tipos de datos escalares

Los tipos de datos escalares son aquellos cuyos valores están formados por una sola unidad indivisible. Existen tipos de datos escalares predefinidos de distintas clases, concretamente tipos enteros, tipos reales, tipos físicos y tipo enumerados. Los tipos reales son continuos en el sentido de que dentro de un intervalo existe un número infinito de valores, por esta razón, como no es posible representar todos los números, se tiene que escoger un conjunto de números representables de manera que cualquier valor se deberá redondear al número representable más próximo produciéndose cierto error. Por el contrario, los tipos enteros y enumerados se conocen como tipos discretos. Al declarar un objeto de tipo escalar, este se inicializará por defecto al valor más a la izquierda del tipo.

2.3.2.2.1 Tipos enteros y tipos reales

Los tipos enteros y reales son tipos de datos predefinidos que, como su nombre indica, sirven para representar números enteros y reales respectivamente. La sintaxis VHDL para declarar un tipo entero o real especificando un rango es la siguiente:

```
type identificador is range literal to | dowto literal;
```

Dependiendo de si se escriben literales enteros o en punto flotante, el tipo de datos declarado será de tipo entero o de tipo real. Con las palabras reservadas ***to*** y ***dowto*** se puede indicar un rango creciente o decreciente respectivamente, de esta forma se determinará la ordenación que tendrán los valores dentro del tipo. Por defecto, un objeto de tipo entero o real se inicializará al valor más a la izquierda de los que forman el tipo, que en este caso coincide con el primer literal del rango. A continuación de dan ejemplos de declaraciones de tipos enteros y reales:

```
type  PrecioProducto is range 1  to  1_000_000;
type  Puntuación is range  0.0  to  10.0;
varible PrecioMesa : PrecioProducto;
variable MiNota : Puntuación;
```

2.3.2.2.2 Tipos físicos

Los tipos físicos sirven para representar medidas del mundo real como pueden ser distancia, tiempo peso. Por esta razón además de un literal numérico llevan asociada una unidad primaria de la medida física que quieren cuantificar. También se pueden definir unidades secundarias múltiplos de la unidad primaria para poder utilizar en cada momento, la unidad más adecuada según el valor que se quiera representar. La sintaxis VHDL para declarar un tipo físico es la siguiente:

```
type  identificador  is range  literal  to |  dowto  literal
   units
       identificador;
       { identificador = literal_físico; }
end units [identifcador];
```

El rango determina el valor mínimo y máximo de unidades primarias del tipo físico. Esta unidad primaria debe ser menor que las unidades secundarias, para las que se tendrá que indicar el número de unidades primarias que contienen mediante el literal físico asociado. Este valor puede especificarse directamente en la función de la unidad primaria o mediante una unidad secundaria previamente declarada.

Se puede definir un tipo físico para cualquier medida física que se quiera cuantificar. Probablemente, el tipo físico más común en VHDL es el tipo *time* Tiempo), declarado en el paquete *standard* de la biblioteca *std*, que sirve para especificar retardos. La declaración del tipo *time* sería

```
type time is range 0 to 1E20
   units
       fs;
       ps = 1000 fs;
       ns = 1000 ps;
       µs = 1000 ns;
       ms = 1000 µs;
      sec = 100 ms;
      min = 60 min;
      hr = 60 min;
end units;
```

Internamente VHDL considera los tipos físicos como enteros con una unidad primaria asociada, por lo que cualquier valor físico que contenga un literal será redondeado a un número entero de unidades primarias.

2.3.2.2.3 Tipos enumerados

Es un tipo de datos en el que se define el conjunto de posibles valores del tipo especificando una lista que contiene todos los valores. Se llaman tipos enumerados porque en la lista se enumeran todos y cada uno de los valores que forman el tipo. La sintaxis para declarar un tipo enumerado es la siguiente:

```
type identificador is ( identificador | carcácter {, ...} );
```

El primer identificador es el nombre del tipo y servirá para referenciarlo. Entre paréntesis y separados por comas se especifican todos los valores del tipo. Como mínimo debe indicarse un valor. A continuación se dan algunos ejemplos de tipos enumerados:

```
type Comandos is ( izquierda, derecha, arriba, abajo, disparar);
type Teclas is (´a´, ´d´, ´w´, ´x´, ´´ );
```

En el paquete *standard* de la biblioteca *std* se definen algunos tipos enumerados de uso bastante común. Concretamente el tipo carácter, el tipo booleano y el tipo bit. El tipo carácter no es más que una enumeración de todos los caracteres de 8 bits estandarizado por la ISO. Los tipos booleanos y bit se definen de la siguiente manera:

```
type boolean is (false, true);
type bit is (´0´, ´1´);
```

El tipo boolenao se utiliza normalmente como resultado de la evaluación de un operador relacional, mientras que el tipo bit se utiliza para el modelado de sistemas digitales.

2.3.2.2.4 Expresiones y operadores

Se puede considerar que una expresión es una fórmula que indica la forma como debe calcular un valor. Para hacerlo, se dispone de una serie de operadores más unos elementos que actúan como operandos. Estos elementos deben ser básicamente literales, identificadores, atributos que determinen un valor, calificaciones y conversiones de tipo y paréntesis para especificar un orden de precedencia. Por lo que a los operadores se refiere, VHDL tiene predefinidos los operadores aritméticos, relaciones, lógicos y de concatenación de cualquier lenguaje de programación común más los operadores de desplazamiento para vectores unidimensionales que fueron añadidos en VHDL-93. La Tabla 2-1 muestra todos los operadores predefinidos:

Cada operador puede aplicarse sobre un conjunto de tipos de datos diferentes, por lo que realmente en VHDL existen varias definiciones distintas del mismo operador. Esta característica, llamada sobrecarga de operadores, resulta muy útil, ya que permite redefinir los operandos predefinidos sobre tipos de datos definidos por el usuario.

LOGICOS	RELACIONALES	ARITMÉTICOS	VHDL-93
NAND	= -- igual que	+, - , * , / , **	**sll** -- despl. lógico izquierda
OR	< -- menor que	**Abs** – valor absoluto	**srl** – despl. lógico derecha
NAND	> -- menor que	**Mod** – módulo	**sla**-- despl. arit. izquierda
NOR	<= -- menor o igual que	**Rem** – resto	**sra** -- despl. arit. derecha
XOR	>= -- mayor o igual que	**&** -- concatenación	**rol** -- rotación izquierda
NOT			**ror** -- rotación derecha

Tabla 2-1. Operadores predefinidos por VHDL

2.4 SENTENCIAS SECUENCIALES

Las sentencias secuenciales son aquellas que nos permiten describir o modelar funcionalidad de un componente, y puede encontrarse en los procesos o en los cuerpos de los subprogramas.

Las podemos clasificar en sentencias de asignación (a señal y a variable), sentencias condicionales (***if, case***), sentencias iterativas (***loop, exit, next***), otras sentencias (***wait, assert, null***) y llamadas a subprogramas.

2.4.1 La sentencia *wait*

La sentencia ***wait*** indica en que punto del flujo debe suspenderse la ejecución de un proceso, al mismo tiempo puede fijar en qué condiciones debe reactivarse dicho proceso. Al ejecutar la sentencia ***wait*** el proceso se suspende y al mismo tiempo se fijan las condiciones para su reactivación.

La primera forma en que se puede utilizar la sentencia ***wait*** es sin ningún tipo de condición para despertar el proceso. Esto significa que el proceso en cuestión ejecutará las sentencias que contenga hasta el ***wait*** y entonces se suspenderá, sin que pueda volver a activarse en toda la simulación. El uso más común de este estilo lo encontramos en los bancos de pruebas a realizar sobre el dispositivo a verificar, y a continuación se termina la simulación.

```
process
begin
        sentencias_secuenciales
        wait;
end process;
```

La segunda forma de usar la sentencia ***wait*** establece a que señales será sensible el proceso (***wait on*** lista_señales). Por lo tanto, siempre que se produzca un evento en alguna de las señales indicadas en la sentencia ***wait*** el proceso se despertará y ejecutará sus sentencias secuenciales hasta ejecutar otra vez una sentencia ***wait***. Es usual utilizar esta forma de la sentencia ***wait*** para modelar lógica combinacional que debe responder a cualquier cambio que se produzca en sus entradas.

```
process
begin
    c<=a and b;
    wait on a, b;
end process;
```

Al suspender un proceso también puede fijarse una condición para su reactivación, esta condición se especifica con la forma ***wait until*** *condicion_booleana*. En *condición_booleana* puede aparecer cualquier expresión que al evaluarse de cómo resultado TRUE o FALSE. Un ejemplo típico de este uso se encuentra en el modelado de elementos sensibles al flanco de un reloj. El siguiente proceso describe el comportamiento de un biestable sensible al flanco de subida del reloj:

```
process
begin
    q<=d and b;
    wait  until Reloj = ´1´;
end  process;
```

Al ejecutar la sentencia **wait** el proceso se suspenderá y no se reactivará hasta que no se produzca un evento en *Reloj* y además *Reloj* pase a valer ´1´. Si al llegar a esta sentencia

wait, *Reloj* ya valiese ´1´, entonces el proceso no se reactivaría, ya que debe cumplirse que haya un evento en la señal además de cumplirse la condición booleana.

