VHDL PARA DISEÑO Y VALIDACIÓN DE SISTEMAS DIGTALES

Jonathan Noel Tombs

Miguel Ángel Aguirre Echánove

Fernando Muñoz Chavero

Hipólito Guzmán Miranda

Universidad de Sevilla

Mayo de 2016

Contenido

CAPÍTULO I. LENGUAJES DE DESCRIPCIÓN DE HARDWARE **3**
1. Introducción histórica 3
2. Los HDL's en la metodología de diseño. 5
3. Jerarquía de un diseño. 7
4. Niveles de abstracción de un HDL 8
5. HDL: Programa o diseño. 8
6. Lenguajes HDL 10
7. Síntesis de Circuitos. 11

CAPÍTULO II. ESTRUCTURA DE UN BLOQUE FUNCIONAL EN VHDL **14**
1. Cuestiones previas 14
2. Un primer ejemplo en VHDL 14
3. Estructura de un diseño VHDL 18
4. La sección ENTITY 19
5. La sección ARCHITECTURE 22
6. La sección CONFIGURATION 25
7. La sección LIBRARY 27

CAPÍTULO III. DESCRIPCIÓN DE LA FUNCIONALIDAD **30**
1. Concepto de concurrencia. 30
2. Estructura de un proceso 31
3. Bloques de sentencias 37
4. Asignaciones concurrentes 38
5. La sentencia generate 40

CAPÍTULO IV. TIPOS DE DATOS Y SEÑALES **42**
1. Definición de tipos de datos. 42
2. Tipo entero 42
3. Tipo matriz 43
4. Definición de tipos compuestos 45
5. Tipos simples 46
6. La librería IEEE 49

CAPÍTULO V. SUBSISTEMAS DIGITALES. EJEMPLOS DE DISEÑO. **53**
1. Codificación de Sistemas Síncronos 53
2. Codificación de sistemas aritméticos. 63
3. Codificación de módulos de memoria. 67

CAPÍTULO VI. VHDL NO SINTETIZABLE. SIMULACIÓN Y MODELADO **70**
1. Mecanismo de simulación. 70
2. Sentencias para modelado. 71
3. Construir un "Test Bench". 74
4. La librería TextIO 76

CAPÍTULO VII. BUENAS PRÁCTICAS CODIFICANDO EN VHDL **82**
7.1 Prácticas de codificación 83
7.2 Tratamiento de la señal de reloj 85
7.3 Reglas para una síntesis segura 88
7.4 Facilitar la revisión de un código 91
7.5 HDL en estándares 94
Bibliografía 95

CAPÍTULO I. LENGUAJES DE DESCRIPCIÓN DE HARDWARE

Este tema tiene por objeto dar una introducción general a los Lenguajes de Descripción de Hardware. Quiere aportar una visión de conjunto acerca del significado de en qué consiste diseñar grandes sistemas electrónicos digitales utilizando los HDL, y acerca de sus herramientas. Tras una breve introducción histórica se procede a analizar el salto cualitativo de diseñar con esquemáticos a diseñar en HDL. Posteriormente introduciremos el concepto de jerarquía de un diseño y de nivel de abstracción. Finalmente se describe en qué consiste una herramienta de síntesis de circuitos mediante HDL, elemento crucial en la nueva metodología para diseñar.

1. Introducción histórica

La necesidad de construir circuitos digitales cada vez más complejos es patente día a día. Ya en el siglo XXI somos capaces de construir microprocesadores de muy altas prestaciones que están compuestos por millones de unidades funcionales (transistores) que realizan tareas de gran responsabilidad en la sociedad. Ingeniería aeroespacial, seguridad bancaria, automoción, identificación electrónica, gestión de recursos,…

En la práctica, el 100% de la electrónica de control y supervisión de los sistemas, elaboración de datos y transferencia de los mismos se realiza mediante circuitos integrados digitales, constituidos por una gran cantidad de transistores: son los llamados circuitos integrados de muy alta escala de integración, o VLSI.

Si en los años cincuenta y sesenta, en los albores de la electrónica integrada los circuitos eran esencialmente analógicos, en los que el número de elementos constituyentes de los circuitos no pasaba de la centena, en la actualidad el hombre dispone de tecnologías de integración capaces de producir circuitos integrados con millones de transistores a un coste no muy elevado, al alcance de una PYME. A mediados de los años sesenta Gordon E. Moore ya vaticinaba un desarrollo de la tecnología planar en el que cada año la escala de integración se doblaría, y de la misma manera aumentaría la capacidad de integrar funciones más complejas y la velocidad de procesamiento de esas funciones. Las predicciones de Moore se han cumplido con gran exactitud durante los siguientes 30 años, y que la tendencia continuará durante los próximos 20. En el año 2012 Intel esperaba integrar 1000 millones de transistores funcionando a 10GHz.

Si bien construir estos circuitos parece una cuestión madura, diseñarlos supone un serio problema. La microelectrónica digital continúa por el camino de la automatización, con herramientas específicas, como son los simuladores digitales o los generadores automáticos de layout, que resuelven el problema de la construcción del circuito y la verificación del mismo. Se utilizan, pues, fundamentos de la ingeniería de computación o CAE en las que se delegan en

herramientas software las tareas de manejo de grandes cantidades de información, bases de datos que, de forma óptima contienen la información acerca de la funcionalidad del circuito, de su geometría y de su conexiones así como de su comportamiento eléctrico. Si bien, por un lado la electrónica digital supone una simplificación funcional de un comportamiento analógico, con vistas a poder manejar grande volúmenes de información.

Con editores de esquemas somos capaces de dar una visión muy precisa y completa del diseño rápidamente, sin embargo esta metodología de captura queda circunscrita casi al mundo analógico.

Conforme al aumento de la complejidad de los circuitos digitales las prestaciones de los editores de esquemas no eran suficientes para responder a una capacidad de diseño tan elevada., que requiere una resolución explícita de los circuitos. Editar un esquema requiere, por tanto, un esfuerzo de desarrollo muy alto.

A principios de los años 90 Cadence Design Systems, líder mundial en sistemas de CAE para microelectrónica, propone el Verilog, un lenguaje alfanumérico para describir los circuitos de forma sencilla y precisa utilizando una sintaxis C: es el primer lenguaje de descripción de hardware en sentido amplio como veremos en epígrafes posteriores. Otros fabricantes de hardware habían propuesto un lenguaje más centrado en la resolución de un problema concreto: generación de una función para un dispositivo programable, resolución del circuito de una máquina de estados finitos a partir de su descripción de la evolución de los estados,... etc. Nacen los conceptos de descripción de alto nivel y de síntesis lógica, que posteriormente formalizaremos.

En el año 1982 el Departamento de Defensa de los Estados Unidos promueve un proyecto para desarrollar un lenguaje de descripción (conocido como MIL-STD-454L) de hardware que:

- Describiera los circuitos digitales de forma amplia: Funcionalidad, tecnología y conexionado
- Permitiera describir y verificar los circuitos a todos los niveles: funcional, arquitectural y tecnológico (posteriormente matizaremos estas tres categorías).
- Describiera la tecnología misma, para poder diseñar circuitos que sean independientes de la propia tecnología o bien durante la puesta a punto del proceso de fabricación.
- Describiera modelos del entorno en el que se va a insertar el circuito de forma que hubiese unas posibilidades de verificación más amplias del propio circuito.

El lenguaje resultante es el VHDL, que responde a las siglas VHSIC HDL (Very High Speed Integrated Circuits, Hardware Description Language), y es ratificado por el Instituto para la Ingeniería Eléctrica y Electrónica (IEEE, en 1987) en la norma IEEE-1076. Aunque en este sentido el Verilog cumple las propuestas anteriormente anunciadas, el VHDL se impone como

lenguaje estándar de diseño. Posteriormente veremos diferencias generales entre uno y otro. Por el momento nos referiremos a los HDL's como lenguajes alfanuméricos comprensibles para describir circuitos electrónicos en sentido amplio. En primer lugar veremos cuál ha sido la aportación de los HDL's en la metodología clásica de diseño.

2. Los HDL's en la metodología de diseño.

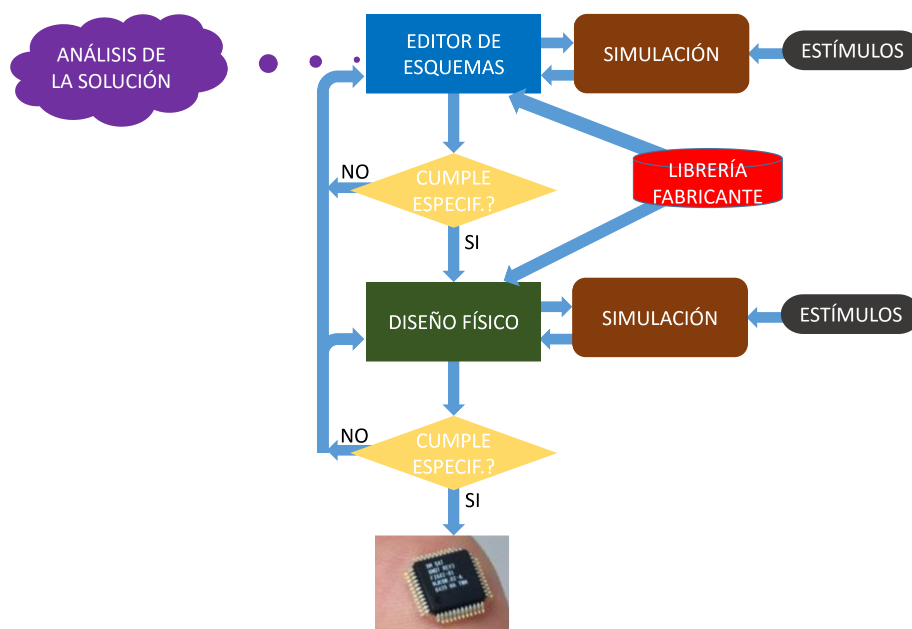


Ilustración 1. Metodología Clásica

2.1. Metodología clásica.

La secuencia de herramientas informáticas que procesan un diseño desde su descripción en un método fácilmente comprensible por el hombre hasta la información final, útil para la fabricación del dispositivo se llama *flujo de diseño*. La figura representa un flujo general que puede incluir más herramientas.

El proceso de elaboración de un diseño es un proceso de análisis jerárquico. Suelen ser diagramas de bloques y las señales que los conectan estableciendo las relaciones entre ellos. A su vez estos bloques son descompuestos en otros más simples, de menor rango en la jerarquía. Este proceso continúa hasta alcanzar funciones basadas en los elementos de menor entidad. A este proceso se le denomina descomposición *Arriba-Abajo* (Up-Bottom), usualmente realizado sin la ayuda de ninguna herramienta informática.

Terminada la etapa de decisiones, se procede a la introducción del diseño. Si utilizamos librerías de elementos básicos proporcionadas por los fabricantes (también llamados *primitivas*) construimos los primeros elementos de la jerarquía a partir de estas primitivas creando y verificando estas funciones del primer estadio jerárquico. Estos primeros bloques se incorporan a la librería de elementos que constituirán nuestro diseño.

Seguidamente construimos y verificamos el segundo nivel, tercero, cuarto, ... hasta llegar al nivel más elevado, que es nuestro diseño y las interconexiones con el sistema exterior. Esta es la fase de la metodología *Abajo-Arriba* (Bottom-Up). Cada unidad funcional es incorporada a la librería de diseño de forma que varias funciones podrían hacer uso de una misma función de inferior rango jerárquico. Este es el concepto denominado *reusabilidad* de una función.

2.2. Metodología basada en HDL's

La introducción de los HDLs como métodos de descripción de circuitos han enriquecido el proceso de creación de un diseño, lo han acelerado y asegurado, abaratando los costes de desarrollo, o permitiendo abordar diseño más complejos.

La figura 2 muestra la modificación del flujo clásico cuando se utiliza esta técnica para la introducción del diseño. Existen varias mejoras sustanciales:

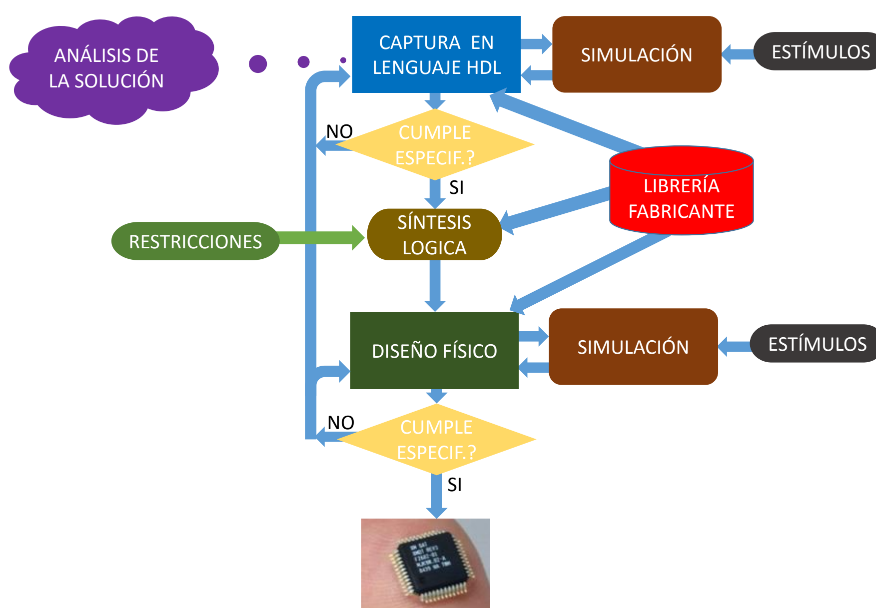


Ilustración 2. Metodología utilizando HDL

La librería de diseño es independiente de la tecnología para la que se diseña, por lo que la reusabilidad de los módulos constituyentes es total. No solo se puede compartir entre diseños sino que puede ser compartido por diferentes procesos de fabricación. Es independiente de la tecnología. El código introducido es de *Alto Nivel* y la simulación para su comprobación es asimismo de *Alto Nivel*. Solamente cuando se obtiene una imagen física del diseño puede predecirse con cierta certeza si cumple o no cumple las especificaciones. El programa de síntesis ha de alimentarse con el diseño, las condiciones de contorno en que funcionará, y la tecnología disponible del fabricante. El resultado será un código HDL de *Bajo Nivel* o lo que es lo mismo una representación del circuito alfanumérica compuesta por primitivas y sus conexiones, lo que se conoce como una *netlist*. Esta *netlist* se podría representar en un plano esquemático, pero no encontraríamos una ordenación racional y comprensible entre las primitivas y sus conexiones. Se utiliza como base de datos para alimentar el flujo de síntesis física.

La penalización ha sido la pérdida de cierto control en la generación del circuito, ya que en la nueva metodología hay una fase de *síntesis* automática en la que se cede a una herramienta software la responsabilidad de la resolución del circuito final. Nos acercamos a la situación ideal de la obtención totalmente automática del circuito a partir de las especificaciones.

En la práctica esta aproximación es la más utilizada en la que el diseñador ha cedido la responsabilidad de diseño de módulos regulares, sin la necesidad de verificar su correcta implementación. Por ejemplo un multiplicador editado en un esquema a base de puertas lógicas elementales (primitivas) había de ser verificado ante prácticamente todas las posibles combinaciones de las entradas. Mediante lenguajes de alto nivel la sentencia *a*b* representa todo el circuito multiplicador correctamente generado automáticamente por un programa.

3. Jerarquía de un diseño.

En este estadio del desarrollo del tema es fácil comprender la idea de descomposición jerárquica de un diseño. Un diseño de gran dimensión, típico de un circuito microelectrónico digital requiere una buena organización del diseño, una descomposición razonada en módulos de inferior rango, que se suceden con una descripción cada vez más detallada de sus funciones, los llamados bloques funcionales. En realidad se sigue la conocida táctica de *divide y vencerás*.

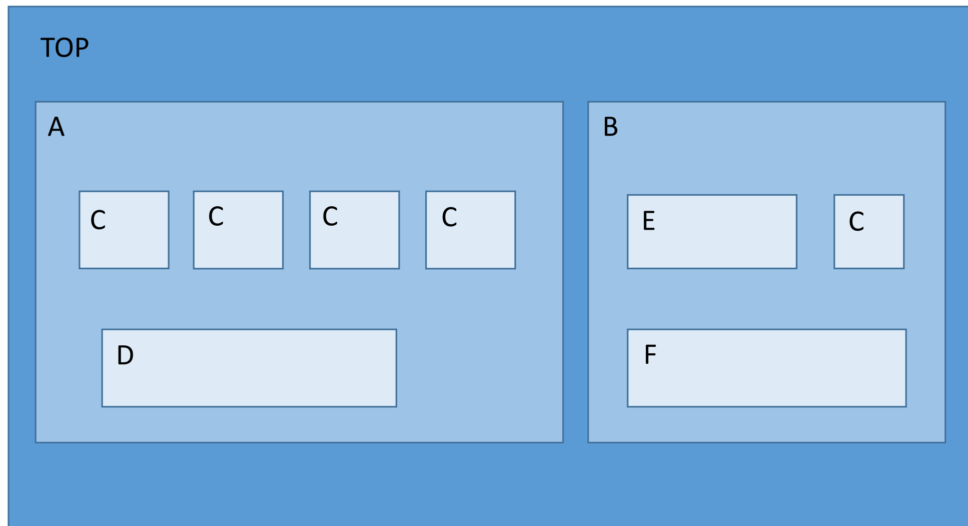


Ilustración 3. Ejemplo de descomposición jerárquica

4. Niveles de abstracción de un HDL

Como se ha comentado un HDL puede expresar un mismo diseño bajo diversos puntos de vista, son los llamados **niveles de abstracción**. El más elevado es una descripción en la que se define el comportamiento del circuito, es el llamado nivel **comportamental** (*behavioral*). El más bajo es un modelo íntimamente relacionado con la tecnología compuesto por primitivas y conexiones, representando la estructura del circuito, es el llamado nivel **estructural** (*structural*). También llamaremos nivel estructural al que expresa una interrelación entre bloques funcionales del mismo nivel jerárquico. Posteriormente veremos que podemos formar estructuras intermedios o **mixtos** compuestos por mezclas de ambos, en el que las unidades de un nivel estructural no son primitivas, sino que son módulos de rango jerárquico inferior.

5. HDL: Programa o diseño.

El lector habrá notado que hemos procurado eludir la palabra “programa” a la hora de referirnos a la elaboración de un código HDL, ya que nos referiremos a él siempre como “diseño HDL”. Como programa conocemos a una secuencia ordenada de comandos y funciones, que realizan unas tareas definidas en una computadora y la propia secuencia determina un comportamiento u otro. Estos comandos y funciones están expresados en un lenguaje alfanumérico que es fácilmente comprensible por el hombre, pero que tiene una proyección directa sobre el lenguaje comprensible por la computadora, y por tanto, fiel a su arquitectura. La traducción de uno a otro la realiza un programa compilador. .

Durante la ejecución del programa se procesan una tras otra las instrucciones secuencialmente condicionadas por la arquitectura de la Unidad Centra de Proceso (CPU).Cualquiera que sea su arquitectura, La estructura del programa toma por tanto la forma de una **secuencia** de instrucciones, accesos a memoria y saltos en flujo de ejecución.

Los lenguajes de descripción de hardware, HDLs, realizan aquello para lo que están diseñados, es decir, para describir un circuito o parte de un circuito hardware: Describen un comportamiento propio de un circuito en sentido amplio y por tanto no existen precedencia entre la activación de una parte del mismo u otra. Representan un comportamiento inherentemente **paralelo**. Esto tiene un conjunto de implicaciones que han de tenerse en cuenta a la hora de elaborar el diseño que particularizaremos al caso del VHDL. La primera implicación es la relación entre el lenguaje y la plataforma de desarrollo, que es una computadora y tiene el comportamiento descrito en el párrafo anterior. Por tanto simulación y síntesis han de estar provistas de mecanismos o artificios que emulen el paralelismo y que inciden directamente en la formulación del código. La segunda implicación es en un cambio de mentalidad en el diseñador a la hora de elaborar la descripción del diseño, estando habituado a realizar programas software. El mencionado ahorro en este caso exige une cuidadosa planificación del diseño previa a su elaboración. Existen usos típicos del programador de software que están absolutamente prohibidos en la formulación de un diseño hardware.

Un caso típico es utilizar variables auxiliares para almacenar temporalmente un valor. Posteriormente es posible reutilizar de nuevo la misma variable en otro punto del programa. Este uso es muy poco aconsejable si estamos diseñando circuitos, ya que una variable de este estilo sería un registro, con diferentes accesos y salidas, implicaría una compartición del mismo registro por varios módulo... en esencia algo poco recomendable. Sin embargo en un software es útil, ya que una variable así es una posición de memoria que puede accederse cuantas veces sea preciso y si se reutiliza el programa resultante es más compacto.

```
int aux;
....
aux=a+b;
...
c=función1(aux, ...);
...
aux=a*b;    //reutilización de la variable
...
j=función2(aux, ...);
```

Otro caso muy corriente es la utilización de módulos aritméticos. En un diseño HDL, una suma o un producto implica necesariamente la síntesis de un circuito digital suma o producto. Cuando se declara la misma operación varias veces en software se utilizan el sumador y el multiplicador de la CPU exhaustivamente, sin más. En HDL el resultado sería un circuito con múltiples operadores que ocuparían gran cantidad de área de silicio. Es importante, por tanto, “ayudar” al programa de síntesis a elaborar el diseño definiendo una estructura que defina el circuito a nivel modular, lo que más adelante llamaremos diseño **rtl** (register transfer level), como nivel de abstracción adecuado para diseñar.