Por último al suspender un proceso se puede especificar un cierto tiempo antes de que éste se reactive. Para ello se utiliza la forma ***wait for***; como ejemplo, el siguiente proceso utiliza esta opción de la sentencia ***wait*** para generar un reloj de un determinado período.

```
process
begin
    Reloj<= not Reloj;
    wait  for 10 ns;
end process;
```

Cada vez que el proceso se suspenda, se fija su reactivación para 10ns más tarde, de modo que lo que hace es generar un reloj de 20 ns de periodo.

2.4.2 Asignación a señal

La asignación a señal como sentencia secuencial (dentro de un proceso o del cuerpo de un subprograma) presenta la siguiente sintaxis en su forma más sencilla:

```
[etiqueta :]  nombre_señal <= valor { after expresión_tiempo}
```

Recordemos que las señales en VHDL son objetos que consisten en un valor actual (el que en cada momento pueden leer los operandos que son sensibles a dicha señal) y un conjunto de pares tiempo valor, que forma la historia futura de la señal o cola de eventos de la señal. Recordemos también que incluso en el caso de ejecutar una asignación a señal sin especificación alguna de retardo, la asignación del nuevo valor no se producirá hasta pasado un retardo delta. En definitiva debe quedar claro que al ejecutarse una de estas sentencias, no se está modificando el contenido de la señal, sino que se está modificando el contenido de su cola de eventos, es decir, se está proyectando un posible cambio del valor futuro de la señal, que algunas veces puede no llegar a producirse.

Para dejar claro este comportamiento de la asignación a señal (que al principio parece extraño al compararlo con la asignación a variable), veamos un ejemplo que nos ayude a entender mejor su funcionamiento. Supongamos que queremos intercambiar el contenido de dos señales, para ello podemos escribir el siguiente proceso:

```
process
begin
    a <= b;
    b <= a;
    wait on a, b;
end  process;
```

Estas dos sentencias de asignación consiguen intercambiar el valor de las dos señales, ya que cuando se ejecuta la segunda *(b<=a),* el valor de la señal *a* no refleja la situación *b<=a*, ya que esta solo ha proyectado un posible cambio en la cola de eventos de *a*. No será hasta la suspensión del proceso (ejecución de la sentencia ***wait***) cuando las señales *a* y *b* tomen el valor que se ha proyectado en su cola de eventos.

La forma en que se comporte la asignación a señal dependerá básicamente del modelo de retardo elegido. VHDL permite escoger entre dos tipos de retardo: el transporte *(**transport**)* y el inercial *(**inertial**)*. El modelo de transporte propaga cualquier cambio que se produzca, mientras que el inercial filtra aquellos cambios de duración inferior a un mínimo. Buscando analogías con el mundo físico, el modelo de transporte es el que refleja una línea de transmisión ideal, mientras que el modelo inercial es el que rige el comportamiento de una puerta lógica.

Por defecto, VHDL supone que las asignaciones a señal siguen el modelo inercial, para usar el modelo de transporte debe especificarse en la asignación, utilizando la siguiente sintaxis:

```
nombre_señal <= [transport ]  valor  { after expresión_tiempo}
```

2.4.3 Asignación a variable

La asignación a variable reemplaza el valor actual de la variable con el valor especificado por una expresión. Su sintaxis general es:

```
[etiqueta :] nombre_variable := expresión;
```

A diferencia de la asignación a señal vista en el apartado anterior, la asignación a variable no tiene ningún retardo asociado, de manera que al ejecutarse, la variable toma el nuevo valor justo en ese momento, deforma que las sentencias que se ejecuten a continuación ya pueden ver ese nuevo valor. Por tanto, si ejecutamos el siguiente par de asignaciones:

```
a :=  b;
b := a;
```

No se consigue intercambiar las dos variables, sino que después de ejecutarse, ambas variables contienen el valor inicial de *b*. Para intercambiar las variables debemos usar una variable temporal:

```
Tmp := a;
a  := b;
b := Tmp;
```

Una variable se declara en un proceso o en un subprograma y sólo es visible en ese proceso o subprograma. Las variables definidas en un proceso existen durante toda la simulación, o sea, nunca se reinicializan. Las variables definidas en un subprograma se reinicializan cada vez que se llama al subprograma. Si la ejecución de un subprograma se suspende por la ejecución de una sentencia ***wait***, entonces sus variables mantienen su valor cuando se reactiva, hasta el momento en que se termine la ejecución del mismo.

2.4.4 La sentencia if

La sentencia ***if*** se utiliza para escoger en función de alguna condición qué sentencias deben ejecutarse. En su forma más simple nos permite ejecutar un conjunto de sentencias en función de una condición:

```
if condicion then
    -- sentencias_secuenciales
end if;
```

Una segunda forma de la sentencia ***if*** permite escoger entre dos grupos de sentencias a ejecutar, dependiendo de una condición; en caso de que la condición se compla, se ejecutará el primer grupo, en caso contario, se ejecutará el segundo grupo.

```
if condicion then
        -- sentencias_secuenciales
else
        -- sentencias_secuenciales
end if;
```

Un ejemplo típico del uso de esta sentencia ***if*** se puede encontrar en el modelado de un *buffer* triestado. Si la señal de habilitación está activa, el buffer deja pasar a la salida el valor que tenga en su entrada, en otro caso deja a su salida en alta impedancia:

```
entity Triestate is
port   ( Habilitación, entrada : in std_logic;
            Salida  : out std_logic);
end Triestate;
architecture Ejemplo of Triestate is
begin
    process
    begin
      if Habilitación=´1´ then
          Salida <= Entrada;
      else
         Salida <= ´Z´;
```

```
            end  if;
            wait on  Habilitación, Entrada;
        end process;
    end  Ejemplo;
```

Una tercera forma de la sentencia ***if*** permite seleccionar entre dos o más conjuntos de sentencias a ejecutar, cada una de ellas en función de distintas condiciones.

```
    if  condicion  then
            -- sentencias_secuenciales
    elsif  condicion  then
            -- sentencias_secuenciales
    elsif  condicion  then
            -- sentencias_secuenciales
    end  if;
```

Como ejemplo veamos el modelo de un biestable por nivel *(Latch)* con entrada de datos *(dato)* , señal de carga *(carga)* y señal de inicialización *(Iniciar)*:

```
process
begin
  if  Iniciar =´0´ then
      Biestable <=´0´;
  elsif   carga =´1´then
       Biestable <= Dato;
end  if;
wait  on  Iniciar, Carga, dato;
end process;
```

En este ejemplo podemos observar que si se cumple la primera condición *(iniciar=´0´)* se colocará el valor *´0´*sobre la señal *Biestable.* Sólo si no se cumple la primera condición se considerará la segunda, y en caso que ésta sea cierta (TRUE) se asignará *Dato* a *Biestable*. Obsérvese que la forma en que se ejecuta una sentencia ***if*** denota prioridad. O sea, el grupo de sentencias que se ejecutan en caso de cumplirse la primera condición son prioritarias sobre el

segundo grupo de sentencias dependientes de la segunda condición, ya que si la primera no se cumple, el segundo grupo no se ejecutará aún cuando su condición de control fuese cierta.

2.4.5 La sentencia case

La sentencia ***case*** se utiliza para escoger qué grupo de sentencias deben ejecutarse entre un conjunto de posibilidades, basándose en el rango de valores de una determinada expresión de selección. Esta puede ser de cualquier tipo discreto (enumerados o enteros) o de tipo vector de una dimensión. La forma general de la sentencia ***case*** es:

```
case  expresión  is
when  valor => sentencias_secuenciales;
{when  valor => sentencias_secuenciales;}
end  case;
```

Los valores especificados para la selección en la sentencia ***case*** deben cumplir dos condiciones. La primera es que nunca puede haber una intersección entre el rango de valores especificados en dos opciones distintas de la sentencia ***case***. La segunda es que la unión de todos los valores especificados debe cubrir todos los valores posibles que pueda tomar la variable selección. Por ejemplo, podemos modelar un multiplexor de tres entradas, en el cual la codificación “00” y “01” seleccionen la misma salida:

```
begin
   case  Seleccion  is
         when “00”  | “01” =>
           Salida <= Entrada1;
         when  “10” =>
           Salida <= Entrada2;
         when  “11” =>
           Salida <= Entrada3;
   end  case;
   wait  on  Seleccion, Entrada1, Entrada2, Entrada3;
end process;
```

Se puede especificar dos o más valores dentro del rango de posibles valores que puede tomar la expresión de selección usando la unión lógica, para la cual se usa el signo "|". Se puede especificar un rango dentro de los valores posibles de la expresión de selección que en este caso deben tomar valores discretos (enteros o enumerados). Para ello se puede usar la partícula ***to***, con lo cual se puede especificar un rango como *valork* ***to*** *valorm*.

Por último, se puede especificar como valor de selección la palabra clave ***others***, cuando en una sentencia ***case*** aparece esta opción, debe ser la última, y significa que en caso de no escoger ninguno de los rangos especificados en las opciones anteriores de la sentencia, se ejecutarán las sentencias que se encuentren en dicha opción.

```
process
begin
   case  Seleccion  is
         when "a"  to "c" =>
           Salida <= Entrada1;
         when  others =>
           Salida <= Entrada2;
   end  case;
   wait  on  Selección, Entrada1, Entrada2;
end  process;
```

Es muy habitual el uso de la cláusula ***others*** en la última opción de la sentencia ***case*** debido a que los posibles valores de la señal de selección deben quedar cubiertos.

2.4.6 La sentencia *loop*

La sentencia ***loop*** se utiliza para ejecutar un grupo de sentencias secuenciales de forma repetida. El número de repeticiones puede controlarse en la misma sentencia usando alguna de las opciones que ésta ofrece. Su sintaxis general es:

```
[etiqueta] : [while condicion_booleana  |  for  control_repeticion]  loop
      sentencias secuenciales
end  loop [etiqueta];
```

Tal como refleja en su sintaxis, existen diversas formas de controlar el número de repeticiones que producirá la sentencia ***loop***, de forma que vamos a estudiar cada una de ellas.
Podemos usar la sentencia ***loop*** sin ningún tipo de control sobre el número de repeticiones del bucle, de forma que se provoca la ejecución infinita del grupo de sentencias especificadas. Aunque en principio pueda parecer no muy adecuada la definición de un bucle infinito en la descripción de un componente, este puede tener sentido al modelar un dispositivo para el cual queremos fijar unos valores iniciales, mediante la ejecución de unas sentencias, para luego pasar a ejecutar el conjunto de sentencias que lo modelan de forma indefinida.

Se puede definir una condición de finalización del bucle con la opción ***while*** *condicion_booleana*. En este caso el grupo de sentencias secuenciales especificadas en la sentencia ***loop*** se ejecutarán mientras la condición sea cierta, cuando la condición sea falsa se abandonará el bucle para pasar a ejecutar las sentencias que aparezcan a continuación. Como ejemplo podemos modelar un contador de 4 bits de la siguiente forma:

```
process
begin
        contador <= 0;
        wait  until  Reloj=´1´;
         while Contador < 15  loop
           Contador <= Contador + 1;
           wait until  Reloj=´1´;
        end loop;
  end process;
```

El bucle anterior se ejecutará mientras el contador no llegue al valor 15; en cuanto el contador tome el valor 15, se cumple la condición de final de bucle y, por lo tanto, pasa a ejecutarse la primera sentencia que se encuentre después de la sentencia ***loop***, de forma que se carga el contador con el valor 0.

Otra forma de controlar el número de iteraciones de la sentencia ***loop*** se deriva del uso de la opción ***for*** *control_repeticion*. Esta nos permite controlar el número de repeticiones,

dependiendo del rango de valores que pueda tomar la expresión de control del bucle. Usando este mecanismo el contador anterior quedaría:

```
process
begin
        Contador <= 0;
        for  I  in  0  to  15  loop
          wait until Reloj=´1´;
          Contador <= Contador + 1;
        end loop;
end process;
```

Este bucle se repetirá mientras la expresión de control (*I* en este caso) tome el rango de valores = hasta 15, en cuanto tome el valor 16 se saldrá del bucle, de forma que se ejecutará otra vez la inicialización del contador. La variable utilizada como control de la sentencia ***loop*** no necesita ser declarada y puede tomar cualquier rango de valores de tipo discreto (enumerados o enteros).

2.4.7 La sentencia exit

La sentencia ***exit*** está muy relacionada con la sentencia con la sentencia ***loop***, de hecho su única utilidad es ofrecer una forma de terminar la ejecución de un bucle. La sintaxis general es:

```
exit  [etiqueta_loop] [when condicion_booleana]
```

La sentencia ***exit*** puede ir acompañada de dos opciones. La primera permite expresar una condición, en caso que esta se cumpla se finaliza la ejecución de la sentencia ***loop*** y se pasa a ejecuatr la primera sentencia que se encuentre a continuación. La segunda opción permite especificar la etiqueta de una sentencia ***loop*** concreta, de forma que el bucle que finaliza su ejecución como consecuencia de la sentencia ***exit*** es aquel que corresponda a la etiqueta especificada. En caso de no indicar ninguna etiqueta se sobreentiende que se finaliza el bucle actual.

2.4.8 La sentencia next

La sentencia ***next*** está íntimamente ligada a la sentencia ***loop***. Se utiliza para detener la ejecución de una sentencia ***loop*** y pasar a la siguiente iteración de la misma. Su sintaxis general es:

```
next  [etiqueta_loop] [when condicion_booleana]
```

Tal como se indica en su sintaxis, se puede especificar una condición para la cual debe pasar a ejecutarse la siguiente iteración de la sentencia ***loop*** y también puede especificarse una etiqueta haciendo referencia a que sentencia ***loop*** deberealizar ese salto de una iteración. Para ejemplificar el uso de la sentencia ***next***, podemos modificar nuestro contador módulo 16 de forma que no pase por el valor 8.

```
process
begin
        Contador <= 0;
        for  I  in  0  to  15  loop
                next  when  I=8;
                Contador <= I;
                wait  until  Reloj=´1´;
        end loop;
end  process;
```

Mientras el valor del índice *I* no toma el valor 8, el contador va tomando los valores de este índice. Para el valor *I*=8 no se ejecutan el cuerpo del bucle y pasa a ejecutarse la siguiente iteración, con lo cual el valor del contador pasa de 7 a 9. Aunque no quede reflejado en este ejemplo, si antes de la sentencia ***next*** hay otras sentencias secuenciales, éstas se ejecutarán aun en el caso de cumplirse la condición de salto de iteración.

2.4.9 La llamada secuencial a subprogramas

Los subprogramas (procedimientos o funciones) que tengamos definidos en alguna parte del código pueden ser llamados para su ejecución en cualquier parte de un código secuencial. La sintaxis de llamada a un procedimiento (***procedure***) es:

```
nombre_procedimiento [{parámetros,}];
```

La sintaxis de llamada a una función (***function***) es:

```
nombre_función  [{parámetros,}];
```

La diferencia radica en que una llamada a función forma parte de la expresión de una asignación o es la asignación en sí misma, mientras que la llamada a procedimiento es una sentencia secuencial por sí sola.

En ambos casos al ejecutarse la llamada a subprograma, este pasa a ejecutarse para retornar el control a la siguiente instrucción secuencial en cuanto finalice su ejecución.

2.5 SENTENCIAS CONCURRENTES

En los siguientes apartados se presentarán algunas de las sentencias concurrentes del VHDL. La característica común a todas ellas es que se ejecutan en paralelo. Principalmente las encontramos en las arquitecturas de los modelos y no estarán contenidas en ningún proceso. Algunas de ellas tienen su equivalente en las sentencias secuenciales, mientras que otras son propias de las sentencias concurrentes, especialmente aquellas cuya utilidad está en la descripción de la estructura del diseño.

2.5.1 La sentencia process

Un proceso es una sentencia concurrente que define su comportamiento a través de sentencias secuenciales. Cualquier sentencia concurrente o secuencial VHDL tiene un proceso equivalente, de hecho el simulador VHDL sólo trabaja con procesos. La sintaxis general de un proceso es:

```
 [etiquta :]  process [(nombre_señal {,...})]
          declaraciones
begin
          sentencias_secuenciales;
end  process  [etiqueta];
```

La etiqueta del proceso es opcional, sirve para identificar el proceso, y puede ser útil en las tareas de depuración de código. La parte de declaraciones se utiliza para declarar datos locales al proceso, también puede contener subprogramas locales al proceso. La parte de sentencias secuenciales es la que define el comportamiento del proceso a través de un algoritmo.

Un proceso es un bucle infinito, al llegar al final (***end process***) vuelve a su primera sentencia secuencial (que aparece después del **begin**) para continuar su ejecución. La ejecución del proceso se detiene (el proceso se suspende) al ejecutar una sentencia ***wait***, al mismo tiempo la ejecución de un sentencia ***wait*** suele fijar las condiciones para despertar al proceso y continuar con su ejecución (fijar a que señales es sensible el proceso). Nótese que un proceso que no contenga ninguna sentencia ***wait*** entra en u bucle infinito que impide que avance el tiempo de simulación.