Otra situación muy típica es la evaluación secuencial de asignaciones:

```
a=a * b;
```

El procesador evalúa a * b y almacena el resultado en un registro propio, para posteriormente transferir el valor a la posición de a. Esto es algo impensable en hardware ya que se trata de describir un multiplicador cuya salida es la entrada, es decir, una realimentación combinacional.

Queda, pues, comentar el proceso que realiza una herramienta de síntesis a la hora de elaborar el circuito a partir de nuestra descripción del hardware:

1. Generación del código en un lenguaje paralelo, comprensible por el usuario (VHDL, Verilog, ABEL, ...).
2. Análisis sintáctico para detección de errores en la generación del código.
3. Elaboración del módulo, identificación y asignación de una imagen hardware del mismo (existen comandos y funciones de los HDL's que no son traducibles a hardware, y que se utilizan para la simulación), lo que se conoce como *modelo de referencia* de una estructura HDL.
4. A partir del modelo de referencia, se genera un circuito basado en primitivas.
5. Optimización del circuito, teniendo en cuenta las condiciones de contorno del diseño, es decir, generación automática del diseño final con las restricciones definidas por el usuario.
6. Elaboración de una descripción en bajo nivel del circuito, una *netlist* en un formato estándar para la síntesis física.

En los lenguajes HDL's se da la doble implicación, la de describir un hardware para implementación de un circuito (síntesis del circuito) y modelado del circuito (simulación del circuito), de la tecnología y del sistema. Por tanto existen funciones que no identifican bien un circuito. Por ejemplo, en VHDL se puede escribir el comando:

```
wait for 47ns;
```

¿existe un circuito capaz de hacer la operación de espera de 47ns exactos? Obviamente no. Sin embargo puede servir para expresar el retardo de una señal o un módulo en simulación. Distinguiremos pues, HDL para síntesis y HDL para modelado y simulación. El primero es un subconjunto del segundo.

6. Lenguajes HDL

Existen un buen número de lenguajes HDL's la mayoría de ellos con un propósito limitado. Los más conocidos son:

- ABEL: Lenguaje promovido por XILINX para la introducción de módulos de carácter específico, como máquinas de estados finitos.

- AHDL: Lenguaje en desuso promovido por ALTERA para facilitar la introducción de diseños en sus FPGA's. En la actualidad AHDL hace referencia a un HDL para diseño de circuitos analógicos.
- EDIF, XNF, ...: No son lenguajes propiamente dichos, en el sentido de no ser fácilmente comprensibles por el usuario, pero se utilizan cómo medios para transferir *netlist* entre herramientas.

6.1 VHDL

El VHDL fue desarrollado por la Departamento de Defensa de los Estados Unidos para modelar y simular sistemas. Emula el paralelismo mediante eventos y utiliza como modelo de formulación el ADA. El Instituto para la Ingeniería Eléctrica y Electrónica (IEEE) ha formulado una norma que regula el VHDL: la norma IEEE 1076-1987.

6.2 Verilog

Es un lenguaje HDL propiedad de CADENCE Design Systems, fabricante de software para diseño de circuitos integrados, de la misma manera emula el paralelismo mediante eventos. Sigue la sintaxis del C.

6.3 VHDL vs Verilog

Llevamos casi un década debatiendo cuál de los dos lenguajes es el más apropiado para diseñar. La comunidad científica ha llegado a la conclusión de que no existe una razón de peso para claramente decantarse por uno o por otro. Por un lado, Verilog nace ante la necesidad de mejorar un flujo de diseño y por tanto es un lenguaje que cubre todas las necesidades de diseño y simulación. El resultado es un lenguaje simple y flexible que es fácil de aprender. Por otro, VHDL es un lenguaje más rígido y con una sistematización en el código que lo hace más legible, aunque las reglas de diseño obligan a proceder de forma, a veces poco ágil. Ante la duda de cuál de ellos enseñar en este curso la respuesta está en la propia comunidad científica: lo ideal es conocer ambos. En este sentido se coincide que el conocedor de Verilog le cuesta aprender VHDL, sin embargo el proceso contrario es en general, más aceptado. El programador de VHDL aprende Verilog en poco tiempo La experiencia de los autores de estos capítulos es precisamente esa, y que conceptualmente es más didáctico el VHDL.

6.4 Nuevos lenguajes de alto nivel

A partir de los años 2000 han nacido nuevos lenguajes de alto nivel cuya sintaxis y método de codificación son idénticos a los lenguajes informáticos más usuales en ingeniería como el C ó C++. Estos lenguajes tienen como objeto facilitar la traslación de los lenguajes de diseño y pruebas de estudio a la descripción hardware. Los más conocidos son Handel-C de Celóxica y SystemC.

Incluso se ha introducido lenguajes de muy alto nivel basados en bloque funcionales. Dado que el coste por puerta lógica ha decrecido, sugieren que no es tan importante la implementación final del código, y por tanto el buen estilo de programación, como que haga la implementación respetando la funcionalidad. Esos lenguajes de muy alto nivel utilzn herramientas como Simulink como captura algorítmica, pero exceden en mucho lo objetivos de este documento,

7. Síntesis de Circuitos.

Hemos mencionado repetidamente la palabra síntesis como un proceso fundamental en la generación del circuito. En el año 1987 A. Sangiovanni-Vincetelli, uno de los mayores contribuidores al CAD/CAE para circuitos VLSI definía síntesis como "generar el circuito desde una descripción algorítmica, funcional o de comportamiento del circuito, una descripción de la

tecnología y una descripción de las condiciones de diseño en una función de costes. El resultado ha de obtenerse en un tiempo razonable de computación y con la calidad similar o mejor a la que el razonamiento humano podría obtener". La síntesis completa de un circuito requiere tres niveles:

1. Síntesis modular o particionamiento razonado: es el análisis a nivel de diseño. Esta labor se la hemos dejado al diseñador.
2. Síntesis lógica, combinacional, secuencial, algorítmica y de comportamiento.
3. Síntesis física. Es decir, generación automática del layout, de la tarjeta PCB, de la programación de la FPGA,...

El tema que nos ocupa se centra fundamentalmente en el punto segundo. La naturaleza del problema estriba en que no existe una única formulación para una misma función lógica, es decir, una misma función puede adoptar muchas expresiones, y su implementación y comportamiento físico es diferente en cada caso. En particular existen dos casos especialmente interesantes: aquel en que el área ocupada por la función es mínima y aquel en que el número de etapas o niveles de la función es mínimo, ya que hay una relación directa entre el número de etapas y el retardo en la propagación de las señales. Debido a que son dos factores normalmente contrapuestos, en cada caso ha de evaluarse la formulación más adecuada o bien una situación intermedia, solución que está definida por las condiciones de diseño. En adelante *síntesis* será un sinónimo de *síntesis lógica*.

Durante los años 80 y 90 las investigaciones en materia de síntesis se han centrado en desarrollar algoritmos que resuelven de una manera eficiente y óptima diferentes estructuras típicas de los circuitos digitales. Podemos identificar:

- Síntesis a dos niveles y síntesis multinivel. Optimización de funciones combinacionales.
- Síntesis de módulos aritméticos. Generación de estructuras regulares que permitan generar circuitos digitales aritméticos de cualquier dimensión.
- Síntesis de circuitos secuenciales: FSM's, Memorias, Contadores, Registros,...

Por tanto se han desarrollado algoritmos específicos para cada estructura y estrategias diferentes para cada tipo de módulo a implementar, por tanto en el proceso de síntesis lógica ha de haber una etapa previa de identificación del módulo en cuestión. Un programa de síntesis es, por tanto, una recopilación de programas específicos de optimización, aplicados a cada estructura digital, sucedido de una optimización combinacional global. De ahí que en el punto 5 de este capítulo descrito la necesidad de obtener un modelo de referencia como paso previo a la síntesis del circuito.

Comercialmente existen varios programas de síntesis en el mercado que merecen mencionarse:

- SYNOPSYS, programa de carácter general que permite realizar síntesis ante multitud de lenguajes y tecnologías. Funciona sobre plataforma UNIX.
- Leapfrog, de CADENCE para VHDL
- VerilogXL, de CADENCE para Verilog.

- Leonardo de MENTOR GRAPHIC's, multilenguaje en plataforma PC
- FPGA-EXPRESS, de Synopsys para PC y centrado en FPGA's. (Obsoleto)
- Sinplify , también de Synopsys

ISE y Vivado, de Xilinx o Quartus de ALTERA

CAPÍTULO II. ESTRUCTURA DE UN BLOQUE FUNCIONAL EN VHDL

1. Cuestiones previas

Una vez introducida la necesidad de diseñar utilizando lenguajes HDL y todo lo que ello significa nos decantamos por presentar el VHDL como lenguaje de diseño. Existen varias cuestiones previas que hay comentar antes de iniciar la exposición formal del lenguaje.

Primeramente, el VHDL es un lenguaje en el que se define el sentido del flujo de las señales, es decir, una señal es de entrada y/o salida definida en el código, no por la evolución de la señal en si misma. La importancia de este comentario radica en que el nivel de descripción más bajo que podemos alcanzar en VHDL es el nivel de puerta lógica, y no de transistor. Por ejemplo, para representar un *switch* en VHDL se ha de incluir una subrutina de cerca de 750 líneas de código, inefienciente a todas luces. En Verilog esto esta bien resuelto por comando *switch*.

En segundo lugar, el VHDL no es sensible a mayúsculas y minúsculas, por tanto la señal PEPE es igual a la señal PePe y a su vez a la señal pepe.

Los comentarios son solamente de línea y van marcados por dos signos menos:

```
--Esto es un comentario en VHDL
```

El lenguaje VHDL está dotado de muy pocos recursos inicialmente, de pocas funciones y tipos de variables. Para darle mayor versatilidad es preciso utilizar librerías estándar que le dotan de mucha mayor potencia para representar comportamientos digitales. Dejaremos a capítulos posteriores la formalización de esta idea.

El mecanismo para realizar la síntesis de un *diseño* descrito en VHDL se realiza, de modo estándar utilizando una librería de compilación donde se ubican los modelos de referencia, que físicamente es un subdirectorio en el directorio de diseño. Esta librería de modelos, por defecto se suele llamar WORK. Ampliaremos esta idea en el apartado referido a librerías.

2. Un primer ejemplo en VHDL

En esta sección vamos a exponer un ejemplo de las diferentes posibilidades del lenguaje VHDL. En particular escribiremos el código que describe un multiplexor 4 a 1, de tal manera que se pongan de manifiesto los diferentes elementos fundamentales del lenguaje. Estos elementos serán debidamente formalizados en secciones posteriores. El multiplexor tiene un bus de entradas de 4 bits llamado *datos* y un bus de selección de dos bits, llamado *selector*. La salida es la señal *salida.* Además estamos interesados en dar la misma descripción desde diferentes puntos de vista.

En primer lugar haremos la descripción basada en la función lógica del multiplexor, que corresponde a una arquitectura de tipo *comportamental* (behavioural)

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux41 IS
    PORT(
        datos: IN std_logic_vector(3 downto 0);
        selector: IN std_logic_vector(1 downto 0);
        salida: OUT std_logic
        );
END mux41;

ARCHITECTURE comport1 OF mux41 IS
BEGIN
Salida <= (((datos(0) AND (NOT selector(0))) OR (datos(1) AND selector(0)) AND (NOT
selector(1))) OR (((datos(2) AND (NOT selector(0))) OR (datos(3) AND selector(0)) AND
selector(1));
END comport1;
```

Esta es una descripción de comportamiento, pero basada en la resolución de la lógica asociada a nuestro multiplexor. El programa de síntesis realmente lo que hace es revisar las funciones y generar un circuito equivalente eliminando redundancias si las hubiere, o buscando la morfología de área mínima, consumo mínimo ó mínimo retardo, dependiendo de las restricciones impuestas al proceso de optimización lógica.

Una segunda versión del multiplexor puede realizarse empleando sentencias de más alto nivel, en la que se describe su comportamiento mediante recursos del lenguaje más cecanos al pensamiento humano. En particular utilizaremos la sentencia *WHEN condición ELSE*, que expresa una asignación bajo el cumplimiento de una condición y su alternativa:

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux41 IS
    PORT(
        datos: IN std_logic_vector(3 downto 0);
        selector: IN std_logic_vector(1 downto 0);
        salida: OUT std_logic
        );
```

```
END mux41;

ARCHITECTURE comport2 OF mux41 IS
BEGIN
Salida <= datos(0) WHEN (selector(0)='0' AND selector(1)='0') ELSE
          datos(1) WHEN (selector(0)='1' AND selector(1)='0') ELSE
          datos(2) WHEN (selector(0)='0' AND selector(1)='1') ELSE
          datos(3);

END comport2;
```

En esta formulación se utiliza un procedimiento más complejo de asignación y no es inmediata su implementación en un circuito. En particular, para la satisfacción de la asignación de datos(3) debe cumplirse la no asignación de las condiciones anteriores, dicho de otra manera, el código expresa una precedencia de datos(0) frente a los demás. Cuando se formula directamente el circuito que lo representa, en buena lógica los datos(0) tiene un menor tiempo de propagación que el resto a través del circuito. La fase de optimización posterior ecualiza los retardo al eliminar las redundancias que hay implícitas.

Una versión alternativa sería:

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux41 IS
    PORT(
        datos: IN std_logic_vector(3 downto 0);
        selector: IN std_logic_vector(1 downto 0);
        salida: OUT std_logic
         );
END mux41;

ARCHITECTURE compor3 OF mux41 IS
BEGIN
Salida <= (datos(0) WHEN selector(0)='0' ELSE datos(1)) WHEN selector(1)='0' ELSE
          (datos(2) WHEN selector(0)='1' ELSE datos(3));

END comport3;
```

Es este caso en la fase de optimización se realizaría menos esfuerzo, ya que la estructura es más parecida a la versión final. Un ejemplo para controlar la estructura del multiplexor es el siguiente:

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux41 IS
    PORT(
        datos: IN std_logic_vector(3 downto 0);
        selector: IN std_logic_vector(1 downto 0);
        salida: OUT std_logic
         );
END mux41;
```

```
ARCHITECTURE compor4 OF mux41 IS
BEGIN
Salida <= datos(0) WHEN (selector(0)='0' AND selector(1)='0') ELSE 'Z';
Salida <= datos(1) WHEN (selector(0)='1' AND selector(1)='0') ELSE 'Z';
Salida <= datos(2) WHEN (selector(0)='0' AND selector(1)='1') ELSE 'Z';
Salida <= datos(3) WHEN (selector(0)='0' AND selector(1)='1') ELSE 'Z';

END comport4;
```

En este caso el circuito se realiza con una estructura muy distinta de las anteriores, basada en puertas triestado. Esta situación típica cuando los multiplexores son de gran tamaño, ya que se vierte sobre el conexionado la función OR del multiplexor. El esfuerzo de optimización se centra en el decodificador del multiplexor.

Finalmente haremos una versión estructural del multiplexor. Para ello supondremos que ya existe una descripción de un multiplexor más simple 2 a 1, descrito bajo la entidad mux21. En este caso obviamos su arquitectura, ya que solamente estamos interesados en su descripción como entidad.

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux21 IS
    PORT(
            datos: IN std_logic_vector(1 downto 0);
            selector: IN std_logic;
            salida: OUT std_logic
             );
END mux21;
```

En ésta versión haremos tres instancias del mux21. Además se precisa la utilización de señales auxiliares internas, llamadas *salida_parcial.* Como se puede comprobar, es la utilización de una descripción puramente conectiva.

```
LIBRARY IEEE;
USE IEEE. Std_logic_1164.all;
ENTITY mux41 IS
    PORT(
            datos: IN std_logic_vector(3 downto 0);
            selector: IN std_logic_vector(1 downto 0);
            salida: OUT std_logic
             );
END mux41;

ARCHITECTURE estruct OF mux41 IS
COMPONENT mux21
    PORT(
            datos: IN std_logic_vector(1 downto 0);
            selector: IN std_logic;
            salida: OUT std_logic
             );
END COMPONENT;
```

```
SIGNAL salida_parcial: std_logic_vector(1 downto 0);
BEGIN
mx1: mux21
PORT MAP(datos =>datos(1 downto 0) , selector=>selector(0),salida=>salida_parcial(0));

mx2: mux21
PORT MAP(datos =>datos(3 downto 2) ,selector=>selector(0),salida=> salida_parcial(1));

mx3: mux21
PORT MAP(datos => salida_parcial , selector=> selector(1) ,salida=>salida );
END estruct;
```

Esta es la situación que eventualmente se obtendría tras un proceso de optimización en el caso de que mux21 fuera un componente *primitiva*, aunque en este caso somos nosotros los que hemos definido la estructura.

3. Estructura de un diseño VHDL

Una de las grandes aportaciones de los lenguajes HDL, como se expuso en el capítulo 1, es la posibilidad de organizar jerárquicamente los diseños, de tal manera que cada elemento, junto con los elementos de inferior nivel jerárquico, es en sí mismo un diseño autocontenido. En consecuencia cada unidad en la jerarquía tiene entidad como circuito, con entradas, salidas y funciones.

La unidad jerárquica en cuestión es designada por la palabra reservada **entity o entidad.** La **entidad** tiene asociado un nombre identificador usualmente relativo a la función que realiza. Cada vez que hagamos uso de este circuito utilizaremos el nombre asociado. Una entidad es modelo de una unidad funcional hardware que puede representar un sistema completo, un subsistema, una tarjeta, un chip, una macro-célula, una puerta lógica o cualquier nivel de abstracción de ellos. Asimismo está definida por señales de enlace con el exterior y una **arquitectura** funcional.

El VHDL ha previsto la posibilidad de modelar diferentes arquitecturas para una misma entidad de ahí que haya que asignar nombres tanto a la entidad como a la arquitectura. Asimismo dispone de un mecanismo para poder acceder de forma automática a las diferentes arquitecturas de una misma entidad desde una entidad de orden jerárquico superior. Es el mecanismo de la configuración o **configuration**

3.1 Código general de una entidad VHDL

El código de una entidad se presenta en el siguiente esquema:

```
        --Zona de declaración de librerías
LIBRARY nombre_librería;
USE paquete_funciones_librería.all;
        --Cabecera de la entidad
ENTITY nombre_entity IS
        GENERIC(..........);
        PORT(................);
        CONSTANT señal : tipo :=valor;
        CONSTANT señal : tipo :=valor;
        …
END nombre_entity;
        --Cuerpo de la entidad
ARCHITECTURE nombre_architecture OF nombre_entity IS
        --Declaración de sub-entidades (componentes) y señales internas
BEGIN
        --Descripción de la funcionalidad
END nombre_architecture;
        --Enlace con las arquitecturas de otras entidades
CONFIGURATION nombre_configuracion OF nombre_entidad IS
        FOR nombre_arquitectura
        --Cuerpo de la configuración
END nombre_configuracion;
```

El desarrollo de este tema consiste en explicar cada uno de los elementos expuestos en el esquema anterior.

4. La sección ENTITY

Una declaración entidad representa una interfaz entre la funcionalidad que representa y su entorno. Identifica la unidad funcional dentro de una jerarquía de forma unívoca. No pueden existir dos entidades del mismo nombre en la jerarquía definidas con un interfaz diferente. Junto con la palabra ENTITY hay asociados dos campos:

- GENERIC: utilizado para pasar parámetros a la entity
- PORT: utilizado para definir las señales que relacionan la entity con el resto del diseño.

Veamos un ejemplo concreto:

```
ENTITY contador IS
        GENERIC(SATURAC: integer:= 1000; N:integer:=10);
        PORT(
                CLK: in BIT;
                RST: in BIT;
                ENABLE: in BIT;
                COUNT: out BIT_VECTOR(N-1 DOWNTO 0);
                SATUR: out BIT
                );
END contador;
```

El código representa una descripción de las entradas y salidas de un contador de tamaño 10. La gran ventaja es que es posible redefinir los parámetros SATURAC y N tantas veces como queramos y extender el contador a la longitud y al valor de saturación que convenga en cada momento sin modificar el código. Cada vez que se utiliza la entidad como parte del diseño se pueden determinar los parámetros que definitivamente definen el diseño. Sin embargo hemos dado –obligatoriamente- un valor por defecto, tanto a SATURAC, 1000, como a N, el valor 10 al tamaño del contador. El motivo es doble. Por un lado el valor por defecto se utilizará en caso de no reasignación del valor del parámetro, pero la más importante es que los parámetros tienen la misión de dar coherencia al modelo del circuito, es decir, un contador de tamaño N y SATURAC no tienen sentido como circuito, sin embargo, al dar un valor sí quedan bien definidos y a la hora de generar su correspondiente modelo hardware de referencia se facilita generar la arquitectura. Los parámetros se utilizan para controlar características estructurales, de flujo de datos o comportamentales de una entidad, o simplemente como documentación. Pueden definirse tantos parámetros como se desee. De los tipos posibles de parámetros hablaremos en el capítulo siguiente. La sintaxis seguida es:

Nombre_parámetro : tipo_parámetro := valor_por_defecto;

La cláusula PORT indica las señales que interrelacionan la **entity** con el resto del diseño. La sintaxis es:

Nombre_señal : dirección *tipo_de_señal;*

El nombre de la señal identifica unívocamente la señal en cuestión. No puede haber otra señal dentro de la arquitectura con el nombre de la señal. El campo “dirección” de la señal indica el sentido del flujo de la misma. Este campo puede tomar los siguientes valores:

- IN para indicar que la señal es una entrada
- OUT para indicar que la señal es una salida
- INOUT para indicar que la señal es una entrada o salida dependiendo de cada instante.
- BUFFER tipo extendido de salida.
- LINKAGE es un tipo enlace genérico de salida que está en desuso. La norma vaticina su desaparición.

Una señal IN recibe sus valores desde el exterior de la entidad. Por tanto, no puede ser reasignada en el interior de la entidad, es decir no puede aparecer a la izquierda de una asignación en la arquitectura:

```
Señal_in<=A;  --Error

A<=Señal_in; --Correcto
```

Una señal OUT genera valores al exterior de la entidad. No puede ser asignada a ninguna otra señal en la arquitectura:

```
A<=Señal_out; --Error

Señal_out<=A; --Correcto
```

Una señal INOUT puede ser asignada en ambos sentidos y es responsabilidad del diseñador determinar en qué condiciones de la función lógica descrita la señal puede ser IN o OUT.

El lector podría pensar que para evitar errores lo más sencillo sería describir todas las señales como INOUT. Sin embargo esto sería atentar contra la naturaleza de los HDL, que es describir de la forma más explícita y precisa posible un circuito. De la otra manera un mal estilo de diseño nos llevaría a una mala descripción del circuito y seguramente a otro tipo de errores.

Una señal BUFFER es señal de salida directamente asignable a otra señal dentro de la arquitectura, pero no puede ser forzada desde el exterior, dicho de otra manera, no es en ningún momento una entrada. Se utilizan fundamentalmente para describir señales que realimentan dentro de la entidad y simultáneamente salen al exterior. Un caso típico es un contador síncrono. Sin embargo en la práctica hay sintetizadores que utilizan la palabra BUFFER para forzar un cierto comportamiento eléctrico de una señal. Por tanto hay sintetizadores que no lo contemplan,

La declaración de tipos **constant** tiene por objeto de la definición de elementos de valor constante dentro de la entidad. Es posible su reasignación tras la declaración como instancia de la entidad.

No entraremos en detalles acerca de los tipos de datos, cuestión que se abordará en el capítulo correspondiente.

5. La sección ARCHITECTURE

La arquitectura de la entidad es la descripción de la funcionalidad. Consta de dos partes:

La parte declarativa, donde entre otros, se especifican los elementos de tipo estructural que van a componer la arquitectura, es decir:

- Nuevos tipos de señales subtipos y constantes.
- Señales internas. Son enlaces entre los diferentes elementos que definen la arquitectura.
- Entidades de orden inferior en la jerarquía, llamadas componentes o **component.**
- Funciones, bloques, grupos,…

La parte descriptiva, donde se define la funcionalidad que concretamente se le asigna a la arquitectura. De las órdenes comandos y funciones que se pueden incluir en una arquitectura hablaremos más adelante. Por ahora profundizaremos en la cuestión de la jerarquía de un diseño y cómo se especifica en una arquitectura.

5.1 Diseño jerárquico (estructural) en una arquitectura.

En primer lugar especificamos en la parte declarativa qué componentes (**component**) se utilizarán en la arquitectura. Hacemos referencia a otras entidades, a sus señales de enlace y a los parámetros.

Luego se especifican las señales mediante la palabra **signal**, seguida de uno o varios nombres y de un tipo según la sintaxis:

SIGNAL nombre1, nombre2, ... : *Tipo de señal*:= *valor inicial*;

Todas las señales especificadas pertenecen al mismo tipo. Como se puede apreciar este es el mismo esquema empleado en la declaración de las señales de enlace de la entidad, con la salvedad de omitir la direccionalidad de la señal. Asimismo, y opcionalmente se puede asignar un valor inicial de la señal. Este valor suele utilizarse en simulación y es omitido por los sintetizadores.

También se pueden declarar valores constantes:

CONSTANT nombre: *Tipo de señal*:= *valor inicial*;

Veamos un ejemplo concreto. Utilizando la declaración de la entidad superior construiremos una arquitectura que llama dos veces al contador anterior. Supongamos un contador doble:

```
ENTITY doble_contador IS
    PORT(
            clk,rst,enable1,clr: IN BIT;
             salida: OUT BIT;
             cuenta: OUT BIT_VECTOR(7 DOWNTO 0)
     );
END doble_contador;

ARCHITECTURE estructura_mod OF doble_contador IS

COMPONENT contador
    GENERIC(N:integer:=10);
    PORT(
        CLK: in BIT;
        RST: in BIT;
        CLEAR: in BIT;
        ENABLE: in BIT;
        COUNT: out BIT_VECTOR(N-1 DOWNTO 0);
        SATUR: out BIT
    );
    END COMPONENT;

SIGNAL enlace1, slogico0: BIT;
SIGNAL cuenta1: BIT_VECTOR(7 DOWNTO 0);
BEGIN
slogic0<='0';
cont1: contador
    GENERIC MAP(N=>8)
    PORT MAP(CLK=>clk, RST=>rst, CLEAR=> CLR, ENABLE=>enable1,
        COUNT=>cuenta1, SATUR=>enlace1);

cont2: contador
    GENERIC MAP(N=>12)
    PORT MAP(CLK=>clk,RST=>rst, CLEAR=>slogic0, ENABLE=>enlace1,
COUNT=>OPEN,SATUR=>salida);
END estructura_mod;
```

El ejemplo muestra la utilización de un mismo componente varias veces. Lo primero que hay que haces notar es la presencia de una etiqueta que identifica cada uno de los componentes de una manera unívoca. cont1 y cont2 son dos instancias de una misma entidad *contador*, y cada uno de ellos tiene un valor diferente del parámetro. De ahora en adelante esta etiqueta, *nombre_instancia*, identificará al componente. La sintaxis es simple:

```
nombre_instancia : nombre_entidad
        GENERIC MAP(...............)
        PORT MAP(...................);
```

En segundo lugar debemos destacar la manera de referenciar señales internas a la entidad. Tras la declaración de los componentes se declaran señales que son interiores al diseño. En el ejemplo aparece la señal *enlace1* como señal interna, común a ambos bloques:

en un caso enlace1 es de salida y en el otro, de entrada. Obviamente en la declaración de la misma no existe direccionalidad, a diferencia de la señales de los puertos de la entidad.

En tercer lugar conviene repasar la sintaxis y la puntuación. La etiqueta de instancia va seguida del nombre de la entidad, en este caso el componente que se trata. Las secciones **generic** y **port** son ahora **generic map** y **port map**, indicando las señales que van a ser asociadas a sus entradas y salidas. En el caso de que hubiese direccionalidad en las señales, es decir, que vengan declaradas desde la entidad (en el ejemplo *clk, rst,enable1, salida,...*) la señal ha de conservar el tipo. Las cláusulas *map* no van separadas por ';'.

Finalmente la manera de asociar señales que se propone es especificando qué señal se asocia a qué entrada, mediante los caracteres '**=>**' y separados por comas. Este método se denomina pasar parámetros por **referencia**. En este sentido conviene hacer algún comentario adicional. Es correcto pasar los parámetros por **orden**, es decir con la sintaxis:

```
cont2: contador
        GENERIC MAP(12)
        PORT MAP(clk,rst,enlace1,OPEN,salida);
```

Si bien es más conciso, es también poco práctico, ya que un error obliga a estar constantemente comparando con el componente y verificando una correcta asociación, es decir, muy poco legible. La experiencia demuestra que siendo más explícito, si bien se escribe más, se tarda en tener éxito mucho menos tiempo. En este caso es obligado que aparezcan todos los parámetros.

La palabra reservada **open**.

La palabra reservada **open** se utiliza para dejar señales sin conectar. La omisión de un parámetro es equivalente a la asociación de **open**.

Es posible asociar **open** a una entrada, que es tanto como decir que se deja sin conectar, lo cual en la práctica no es nada aconsejable, ya que se deja libertad al proceso de síntesis para que realice las operaciones sin nuestro control. En la práctica se debe asociar un valor lógico concreto a la entrada para evitar resultados impredecibles. En VHDL no se admite que en la referencia a un componente se asocie un valor lógico directamente, por lo que hay que declarar una señal auxiliar.

```
constant clogic0: std_logic :='0'; --Este valor no se puede asignar

signal slogic0: std_logic; --Este es el modo de asignación
….
slogic0<='0';
….
port map (clr=> slogic0,....)
```

La asociación para la conexión de componentes se realiza mediante señales del mismo tipo por tanto no es válido asociar constantes con señales de puertos, de ahí que la asociación de *clogic0* no sería válida.

La arquitectura presentada es un ejemplo de arquitectura estructural, ya que está compuesta únicamente de componentes y sus enlaces. Si hacemos referencia al paralelismo entre ellos, este caso es un ejemplo claro que no existe precedencia en el comportamiento de un contador u otro, ya que ambos se activan de igual manera, aunque el segundo dependa del primero.

5.2 Diseño de comportamiento en una arquitectura.

En el nivel más bajo del diseño ha de haber una información acerca de la funcionalidad del propio diseño. No se hacen referencia a otros componentes, sino que en la propia arquitectura se definen las operaciones que se realizan sobre las señales. Sin entrar excesivamente en detalles, ya que serán motivos de capítulos posteriores.

5.3 Diseño con primitivas.

Las primitivas son las unidades funcionales elementales que describen el diseño después de realizar la síntesis. Están asociadas a la tecnología elegida y están representadas como componentes en una librería que el fabricante de circuitos integrados proporciona. Es posible realizar una descripción de un diseño a partir de una descripción mediante primitivas, lo que equivale a realizar un esquemático escrito.

6. La sección CONFIGURATION

La sección CONFIGURATION responde a un complejo mecanismo de selección entre las diferentes arquitecturas posibles de una misma entidad utilizada en un nivel jerárquico superior. Una configuración está asociada a una entidad y una arquitectura. Por tanto, mediante la selección de la configuración determinaremos qué arquitectura es seleccionada para esa entidad. Podemos definir tantas configuraciones como sean necesarias, ya que a ella se le asocia un nombre.

La sintaxis es:

```
CONFIGURATION nombre_configuracion OF nombre_entidad IS
        FOR nombre_arquitectura
            FOR nombre_instancia :
nombre_entidad   USE CONFIGURATION WORK.nombre_configuracion;
            END FOR;
            ......................
        END FOR;
END nombre_configuracion;
```

Es en el nivel jerárquico inmediatamente superior donde se particulariza qué configuración se desea para determinada entidad. Veamos un ejemplo:

```
--Entidad inferior
ENTITY mi_contador IS
            PORT(
clk,rst,enable: in std_logic;
cuenta: out std_logic_vector(7 downto 0)
      );
END contador;
--definimos dos arquitecturas diferentes para la misma entidad
ARCHITECTURE comport OF contador  IS
BEGIN
      ....................
END comport;

ARCHITECTURE estruct OF contador IS
BEGIN
      ....................
END estruct;
--asociamos una configuración a cada una de las arquitecturas
CONFIGURATION comport_conf OF contador  IS
            FOR comport
            END FOR;
END comport_conf;
CONFIGURATION estruct_conf OF contador IS
            FOR estruct
            END FOR;
END estruct_conf;

      -- configuración de la entidad superior que referencia el componente
CONFIGURATION estruct_top OF top IS
            FOR estruct
              FOR cont1: contador USE CONFIGURATION WORK.estruct_conf;
              END FOR;
  FOR cont2: contador USE CONFIGURATION WORK.comport_conf;
              END FOR;
             ................
            END FOR;
END estruct_conf;
```

En este ejemplo hemos utilizado dos arquitecturas diferentes del mismo componente. En el caso de que no exista la necesidad de variar las configuraciones se puede resumir en la cláusula ALL:

```
      -- configuración de la entidad superior que referencia el componente
CONFIGURATION estruct_top OF top IS
            FOR estruct
              FOR ALL: contador USE CONFIGURATION WORK.estruct_conf;
              END FOR;
              ................
            END FOR;
END estruct_conf;
```

6.1 Estilo par entidad-arquitectura

Existe una sintaxis alternativa que permite una asociación más explícita:

```
CONFIGURATION estruct_top OF top IS
        FOR estruct
          FOR cont1: contador USE ENTITY WORK.contador(estruct);
          END FOR;
          FOR OTHERS: contador USE ENTITY WORK.contador(comport);
          END FOR;
          ................
        END FOR;
END estruct_conf;
```

En este caso hemos introducido también la cláusula OTHERS para identificar a aquello que no se ha hecho referencia previamente. Este primer ejemplo de una palabra que es muy utilizada en VHDL.

En la práctica este potente mecanismo es poco utilizado, dado que no se ofrecen muchas situaciones de varias arquitecturas para una misma entidad. Si solamente existe una única arquitectura en todos y cada uno de los componentes del diseño, la configuración será única y por tanto se omite la sección de configuraciones.

7. La sección LIBRARY

Tiene un especial interés el dedicar un párrafo completo a las librerías en VHDL, dado que en la práctica todas las extensiones del lenguaje se realizan utilizando este potentísimo mecanismo.

Mediante bibliotecas podemos definir componentes, funciones y procedimientos, es decir, unidades del lenguaje que pueden ser útiles en múltiples aplicaciones, siendo un elemento de reutilización del lenguaje. De esta manera, un componente de una librería ya analizada y sintetizada puede ser llamada en un código sin necesidad de tener que volver a declararlo como componente. Las librerías se estructuran en paquetes (packages) que permiten una fácil clasificación de los elementos que las componen.

La declaración de una librería sigue la siguiente sintaxis:

```
library MiLibreria;
use MiLibreria.Paquete1.all;
use MiLibreria.Paquete2.all; --Enmascara las funciones de Paquete1
```

De esta manera quedan accesibles todos (**all**) los elementos del *Paquete1* y *Paquete2* de la librería *MiLibreria*. En el capítulo seis nos dedicaremos en detalle a definir, declarar y construir nuestras librerías. Sin embargo nos detendremos en explicar algunos aspectos importantes de las librerías más utilizadas. Solo dos palabras para comentar que las funciones VHDL disponen del **mecanismo de sobrecarga** en sus llamadas, de forma que no solo se atiende a la

coincidencia del nombre de la función en la llamada, sino que además se verifica la coincidencia de los tipos de argumento de las mismas.

Finalmente, la presencia de funciones con igual nombre y tipo de argumentos en *Paquete1* y *Paquete2* hace que las funciones del primero queden enmascaradas por las del segundo, de ahí que si se desea utilizar parcialmente una de ellas, bien debemos cuidar el orden de declaración o bien sustituir la palabra **all** por las funciones que nos interesen.

7.1 La necesidad de la librería *std_logic_1164*

El VHDL posee un limitado número de tipos y funciones que resultan insuficientes a la hora de describir completamente el comportamiento de un circuito digital. Por tanto se hace preciso que la extensión del lenguaje contemple un número más amplio de situaciones. En particular, los tipos de datos que maneja el VHDL de modo natural se reflejan en la siguiente tabla:

- **BIT** , {0,1}
- **BOOLEAN**, {false, true}
- **CHARACTER**, {la tabla ASCII desde 0 a 127}
- **INTEGER**, {-2147483647, 2147483647}
- **NATURAL**, {0, 2147483647}
- **POSITIVE**, {1, 2147483647}
- **STRING**, array {POSITIVE range <>} de CHARACTER
- **BIT_VECTOR**, array {NATURAL range <>}de BIT
- Tipos físicos, como **TIME**
- **REAL**, {-1E38, 1E28}
- **POINTER**, para accesos indirectos a datos.
- **FILE**, para accesos a ficheros y pilas de datos del disco duro.

Ni que decir tiene que muchos de ellos corresponden a estructuras del lenguaje que no son sintetizables. De hecho, durante la síntesis de bloques VHDL casi nunca se utilizan de manera explícita y completa estos tipos de datos. Por tanto el VHDL de forma sistemática ha de extenderse mediante tipo de datos que presentan un campo de variabilidad mucho más cercano al funcionamiento eléctrico de un circuito. Se trata del paquete *std_logic_1164* de la librería IEEE. Normalmente esta librería aparece dando sentido a un nuevo tipo de datos, el *std_logic*, que es un tipo enumerado y resuelto. Los detalles acerca de este tipo de datos se exponen en el capítulo cuarto de estos apuntes.

La cabecera de librerías estándar es típicamente:

library IEEE;

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

En esta librería se introducen los nuevos operadores relacionales asociados al nuevo tipo de datos.

7.2 Librerías aritméticas

El VHDL no tiene definidos de modo estándar operaciones aritméticas que permitan el uso de operadores. Es preciso, pues, la inclusión de librerías que faciliten la inclusión de operadores de tipo aritmético. La librería estándar para este caso es la *numeric_std*. En el capítulo dedicado a comentar la utilización de las funciones aritméticas desarrollaremos las referencias a esta librería. Se estudiará a fondo en los capítulos cuatro y cinco.

CAPÍTULO III. DESCRIPCIÓN DE LA FUNCIONALIDAD

1. Concepto de concurrencia.

Un circuito electrónico es, por naturaleza paralelo, es decir, no existe un mecanismo inherente a la tecnología soporte que establezca precedencias en la actividad de sus bloques constituyentes. Esto significa que la descripción del mismo ha de preservar esta propiedad. Cuando se introducen sentencias o bloques (que más adelante llamaremos procesos), todos ellos son concurrentes, es decir, no existe orden de prelación entre unos u otros (en lo que afecta a su funcionalidad, que no es lo mismo que su legibilidad e interpretación). Por ejemplo:

```
      A<=B and C;
D<=A and F;
```

es equivalente a:

```
D<=A and F;
A<=B and C;
```

sin embargo existen gran cantidad de situaciones en las que es muy fácil describir un comportamiento de un circuito de modo secuencial.

Nos obstante, formalmente existen en VHDL cinco tipos de elementos concurrentes:

- Bloques (**block**). Grupos de sentencias concurrentes.
- Instancias de componentes. En diseños estructurales, cada vez que se produce la utilización de un componente se realiza de modo concurrente.
- Sentencias **generate**. Generación de copias de bloques de hardware.
- Llamadas a procedimientos y funciones (**procedure** y **function**). Los procedimientos son algoritmos combinacionales que computan y asignan valores a señales. Suelen formar parte de librerías.
- Declaración y definición de procesos (**process**). Internamente definen algoritmos secuenciales que leen valores de señales y asignan valores a señales, aunque externamente el proceso se ve como una instancia concurrente.
- Asignaciones de señales computadas a través de expresiones booleanas o aritméticas.

Resulta, pues muy útil la presencia de mecanismos que permitan introducir código secuencial. El más común es el proceso o **process**, que detallaremos en el siguiente apartado. Un proceso es un bloque que contiene código secuencial pero externamente se contempla como un bloque concurrente, y tiene el mismo tratamiento que el ejemplo anterior. Las

instancias de componentes se han tratado en el capítulo anterior y los procedimientos y las funciones se explicarán en el capítulo seis cuando se explican las librerías, ya que es en ellas donde más se utilizan.

2. Estructura de un proceso

Para comprender el funcionamiento de un proceso conviene introducir una doble perspectiva del mismo: la simulación y la síntesis. Desde el punto de vista de simulación el proceso contiene una secuencia de transformaciones sobre las señales que se producen una vez activado el mismo. Desde el punto de vista de síntesis conviene no olvidar que un proceso representa una porción de un circuito y que es preciso asociar una imagen hardware al conjunto de sentencias que lo componen.

El mecanismo del **process** se entiende como una manera fácil de transferir algoritmos ensayados en lenguajes de alto nivel secuenciales a VHDL. Esta idea condiciona enormemente su sintaxis y sus posibilidades.

2.1 Elementos de un proceso.

El siguiente listado recoge todos los elementos de un proceso