Existe una sintaxis simplificada que equivale a definir un proceso con una única sentencia ***wait*** como última sentencia del mismo. Así, el proceso que se escribe a continuación:

```
process
begin
          sentencias secuenciales;
wait  on  lista de sensibilidad;
end process;
```

es equivalente al siguiente proceso:

```
process  (lista sensibilidad)
begin
          sentencias secuenciales;
end process;
```

2.5.2 Asignación a señal concurrente

La asignación a señal puede darse también en el mundo concurrente. En este caso la asignación concurrente a señal tiene un proceso equivalente con una asignación secuencial a señal. La forma más sencilla de su sintaxis es:

```
[etiqueta :] nombre_señal <= [transport] forma_onda;
```

La etiqueta es opcional, sirve para identificar al proceso y puede ser útil en las tareas de depuración de código. La asignación a señal concurrente es una forma compacta de escribir una asignación a señal secuencial, se comporta de la misma forma que ésta y su principal diferencia radica en que se encuentra en las arquitecturas *(architecture)*, en lugar de encontrase en procesos o subprogramas.

La asignación concurrente es sensible a las señales que se encuentren a la derecha de la asignación, o sea, que formen parte de la expresión que se asigne. De forma que la siguiente asignación concurrente : a <= b; es equivalente al proceso

```
process (b)
begin
    a<=b;
end process;
```

Aunque en este ejemplo se ha presentado una asignación muy simple, la asignación concurrente permite algunas opciones potentes que le dan gran flexibilidad y que hacen que pueda tener cierta analogía con las sentencias secuenciales condicionales. Por ello estas formas compuestas de asignación concurrente se estudian en las siguientes secciones.

2.5.3 Asignación concurrente condicional

La asignación concurrente condicional es una forma compacta de expresar las asignaciones a señal usando la sentencia ***if*** secuencial. Su sintaxis es:

```
[etiqueta :]  nombre_señal <= [transport]
                    {forma_onda  when  expresión_booleana  else}
                    forma_onda [when expresión_booleana];
```

La sentencia funciona de forma que el valor que se asigna a la señal es el que se especifica en la señal que se cumple. Esta asignación concurrente se ejecutará cada vez que se produzca un evento en las señales que forman parte de la expresión de asignación *(forma_onda)* o en las señales que forman parte de la condición *(expresión_booleana)*

Como ejemplo típico del uso de una asignación condicional concurrente podemos modelar un multiplexor 2-1:

```
Salida <= Entrada  when  Sel=´0´ else
          Entrada2;
```

Cuyo proceso equivalente es:

```
process (Sel, Entrada1, Entrada2)
begin
   if  Sel=´0´ then
      Salida <= Entrada1;
   else
      Salida <= Entrada2;
   end if;
end process;
```

2.5.4 Asignación concurrente con selección

La asignación concurrente con selección es una forma compacta de expresar las asignaciones a señal usando la sentencia ***case*** secuencial. Su sintaxis es:

```
[etiqueta : ] with  expresion  select
nombre_señal <= {forma_onda} when valor, }
                {forma_onda} when valor;
```

La sentencia funciona de manera que la forma de onda que se asigna a la señal es la que se especifica en la opción correspondiente al valor que toma la expresión de selección. Esta asignación concurrente es ejecutará cada vez que se produzca un evento en las señales que forman parte de la expresión de selección *(expresión)* o en las señales que forman parte de la expresión de asignación *forma_onda).* Así por ejemplo un multiplexor 4 –1, sería:

```
entity  selsig  is
port ( d0, d1, d2, d3 : in bit;
        s             : in  integer range  0  to  3
        output:  out  bit);
 end  selsig;
 architecture  maxpld  of  selsig  is
 begin
   with  s  select
   output <= do when  0;
             d1 when  1;
             d2  when  2;
             d3  when  3;
 end  maxpld;
```

2.5.5 Sentencias estructurales

VHDL proporciona un conjunto de sentencias dedicadas a la descripción de estructura hardware. Son también sentencias concurrentes, ya que se ejecutan en paralelo con cualquier otra sentencia concurrente de la descripción y aparecen en la arquitectura de un modelo fuera de cualquier proceso.

2.5.5.1 Componentes

Para realizar la descripción estructural de un sistema es necesario conocer qué subsistemas o componentes lo forman, indicando las interconexiones entre ellos. Para este propósito VHDL proporciona el concepto de componente.

Para poder usar un componente VHDL exige que se realicen dos pasos: primero, debe declarase el componente, después, ya puede hacerse referencia o copia del componente.

La declaración de un componente puede aparecer en un arquitectura o en un paquete. Si aparee en la arquitectura, entonces se pueden hacer copias del componente en dicha arquitectura; si aparece en un paquete, se pueden hacer copias del componente en todas aquellas arquitecturas que usen ese paquete. La sintaxis de la declaración de un componente es:

```
component nombre_componente
    [generic (lista_generic);]
    [port (lista_puertos);]
end component [nombre_componente];
```

Habitualmente al declarar un componente se hará referencia a un diseño desarrollado anteriormente y ya compilado en alguna biblioteca. En este caso la declaración de componente coincide con la declaración de la entidad de dicho diseño, es decir, debe tener los mismo puertos, definidos del mismo tipo y dirección. Pero VHDL también ofrece la posibilidad de declarar un componente que aún no se ha desarrollado, en este caso los puertos que contenga y su tipo quedarán definidos en la declaración de componente.

Una vez ya se ha declarado un componente, ya podemos realizar tantas copias o referencias a él como se quiera. La referencia componente es una sentencia concurrente, que puede aparecer en una arquitectura, y que se ejecuta en paralelo con las demás sentencias concurrentes que pueda contener esa arquitectura. Cada copia o referencia a un componente se ejecutará cada vez que se produzca un evento en alguna de las señales conectadas a sus puertos de entrada o de entrada/salida. La sintaxis de la referencia a componentes es:

```
etiqueta_referencia : nombre_componente
        [generic map (lista_asociación);]
        [port map (lista_asociación);]
```

Al referenciar un componente debemos especificar qué valores queremos dar a cada uno de sus genéricos y que señales queremos conectar a cada uno de sus puertos. De esta manera queda completamente especificada la estructura que estamos definiendo, ya que habremos indicado qué componentes la forman y como se conectan entre ellos. Como ejemplo del uso de componentes, a continuación modelamos un sumador completo de un bit a partir de dos semisumadores de un bit y una puerta *or*. La Figura 2-1 muestra el esquema lógico de la descripción.

```
entity  SumadorCompleto is
      port (X, Y, CIn  : in bit;
              COut, Sum : out bit);
end SumadorCompleto;
architecture Estructura of  SumadorCompleto is
        signal A, B, C : bit;
        component SemiSumador
        port (I1, I2  :  in bit;
              COut, Sum : out bit);
        end component;
        component  PuertaOr
        port (I1, I2 : in bit;
               O : out bit);
        end component;
        signal A, B, C : bit;
begin
        U1 : Semisumador port map (X, Y, A, B);
        U2 : Semisumador port map (B, CIn, C, Sum);
        U3 : PuertaOr port map (A, C, COut);
end  Estructura;
```

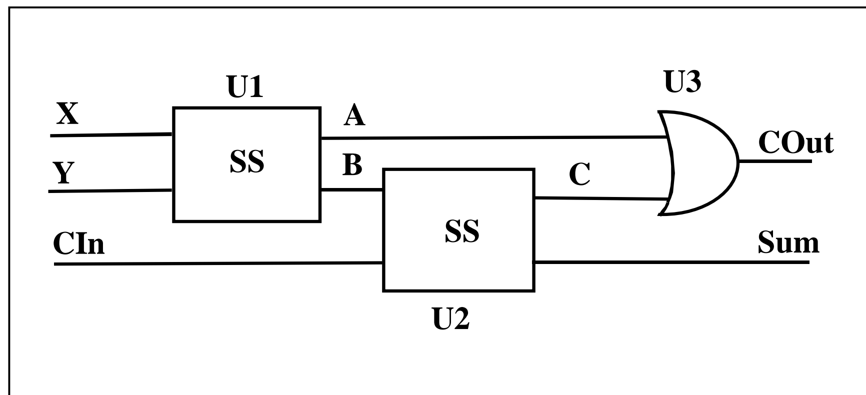


Figura 2-1 Esquema lógico equivalente a la descripción VHDL

Existen dos formas de asociar los puertos locales (los que aparecen en la declaración del componente) con los puertos actuales (aquellos que aparecen en cada copia de un componente).

La primera forma es la que se muestra en el ejemplo del sumador completo: asociación posicional. Se asocia cada puerto local con su puerto actual por la posición en que aparece. Por ejemplo, la instancia *U1* del semisumador conecta el puerto *X* del sumador a la entrada *I1* del semisumador, ya que aparece en la primera posición en la lista de puertos del componente.