```
etiqueta_proceso: PROCESS(lista de sensibilidad)
VARIABLE ...
BEGIN
                --secuencia de ordenes
END PROCESS etiqueta_proceso;
```

Los procesos se introducen en la zona de arquitectura y no existe restricción alguna acerca de su número y extensión.

2.2 Lista de sensibilidad.

La lista de sensibilidad es un artificio para realizar computacionalmente el paralelismo, principalmente en simulación, aunque también es útil en síntesis. La lista de sensibilidad es un conjunto de señales que activan el proceso, es decir, mediante un cambio en un cualquiera de ellas, el proceso es evaluado completamente. Esto permite resolver el problema de la concurrencia mencionado, ya que si una sentencia concurrente es evaluada cuando los argumentos de la sentencia varían, igualmente ocurre con una lista de sensibilidad.

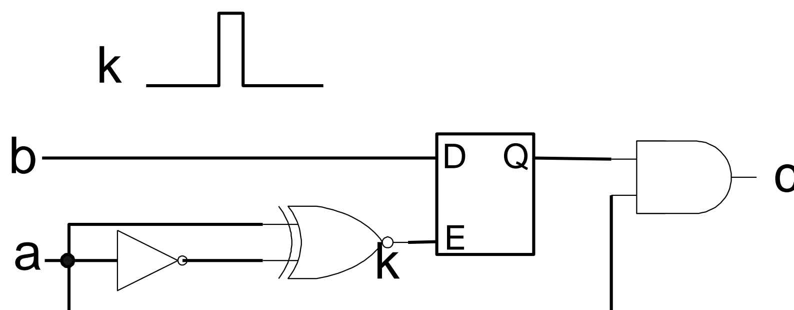
Existen varias cuestiones que conviene aclarar:

- La lista de sensibilidad debe contener todas las señales que afecten a la evaluación del proceso, es decir, señales que usualmente se ubican a la derecha en una sentencia de asignación o evalúan una sentencia condicional.
- La omisión de una señal en la lista de sensibilidad afecta necesariamente a la morfología del circuito que representa.
- Si no existe lista de sensibilidad el proceso se denomina **infinito** y se ejecuta constantemente. Para su control se precisa el uso de sentencias tipo *wait* que son típicas de construcciones no sintetizables y se estudiarán en el capítulo correspondiente.
- Si una señal que se computa a la **izquierda** de una asignación dentro del proceso aparece en la lista de sensibilidad, este se evaluará de nuevo, y se producirá sucesivamente hasta que el valor de la señal se estabilice.
- Si se omitiese una señal de entrada en la lista de sensibilidad, este se evalúa cuando una de las señales restantes de la lista cambie, por tanto dicha señal habría de almacenarse hasta el cambio. Se generaría, pues, un *latch* no deseado.

Por ejemplo, si en la lista de sensibilidad se omite una señal, hemos dicho que afecta decididamente a la imagen hardware que se representa. Imaginamos el siguiente ejemplo:

```
pr1: PROCESS( a ) –omitimos la señal b
BEGIN
        c<=a and b;
END PROCESS pr1;
```

Se ha omitido la señal *b*. Cuando *a* varía el proceso se evaluará realizando la operación *and* de *a* y *b*, con el valor que *b* tenga en ese instante. Sin embargo, cuando *b* varía *c* no se actualizará, ya que no está en la lista de sensibilidad. De esta manera, el circuito que produce el valor de c ha de almacenar el valor de b cuando se produzca una variación en a. La figura representa un circuito equivalente a la situación expresada. El circuito activa el match es un detector de flanco, que se basa en la presencia de un retardo en la entrada de la puerta XNOR, que genera la forma de onda indicada en el punto k, con un pulso de anchura el retardo del inversor. Esta situación no es admisible desde el punto de vista de un diseño, ya que el retardo del inversor es dependiente de la tecnología, y es tan estrecho que es probable que viole las condiciones temporales de la entrada de habilitación E del *latch*.



Dado que este circuito no es aceptable casi todos los programas de síntesis producen un aviso de ausencia de una señal de la lista de sensibilidad, aviso que es preciso eliminar, y

dejan a que el diseñador, si quiere implementar estructuras como estas, describa explícitamente el circuito.

2.3 Declaración del tipo **variable**

En la implementación secuencial se suele hacer uso de variables temporales que no trascienden fuera del proceso. Su uso está limitado a asignaciones en su interior. Este recurso del lenguaje se le conoce como **variable**. El mecanismo de asignación es igual que el de una señal, con la salvedad de que en la asignación se utilizan los símbolos ':=' en lugar del '<='. Por ejemplo:

```
asig: PROCESS(b,c)
VARIABLE a: std_logic;
BEGIN
        a:= b and c;
        d<= a xor b;
END PROCESS asig;
```

La variable 'a' aparece en una situación transitoria, y tiene una misión aclaratoria, es decir, un proceso se activa con señales y resuelve los valores de señales. Existen, además, otras diferencias cualitativas que describiremos en el apartado siguiente.

2.4 Asignaciones dentro de un proceso

En el cuerpo de un proceso se admiten asignaciones simples. La principal diferencia en el tratamiento estriba en que el proceso se evalúa en un instante de tiempo cero, es decir, que todas las asignaciones y condiciones se producen sin retraso. De la misma manera las asignaciones a señales (que no a variables) dentro de un proceso no se hacen efectivas hasta que el proceso no se concluye. Por ejemplo:

```
ARCHITECTURE comport OF ejemplo
SIGNAL q : INTEGER RANGE 0 TO 3;
BEGIN
        ej: PROCESS(A,B,..)
        BEGIN
                q<=0;
                IF(A='1') THEN q<=q+1; END IF;
                IF(B='1') THEN q<=q+2; END IF;
                CASE q =>
                        WHEN 0 =>
                        ....
                END CASE;
        END PROCESS ej;
END comport;
```

El valor de *q* no se actualiza hasta que el proceso ha concluido. Esto significa que en la evaluación realizada en la estructura CASE el valor de *q* es el que tuviese antes de la evaluación del proceso. Ni siquiera la asignación incondicional afectaría a dicha evaluación. Para realizar

una evaluación secuencial tal y como se realiza en un programa desarrollado para una CPU es preciso utilizar variables en vez de señales, es decir:

```
ARCHITECTURE behav OF ejemplo
BEGIN
        ej: PROCESS(A,B,..)
        VARIABLE q : INTEGER RANGE 0 TO 3;
        BEGIN
                q:=0;
                IF(A='1') THEN q:=q+1; END IF;
                IF(B='1') THEN q:=q+2; END IF;
                CASE q =>
                        WHEN 0 =>
                        ....
                END CASE;
        END PROCESS ej;
END behav;
```

Mediante esta modificación realizamos la asignación deseada y la funcionalidad que resulta es la requerida.

2.5 Sentencias propias de un proceso

Dentro de un proceso existen un conjunto de sentencias que son propias de su estructura. Fuera de él carecen de significado. Son condiciones no concurrentes:

Sentencia IF THEN ELSIF ELSE

La condicional se resuelve mediante la sentencia IF. Su sintaxis es:

```
IF condición 1THEN
                Consecuencia 1;
ELSIF condición 2 THEN
                Consecuencia 2;
ELSIF ... THEN
                ...
ELSE
                Consecuencia por defecto;
END IF;
```

La condición puede ser lógica o aritmética. Merece especial atención al concepto inherente de jerarquía que introduce la sentencia IF. La condición 1 es previa a la condición 2, que nunca será evaluada en caso de cumplimiento de la primera. Algo parecido ocurre con la siguiente estructura anidada, también válida:

```
IF condición 1THEN
        IF condición 2 THEN
                        Consecuencia 2.1;
        ELSE
                        Consecuencia 2.2;
        END IF;
ELSE
                Consecuencia por defecto;
END IF;
```

La diferencia entre ambas estructuras estriba en que en la segunda las consecuencias 2.1 y 2.2 se validan mediante un circuito que comprueba Condición 1 y Condición 2 encadenadas. En la primera de las estructuras la Consecuencia 1 tendría un camino crítico menor.

Las condiciones en una sentencia IF se evalúan a través de expresiones booleanas que devuelven los valores 'verdadero' o 'falso'. En la siguiente tabla se resumen los tipos de expresiones:

Tipo de condición	**Ejemplo**	C
=	**If (a = b) then**	Igualdad.
/=	**If (a /= b) then**	Desigualdad
<,<=,>,>=	**If (a<b) then**	Menor, menor o igual, mayor, mayor o igual

Inferencia de elementos de memoria

Las sentencias condicionales tienen una particularidad que las hacen de especial interés en la síntesis de hardware secuencial. Sea el ejemplo siguiente:

```
sec: PROCESS(ena,d)
BEGIN
        IF (ena='1') THEN
                q<=d;
                qz<=NOT d;
        END IF;
END PROCESS sec;
```

La estructura inferida hace que q tome el valor d cuando ena tiene el valor '1'. Pero, ¿qué ocurre cuando *ena* vale '0'? Si cambia d, q permanece con el valor de d cuando *ena* valía '1'. Existe un dispositivo hardware que permite realizar la funcionalidad expuesta, consistente en un elemento de memoria activo por nivel o *latch*. Del ejemplo se infiere un *latch* para la señal *q* y otro para la *qz*, que en un posterior proceso de optimización sería agrupado en uno solo con la negación de *q*.

El modelo de inferencia de un *latch* se reproduce continuamente durante el proceso de diseño, en situaciones deseadas y en situaciones no deseadas. Un ejemplo típico de situación deseada es el *biestable*, donde la señal *ena* es el reloj, y por tanto se ha de utilizar un mecanismo para especificar el flanco del reloj, mecanismo que se expondrá en el capítulo cuarto. El *latch* rara vez se utiliza en diseño síncrono y suele ser una fuente de problemas. Normalmente cuando los programas de síntesis detectan una estructura de un *latch* (no deseada, el biestable es un elemento siempre deseado) suelen proporcionar un aviso de que se infiere. En general es una estructura condicional que no tiene solución hardware para todas las posibles situaciones. Por tanto es recomendable que en las estructuras que dependen de la evaluación de una condición antecedente dispongan al menos de una valor por defecto para los consecuentes.

Sentencia CASE WHEN

Una selección en función del valor que toma una señal se construye mediante la selección

```
CASE objeto IS
        WHEN caso1 =>
                Código1;
        WHEN caso2 =>
                Código2;
        WHEN caso3 | caso4 =>
                Código34;
        WHEN OTHERS =>
                Código para el resto de los casos;
END CASE;
```

En los casos en que las dos formas son posibles, es recomendable el uso de CASE antes que IF/ELSIF dado que el IF introduce un elemento de prioridad que puede resultar menos eficiente (más área).

WHEN OTHERS representa la condición por defecto. No puede haber dos valores de selección iguales y la cláusula **others** debe aparecer si no se cubren todos los posibles valores de selección.

Es importante que exista una asignación para todas las salidas en cada uno de los casos ya que de esta forma evitamos la formación de *latches* indeseables.

Bucles dentro de procesos

VHDL permite la construcción de bucles para reducir el tamaño de código repetitivo. Estas sentencias han de ir en dentro de un proceso.

```
Etiqueta: FOR parámetro IN lista_de_valores LOOP
		Código;
END LOOP Etiqueta;

Etiqueta: WHILE condición LOOP
		Código;
END LOOP Etiqueta;
```

Cuando el objeto a evaluar es un índice, no es precisa su declaración previa. En ambos casos es posible interrumpir la evaluación normal con las instrucciones NEXT y EXIT, cuando la condición se satisface. NEXT salta el resto del código secuencial dentro del bucle y obliga a la evaluación de un nuevo índice. EXIT salta el resto del código secuencial y obliga a la terminación del mismo, continuando con la secuencia del proceso.

```
NEXT etiqueta WHEN condición;  --saltar el resto del bucle
EXIT etiqueta WHEN condición;  --salir del bucle
```

por ejemplo:

```
FOR i IN 0 TO 7 LOOP
		NEXT WHEN i=6; -- Dejamos el bit 6 como estaba
		Bus(i) <= '0';
END LOOP;
```

Otra alternativa es:

```
i:=0;
WHILE i < 8 LOOP
	Bus(i) <= '0';
	i:=i+1;
END LOOP;
```

3. Bloques de sentencias

Es una agrupación de sentencias concurrentes. La sintaxis es:

```
etiqueta: BLOCK
	BEGIN
		Sentencia Concurrente 1;
Sentencia Concurrente 2;
Sentencia Concurrente 2;
		......
	END BLOCK etiqueta;
```

Normalmente un bloque no crea un nuevo nivel de jerarquía. Su función es puramente aclaratoria y en la práctica es poco utilizado.

4. Asignaciones concurrentes

Es posible realizar operaciones de asignación concurrente equivalentes a las que se han definido dentro de las estructuras secuenciales, en las que las restricciones a la hora de realizar una inferencia son exactamente las mismas que en las mencionadas estructuras, con la ventaja de que no es preciso la realización de una estructura del tipo proceso.

NOTA: Hemos comentado que realmente todos los elementos concurrentes son tratados dentro del complejo tratamiento informático del lenguaje como **process**, concurrentes entre sí.

4.1 Asignación simple

Es la sentencia concurrente más elemental y se realiza mediante el asignador '**<=**'. La condición para una correcta asignación es la igualdad en los tipos de las señales que se asignan. Su sintaxis es:

```
etiqueta :  Señal Destino <= Señal Origen;
```

Donde la señal destino puede ser una señal declarada previamente mediante **signal** o bien una señal que se encuentra en el puerto (**port**) de la entidad (**entity**) como salida, tipo **out** , entrada-salida **inout** (ó **buffer**, cuando se admita).

La señal origen puede ser una señal declarada previamente mediante **signal** o bien una señal que se encuentra en el puerto (**port**) de la entidad (**entity**) como entrada (**in**), entrada-salida (**inout**) (ó **buffer**, cuando se admita). Además la asignación simple permite la evaluación de expresiones siguiendo una sintaxis igual:

```
etiqueta :  Señal Destino <= Expresión Funcional;
```

La expresión funcional representa una función booleana o aritmética que es combinación de señales de entrada o señales. La siguiente tabla representa las funciones booleanas que se admiten en una asignación:

Operador	**Ejemplo**	Tipo
NOT	**a <= NOT b;**	Operador lógico

AND, NAND	**a <= b AND c;**	Operadores lógicos
OR,NOR,XOR	**a <= b OR c;**	Operadores lógicos

Asimismo se admiten operaciones aritméticas. Estas operaciones han de estar previamente definidas en librerías. En el capítulo cuarto se tratan más en profundidad la manera de trabajar con estas funciones. Por ahora nos limitaremos a exponer una relación de las mismas:

Operador	Uso	Descripción
*	**a <= b * c;**	Multiplicación
**	**a <= b ** 2;**	Exponencial
/	**a <= b/c;**	División (c ha de ser potencia de 2)
+	**a <=b+c;**	Suma
-	**a <=b-c;**	Resta
MOD	**a <= b MOD c;**	Calcula **b** en modulo **c**
REM	**a <= b REM c;**	Calcula la resta entera de b / c

Es posible una asociación y una inferencia de jerarquía en las funciones jerarquía mediante la utilización de paréntesis. Existe una prioridad en la definición de las funciones. Se resume en la siguiente tabla:

Máxima: ** ABS NOT

* / MOD REM

\- (negación)

\+ - &

= /= < <= > >=

Mínima AND OR NAND NOR XOR

4.2 Asignación condicional

Es una expresión concurrente que realiza las mismas operaciones que la cláusula IF dentro de un proceso. Su sintaxis es:

```
etiqueta: Señal Destino <= Expresión1 WHEN Condición ELSE Expresión2;
```

No existen diferencias cualitativas entre ambos tipos de sentencias. La forma IF admite, de modo mucho más legible, más de una consecuencia y consecuencias mucho más complejas que la forma concurrente WHEN.

De la misma forma es posible encadenar la forma WHEN:

```
etiqueta: Señal Destino <= Expresión1 WHEN Condición1 ELSE
Expresión2 WHEN Condición2 ELSE
Expresión3 WHEN Condición3 ELSE
        .............
ExpresiónN WHEN CondiciónN ELSE  Expresión;
```

4.3 Selección en forma concurrente

También existe una sentencia equivalente a **case** en forma concurrente. Su sintaxis es:

```
etiqueta: WITH expresión_selección SELECT
Señal Destino <= Expresión1 WHEN Valor_Selección1,
                 Expresión2 WHEN Valor_Selección2,
                      ....
                 ExpresiónN WHEN OTHERS;
```

No puede haber dos valores de selección iguales y la cláusula **others** debe aparecer si no se cubren todos los posibles valores de selección.

5. La sentencia **generate**

Uno de los mecanismos de generación de hardware más potentes del VHDL es el basado en la sentencia **generate**. Los detractores del Verilog utilizan esta sentencia como argumento de valor, ya que en no existe en Verilog una forma equivalente.

5.1 Bucles utilizando **generate**

Mediante la sentencia **generate** podemos realizar cero, una o más copias de un conjunto de sentencias concurrentes en función de un parámetro que recorre un rango de valores. La sintaxis es como sigue:

```
etiqueta: FOR parámetro IN rango_de_valores GENERATE
                {
Sentencias concurrentes asociadas al parámetro
                }
```

```
END GENERATE etiqueta;
```

En este caso la etiqueta es obligatoria, ya que se permite anidar bucles **generate**. El parámetro no se declara externamente. Es local al bucle. No es posible asignar un valor al parámetro ni este puede ser asignado.

El rango de valores se recorre en forma creciente o decreciente:

```
entero_menor TO entero_mayor
entero_mayor DOWNTO entero_menor
```

El uso más corriente de la sentencia **generate** es la generación de copias de componentes, procesos o bloques. El ejemplo representa la llamada múltiple a un componente, y genera 8 instancias del mismo con valores diferentes de los parámetros:

```
COMPONENT MiComp
        GENERIC(N: integer:=8);
        PORT(X: in std_logic;
             Y: out std_logic );
END COMPONENT;
....
    SIGNAL A,B: bit_vector (7 downto 0);
....
    Bucle_generate: FOR i IN 0 TO 7 GENERATE
        copia: MiComp
                GENERIC MAP (N=>2*i)
                PORT MAP(X=>A(i), Y=>B(7-i));
        END GENERATE Bucle_generate;
....
```

En el ejemplo hemos jugado con diferentes posibilidades del mecanismo **generate**, con la idea de poner de manifiesto sus posibilidades. Hemos asignado a cada componente un valor diferente del parámetro, en función del mismo. Asimismo hemos realizado operaciones con el índice del vector B.

5.2 **Generate** condicionado.

Mediante la evaluación de una expresión booleana es posible construir un componente:

```
Etiqueta: IF expresión GENERATE
{
    Sentencias concurrentes
}
END GENERATE Etiqueta;
```

Es preciso destacar que esta forma de IF se presenta como sentencia concurrente y la ausencia de una condición alternativa tipo ELSE ni ELSIF. Esta estructura tiene mejor uso anidada dentro de otro bucle **generate**.

CAPÍTULO IV. TIPOS DE DATOS Y SEÑALES

En este capítulo vamos a presentar los tipos de datos más corrientes en VHDL y planteamos todos conceptos básicos necesarios para poder manejar el lenguaje, dejando a capítulos siguientes las particularizaciones sobre elementos más comunes. Haremos un recorrido por todos los elementos típicos y la librería std_logic_1164 de gran importancia en la generación de códigos sintetizables. Partimos de lo reseñado en el apartado 6.1 del capítulo segundo de estos apuntes.

1. Definición de tipos de datos.

El VHDL admite la definición de nuevos tipos de datos de una forma muy simple. Dentro de la parte declarativa es posible escribir de manera explícita todos los posibles valores que puede tomar una señal de un determinado tipo. Se le conoce como tipo enumerado:

TYPE nombre_tipo **IS** (Enum1,....,EnumN);

Donde Enum1 a EnumN puede ser un identificador (un nombre cualquiera) o bien un carácter. Por ejemplo:

TYPE equipo **IS** (Sevilla, Málaga, Huelva, Betis, Granada, Córdoba, Jaén, Jerez)
....
SIGNAL *ligaandaluza*: equipo;

Realmente la señal *ligaandaluza* en un tipo enumerado porque realmente es un subtipo de INTEGER donde Sevilla representa el valor 0, Málaga el valor 1, hasta Jerez el valor 8. De esta forma se obtiene una evaluación positiva de la condición:

Betis > Sevilla

Este mecanismo permite realizar operaciones con valores de las señales que están representados de una forma clara por un estado, nombre o identificador. En el tema dedicado a las máquinas de estados finitos se comprenderá su potencia y utilidad.

2. Tipo entero

El VHDL admite en su forma estándar el uso del tipo entero. Una señal declarada como entero es un registro de 32 bits que recorre el rango desde $-(2^{31}-1)$ hasta $+(2^{31}-1)$. El registro trata los valores negativos en complemento a 2. Es posible definir un intervalo de enteros mediante la definición de un nuevo tipo:

TYPE nombre_tipo **IS RANGE** intervalo;

El intervalo debe estar contenido entre los valores anteriormente expuesto. Por ejemplo:

TYPE puntos_liga **IS RANGE** 0 **TO** 144;

Todas las señales que sean definidas como puntos_liga podrán tomar valores dentro de este intervalo. Internamente estas señales serán tratadas mediante registros de 8 bits.

3. Tipo matriz

Una matriz es un conjunto ordenados de elementos del mismo tipo. El acceso a uno de esos elementos se realiza a través de un índice. Un tipo matriz se define:

TYPE matriz_nombre_tipo **IS ARRAY** (intervalo de enteros) **OF** nombre_tipo;

Por ejemplo:

TYPE clasificacion **IS ARRAY** (1 **TO** 7) **OF** equipo;

VHDL soporta de matrices multidimensionales, es decir, cuyos elementos dependen del valor de uno o más parámetros. La mayoría de programas de síntesis no contemplan esta posibilidad y se restringen al tratamiento de matrices unidimensionales. No obstante a través de varias declaraciones de tipo unidimensional se pueden construir.

El valor del intervalo de enteros permite la definición de tipos restringidos, como el del ejemplo, y no restringidos que pueden tomar valores en todo el rango de variabilidad de los enteros. Su sintaxis es:

TYPE matriz_nombre_tipo **IS ARRAY**(Tipo_predefinido **RANGE** <>) **OF** nombre_tipo;

La ventaja de este mecanismo consiste el que dejamos a las herramientas de síntesis VHDL tomar las decisiones acerca del intervalo de variabilidad final. Por ejemplo:

TYPE Bit_Vector **IS ARRAY** (**INTEGER RANGE** <>) OF bit;
...
VARIABLE MiVector: Bit_Vector (-5 **TO** 5);

Siendo la variable la que define finalmente el intervalo en el momento de su declaración.

3.1 Atributos de las matrices.

Los atributos son valores asociados a una señal o variable que proporcionan información adicional, independientemente del valor que la señal o variable tenga en un instante dado. Son de gran utilidad en la elaboración de código que depende intensivamente de parámetros. En este apartado se exponen los atributos relacionados con el aspecto matricial. En apartados posteriores se exponen otros atributos asociados al comportamiento de las señales. Estos son los atributos matriciales definidos en VHDL estándar:

- LEFT. Identifica el valor definido a la izquierda del intervalo de índices.

- RIGTH. Identifica el valor definido a la derecha del intervalo de índices.
- HIGH. Identifica el valor máximo del intervalo de índices.
- LOW. Identifica el valor mínimo del intervalo de índices.
- LENGTH. Identifica el valor total del intervalo de índices.
- RANGE. Copia el intervalo de índices
- REVERSE_RANGE. Representa el mismo intervalo de índices pero en sentido inverso.
- La manera de acceder a los valores de los atributos es la siguiente:

nombre_de_señal'NOMBRE_ATRIBUTO

Veamos un ejemplo que presenta todos los valores de los atributos anteriormente expuestos:

```
....
signal MiSenal : std_logic_vector (7 downto –3);
....

MiSenal'left toma el valor 7
MiSenal'rigth toma el valor -3
MiSenal'high toma el valor 7
MiSenal'low toma el valor –3
MiSenal'length toma el valor 11 (El 0 también se cuenta!!!)
MiSenal'range toma el valor (7 downto –3)
MiSenal'reverse_range toma el valor (-3 to 7)
```

Podremos utilizar algunos de estos valores en el bucle en el código siguiente:

```
Bucle: FOR i IN MiSenal'reverse_range LOOP
        A(MiSenal'high - i)<=MiSenal(i)
END LOOP Bucle;
```

Esta función trabaja con una matriz con independencia de su tamaño.

3.2 Tipo vector

Probablemente el tipo más utilizado es la matriz unidimensional de bits, tipo que representa físicamente un BUS o conjunto ordenado de líneas eléctricas. Habitualmente estos tipos se predefinen con un tipo infinito para posteriormente se particularizado. Estos tipos predefinidos suelen caracterizarse por llevar el sufijo _VECTOR.

En la librería correspondiente está definido el tipo

```
TYPE Bit_Vector IS ARRAY (POSITIVE range <>) OF BIT;
```

En nuestro código podremos definir señales:

```
.....
SUBTYPE MiBus : Bit_Vector (15 downto 0);
SIGNAL x,y : MiBus;
SIGNAL a : BIT;
```

```
......

IF(x=y)THEN
.....

x(3)<=a; -- Asignación de un índice particular
y<="1110001010001 10"; -- Asignación de un valor constante
.....
VARIABLE MaxIndex : INTEGER;
.....
MaxIndex:=x'HIGH; --Vale 15
.....
```

En el ejemplo anterior se muestra un conjunto de asignaciones y manipulaciones posibles de un VECTOR. Hemos definido un subtipo (subconjunto de un tipo), aunque no es preciso realizar tal operación, ya que podemos trabajar directamente con el tipo Bit_Vector(15 downto 0).

La asignación de un valor constante se realiza utilizando las dobles comillas en lugar de la comilla simple, utilizado en el tipo *std_logic*. La razón estriba en que se asocian cadenas de caracteres.

Otra cuestión interesante es la forma que hemos dado al VECTOR: **(15 downto 0)**. Es norma del buen programador el ser sistemático a la hora de definir las formas de los buses. Normalmente el índice que se asigna al bit más significativo debe ser el más alto y al bit menos significativo el más bajo. Si se utiliza la forma **(0 to 15),** esta regla no se respeta y llevará seguro a confusiones. Es por tanto muy recomendable utilizar la forma DOWNTO a la hora de definir buses y olvidar la forma TO.

4. Definición de tipos compuestos

Un tipo compuesto (**record**) consiste en una agregación de tipos que forman uno nuevo. La manera de definirlos es:

```
TYPE nombre_de_tipo IS
    RECORD
        nombre_de_subtipo1 : tipo;
        nombre_de_subtipo2 : tipo;
        ............
    END RECORD;
```

La manera de acceder a los tipos individualmente es mediante una referencia al nombre del subtipo. Por ejemplo:

```
TYPE Entero_Complejo IS
    RECORD
        EnteroR : INTEGER RANGE –100 TO 100;
        EnteroI : INTEGER  RANGE –100 TO 100;
```

```
                END RECORD;

        ....
        SIGNAL x,y,z : Entero_Complejo
SIGNAL real, imaginaria : INTEGER RANGE –100 TO 100;

.....
x.EnteroR <= real;
.....
imaginaria <=x.EnteroI;
.....
y<=z;
.....
```

En este ejemplo se han realizado asignaciones parciales y completas del tipo definido. También se pueden realizar agregaciones:

```
X<=( EnteroR => 20, EnteroI => 40);
Y<=(30,-10);
```

En el ejemplo se pasan los argumentos por referencia y por orden.

5. Tipos simples

Como se ha comentado en capítulos anteriores el VHDL dispone de tipos de datos estándar que de manera natural soportan todas las posibilidades del lenguaje. Estos tipos son:

- CHARACTER, tipo enumerado que soporta toda la secuencia de caracteres ASCII desde 0 a 128.
- BOOLEAN, tipo enumerado que toma valores TRUE ó FALSE, donde TRUE>FALSE.
- BIT, tipo enumerado, que toma valores '0' ó '1'. Las funciones lógicas devuelven también un valor del mismo tipo.
- INTEGER que toma valores en el intervalo $-(2^{31}-1)$ a $+(2^{31}-1)$.
- NATURAL, un subtipo del INTEGER que toma valores entre 0 y $+(2^{31}-1)$.
- POSITIVE, un subtipo del INTEGER que toma valores entre 1 y $+(2^{31}-1)$.
- STRING, una matriz de intervalo no definido del tipo CHARACTER.
- BIT_VECTOR, una matriz de intervalo no definido del tipo BIT.

En la práctica ninguno de estos tipos se utiliza para describir hardware. Todos los programas de simulación y síntesis VHDL proporcionan una librería de tipos y funciones que completan la librería estándar para describir de manera más precisa en comportamiento de circuitos digitales. Se trata de las librerías IEEE.

4.1 Los atributos de las señales.

Independientemente del valor que una señal tenga en un instante dado el VHDL mantiene una serie de registros asociados a la señal llamados atributos. La diferencia con los

atributos asociados a una matriz estriba en que son dinámicos, es decir, están asociados a la evolución de la señal en el tiempo y son extremadamente útiles a la hora de describir situaciones reales. Se basan en el mecanismo comentado en el capítulo primero en el que se define una base de tiempos incremental (el tiempo mínimo es un paso de simulación) para emular el paralelismo inherente a un circuito, y describir adecuadamente la evolución de los sistemas digitales. Los atributos predefinidos en VHDL son:

- DELAYED(t). Valor de la señal retrasada t unidades de tiempo.
- STABLE(t), verdadero si la señal permanece invariable durante t unidades de tiempo.
- QUIET(t), verdadero si la señal no ha recibido ninguna asignación en t unidades de tiempo.
- TRANSACTION, tipo bit, a '1' cuando hay una asignación a la señal.
- EVENT, verdadero si ocurre un cambio en la señal en el paso de simulación.
- ACTIVE, verdadero si ocurre una asignación a la señal en el paso de simulación.
- LAST_EVENT, unidades de tiempo desde el último evento.
- LAST_ ACTIVE, unidades de tiempo desde la última asignación.
- LAST_VALUE, valor anterior de la señal.
- DRIVING, verdadero si el proceso actual determina el valor de la señal.
- DRIVING_VALUE, valor que toma la señal tras el proceso.

Ni que decir tiene que la mayoría de estos atributos no se utilizan en el subconjunto de síntesis de VHDL. Son válidos para modelado de dispositivos. Sin embargo existe uno de ellos en particular extremadamente útil a la hora de describir sistema síncronos. Se trata del atributo EVENT, que estudiamos en detalle en la sección siguiente.

La existencia de atributos marcan claramente la diferencia entre un objeto tipo **signal**, de los tipo **variable** y **constant**. Se recuerda que los elementos de **port map** tiene un tratamiento equivalente al de **signal**.

4.2 El atributo EVENT

Es de una gran importancia en la elaboración de código VHDL. Veamos por qué. X'EVENT significa que si una señal sufre una modificación se produce un valor 'verdadero' en el atributo EVENT durante el instante de evaluación de la señal, es decir durante el paso de simulación. Después, si ya no se produce cambio alguno, el atributo toma el valor 'falso'.

Este atributo permite detectar como caso particular una transición de una señal '0 a 1' y '1 a 0'. Si se combina con una condición adicional podemos describir la transición de una señal a un valor concreto, es decir:

X'**EVENT AND** X='1'

es una manera de detectar un flanco de subida, ya que la señal X a cambiado y su valor de cambio es '1'. Si este mecanismo lo aprovechamos para describir un elemento de memoria que se activa con el flanco:

```
sinc: PROCESS(CLK)
BEGIN
    IF CLK'EVENT AND CLK='1' THEN
        Q<=D;
    END IF;
END PROCESS sinc;
```

Mediante este código se describe un elemento de memoria que registra a través de un flanco de la señal *clk*. A este elemento se le conoce como biestable o flip-flop tipo D. Lo más interesante es que este mecanismo nos permite describir de forma clara todos los sistemas digitales síncronos, que en realidad son la práctica totalidad de los sistemas digitales.

Aún así hay cuestiones que resultan interesantes de destacar. Veamos el siguiente código, parecido al anterior:

```
sinc_mal: PROCESS(CLK)
BEGIN
    IF CLK'EVENT THEN
        Q<=D;
    END IF;
END PROCESS sinc_mal;
```

El atributo EVENT evalúa los cambios en CLK cada vez que se produce una transición valida la condición. La pregunta: ¿existe algún módulo hardware digital cuyo comportamiento coincida con la descripción del código anterior? Evidentemente no. El artificio EVENT tiene sentido a la hora de realizar una síntesis si va ligado al valor de la señal.

Otra cuestión es que intencionadamente hemos omitido el valor de D en la lista de sensibilidad. Esto es posible debido a que la evaluación tiene lugar en instantes muy precisos de tiempo que dependen sólo de CLK por tanto no tiene importancia el valor de D en el resto del tiempo.

Finalmente es posible escribir este código:

```
sinc: PROCESS(CLK)
BEGIN
    IF CLK'EVENT AND CLK='1' THEN
        CUENTA<=CUENTA + 1; -- ¿Realimentación combinacional?
    END IF;
END PROCESS sinc;
```

En el apartado 6 del capítulo primero describimos como una realimentación combinacional la línea del código anterior. Realmente el bucle existe pero está seccionado con un registro, situación que es del todo válida. Hemos descrito un contador. Aunque es válido no recomendamos la generación de códigos como el anterior. En el capítulo cinco expondremos y analizaremos a fondo un el código de un contador.

4.3 La función RISING_EDGE

La fórmula del **IF** CLK'**EVENT AND** CLK='1' **THEN** está cada vez más en desuso. CLK'EVENT representa un cambio de la señal CLK, ese cambio puede ser desde cualquier valor de los posibles de partida hasta cualquiera de ellos. Sólo se limita a los que el valor de destino es un '1'. Por ejemplo si el de partida es 'H' y el de destino es '1' la condición sería válida.

En su lugar se tiende a usar una función contenida en la librería IEEE.std_logic_1164 destinada a identificar un flanco de reloj. Esta función es de tipo booleano tomando valores entre 'true' y 'false', y a forma es RISING_EDGE(CLK).

Sin embargo, en la práctica, la fórmula **IF** CLK'**EVENT AND** CLK='1' **THEN** es mantenida por casi todo los programa de síntesis como la detección de un flanco, así que no podemos darla por incorrecta, sólo diremos que existen formulaciones como RISING_EDGE(CLK), mejores, y por tanto se utilizarán a partir de ahora en éste texto. Ni que decir tiene la existencia de FALLING_EDGE(CLK) para determinar el flanco de bajada.

6. La librería IEEE

Como se ha comentado la definición BIT resulta insuficiente para describir completamente el conjunto de valores que cualitativamente toma una señal digital. De ahí que la librería IEEE proporciona un paquete llamado "std_logic_1164". La librería extiende, como se ha dicho, el paquete estándar de VHDL, contemplando las situaciones a nivel de función lógica. No existe un tratamiento a nivel de interruptor o **switch** (cosa que sí existe en Verilog). Prácticamente es obligatorio disponer de una cabecera que declare el uso de esta librería en todos nuestros diseños:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

En esta librería se define un nuevo tipo de señal enumerado, el *std_ulogic,* *(*uninitialized std_logic). Este nuevo tipo de contempla los siguientes valores:

- 'U', no inicializado, es el valor por defecto.
- 'X', desconocido. Es el valor que toma en casos de conflicto.
- '0', el valor lógico bajo.
- '1', el valor lógico alto.
- 'Z', representa el estado de alta impedancia.
- 'W', estado desconocido débil.
- 'L', valor lógico bajo débil.
- 'H', valor lógico alto débil.
- '-', valor que no importa

Las situaciones tipo débil suelen representar situaciones de tipo resistivo. Por ejemplo, una señal con valor 'L' se conecta directamente a una con valor '1' el resultado será una señal

de valor ‘1’. El tipo std_logic, derivado del std_ulogic, es de los llamados tipo resuelto o *resolved*.

Es importante recordar cuales son estos valores y qué representan. El valor ‘z’ (¡¡minúscula!!) no existe, ya que no está contemplado en este conjunto enumerado. En la librería se contemplan las manipulaciones precisas que permiten el adecuado tratamiento de las señales y sus valores.

De la misma manera están definidos los tipos VECTOR *std_logic_vector*. para soportar conjuntos ordenados de del tipo *std_logic*. Dentro del paquete se encuentran definidas las operaciones booleanas y los comparadores = y /=. Asimismo proporciona mecanismos que permiten la conversión entre tipos. Por ejemplo:

```
SIGNAL x,y,z : std_logic_vector (8 DOWNTO 0);
SIGNAL b,c : BIT;
.....
x(5)<=Conv_Std_Logic(b);
c<=Conv_bit(z(3));
x<=y xor z;
```

6.1 Los tipos SIGNED y UNSIGNED.

La utilización del tipo *std_logic_vector* en un diseño en el que se realizan operaciones de cálculo tiene dos dificultades: La primera consiste en que la librería no contiene los operadores aritméticos necesarios para poder realizar los cálculos y la segunda es debida a que el valor que se representa plantea ciertas dificultades debido a la naturaleza ambigua de la representación binaria. Por ejemplo si un vector tiene el valor “1101” estamos representando ¿qué valor? ¡Depende! Si trabajamos en valores positivos tenemos el 13, pero si trabajamos con representación en complemento a 2 tenemos el –3. No disponemos con la librería *std_logic_1164* de una manera de tratar esta situación, ya que se limita a las funciones booleanas.

La librería IEEE contiene dos paquetes de funciones, la *std_logic_arith y la numeric_std,* que contienen las funciones aritméticas para resolver la primera parte del problema. La *std_logic_arith* se encuentra en desuso, por su manera de tratar los valores, por tanto introduciremos al lector en la forma *numeric_std.* Mediante la inserción en la cabecera de

```
USE IEEE.numeric_std.ALL;
```

disponemos de todos los operadores aritméticos (+,-,*,ABS) y relacionales (<,>,<=,>=,= y /=).

Se han definido nuevos tipos de datos: UNSIGNED y SIGNED. Son tipos de datos std_logic_vector con la atribución de contemplar el valor de su contenido como entero sin signo el primero y entero con signo y representación en complemento a 2 el segundo. También se incluyen las funciones de conversión correspondientes para realizar correctamente las asignaciones. Por ejemplo podremos definir:

```
SIGNAL a,b: UNSIGNED(8 downto 0);
SIGNAL x,y: SIGNED(8 downto 0);
SIGNAL d,e: std_logic_vector(8 downto 0);
.....
```

Realizamos las asignaciones, por ejemplo:

```
a <= b;   --Son del mismo tipo y funciona
x <= a;   --Se obtiene un error!! Diferente tipo
x <= signed(a);  --También funciona la función conv_signed.
b<= unsigned(x) + unsigned(d); --gracias al mecanismo de sobrecarga
d <= std_logic_vector(a) * std_logic_vector(y) --Se obtiene un error!! Operador ambiguo
```

Mediante este procedimiento podemos definir y controlar la morfología del hardware de los operadores. Las funciones de conversión son útiles y se estudiarán en el apartado siguiente. No existe una razón para que no se puedan usar como tipos de entrada o salida, aunque por cuestiones de compatibilidad y portabilidad se suelen convertir avalores tipo *std_logic_vector*:

6.2 Las funciones de conversión

Como se ha indicado anteriormente, las asignaciones solamente se pueden realizar entre tipos idénticos. En las librerías se incluyen funciones que permiten la conversión de un tipo en otro. Cuando se trata de convertir entre tipos signed, unsigned o std_logic_vector la función de conversión sigue la forma:

```
señal1 <= tipo_señal1(señal2);
```

Podemos reflejar todas las coversiones en la siguiente tabla:

	Unsigned	Signed	Std_logic_vector	Integer
Unsigned	x	Signed()	Std_logic_vector()	**To_integer()**
Signed	Unsigned()	x	Std_logic_vector()	**To_integer()**
Std_logic_vector	Unsigned()	Signed()	x	-
integer	**To_unsigned(,length)**	**To_signed(,length)**	-	x

Donde en columnas especificamos el tipo de partida, en filas el tipo de destino, y las diferentes casillas representan las funciones de conversión entre unas y otras.

Donde la conversión entre tipo Signed, Unsigned y Std_logic_vector es una conversión entre matrices (arrays) de valores std_logic, mientras que en negrita, las conversiones a o desde enteros son funciones.

En el paréntesis hay que incluir la señal que se está convirtiendo. Para convertir desde entero a Signed o Unsigned debemos especificar el parámetro del tamaño del vector al que se convierte. Por ejemplo:

```
a<=Unsigned(b);
```

donde a es un vector Unsigned del mismo tamaño que b.