La segunda forma es más explícita y proporciona más información. Se asocia cada puerto actual con su puerto local especificando los nombres de ambos. Si representamos la referencia *U2* del ejemplo anterior con la asociación por nombres tendremos:

```
U2 : semisumador port map (I1=>B, I2=>Cin, Cout=>C, Sum=>Sum);
```

Usando esta nomenclatura podemos especificar la asociación de puertos de forma independiente al orden que estos ocupan en la declaración del componente. De esta forma la siguiente referencia:

```
U2 : semisumador port map (Cout=>C, I1=>B, I2=>Cin, , Sum=>Sum);
```

es equivalente a la anterior.

En cualquiera de las dos notaciones existe la posibilidad de dejar puertos de salida desconectados usando la palabra reservada ***open*** en el puerto actual.

2.5.5.2 Configuración de un diseño

Un modelo VHDL tiene una única entidad, pero puede tener tantas arquitecturas como se quiera. Por supuesto, al simular el modelo debemos escoger cuál de las posibles arquitecturas va a ser usada. El mecanismo que permite relacionar entidad y arquitectura es la configuración *(**configuration**)*.

La configuración de un diseño puede aparecer en dos entornos distintos: puede aparecer en una arquitectura, entonces se llama especificación de configuración o puede aparecer como una unidad de diseño independiente, entonces se llama declaración de configuración.
Si dentro de la misma arquitectura, en la cual se usa referencia a otros componentes, se quiere indicar qué arquitectura se quiere utilizar para cada uno de los componentes referenciados, se puede usar la especificación de configuración. La sintaxis general para la especificación de configuración es la siguiente:

```
for (ref_componente {,...} |  others | all )  :  id_componente
     use  entity id_entidad [(id_arquitectura); |
     use  configuration  id_configuration;]
```

Usando la especificación de configuración, podemos escribir de nuevo el ejemplo del sumador completo indicando qué arquitectura usar para cada uno de los componentes que lo forman:

```
architecture  Estructura  of  Sumador_completo  is
...
declaraciones de los componentes
...
for all :  Semisumador use entity work.Semisumador (Estructural);
for U3 :  PuertaOr  use entity  work.PuertaOr (Logica);
begin
        U1 : Semisumador  port map  (X, Y, A, B);
```

```
        U2 : Semisumador  port map  (B, CIn, C, Sum);
        U3 : PuertaOr  port map  (A, C, COut);
    end  Estructura;
```

En el ejemplo se especifica que para todos aquellos componentes ***(for all)*** del tipo *Semisumador* se usará la arquitectura *Estructural* de dicho componente. También se especifica que para el componente *U3* del tipo *PuertaOr* se usará su arquitectura *Lógica.*

La especificación de configuración puede aparecer en la parte declarativa de una arquitectura o también en la parte declarativa de una sentencia ***block***. Puede ser tan simple como la que se ha visto en este ejemplo, el el cual únicamente se especifica que arquitectura debe usarse para cad componente, o bien puede introducir cambios en las conexiones de los puertos y en los valores de los genéricos de cada componente, o sea, puede contener una cláusula ***generic map*** y una cláusula ***port map***.

2.6 SUBPROGRAMAS

Los subprogramas se usan para escribir algoritmos reutilizables. En general un subprograma constará de una parte declarativa, en la cual definirá estructuras de datos locales al programa y una parte de sentencias (secuenciales) que describirán las operaciones.

Otra característica de un subprograma es su lista de parámetros, esto es, una lista que usa para especificar sobre qué datos externos al subprograma debe implementar su funcionalidad en el momento en que éste se llama para que se ejecute. Los parámetros que aparecen en la definición de un subprograma se llaman parámetros formales, mientras que los parámetros que aparecen en las llamadas a subprogramas sin los parámetros actuales.

VHDL ofrece dos tipos de subprogramas que a pesar de tener muchas características en común también presentan importantes diferencias, por lo tanto los vamos a estudiar por separado.

2.6.1 Funciones

Las funciones están orientadas a realizar cálculos, podemos pensar en ellas como en una generalización de las expresiones

```
function nombre_funcion [ (lista_parametros)] return tipo is
        {declaraciones}
begin
        {sentencias secuenciales}
end nombre_funcion;
```

Los parámetros formales de una función solamente pueden ser del tipo ***in*** (que es el tipo que toman por defecto) y la clase de objetos que pueden formar parte de dichos parámetros, por defecto se asume que son constantes.

En la parte declarativa de las funciones se pueden declarar todas aquellas estructuras de datos que se requieran (tipos, subtipos, constantes y variables), pero éstas solamente existirán cuando la función esté activa, y se crearán e inicializarán de nuevo cada vez que esta sea llamada. Por este motivo no se pueden declarar señales en la parte declarativa.
La parte declarativa de una función también puede contener definiciones de subprogramas; éstos, al igual que cualquier tipo de datos que se haya declarado, serán locales a la función y, por lo tanto, invisibles fuera de ella.

En las sentencias secuenciales que se encuentran en la parte de sentencias de una función siempre debe haber al menos una sentencia ***return***. Esta sentencia será la que retornará el único valor que una función puede devolver como resultado de su ejecución.

En la parte de sentencias de una función no se puede modificar (no se pueden realizar asignaciones) variables o señales externas a la función, aunque sí puede usarse para formar parte de expresiones utilizadas en la función. En la parte de sentencias de una función tampoco puede aparecer ninguna sentencia ***wait***. A continuación se muestra un sencillo ejemplo de función.

```
function  BOOL_TO_SL (X : boolean)
                return  std_ulogic  is
   begin
      if  X  then
         return '1';
      else
         return '0';
      end  if;
   end  BOOL_TO_SL;
```

Una vez definida esta función puede llamarse desde cualquier parte del modelo usando bien la llamada a función secuencial a la llamada a función concurrente. En ambos casos la llamada a función no será en si misma una sentencia sino que formará parte de una expresión.

2.6.2 Procedimientos

Los procedimientos ***(procedures)*** están orientados a realizar cambios en los datos que tratan, ya sea sus parámetros ya sean datos externos a los que pueden acceder. La sintaxis general es:

```
procedure  nombre_procedimiento  [(lista_parametros)]  is
        {parte declarativa}
begin
        {sentencias secuenciales}
end  nombre_procedimiento;
```

Los parámetros formales de un procedimiento pueden ser de tres tipos distintos***: in, out*****,** e ***inout*** , por defecto se consideran de tipo ***in***. La clase de objetos que pueden formar parte de los parámetros formales son constantes, variables y señales. Por defecto, los parámetros de tipo ***in*** se consideran constante, mientras que los de tipo ***out*** e ***inout*** se consideran variables.

Los parámetros de modo entrada ***(in)*** se usan para pasar valores al procedimiento, que éste puede usar, pero nunca puede variar su valor. Los parámetros de modo salida ***(out)*** son aquellos que el procedimiento solo puede usar realizando una asignación sobre ellos, o sea,

puede modificar su valor pero no usar o leer su valor. Por último, los parámetros de modo entrada/salida ***(inout)*** son aquellos que pueden usarse o leerse dentro del procedimiento u que también puede variarse su valor mediante asignación.

La parte declarativa de un procedimiento también puede contener definiciones de subprograams; éstos, al igual que cualquier tipo de datos que se haya declarado, serán locales al procedimeinto y, por lo tanto, invisibles fuera de él.

Un procedimiento retornará el flujo del programa al lugar donde fue llamado al llegar al final del mismo (a su sentencia **end**). Como puede haber ocasiones en que queramos salir de un procedimiento antes de llegar a su final, se puede usar la sentencia ***return*** en un procedimiento se usa sin ir acompañada de ninguna expresión e indica que debe devolverse el control del programa al punto de llamada.

La parte de sentencias de un procedimiento puede modificar (realizar asignaciones) a variables o señales externas al procedimiento. En las sentencias de un procedimiento pueden aparecer sentencias ***wait***. A continuación se muestra un ejemplo de procedimiento:

```
procedure  display_muxC
                    (alarm_time, current_time: : in digit;
                   show_a              : in std_ulogic;
signal  display_time : out digit) is
begin
   if (SHOW_A = '1') then
       display_time <= alarm_time;
  else
      display_time<= current_time;
  end if;
end  display_mux;
```

CAPÍTULO III:
DESCRIPCIÓN DE CIRCUITOS DIGITALES

El objetivo de este capítulo es el de detallar la descripción en VHDL de los componentes fundamentales de un sistema digital. Este conocimiento es imprescindible a la hora de asegurar que la implementación propuesta por la herramienta de síntesis coincide funcionalmente con la que se desea. La carencia de una metodología estándar de síntesis desde VHDL impide la descripción general del uso del lenguaje sin hacer referencia a particularidades y excepciones concretas propias de cada herramienta.