```
c<=To_signed(d, 8);
```

donde d es un entero que se convierte a un vector signed de 8 bits.

La librería numeric_std es más rica en funciones, así podemos enumerar funciones:

a) de comparación: <, >, <=,=>, /=, = entre signed, unsigned, integer y el subconjunto natural.
b) Aritméticas: +, -, *, / por 2**N, rem y mod, sobre los mismos operandos.
c) Manipulación de bits: sll, srl, rol y ror, sobre unsigned y signed
d) Funciones directas de conversión, que se han especificado anteriormente.

CAPÍTULO V. SUBSISTEMAS DIGITALES. EJEMPLOS DE DISEÑO.

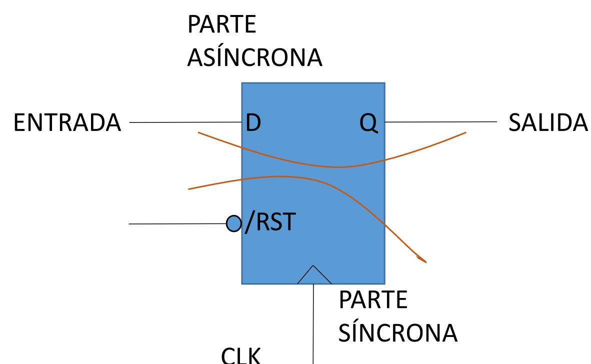
Este capítulo incide en aspectos prácticos relacionados con la codificación utilizando el lenguaje VHDL y se basa fundamentalmente en los conceptos introducidos en capítulos anteriores. Abordaremos dos aspectos esenciales en la codificación de sistemas digitales, por un lado los sistemas síncronos y por otro, en los sistemas aritméticos. Perseguimos un alto grado de sistematización en la escritura del código de cualquier sistema síncrono, facilitando que la depuración se centre en la funcionalidad, y por tanto esta sea más fácil. No está demás recordar que el VHDL es una manera de describir hardware y que estamos conformando un circuito electrónico digital.

Existen otros estilos de programación más económicos en cuanto a cantidad de código para expresar una misma funcionalidad. No pretenderíamos presentarlo como un error, sino como un estilo no recomendable, ya que si bien el resultado de la síntesis es muy parecido, hay pérdida de legibilidad.

1. Codificación de Sistemas Síncronos

La descripción de sistemas síncronos es una parte fundamental del diseño de sistemas digitales. Se basan en la utilización de una señal de reloj que coordina los cambios en el circuito, sucediéndose en instantes muy precisos, concretamente en los flancos activos (de subida o bajada).

Todo sistema síncrono se puede descomponer en dos procesos concurrentes, uno que recoge la funcionalidad característica del sistema y otro, generador de un registro o banco de flip-flops, también llamado proceso de sincronismo. Nuestra experiencia nos aconseja mantener este esquema de codificación en dos procesos, ya que si bien requiere la introducción de un número mayor de líneas de código la descripción es más clara, más legible y más fácil de depurar. Es más, hemos detectado algunos sintetizadores que no admiten otra manera de introducir la descripción. Todos los ejemplos que propondremos seguirán esta estructura yse acompaña un esquema para mejor comprensión.



Ejemplo 1. Codificación de un registro simple

El código es el siguiente:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY registro_simple IS
        GENERIC (N : INTEGER := 5);
        PORT(
                rstz : IN std_logic;
                clk : IN std_logic;
                entrada : IN std_logic_vector (N-1 DOWNTO 0);
                salida : OUT std_logic_vector (N-1 DOWNTO 0)
        );
END registro_simple;

ARCHITECTURE comport OF registro_simple IS
BEGIN
  Sinc: PROCESS(rstz,clk)
  BEGIN
        IF (rstz='0') THEN
                        Salida<=(OTHERS=>'0');
        ELSIF RISING_EDGE(clk) THEN
                        Salida <= entrada;
                END IF;
  END PROCESS sinc;
END comport;
```

En este ejemplo hemos introducido algunos elementos que merecen ser destacados. En primer lugar la señal de reset asíncrono, *rstz,* es activa a nivel bajo. Es buena práctica que aquellas señales definidas como activas a nivel bajo lleven un distintivo, de ahí que optemos por la terminación *z*. En segundo lugar la utilización siempre que sea posible de parámetros en **generics** ya que su impacto en la complejidad del código en prácticamente nula y permite la reutilización del mismo. El análisis de la estructura de sincronismo indica prioridad en la señal de *rstz* sobre el flanco de reloj *clk*. Esto indica que el registro cargará un '0' ante un valor '0' de la señal *rstz* aunque *clk* fluctúe. Finalmente hemos omitido la señal *entrada* de la lista de sensibilidad, ya que solamente se actualiza en flancos de *clk*.

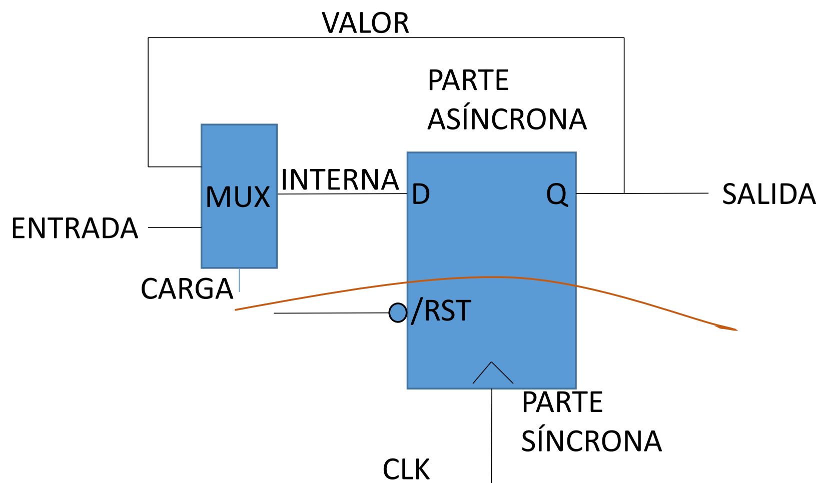
Ejemplo 2. Codificación de un registro con señal de carga

El código es el siguiente:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY registro_carga IS
        GENERIC (N : INTEGER := 5);
        PORT(
                rstz : IN std_logic;
                clk : IN std_logic;
                carga : IN std_logic;
                entrada : IN std_logic_vector (N-1 DOWNTO 0);
```

```
                    salida : OUT std_logic_vector (N-1 DOWNTO 0)
          );
END registro_carga;

ARCHITECTURE comport OF registro_carga IS
SIGNAL interna, valor : std_logic_vector (N-1 DOWNTO 0);
BEGIN
  Mux: interna <= entrada WHEN (carga ='1') ELSE valor; --realimentación
  Sinc: PROCESS(rstz,clk)
  BEGIN
                    IF (rstz='0') THEN
                              valor<=(OTHERS=>'0');
                    ELSIF RISING_EDGE(clk) THEN
                              valor <= interna;
                    END IF;
  END PROCESS sinc;
  Salida <= valor;
END comport;
```

En éste ejemplo hemos utilizado la técnica de dos procesos concurrentes. El proceso *sinc* es una estructura idéntica a la del ejemplo1.1. Hemos hecho uso de dos señales auxiliares, *interna* y *valor*. La señal *interna* es la salida del multiplexor que realimenta el valor almacenado. La señal *valor* se puede quitar, ya que su función es simplemente la de evitar que la realimentación se produzca sobre la señal *salida*, que al ser de tipo **out** no puede aparecer a la derecha en una asignación.

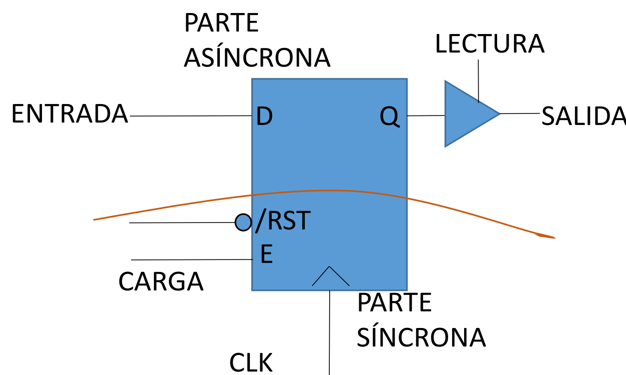


Podría construirse cambiando salida al tipo **inout** o al tipo **buffer**, que es más adecuado. Sin embargo no se utiliza en tipo **inout** porque no describe bien el tipo de la señal. Tampoco lo definimos tipo **buffer** porque existen algunos sintetizadores que tratan el tipo **buffer** de una manera no estándar, reservando la palabra **buffer** para forzar la inclusión de un circuito de amplificación de corriente. El esquema presenta la posición de las diferentes señales en el registro.

La versión con un solo proceso y con salida en alta impedancia:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY registro_carga IS
        GENERIC (N : INTEGER := 5);
        PORT(
                rstz : IN std_logic;
                clk : IN std_logic;
                carga : IN std_logic;
                lectura: IN std_logic;
                entrada : IN std_logic_vector (N-1 DOWNTO 0);
                salida : INOUT std_logic_vector (N-1 DOWNTO 0)
        );
END registro_carga;

ARCHITECTURE comport OF registro_carga IS
SIGNAL valor : std_logic_vector (N-1 downto  0);
BEGIN

  salida<=valor WHEN lectura='1' ELSE (OTHERS=>'Z');

  Sinc: PROCESS(rstz,clk)
  BEGIN
        IF (rstz='0') THEN
                valor<=(OTHERS=>'0');
        ELSIF RISING_EDGE(clk) THEN
                IF (carga ='1') THEN
                        valor <= entrada;
                END IF;
        END IF;
  END PROCESS sinc;
END comport;
```

Este código realiza una función idéntica al anterior, pero una descripción más compleja y menos legible. Hemos introducido una señal, *lectura*, que pone el *valor* a la salida, y en caso contrario, la deja en alta impedancia. Esto es útil en el caso de la construcción de memorias, como la del **ejemplo 8.** El ejemplo se refleja en el siguiente esquema. La salida está basado en una puerta triestado que depende de una señal *lectura*. El esquema presenta una primitiva que refleja la arquitectura propuesta, aunque el proceso síncrono ha sido modificado



Ejemplo 3. Codificación de un contador

El código propuesto es:

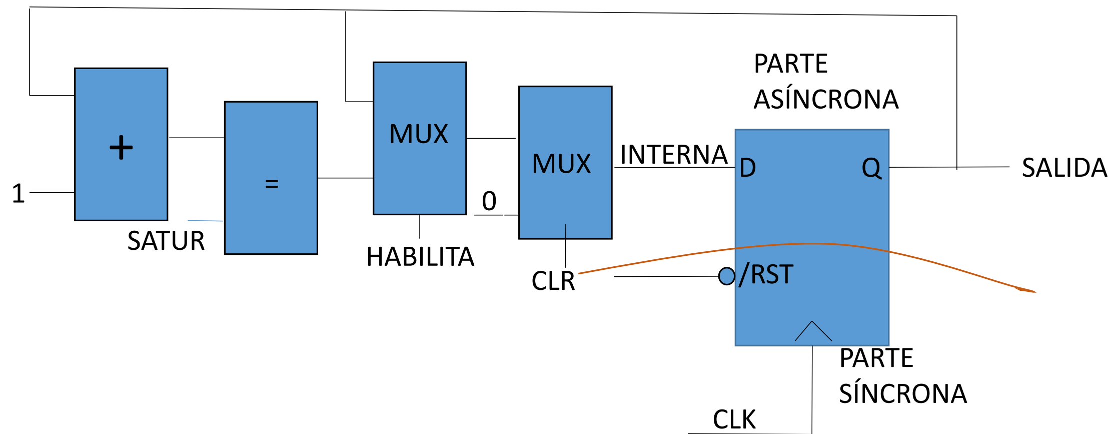
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

ENTITY contador IS
        GENERIC (satur : INTEGER := 13; N : INTEGER := 5);
        PORT(
                resetz : IN std_logic;
                clk : IN std_logic;
                habilita : IN std_logic;-- poner en marcha el contador
                clr: IN std_logic;        -- puesta a '0' síncrona
                salida : OUT std_logic_vector (N-1 DOWNTO 0)
        );
END contador;

ARCHITECTURE comport OF contador IS
SIGNAL interna, valor : unsigned (N-1 DOWNTO 0);
BEGIN
  Cont: PROCESS(clr,habilita,valor)
  BEGIN
        IF (clr='1') THEN
                interna<=(OTHERS=>'0');
        ELSIF (habilita='1') THEN
                IF (valor=satur) THEN -- comparación de la saturación
                        interna<=(OTHERS=>'0');
                ELSE
                        interna<=valor + 1;
                END IF;
        ELSE
                interna<=valor;
        END IF;
  END PROCESS cont;

  Sinc: PROCESS(resetz,clk)
  BEGIN
        IF (resetz='0') THEN
                valor<=(OTHERS=>'0');
        ELSIF RISING_EDGE(clk)  THEN
                valor <= interna;
        END IF;
  END PROCESS sinc;
  Salida <= Std_logic_vector(valor);
END comport;
```

En este ejemplo hemos mantenido la estructura de dos procesos concurrentes y el proceso síncrono es una copia del ejemplo 2. Además hemos introducido varios elementos nuevos. El valor de saturación se ha introducido en el campo **generic**, como parámetro, aunque podría incluirse en el campo **port** como señal. Hay que puntualizar ciertos aspectos acerca de la comparación del valor corriente con el valor de saturación:

a) La comparación se satisface cuando el contador alcanza el valor *satur*, que debe ser alcanzable por la señal *valor*. (p.e. *satur* = 35 generaría un error).
b) La comparación se realiza entre un tipo *unsigned* y un entero. Esta función está incluida en la librería.
c) El número de ciclos de reloj necesario para saturar el contador es *satur*.

En cuanto al operador suma hemos utilizado un esquema siguiendo las recomendaciones dadas en apartados anteriores, sobre la librería *numeric_std*. El operador '+' representa la función suma entre los tipos *unsigned* y *integer*, función que existe en la librería también.

El contador dispone de una señal *clr* de 'puesta a 0 síncrona'. Su comportamiento difiere del de *rstz*, ya que si *clr* toma el valor '1', el contador evolucionará al valor 0 en el siguiente flanco activo de reloj *clk*.



Ejemplo 4. Codificación de una máquina de estados finita tipo Moore.

Una máquina de estados finitos tipo Moore es una máquina de estados que evoluciona síncronamente en función del estado y las entradas y cuyas salidas toman valores en función del estado en que se encuentra. Para sistematizar su codificación cada uno de los elementos de la máquina puede ser definido por un proceso concurrente:

a) Proceso de la evolución de la máquina
b) Proceso síncrono
c) Proceso de las salidas.

El código correspondiente a este ejemplo es el siguiente:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY MaqMoore IS
        PORT(
                rstz : IN std_logic;
                clk : IN std_logic;
                activacion : IN std_logic;-- poner en marcha la máquina
                retorno: IN std_logic;        -- retorna a reposo
                salida : OUT std_logic                );
END MaqMoore;
ARCHITECTURE comport OF MaqMoore IS
TYPE estado IS (reposo, activo, espera);
SIGNAL actual, futuro: estado;
```

```
BEGIN
SubsistemaFsm: PROCESS ( actual, activacion, retorno)
BEGIN
        CASE actual IS
                WHEN reposo =>
                        IF (activacion='1') THEN
                                futuro<=activo;
                        ELSE
                                futuro<=reposo;
                        END IF;
                WHEN activo =>
                        futuro<=espera;
                WHEN retorno =>
                        IF (retorno='0') THEN
                                futuro<=activo;
                        ELSIF (activacion='1') THEN
                                futuro<=espera;
                        ELSE
                                futuro<=reposo;
                        END IF;
                WHEN OTHERS =>
                        futuro<=reposo;
        END CASE;
END PROCESS SubsistemaFsm;

SubsistemaSinc: PROCESS(resetz,clk)
  BEGIN
        IF (resetz='0') THEN
                actual<=reposo;
        ELSIF RISING_EDGE(clk) THEN
                actual <= futuro;
        END IF;
  END PROCESS SubsistemaSinc;

SubsistemaOUT: salida<='1' WHEN actual=activo ELSE '0'; -- valor salida

END comport;
```

Este ejemplo muestra un método general para la resolución de una máquina de estados. En primer lugar el subsistema de la evolución de los estados que se definen como un tipo enumerado, y su formulación física depende de qué codificación se esté utilizando. Esto significa que la relación entre el código del estado y su nombre viene determinado por su ubicación en la lista de definición del tipo estado. Supongamos una codificación binaria: así el código de reposo es “00” (0), el de activo es “01” (1) y el de espera es “10” (2). El código “11” (3) no es alcanzable, por ello se le da una salida a un hipotético alcance de ese estado. WHEN OTHERS => es una cláusula preventiva para este caso, y se considera una buena práctica su inclusión.

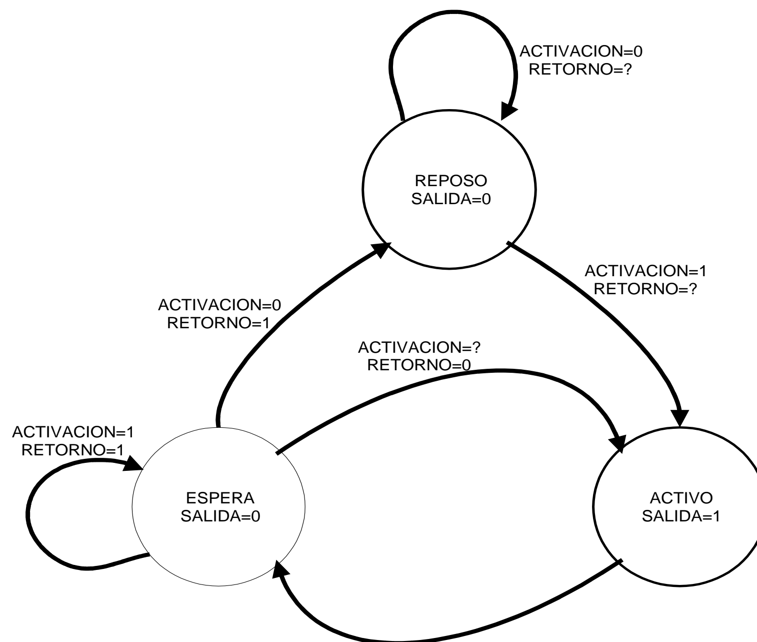
Otros tipos de codificación de la máquina de estados posibles son ‘one hot’, ‘zero hot’, ‘two hots’, Humming, Gray, Johnson, Secuencial… Cada uno realizará una proyección hardware distinta durante la síntesis.

El subsistema de sincronismo es igual que en los ejemplos anteriores y es una buena práctica mantener una misma estructura. Finalmente el subsistema que define las salidas que

es función de que la máquina se encuentre en un estado. Tradicionalmente se incluye en la máquina, aunque por claridad se recomienda que se trate aparte.

Esta forma de sistematización de redacción del código de las máquinas de estado tiene una ventaja: es muy fácil realizar la depuración, quedando claro a qué parte del código corresponden los elementos específicos de una máquina Moore. Además conociendo cómo se realiza entre ingenieros de un mismo equipo se facilita esta.

Finalmente la evolución se produce en flancos activos de reloj mediante el proceso de



sincronismo, que tiene una estructura idéntica al de los ejemplos anteriores, lo que favorece la sistematización.

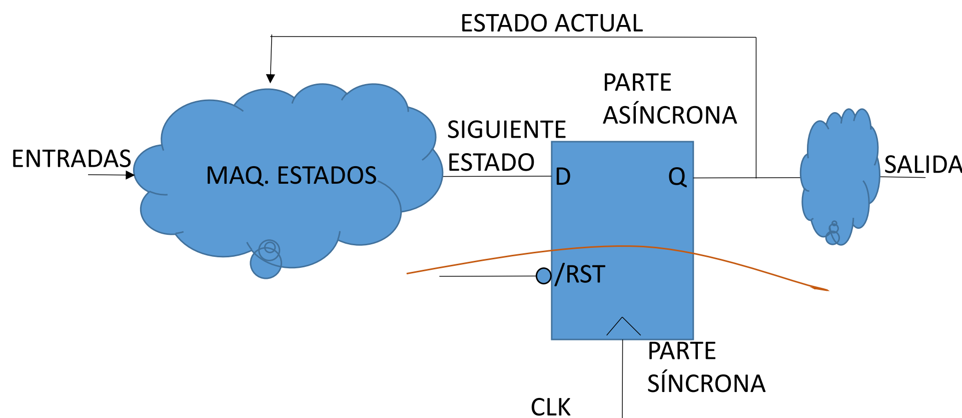


Figura 4. Ejemplo de máquina de Moore

Ejemplo 5. Codificación de una máquina de estados finita tipo Mealy

Una máquina de estados finitos tipo Mealy es una máquina de estados que evoluciona síncronamente en función del estado y las entradas y cuyas salidas toman valores en función del estado en que se encuentra y de las propias entradas.

El código correspondiente a este ejemplo es el siguiente:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY MaqMealy IS
        PORT(
                resetz : IN std_logic;
                clk : IN std_logic;
                activacion : IN std_logic;-- poner en marcha el contador
                retorno: IN std_logic;        -- puesta a '0' síncrona
                salida : OUT std_logic
                );
END MaqMealy;
ARCHITECTURE comport OF MaqMealy IS
TYPE estado IS (reposo, activo, espera);
SIGNAL actual, futuro: estado;
BEGIN
fsm: PROCESS ( actual, activacion, retorno)
BEGIN
        CASE actual IS
                WHEN reposo =>
                        IF (activacion='1') THEN
                                salida<='1';
                                futuro<=activo;
                        ELSE
                                salida<='0';
                                futuro<=reposo;
                        END IF;
                WHEN activo =>
                        salida<='1';
                        futuro<=espera;
                WHEN espera =>
                        IF (retorno='0') THEN
                                salida<='1';
                                futuro<=activo;
                        ELSIF (activacion='1') THEN
                                salida<='0';
                                futuro<=espera;
                        ELSE
                                salida<='0'; -- valor salida
                                futuro<=reposo;
                        END IF;
                END CASE;
END PROCESS fsm;

Sinc: PROCESS(resetz,clk)
  BEGIN
```

```
        IF (resetz='0') THEN
                actual<=reposo;
        ELSIF RISING_EDGE(clk) THEN
                actual <= futuro;
        END IF;
    END PROCESS sinc;
END comport;
```

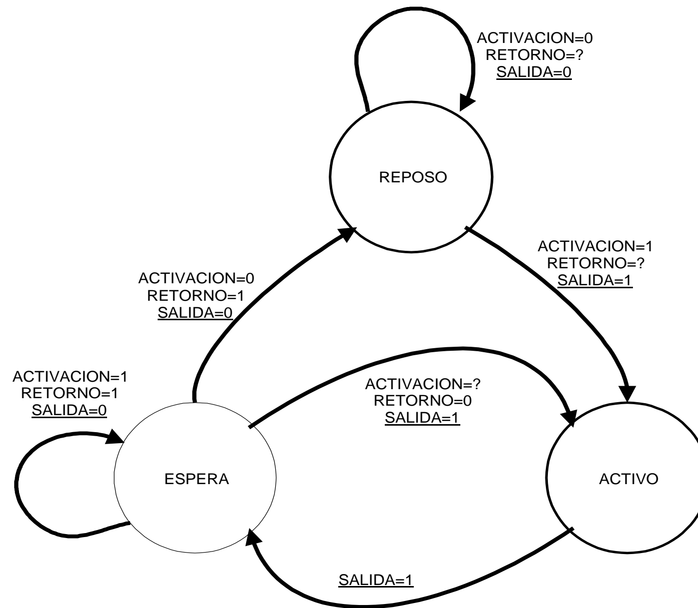
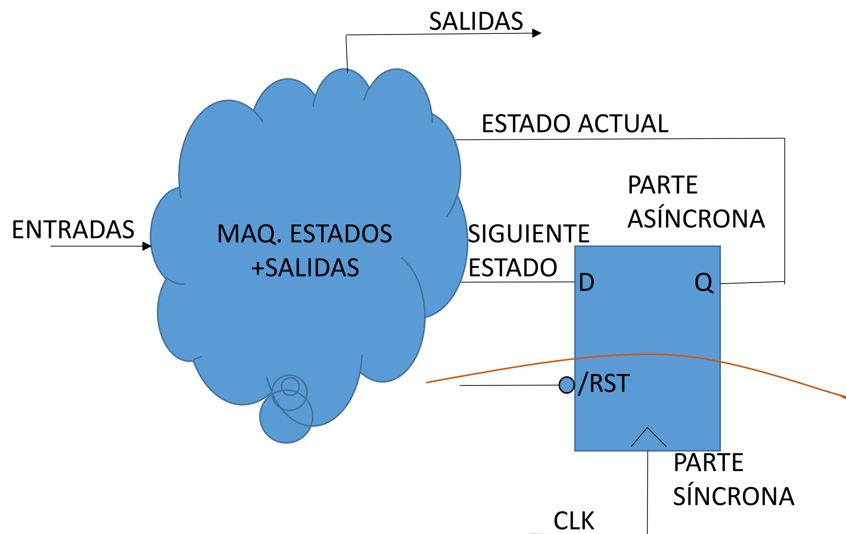


Figura 5. Ejemplo de máquina de Mealy

Desde un punto de vista puramente formal no existen razones para recomendar el uso de un tipo de máquina u otro, incuso cuando la dimensión de la máquina es grande, se produce un ligero ahorro en el hardware resultante, pero desde la perspectiva de una buena implementación se recomienda la forma Moore. La forma Mealy presenta problemas con los pulsos espúreos (*glitches*) en las entradas ya que los transfiere a la salida. La forma Moore presenta una barrera a este tipo de efectos.



2. Codificación de sistemas aritméticos.

Una de las grandes aportaciones de los lenguajes de descripción de hardware es el la de poder trabajar con operadores aritméticos sin preocuparse de su forma o estructura, que por regla general sigue un patrón regular. Sin embargo por una parte este hecho supone una pérdida en el control de la topología de los propios circuitos, aunque existen mecanismos para recuperarla. De alguna manera, mediante la inserción de un signo + ó *, dejamos que la herramienta de síntesis automática haga el trabajo más tedioso. En la era de los esquemáticos había que realizar una verificación exhaustiva del módulo en cuestión mediante un test funcional que prácticamente contemplara todas las posibles combinaciones de las entradas.

En este apartado introducimos unos ejemplos de utilización de estos operadores y de cómo su utilización afectaría al resultado.

Ejemplo 6. Operador aritmético sin signo.

Proponemos un sumador:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

ENTITY Sumador IS
  GENERIC(S1: integer:=8; S2: integer:=5);
  PORT(
        A : IN std_logic_vector(S1-1 downto 0);
        B : IN std_logic_vector(S2-1 downto 0);
        SUMA: OUT std_logic_vector(S1-1 downto 0) --suponiendo que S1>S2
  );
END Sumador;
ARCHITECTURE comport OF Sumador IS
Signal Au : Unsigned(S1-1 downto 0);
Signal Bu : Unsigned(S2-1 downto 0);
Signal SumaU: Unsigned(S1-1 downto 0);

BEGIN
        SumaU <= Au + Bu;

        Au <= Unsigned(A);
        Bu <= Unsigned(B);
        SUMA <= Std_logic_vector(SumaU);
END comport;
```

El ejemplo propone un sumador totalmente basado en números sin signo. En este caso los tipos *std_logic_vector* se convierten a tipo *Unsigned*. Luego se realiza la conversión inversa al tipo *Std_logic_vector*, todo mediante sentencias concurrentes.

Por otra parte es interesante observar el tratamiento del bit de acarreo. La función suma estándar no contempla la posibilidad de que el valor del resultado SUMA exceda de la capacidad de representación del mayor de los valores de A ó B. Es por tanto muy peligroso utilizar estas funciones sin haber protegido esta posibilidad. Por tanto proponemos este código:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

ENTITY Sumador IS
  GENERIC(S1: integer:=8; S2: integer:=5);
   PORT(
        A : IN std_logic_vector(S1-1 downto 0);
        B : IN std_logic_vector(S2-1 downto 0); Asumimos S1>S2 siempre
        SUMA: OUT std_logic_vector(S1 downto 0) --suponiendo que S1>S2
  );
END Sumador;
ARCHITECTURE comport OF Sumador IS
SIGNAL Au, SumaU: Unsigned(S1 downto 0);
SIGNAL Bu : Unsigned(S2 downto 0);

BEGIN
        Au<= ”0” & Unsigned(A); --concatenar un cero por delante por ser unsigned.
        Bu<= ”0” & Unsigned(B);
        SumaU <= Au + Bu;

        SUMA <= Std_logic_vector(SumaU);

END comport;
```

Resulta, pues, muy útil revisar los códigos fuente de las librerías estándar para evitar problemas relacionados con los operadores de este tipo. La concatenación se realizaría:

```
auxA<= A(S1-1) & A;
```

en el caso de trabajar con señales del tipo *signed.* El resultado es un sumador de un bit más

Ejemplo 7. Operador aritmético con el signo extendido.

El código del ejemplo anterior se puede escribir:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;
```

```
ENTITY Sumador IS
  GENERIC(S1: integer:=8; S2: integer:=5);
  PORT(
        A : IN std_logic_vector(S1-1 downto 0);
        B : IN std_logic_vector(S2-1 downto 0);
        SUMA: OUT std_logic_vector(S1-1 downto 0) --suponiendo que S1>S2
  );
END Sumador;
ARCHITECTURE comport OF Sumador IS
SIGNAL sgA: signed(S1 downto 0);
SIGNAL usgB: unsigned(S2 downto 0);
SIGNAL intB: interger range 0 to 2**(S2-1);
SIGNAL SumaS: signed(S1 downto 0);

BEGIN
        sgA <= A(S1-1) & signed(A);
        usgB <= B(S2-1) & unsigned(B);
        intB <= to_integer(intB);

        SumaS <=sgA + intB;

        SUMA <= Std_logic_vector(SumaS);

END comport;
```

Utilizamos los tipos vector con atributo *signed* e integer para poder operar con el tipo *unsigned*, ya que lalibrería no contempla la suma entre estos dos tipos. En este caso no podríamos mezclar los operandos dentro de la misma entidad. El mecanismo de sobrecarga se encargará de seleccionar la operación deseada.

Ejemplo 8. Control de la síntesis de algunos operadores aritméticos.

Uno de los errores más comunes en la elaboración de códigos HDL es la pérdida de perspectiva acerca de los recursos hardware que se emplean. Los diseñadores sin experiencia tienen la tendencia a asociar conocimientos y estructuras software utilizando sintaxis VHDL. El siguiente ejemplo tiene por objeto poner de manifiesto esa idea.

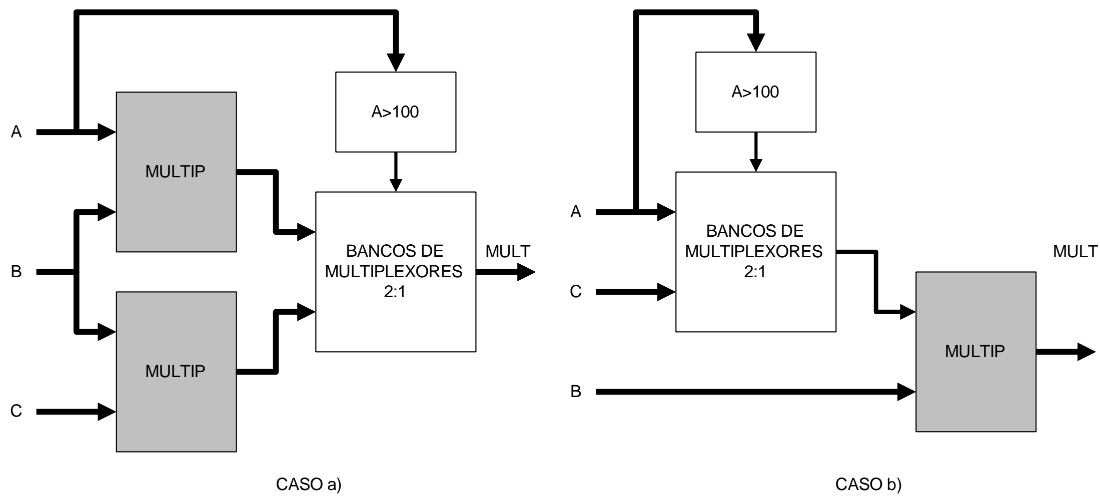
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

ENTITY MultMux IS
  GENERIC(Mb: integer:=8);
  PORT(
        A : IN std_logic_vector(Mb-1 downto 0);
        B : IN std_logic_vector(Mb-1 downto 0);
        C : IN std_logic_vector(Mb-1 downto 0);
        MULT: OUT std_logic_vector(2*Mb-1 downto 0)
   );
END MultMux;

ARCHITECTURE comport OF MultMux IS
SIGNAL sgA,sgB,sgC: SIGNED(Mb-1 downto 0);
SIGNAL sgMULT: SIGNED(2*Mb-1 downto 0);
BEGIN
sgA <= Signed(A);
```

```
sgB <= Signed(B);
sgC <= Signed(C);
mm: PROCESS(sgA,sgB,sgC)
BEGIN
        IF(sgA>100) THEN
                sgMULT<= sgA * sgB;
        ELSE
                sgMULT<= sgC * sgB;
        END IF;
END PROCESS mm;
MULT <= Std_logic_vector(sgMULT);
END comport;
```

Este ejemplo es, desde el punto de vista arquitectural y sintáctico correcto. Sin embargo se precisarán dos multiplicadores para su resolución. Veamos una versión más correcta:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.all;

ENTITY MultMux IS
  GENERIC(Mb: integer:=8);
  PORT(
        A : IN std_logic_vector(Mb-1 downto 0);
        B : IN std_logic_vector(Mb-1 downto 0);
        C : IN std_logic_vector(Mb-1 downto 0);
        MULT: OUT std_logic_vector(2*Mb-1 downto 0)
   );
END MultMux;

ARCHITECTURE comport OF MultMux IS
SIGNAL sgA,sgB,sgC: SIGNED(Mb-1 downto 0);
SIGNAL sgMULT: SIGNED(2*Mb-1 downto 0);
BEGIN
sgA <= Signed(A);
sgB <= Signed(B);
sgC <= Signed(C);
mm: PROCESS(sgA,sgB,sgC)
VARIABLE sgQ: SIGNED(Mb-1 downto 0);
BEGIN
        IF(sgA>100) THEN
                sgQ:= A;
        ELSE
                sgQ:= C;
        END IF;
        sgMULT<= sgQ * sgB;
END PROCESS mm;
MULT <= Std_logic_vector(sgMULT);
END comport;
```

La figura representa un esquema de la imagen hardware en ambos casos a) y b), siendo el caso b) de los ejemplos mucho más compacto. Se han mantenido las distintas conversiones para que sean sintácticamente correctas.



3. Codificación de módulos de memoria.

Un ejercicio muy completo de codificación consiste en la creación de bloques de memoria parametrizados. Utilizaremos el registro del ejemplo 2 como un componente de este código. Además construiremos el decodificador de entrada y de salida.

Ejemplo 9. Memoria RAM síncrona de tamaño genérico.

El código del decodificador de N entradas puede ser:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY decod IS
   GENERIC(N: INTEGER :=2);
   PORT (
            enable : IN std_logic;
            address: IN std_logic_vector(N-1 DOWNTO 0);
            pointers: OUT std_logic_vector (2**N-1 DOWNTO 0)
            );
END decod;

ARCHITECTURE comport OF decod IS
SIGNAL aux: INTEGER;
SIGNAL usgaddress(N-1 DOWNTO 0);
BEGIN
usgaddress <= unsigned(address);
aux<=TO_INTEGER(usgaddress);
dec: PROCESS(enable,aux)
BEGIN
    IF (enable='1') THEN
            FOR i IN 0 TO 2**N-1 LOOP
             IF aux=i THEN
                        pointers(i)<='1' ;
              ELSE
```

```
                pointers(i)<='0' ;
            END IF;
          END loop;
    ELSE
          pointers<=(others=>'0');
    END IF;
END PROCESS dec;
END comport;
```

Ahora construimos un código para la memoria. Utilizaremos tres parámetros libres, Nbdirecc, Npalabras y Nbpalabra, para dimensionar la memoria. Con ellos definimos, respectivamente el tamaño del bus de direcciones, el número efectivo de registros y el tamaño de cada uno de ellos.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY memoria IS
   GENERIC(Nbdirecc: INTEGER :=2;Npalabras: INTEGER :=2; Nbpalabra: INTEGER :=2);
   PORT (
          clk: IN std_logic;
          rstz: IN std_logic;
          escribe: IN std_logic;
          Dato: INOUT std_logic_vector(Nbpalabra -1 DOWNTO 0);
          Direccion: IN std_logic_vector(Nbdirecc -1 DOWNTO 0)
          );
END memoria;

ARCHITECTURE comport OF memoria IS
COMPONENT decod
  GENERIC(N: INTEGER :=2);
   PORT (
          enable : IN std_logic;
          address: IN std_logic_vector(N-1 DOWNTO 0);
          pointers: OUT std_logic_vector (2**N-1 DOWNTO 0)
          );
END COMPONENT;

COMPONENT  registro_carga
  GENERIC (N : INTEGER := 5);
  PORT(
          rstz : IN std_logic;
          clk : IN std_logic;
          carga : IN std_logic;
          lectura: IN std_logic;
          entrada : IN std_logic_vector (N-1 DOWNTO 0);
          salida : OUT std_logic_vector (N-1 DOWNTO 0)
          );
END COMPONENT;
SIGNAL lee : std_logic;
SIGNAL punterolee, punteroescribe : std_logic_vector(2** Nbdirecc-1 DOWNTO 0);
SIGNAL DatoIn, DatoOut: std_logic_vector(Nbpalabra-1 DOWNTO 0);
BEGIN
decodEscr: decod
  GENERIC MAP(N=>Nbdirecc)
   PORT MAP(enable => escribe, address=>direccion, pointers=> punteroescribe);

lee<= NOT escribe;
```

```
decodLect: decod
  GENERIC MAP(N=>Nbdirecc)
   PORT MAP(enable => lee, address=>direccion, pointers=> punterolee);

FOR i IN 0 TO Npalabras-1 GENERATE
reg: registro_carga
  GENERIC MAP(N => Nbpalabra)
  PORT MAP(rstz =>rstz, clk =>clk, carga => punteroescribe(i), lectura=> punterolee(i),
         entrada =>Datoin, salida =>Datoout);
END GENERATE;


--Para evitar dejar las señales DatoOut a 'Z' mientras se escribe
DatoOut <= (OTHERS=>'0') WHEN(lee='0') ELSE (OTHERS=>'Z');

--La entrada/salida de Dato es bidireccional. Esto se resuelve mediante las sentencias
DatoIn<=Dato; --Entrada
Dato <= DatoOut WHEN(lee='1') ELSE (OTHERS=>'Z'); --Salida

END comport;
```

En este ejemplo hemos construido un multiplexor realizado con puertas triestado. Está construido implícitamente utilizando cada registro del bucle **generate** conectando sus salidas a la señal DatoOut. La activación se realiza mediante la presencia de un decodificador. Sin embargo existe el riesgo de dejar el bus en alta impedancia. Para este caso hemos puesto una salvaguarda de dejarlo a cero en caso de que se esté en proceso de escritura. En este caso el decodificador no se habilitaría y el bus de salida quedaría en alta impedancia. Las sentencias finales son un ejemplo de cómo realizar un bus bidireccional típico.

CAPÍTULO VI. VHDL NO SINTETIZABLE. SIMULACIÓN Y MODELADO

En este capítulo haremos una breve introducción a conceptos de VHDL relacionados con su capacidad para describir tecnologías y entornos. El objetivo de este capítulo es ofrecer un perspectiva diferente a la hasta el momento expuesta, ya que la visión del VHDL sintetizable ofrece una perspectiva muy parcial del lenguaje, si bien es la más útil desde el punto de vista de un diseñador.

Ahora podremos desarrollar una perspectiva completa de un sistema, ya que podremos integrar en el mismo código Sistema y Circuito Digital, coexistiendo ambos en igual ámbito de simulación.

1. Mecanismo de simulación.

El VHDL así como el resto de los lenguajes de simulación utilizan un sistema basado en *tabla de eventos* para evaluar las sentencias del código y simulación de una entidad que ejecuta las instrucciones paralelamente. La lista de sensibilidad representa un mecanismo para vigilar la evaluación de un proceso cuando la señal sufre un cambio y por tanto emula el paralelismo inherente al hardware.

El sistema de simulación se basa en la evolución de una variable tiempo que se incrementa en un tiempo mínimo de discretización o *delta_time*, y por tanto configura el entorno de simulación. Esa base de tiempo es el nivel máximo de detalle que permite discriminar la simulación. Por tanto podremos introducir modelos que describen un comportamiento de los elementos hardware sin más que implementar sus respuestas funcionales y temporales.

Simulación funcional, tiempo cero

Es una simulación con retrasos en la lógica combinacional nulos, y en la lógica secuencial mayores que cero, es decir, los tiempos de propagación de las señales a través de la lógica se consideran cero y por tanto solamente se tiene en cuenta su respuesta funcional. Para poder hacer funcionar la lógica secuencial es imprescindible dotar a los elementos de memoria con un retardo no nulo. Es fácil ver qué pasaría en caso de no proceder así. Por ejemplo un registro de desplazamiento no desplazaría cada ciclo como corresponde a su funcionamiento.

Esta simulación sirve para realizar una verificación estrictamente funcional.

Simulación temporal

Es una simulación característica de sistemas ya sintetizados donde se trabaja con circuitos descritos con primitivas y asociados a una tecnología de un fabricante. Estos modelos de

primitivas disponen ya de datos temporales. En este caso la simulación genera formas de onda mucho más cercanas al comportamiento final. A partir de una simulación de este tipo se manifiestan diversos comportamientos típicos, retardos, como spikes, pulsos,... etc.

Hay dos tipos de retardos: inerciales y de transporte. Los retardos inerciales consisten en que el retardo tiene inercia, y esa ha de ser superada para que el circuito descrito manifieste su valor a la salida. Por ejemplo, si una puerta tiene un retardo inercial de 20ns y le llega un pulso en una entrada menor que 20ns, por ejemplo 10ns, este pulso no tendría suficiente tiempo (y por tanto anchura del pulso) para vencer la inercia. Este es el retardo por defecto en VHDL.

El otro modelo de retardo es el de transporte, más bien referido a conexionados, es decir, se simulan los retrasos de las puertas lógicas y se incorporan retardos de las líneas obtenidos del programa de ruteado siendo un comportamiento muy próximo al real. Este modelo de retardo requiere una definición adicional: **transport**.

2. Sentencias para modelado.

El control del flujo de simulación se realiza mediante los siguientes comandos no sintetizables AFTER y WAIT.