Se concretan también un conjunto de recomendaciones que facilitan al diseñador el aprovechamiento al máximo de las prestaciones de la herramienta de síntesis.

3.1 DESCRIPCIÓN DEL SISTEMA

El sistema digital a sintetizar se va describir mediante un arquitectura asociada a una determinada entidad en la que se definen los puertos del sistema. El proceso de síntesis generará una nueva arquitectura asociada a la misma entidad.

Tal como vimos en el Capítulo 2, la arquitectura VHDL está compuesta de las siguientes sentencias concurrentes:

- Sentencias de aserción concurrente que son ignoradas,
- Bloques
- Llamadas concurrentes a procedimientos,
- Asignaciones concurrentes a señal,

- Referencia a componentes
- Procesos, y
- Sentencias de generación que se sustituyen por el código iterativo equivalente

Salvo en lo que respecta a las sentencias de referencia a componentes, la jerarquía se elimina por defecto y todas las asignaciones de señal resultantes y llamadas a procedimientos se sustituyen por los procesos equivalentes.

3.2 LÓGICA COMBINACIONAL

En VHDL, la lógica combinacional puede ser descrita mediante asignaciones concurrentes de señal o procesos. Recordemos que para cualquier asignación concurrente existe un proceso equivalente. Un conjunto de asignaciones concurrentes de señal describen lógica combinacional siempre que:

a) La señal destino no interviene en la forma de onda de ninguna asignación.

Así, por ejemplo, la asignación:

a <= b **and** c **when** e =´0´ **else** ´1´;

se corresponde con lógica combinacional. Una posible implementación sería el circuito de la Figura 3-1.

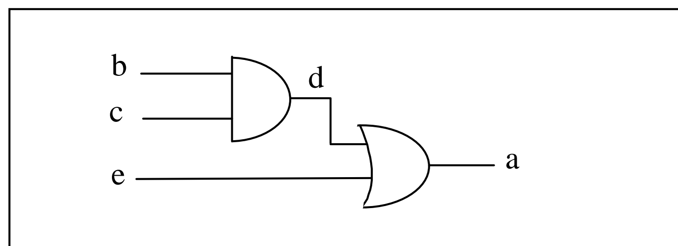


Figura 3-1. Implementación de asignación combinacional

Por el contrario, las asignaciones siguientes, se corresponden con lógica secuencial.

a <= b **and** c **when** a =´0´ **else** ´1´;

a <= b **and** c **when** e =´0´**else** a;

b) El conjunto de asignaciones concurrentes no contienen lazos combinacionales.

Así, el conjunto de asignaciones:

d <=b **and** c;

a <=d **or** e;

constituyen una descripción alternativa de la implementación de la Figura 3-1. Por el contrario, las asignaciones siguientes no, ya que la señal *a* se realimenta como entrada del circuito.

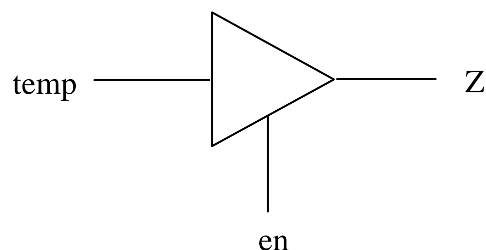
d <= b **and** a;

a <= d **or** e;

Tanto las asignaciones de señal seleccionadas como condicionales son soportadas. La asignación de señal seleccionada se corresponde funcionalmente con un multiplexor en el que la expresión se corresponde con la señal de control. Esta será la implementación final, siempre que durante el proceso de optimización no se encuentre una implementación alternativa mejor en términos de coste o velocidad.

A partir de los paquetes de síntesis, la utilización del valor *std_logic ´Z´*, implica la descripción de un buffer triestado:

z <= temp **when** (en =´1´) **else** ´Z´;



La segunda forma de describir lógica combinacional es mediante procesos. Un proceso describe lógica combinacional siempre que:

a) Todas las señales utilizadas en la forma de onda de alguna de las asignaciones secuenciales de señal o variable se incluyen en la lista de sensibilidad del proceso. Así, por ejemplo, el proceso siguiente:

```
process  (b, c, e)
   variable d: ...;
begin
   d := b and c;
   a <= d or e;
end  process;
```

cumple la condición. En algunas herramientas se admite igualmente el proceso equivalente al anterior, es decir, un proceso en el que la lista de sensibilidad es reemplazada por una sentencia *wait* equivalente:

```
process
   variable d: ...;
begin
   d :=b and c;
   a<= d or e;
   wait on b, c, e;
end process;
```

Esta es la única sentencia *wait* permitida en este tipo de procesos. En ambos casos la lógica descrita podría implementarse con el circuito de la Figura 3-1.

Algunas herramientas de síntesis ignoran la lista de sensibilidad. El proceso se considera combinacional cuando carece de sentencias de espera y secuencial en caso contrario.

b) Bajo cualquier condición de ejecución del proceso, todas las variables y señales son asignadas. Así, por ejemplo, el proceso siguiente constituye una descripción alternativa a la implementación de la Figura 3-1.

```
process  (b, c, e)
   variable d: ...;
begin
    if  (b = ´1´)  then
        d :=c;
   else
        d :=´0´;
   end if;
         a <= d  or e;
end process;
```

Por el contrario, en el proceso siguiente la variable “d” no cumple la condición:

```
process  (b, c, e)
   variable d: ...;
begin
    if  (b=´1´)  then
        d :=c;
   else if;
       a<= d or e;
end process;
```

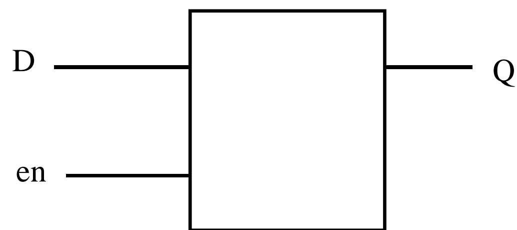
La descripción combinacional mediante procesos es mucho más flexible que mediante asignaciones a señales concurrentes. La causa de este hecho reside en la mayor flexibilidad del código secuencial en VHDL que la del código concurrente. Mientras la estructura de una asignación a señal seleccionada está fija, su equivalente secuencial, la sentencia ***case***, permite incluir cualquier tipo de código secuencial en cada alternativa. Esta mayor flexibilidad es posible encontrarla también en la sentencia ***if ... then... else*** frente a la sentencia de asignación condicional. En consecuencia, aunque también aquí es posible encontrar una relación funcional entre el código y el *hardware*, esta relación suele ser menos directa que en el caso concurrente.

3.3 LÓGICA SECUENCIAL

3.3.1 Descripción de "latches"

La mayoría de las herramientas infieren *latches* cuando en un proceso se relaja la condición b). Un *latch* se puede inferir de una sentencia ***if*** no especificada totalmente. Así, por ejemplo, el código siguiente sería la descripción explícita de un *latch* tipo D.

```
latch: process (D, en)
    begin
        if  (en = ´1´)  then
            Q<= D;
        end if;
    end process  latch;
```



3.3.2 La señal de reloj

Se identifican como señales de reloj aquellas señales de comportamiento correcto con especificación de evento y flanco. La forma más usual es:

```
Reloj ´EVENT and  reloj = ´1´
```

Sobre la señal de reloj se imponen usualmente una serie de condiciones de uso que la caracterizan como una señal de comportamiento correcto. Las más usuales son las siguientes:

- Generalmente, sólo se admite la especificación de un evento por proceso (o bloque) lo que se traduce en un único reloj por proceso. La siguiente descripción sería, por tanto, incorrecta:

```
process ( ... )
        if (reloja ´event and reloja= ´1´) then
        ...
        end if;
        if (relojb´event and relojb= ´1´) then
        ...
        end if;
end process;
```

- Cuando la señal de reloj se utiliza en una construcción condicional, no se pueden especificar acciones en la cláusula *else*. La siguiente descripción sería por, por tanto, incorrecta:

```
if ( reloj ´ event and reloj = ´1´) then
    x <= a;
else
    x <= b;
end if;
```

- La especificación de reloj no se puede utilizar como operando. La siguiente descripción sería, por tanto, incorrecta:

```
if not (reloj ´event and reloj = ´1´) then ...
```

3.3.3 Registros

Se identifican como registros aquellas señales cuya asignación depende de una señal de reloj. Puede utilizarse un proceso con sentencia de espera, como en los siguientes ejemplos:

```
process
begin
     wait until (reloj = ´1´);
       Q<=D;
```

```
end process;

process
begin
    wait until  (reloj ´event  and  reloj = ´1´);
        Q<D;
end process;
```

En un proceso con lista de sensibilidad, ésta puede ser utilizada para especificar el evento de la señal de reloj:

```
process  (reloj)
begin
    if  (reloj ´event  and  reloj = ´1´)  then
        Q<D;
        end  if;
end  process;
```

Este último estilo admite la inclusión de señales de *set* y *reset*, ya sean síncronas:

```
process  (reloj)
begin
    if  (reloj ´event  and  reloj = ´1´)  then
        if  (reset = ´1´)  then
        Q<´0´;
        else
        Q<D;
        end if;
    end if;
end process;
```

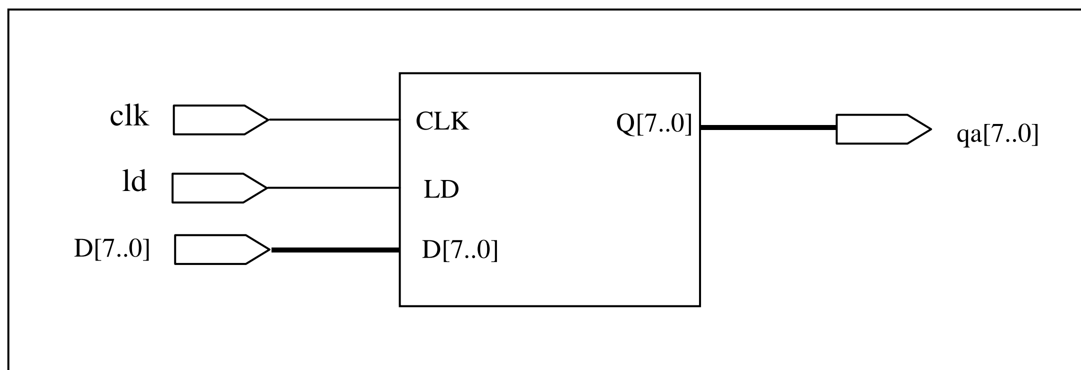
o asíncronas:

```
process  (reloj, reset)
begin
    if  (reset = ´1´)  then
        Q<´0´;
    elsif (reloj ´event  and  reloj = ´1¨)  then
        Q<D;
    end if;
end process;
```

3.3.4 Contadores

Los contadores representan otro elemento muy frecuente en diseño digital. La forma más usual para describirlos en VHDL es mediante operaciones de incrementación y/o decrementación. A continuación se incluyen la implementación, mediante la sentencia ***if***, de varios contadores de 8 bits, controlados por las señales *clk, clear, ld, enable,* y *up/down*.

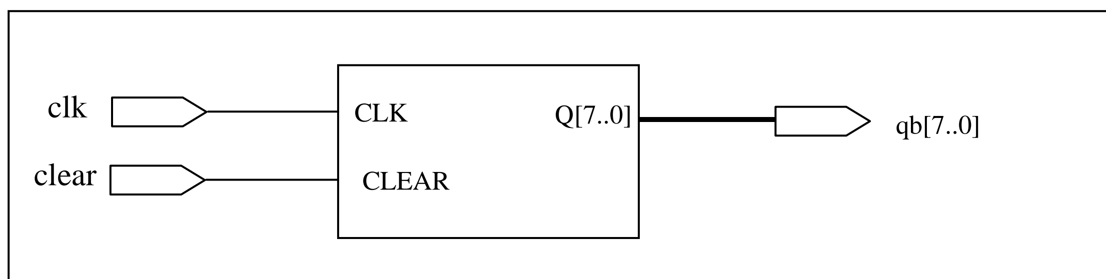
a) Contador síncrono con carga paralelo *(ld)*




```
process (clk)
    variable  cnt  : integer range  0  to 255;
begin
   if  (clk ´event  and  clk =´1 ´)  then
       if  ld = ´0´  then
```

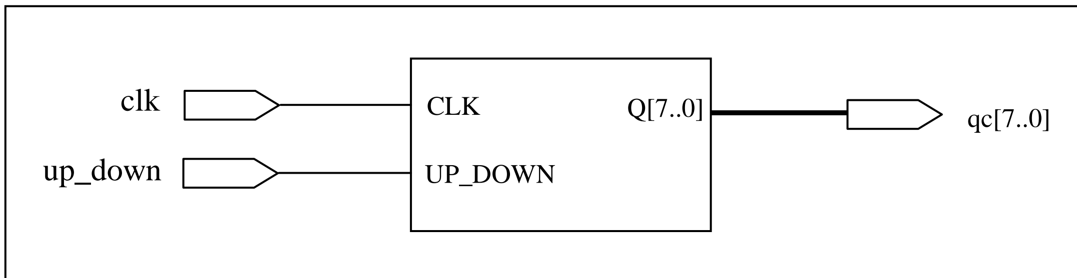
```
                cnt := d;
            else
                cnt := cnt +1
            end if;
        end if;
            qa <=  cnt;
    end process;
```

b) Contador síncrono con *clear*



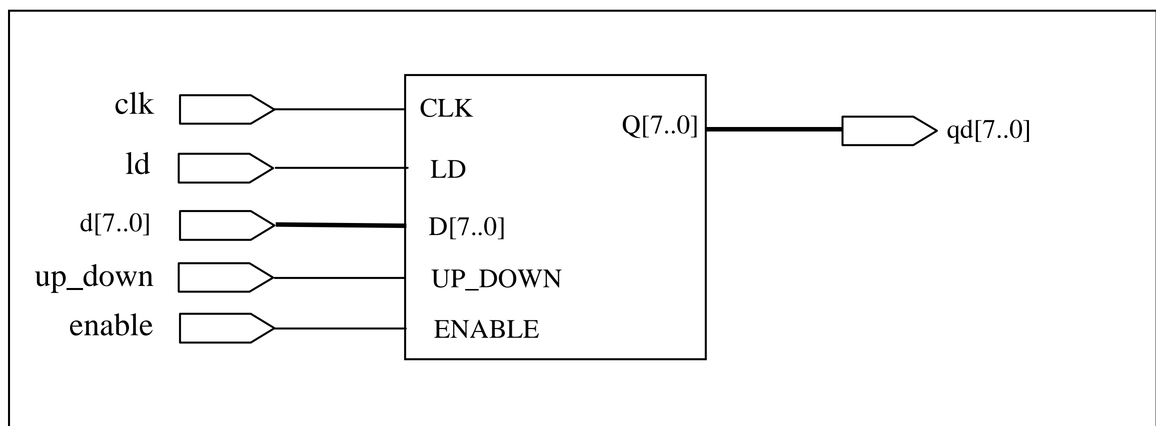

```
process (clk)
    variable  cnt  : integer range 0  to 255;
begin
   if  (clk ´event  and  clk =´1´)  then
      if  clear = ´0 ´ then
          cnt := 0;
      else
          cnt := cnt +1
      end if;
   end if;
    qb <=  cnt;
end process;
```

c) Contador síncrono up/down




```
process (clk)
    variable  cnt        : integer range  0  to 255;
    variable  direction : integer;
begin
if  (up_down = ´1´)  then
    direction := 1;
else
    direction := -1;
end if;
if  (clk ´event  and  clk =´1 ´)  then
    cnt :=  cnt + direction;
end if;
qc <=  cnt;
end process;
```

d) Contador síncrono con *enable* , *load (ld)*, *up_down*



```
process (clk)
    variable  cnt          : integer range  0  to 255;
    variable  direction  : integer;
begin
      if  (up_down = ´1´)  then
          direction := 1;
     else
          direction := -1;
     end if;

      if  (clk ´event  and  clk =´1 ´)  then
          if  ld = ´0 ´ then
            cnt := d;
          else
             if  enable = ´1´ then
                 cnt :=  cnt + direction;
             end if;
          end if;
     end if;
     qd <=  cnt;
end  process;
```

3.3.5 Descripción de FSMs

Existen varios estilos de descripción de FSMs. Cada uno de ellos será el más apropiado, dependiendo de la herramienta y del tipo de máquina que se quiera describir. Los resultados de la síntesis en cada caso van a ser diferentes.

La herramienta de síntesis MAX+PLUS describe las FSMs utilizando un único proceso secuencial con las señales de reloj, *reset*, registros de estado y entradas en la lista de sensibilidad. El código secuencial se divide en cuatro partes diferenciadas de las que se infiere distinto tipo de lógica:

```
architecture  estilo_Altera  of  FSM  is
    type  estados  is  (s0, ...sp);
    signal  estado: estados := s0;            - registro de estados
begin
Sm: process  (reset, reloj, estado, x1, ...xn)
    begin

       •                               - lógica combinacional
       •
      if  (reset = ´0´)  then
        estado <= s0;                         - reset asíncrono
      elsif  Rising_Edge (reloj)  then
        case  estado is
           when  s0 =>               - estado s0
              •                      - acciones del estado s0
              •                      - transiciones de estado y
              •                      - asignaciones a registro de salida
           when sp =>              - estado sp
              •                    - acciones del estado so:
              •                      - transición de estado y
              •                     - asignaciones a registro de salida
        end  case;
      end  if;
       •
       •                    - lógica combinacional
    end  process  sm;
end estilo_Altera;
```

Este estilo es el más flexible, ya que permite describir en un único proceso lógica combinacional, lógica secuencial e inicialización síncrona o asíncrona de los elementos de memoria. De hecho , todas las asignaciones fuera del cuerpo de la sentencia:

if (reset = ´0´) **then ... elsif** Rising_Edge (reloj) **then ...end if;**

tanto al principio como al final del proceso, van a producir lógica combinacional. Las asignaciones en el cuerpo de la sentencia:

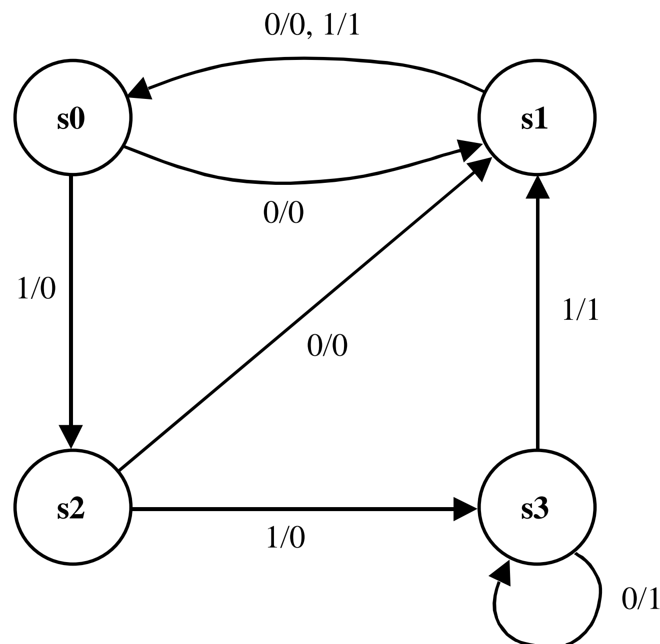
if (reset = ´0´) **then...**

van a corresponderse con la inicialización asíncrona. Las asignaciones en el cuerpo de la sentencia:

elsif Rising_Edge (reloj) **then...**

van a corresponderse con la carga síncrona de los elementos de memoria. En esta sección podría incluirse la inicialización síncrona de los elementos de memoria.

- **Ejemplo:** Dada la máquina definida por el diagrama de estados de la Figura realizar la descripción de la misma siguiendo el estilo explícito anterior.




```
architecture ejemplo of FSM is
    type estados is (s0, s1, s2, s3);
    signal estado: estados := s0;          - registro de estados
begin
sm: process (reset, reloj, estado, x)
    begin
       case estado is                      - lógica combinacional
          when s1 =>
             z <= x;
          when s3 =>
```

```
            z  <= ´1´;
        when others =>
            z  <= ´0´;
      end case;
      if  (reset = ´0´)  then
        estado <= s0;                    - reset asíncrono
      elsif  Rising_Edge (reloj)  then
        case  estado is
          when  s0 =>               - estado s0
            if  (x =´0´)  then       -  asignación de estado próximo
                estado <= s1;
            else
                estado <= s2;
            end  if;
          when  s1 =>               - estado s1
                estado <= s0;       - asignación de estado próximo
          when  s2 =>                  - estado s2
            if  (x =´0´)  then       -  asignación de estado próximo
                estado <= s1;
            else
                estado <= s3;
            end  if;
          when  s3 =>                  - estado s3
            if  (x =´1´)  then       -  asignación de estado próximo
               estado <= s1;
            end  if;
        end case;
      end if;
    end  process sm;
end ejemplo;
```

3.4 RECOMENDACIONES GENERALES

El código VHDL desarrollado como descripción de la arquitectura RT a sintetizar va a tener, adicionalmente, otras aplicaciones, como verificación funcional por simulación, documentación y mantenimiento, etc. En parte o en todo, el código debe ser reutilizable en proyectos futuros al objeto de minimizar el esfuerzo de diseño necesario.

Dependiendo de la aplicación, caben recomendaciones particulares que permiten optimizar el código a desarrollar. Así, se pueden proponer recomendaciones orientadas a minimizar el tiempo de simulación, mejorar la legibilidad del código que favorezca el mantenimiento del mismo y sus uso en la documentación del proyecto o que faciliten su documentación posterior. Algunas de estas recomendaciones pueden ser contradictorias, de tal forma que dependiendo, dependiendo de la aplicación más importante, habrá que establecer una prioridad entre ellas.

En este capítulo suponemos que la aplicación más importante es síntesis, que por otro lado, va a constituir de hecho la situación más frecuente en la mayoría de los casos. En consecuencia, las recomendaciones que hagamos estarán dirigidas a mejorar los resultados de la herramienta de síntesis que se utilice.

3.4.1 Recomendaciones para síntesis

Vamos a hacer a continuación una breve descripción de las recomendaciones de carácter general que hay que tener en cuenta a la hora de asegurar una calidad mínima en el código VHDL para la síntesis que asegure resultados satisfactorios.

3.4.1.1 Descripción *"hardware"*

Las herramientas comerciales cubren las etapas de síntesis RT y lógica. En consecuencia, el diseño arquitectual corresponde actualmente al diseñador. Así pues, la arquitectura del sistema compuesta por unidades de control y de datos, máquinas de estados finitos, unidades operacionales, interconexiones, entradas y salidas, etc., deben de decidirse adecuadamente al objeto de asegurar con el mayor grado de certidumbre posible que la implementación final va a satisfacer los requerimientos de área, velocidad, consumo, etc., etc., de forma óptima. Unicamente cuando el diseño arquitectual se haya completado cabe generar el código VHDL correspondiente. Abordar el diseño del sistema directamente en VHDL no es recomendable.

Aunque VHDL tenga similitudes sintácticas con lenguajes de programación (particularmente ADA), se trata de un lenguaje de descripción de *hardware* . La descripción VHDL refleja una arquitectura. Si esta no ha sido cuidadosamente diseñada, los resultados de la síntesis no van a ser óptimos. Código elegante desde el punto de vista de programación puede ser ineficiente en

síntesis. En este sentido la utilización de código simple y fuertemente ligado a la implementación va a asegurar la obtención de resultados óptimos.

3.4.1.2 Limpieza del código

El código VHDL generado para síntesis debe ser lo más claro posible al objeto de asegurar al máximo el control sobre el resultado. En este sentido cabe hacer las siguientes recomendaciones:

- Utilizar al máximo tipos estándar reconocidos e implementados eficientemente por la herramienta de síntesis. Salvo en el caso de modelar *buses,* la utilización de tipos enteros restringidos es recomendable.

- No usar variables para modelar registros. Los elementos de memoria constituyen recursos *hardware* que deben decidirse antes de la síntesis y declararse como señales en la descripción VHDL. El uso de variables es recomendable frente al de señales, por razones de legibilidad y eficiencia en la simulación.

- Tanto por razones de legibilidad del código como de eficiencia en síntesis, las sentencias *case* son preferibles al encadenamiento de sentencias *if..then..else*. Utilizar la sentencia *elsif* en vez de *elese if*. Sin embargo, es preferible utilizar sentencias limpias del tipo:

```
if  (a= ´1´)  then
      ...
else
    ...
end if;
```

que:

```
if  (a= ´1´)  then
      ...
elsif  (a= ´0´)  then
    ...
else
   ....
end  if;
```

- Por razones de legibilidad, es recomendable dar el mismo nombre a las referencias de los componentes que los de las entidades con las que se asocian.

3.4.1.3 Limpieza del código

Al objeto de sacar el máximo partido de una determinada herramienta de síntesis, es necesario conocer en detalle la metodología de síntesis que esta herramienta utiliza. Concretamente, los algoritmos de síntesis que implementa y su eficacia.

IV. BIBLIOGRAFÍA

[1] LLUÍS TERÉS, YAGO TORROJA, SERAFÍN OLCOZ, EUGENIO VILLAR. VHDL: Lenguaje Estándar de Diseño Electrónico. McGraww-Hill. 1998

[2] Institute of Electrical and Electronics Engineers: *The IEEE Standard VHDL Language Reference Manual,* IEEE Std 1076- 1987, 1988.

[3] Institute of Electrical and Electronics Engineers: *The IEEE Standard VHDL Language Reference Manual,* ANSI/IEEE Std 1076- 1993, 1994.

[4] P.J. ASHENDEN: *The Designer´s Guide to VHDL,* Morgan Kaufmann Publishers, Inc., 1995

[5] R. LIPSETT, C. SCHAEFER, C. USSERY: VHDL: *Hardware Description and Design,* Kluwer Academic Publishers, 1989.

[6] S. MAZOR, P. LANGSTRAAT: *A Guide to VHDL,* Kluwer Academic Publishers, 1993

[7] FRANK SCARPINO. *VHDL and AHDL Digital System Implementation.*Prentice Hall. 1998.

[8] ALTERA MAX + PLUS II VHDL. 1996