Sentencia AFTER

Se utiliza para controlar la evolución de una señal a lo largo del tiempo. Los incrementos temporales han de ser variables del tipo **time.**

```
Señal <= valor AFTER tiempo, valor AFTER tiempo, ... ;
```

Por ejemplo una señal oscilante puede construirse:

```
clock <= NOT clock AFTER 30 ns;
```

Una señal que realiza un pulso a nive bajo durante 1 microsegundo.

```
Rstz<='1','0' AFTER 10 ns, '1' AFTER 1 us;
```

Un comportamiento realista de una primitiva de una puerta AND se describe en el siguiente ejemplo, representando un comportamiento inercial:

```
ENTITY and2 IS
        PORT(a, b: IN STD_LOGIC;
             Y: OUT STD_LOGIC );
```

```
END and2;
ARCHITECTURE comport OF and2 IS
BEGIN
	y <= a AND b AFTER 5 ns;
END comport;
```

Si a sufre un pulso de tamaño menor que 5ns la puerta no sufriría ningún cambio incluso en el caso de que la función lógica obligara a un cambio en la salida.

Para simular un retardo de transporte en el que el valor se manifiesta al cabo de un cierto tiempo:

```
b<=TRANSPORT a AFTER 10 ns;
```

Sentencia WAIT

Permite al diseñador suspender la ejecución secuencial de un proceso. Es por tanto una sentencia que está siempre insertada dentro de la ejecución de un process. Dado que establece un control sobre el mismo, es necesario que el proceso sea de ejecución infinita, es decir, no exista la lista de sensibilidad.

Existen tres formas de trabajar con la sentencia wait:

- WAIT ON lista de señales. Se vigilan cambios en las señales.
- WAIT UNTIL condición. Se vigila el cumplimiento de una sentencia.
- WAIT FOR tiempo. Detiene el proceso durante un tiempo

Igualmente es posible realizar combinaciones de distintas Veamos algunos conceptos asociados a estas sentencias.

WAIT ON lista de señales. Se detiene el proceso a la espera que las señales que contiene la lista sufran algún evento. Realmente cuando existe más de una señal se vigila el *or* de los atributos *EVENT* de las señales.

Por ejemplo el típico proceso de sincronismo se podría escribir:

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

```
        WAIT ON rstz,clk;
END PROCESS;
```

Realizando la sentencia WAIT ON la tarea de la lista de sensibilidad.

WAIT UNTIL condición. La condición que se vigila en este caso es el cumplimiento de una sentencia booleana, de lo contrario el proceso permanece suspendido. Por ejemplo, si a y b son cantidades numérica y c un std_logic, se puede escribir:

```
WAIT UNTIL (a>b) AND (c='1');
```

WAIT FOR tiempo. La variable debe ser del tipo **time**. También se pueden introducir expresiones complejas que devuelven un valor del tipo time. Por ejemplo, si a y b son del tipo time:

```
WAIT FOR (a-b);
```

es una expresión válida.

Condiciones complejas. Se pueden realizar complejas combinando las diferentes formas anteriores. En este caso se realiza el *and* de todas las combinaciones:

```
WAIT ON a,b UNTIL (a='1') AND (b='0') FOR 5 ns;
```

Se esperaría un cambio en a y b, continuará cuando este cambio satisfaga la condición pero al cabo de 5 ns.

2.1 Lista de sensibilidad y WAIT ON.

Se ha comentado el hecho de que la lista de sensibilidad y la sentencia **WAIT ON** al final del proceso son equivalentes. Pero, ¿por qué al final del proceso y no al principio? La respuesta es sencilla. Durante la inicialización de los mecanismos de simulación todos los procesos se ejecutan una vez, independientemente de su lista de sensibilidad. Por tanto para evitar que no se detenga la ejecución del mismo durante los tiempos iniciales es preciso situar **WAIT ON** al final.

2.2 Concurrencia de procesos.

En el capítulo 3 se explica cómo se producen las asignaciones de señales (tipo **signal**) dentro de los procesos. Una asignación de una señal dentro del proceso no es efectiva hasta la resolución del mismo al final. Esto no ocurre así con las variables temporales (tipo **variable**). Veamos un pequeño ejemplo:

```
PROCESS(a)
BEGIN
        k<=0;
        IF(a='1') THEN k<='1';
        END IF;

        IF (k='0') THEN b<=a;
        ELSE b<='0';
        END IF;

END PROCESS;
```

En este caso k no toma un nuevo valor hasta el final del proceso, es decir, k no evaluará correctamente la expresión de b<=a, ya que se tomará el valor que k antes de entrar evaluar el proceso.

La sentencia **WAIT** permite resolver este conflicto, si se inserta una sentencia **WAIT ON** después de cada sentencia secuencial. El efecto es que se puede evaluar dicha sentencia debido a que se espera durante un paso de simulación y el simulador la evalúa.

```
PROCESS
BEGIN
        k<=0;
        WAIT FOR 0 ns;
        IF(a='1') THEN k<='1';
        END IF;
        WAIT FOR 0 ns;

        IF (k='0') THEN b<=a;
        ELSE b<='0';
        END IF;
        WAIT ON a;
END PROCESS;
```

Para poder insertar la sentencia **WAIT** es preciso eliminar la lista de sensibilidad. Obviamente la alternativa más elegante y correcta es la de transformar k en una variable.

3. Construir un "Test Bench".

En la práctica una simulación se realiza en lo que se conoce como un "test bench". Éste consiste en un banco de pruebas de nuestro circuito, que se estimula determinando los valores de las señales externas al mismo. Típicamente las dos señales más comunes son el reloj (clk) y la inicialización asíncrona (rstz), en este caso activa en valor '0'. Además suele introducirse un

modelo del sistema exterior al circuito, por ejemplo, un motor eléctrico, las valores de los pixeles de una imagen, una señal muestreada de un convertidor A/D, ... etc.

Asimismo se establecen condiciones temporales que gobiernan señales y modelos. Una fuente de datos suelen ser los ficheros, que usualmente contienen medidas reales de un sistema. El VHDL permite su lectura, escritura y el gobierno de la misma. Al manejo de ficheros dedicaremos la sección 4.

3.1 Modelo de "test bench".

Un entorno de simulación puede registrar señales internas a un circuito y representarlos en formas de onda. La figura representa un esquema de los posibles elementos de un "test bench". También un sistema de ficheros es un mecanismo útil para registrar resultados.

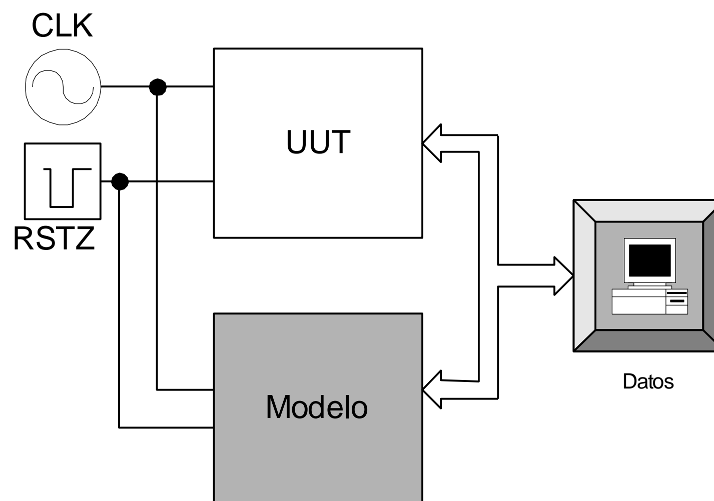


Ilustración 6. Modelo general de un "test bench"

Típicamente un test bench es una entidad sin entradas ni salidas aparentes,aunque pueden determinarse un valor para los parámetros del circuito. Por ejemplo si existe un contador de valor N

```
ENTITY tb IS
        GENERIC( N : integer :=16);
END tb;
```

Un segundo elemento es el reloj. En nuestro caso lo denominamos *clk*:

```
SIGNAL clk : std_logic :='0';
…..
clk <= NOT clk AFTER 20 ns;
….
```

Definimos *clk* como una señal que oscila a 50 Mhz. La asignación es concurrente con el componente que simulamos. Asimismo precisa de un valor inicial, ya que todas las señales en simulación arranca desde un valor desconocido 'Uninitialized' y éste se ha de especificar en su declaración inicial.

El tercer elemento es la señal de inicialización asíncrona. Todo buen diseñador debe poner cuidado en conseguir que el circuito funcione con un estado inicial bien conocido, de ahí que sea aconsejable trabajar siempre de esta forma.

```
SIGNAL rstz : std_logic;
….
rstz <= '1', '0' AFTER 10 ns, '1' AFTER 40 ns;
….
```

Básicamente hemos representado un monostable.

Obviamente el cuarto elemento es el propio circuito objeto de diseño, que en este caso actúa como una unidad de jerarquía inferior del entorno de simulación.

Veamos un ejemplo completo:

```
ENTITY testbench IS
END ENTITY;

ARCHITECTURE ej OF testbench IS
COMPONENT minand
  PORT (
        A, B: IN std_logic;
        Y: OUT std_logic);
END COMPONENT;
SIGNAL Ai, Bi, Yo: std_logic;
BEGIN
  Uut: minand    PORT MAP(A=>Ai, B=>Bi, Y=>Yo);

  Ai <= '0', '1' AFTER 50 ns, '0' AFTER 100 ns, '1' AFTER 150 ns;
  Bi <= '0', '1' AFTER 100 ns;
END ARCHITECTURE ej;
```

4. La librería TextIO

Uno de los paquetes de librerías predefinidas es el que permite acceso a ficheros que almacenan datos y permite, por tanto la inserción de datos en el circuitos generados para la realización, por ejemplo de pruebas complejas. La librería permite tanto escribir como leer datos formateados de ficheros. Son ficheros de caracteres ASCII. Los ficheros han de ir formateados según especifique el usuario, es decir, debe repetirse la estructura de una línea.

Una línea es una cadena de caracteres, datos separados por espacios, que concluye con un retorno de carro. La línea es la unidad de transferencia fundamental.

El procesamiento de un fichero se produce dentro de un proceso, cuya ejecución ha de coordinarse cuidadosamente con señales que lo activan y los valores obtenidos del fichero.

Para poder acceder a un objeto del tipo fichero ha de declararse la librería estándar TextIO:

```
USE std.textio.all;
```

En ella existe el tipo básico FILE, que se utiliza dentro de la sección declarativa de un proceso:

```
FILE infile : TEXT IS IN "fichero.in";
FILE outfile : TEXT IS OUT "fichero.out";
```

y de la misma manera se declara la línea como una variable del tipo line:

```
VARIABLE lineaent, lineasal: LINE;
```

Para controlar el fin de fichero se utiliza la función ENDFILE:

```
WHILE NOT( ENDFILE( infile )) LOOP
……..
END LOOP;
```

Para copiar una línea del fichero de entrada sobre la variable tipo line se utiliza la función READLINE y para escribir sobre el fichero de salida una línea se utiliza la función WRITELINE. Su sintaxis es:

```
READLINE(infile, lineaent);
WRITELINE(outfile,lineasal);
```

Finalmente para extraer un argumento de la línea leída se utiliza la función READ

```
READ(lineaent, argumento1);
READ(lineaent, argumento2);
```

y para escribirlo WRITE:

```
WRITE(lineasal, argumento1);
WRITE(lineasal, argumento2);
```

Un ejemplo completo:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE std.textio.all;

ENTITY testbench IS
END testbench;

ARCHITECTURE tbench_1 OF testbench IS
COMPONENT sf
PORT (
       clk:  IN std_logic;
       rst:   IN std_logic;
       S_a: IN std_logic; -- sensores A
       S_b: IN std_logic; -- B
       S_c: IN std_logic; -- C
       S_d: IN std_logic; -- D
       rav0: OUT std_logic _vector (2 DOWNTO 0); -- Luces calle 0
       rav1: OUT std_logic _vector (2 DOWNTO 0) -- Luces calle 1
    );
END COMPONENT ;

SIGNAL t_clk : std_logic := '0';
SIGNAL t_rst  : std_logic ;
SIGNAL t_sa, t_sb, t_sc, t_sd: std_logic := '0';
SIGNAL rav0,rav1: std_logic _vector (2 DOWNTO 0);

TYPE luces IS (rojo, amarillo, verde, error);
SIGNAL calle0, calle1: luces;

BEGIN

 uut : sf
PORT MAP ( clk=>t_clk, rst=>t_rst, S_a=>t_sa, S_b=>t_sb, S_c=>t_sc,
       S_d=>t_sd, rav0=>rav0, rav1=>rav1) ;

 t_clk <= NOT t_clk AFTER 50 ns;
 t_rst <= '0', '1' AFTER 55 ns;
 t_sa <= '0', '1' AFTER 1000 ns;

PROCESS(rav0, rav1)
BEGIN
       IF (rav0 = "100") THEN
              calle0 <= rojo;
       ELSIF (rav0 = "010") THEN
              calle0 <= amarillo;
       ELSIF (rav0 = "001") THEN
              calle0 <= verde;
       ELSE
              calle0 <= error;
       END IF;

       IF (rav1 = "100" ) THEN
              calle1 <= rojo;
       ELSIF (rav1 = "010" ) THEN
              calle1 <= amarillo;
       ELSIF (rav1 = "001" ) THEN
```

```
                    calle1 <= verde;
            ELSE
                    calle1 <= error;
            END IF;
        END PROCESS;

        textOUT: PROCESS(rav0, rav1)
        FILE fout: TEXT IS OUT "fout.dat"; --declaración de un fichero de salida
        VARIABLE linea: LINE;
        VARIABLE str0: string(1 TO 5) := "rav0=";
        VARIABLE str1: string(1 TO 5) := "rav1=";
        VARIABLE espacio: string(1 TO 5) := "     ";
        BEGIN
            WRITE(linea,str0);
            WRITE (linea, rav0);
            WRITE (linea, espacio);
            WRITE (linea,str1);
            WRITE (linea, rav1);
            WRITELINE(fout,linea);
        END PROCESS;
    END tbench_1;
```

Un caso de lectura y escritura de un fichero con diferentes estructuras es el siguiente:

```
LIBRARY ieee;
  USE IEEE.STD_LOGIC_1164.ALL;
  USE IEEE.STD_LOGIC_MISC.ALL;
  USE IEEE.STD_LOGIC_ARITH.ALL;
  USE IEEE.STD_LOGIC_TEXTIO.ALL;
  USE IEEE.STD_LOGIC_UNSIGNED.ALL;


ENTITY b13_tb IS
END b13_tb;

ARCHITECTURE tb OF b13_tb IS
COMPONENT b13
PORT(
  reset : in STD_LOGIC;
  eoc : in STD_LOGIC;
  soc : out STD_LOGIC;
  load_dato,add_mpx2 : out STD_LOGIC;
  canal : out STD_LOGIC_VECTOR (3 downto 0);
  mux_en : out STD_LOGIC;
  clock : in STD_LOGIC;
  data_in       : in STD_LOGIC_VECTOR ( 7 downto 0 );
  dsr : in STD_LOGIC;
  error : out STD_LOGIC;
  data_out : out STD_LOGIC
);
END COMPONENT;

  SIGNAL clock : STD_LOGIC;
  SIGNAL rst : STD_LOGIC;

 SIGNAL eoc, dsr : STD_LOGIC;
 SIGNAL soc, load_dato, add_mpx2, mux_en, error, data_out : STD_LOGIC;
  SIGNAL canal : STD_LOGIC_VECTOR (3 downto 0);
```

```
    SIGNAL data_in: STD_LOGIC_VECTOR (7 downto 0);


  TYPE st_type IS (initial,read_line_input,STOP);
  SIGNAL st : st_type:=initial;

BEGIN  -- Rtl
b13_map : b13 PORT MAP(reset=>rst, eoc=>eoc, soc=>soc, load_dato=>load_dato,
    add_mpx2=>add_mpx2, canal=>canal, mux_en=>mux_en, clock=>clock,
                    data_in=>data_in, dsr=>dsr, error=>error, data_out=>data_out);

--main process
p_main: PROCESS (clock,rst)

VARIABLE line_in,line_out : line;
VARIABLE v_eoc,v_dsr: STD_LOGIC;
VARIABLE v_data_in : STD_LOGIC_VECTOR(7 downto 0);
FILE filein : TEXT OPEN read_mode is "../b13.inp";
FILE fileout : TEXT OPEN write_mode is "../b13.out";
BEGIN
    IF  RISING_EDGE(clock) THEN  -- rising clk edge (don't like this coding style at all
        CASE st IS
            WHEN initial =>
                st<=read_line_input; --Just for future usues
                READLINE(filein,line_in);
                IF(line_in(1)='#') THEN rst <= '1', '0' after 22 ns;
                ELSE rst <= '0';
                END IF;
                data_in<=(others=>'0');
                eoc <= '0';
                dsr <= '0';
            WHEN read_line_input =>
                data_in<=(OTHERS=>'0');
                eoc <= '0';
                dsr <= '0';
                READLINE(filein,line_in);
                IF(line_in(1)='#') THEN rst <= '1', '0' after 20 ns;
                ELSIF(line_in(1)='.') THEN
                    FILE_CLOSE(filein);
                    FILE_CLOSE(fileout);
                    assert false report "Simulation completed" severity failure;
                    st <= STOP;
                ELSE --module that should change for each benchmark
                    READ(line_in,v_eoc);
                    READ(line_in,v_data_in);
                    READ(line_in,v_dsr);
                    eoc <= v_eoc;
                    data_in <= v_data_in;
                    dsr<= v_dsr;
                END IF;
                WRITE(line_out,soc);
                WRITE(line_out,load_dato);
                WRITE(line_out,add_mpx2);
                WRITE(line_out,canale);
                WRITE(line_out,mux_en);
                WRITE(line_out,error);
                WRITE(line_out,data_out);
```

```
                        WRITELINE(fileout,line_out);
                WHEN STOP =>
                        null;
                END CASE;

    END IF;
END PROCESS;

clkgen : PROCESS
    BEGIN
        clock <= '1';
        LOOP
                        WAIT FOR 10 ns;
                        clock<=NOT clock;
        END LOOP;
    END PROCESS;

end tb;
```

CAPÍTULO VII. BUENAS PRÁCTICAS CODIFICANDO EN VHDL

Se han publicado numerosos libros de VHDL, demostrando que los HDLs son muy flexibles. Existen múltiples maneras de codificar un diseño en VHDL, pero hay razones para que la codificación sea siguiendo unas pautas a la hora de generar un código. Estas reglas generales se basan en:

- El código debe sintetizarse en todos los entornos independientemente del fabricante.
- El código debe ser claro, ya que es habitual trabajar en equipos y por tanto el estilo debe facilitar su legibilidad por otros miembros del equipo.
- El código ha de cumplir unas normas de estilo. Seguir unas directrices únicas facilita su interpretación y la reusabilidad de un diseño.
- El diseñador debe controlar bien cuanto sea posible el circuito que se sintetizará desde el código. Esto significa que se debe seguir unas pautas de codificación, que lleven a un único circuito con la misma arquitectura, considerando cualquier sitetizador.
- El código debe ser fácilmente depurable. Cuanto más rigurosa sea la escritura del código más fácil será el proceso de la detección de errores.

Éstas y otras razones llevan a definir unas reglas de estilo de codificación, usualmente en grado de recomendación, pero que si desea calificar el código de un diseño bajo norma o se pretende que se aceptado por los grandes centros de diseño se deben seguir fielmente. Cada una se le designa por un código, que en nuestro caso es propio de este capítulo. En particular serán una mezcla de dos juegos de reglas, las numeradas por 0XX son recomendaciones para la norma DO-254 y las 1XX son aquella que adicionalmente recomienda la agencia espacial francesa, CNES, para realizar diseños. Se ha evitado duplicar reglas, por tanto aquellas ya escritas para la norma DO-254 se omiten en las de CNES. Las reglas se toman de estos dos documentos y se mencionan como bibliografía.

Las reglas de estilo se pueden dividir en cuatro grandes grupos de reglas permitiendo una clasificación inicial:

1. Prácticas de codificación,
2. Tratamiento de la señal de reloj
3. Reglas para una síntesis segura
4. Facilitar la revisión de un código

Cada regla tiene un grado de cumplimiento. El seguimiento o no de una regla tiene diferente calificación, que categorizaremos como 'Error', 'Warning' ó 'Note', lo que provee una medida del impacto de la regla en la calidad de un código.

Como norma general, los errores deben ser corregidos, las alarmas deben también corregirse, aunque se admiten excepciones bien justificadas y documentadas. Las notas deben ser simplemente verificadas.

El seguimiento de las normas de estilo es sólo parte del problema. La otra parte se encuentra en la adecuada elección de los parámetros de síntesis, aunque esto ya depende de la herramienta que se esté utilizando.

7.1 Prácticas de codificación

PC001. Evitar el usar tipos incorrectos de VHDL. Categoría: Error

No usar declaraciones de señal incorrecta, en cuanto a tipos, rangos y/o cotas.

PC002. Evitar hacer asignaciones duplicadas. Categoría: Warning

La misma señal no debe asignarse más de una vez en la misma región de código.

PC003. Evitar asignaciones a valores de tamaño prefijado. Categoría: Warning

Para diseñar IPs reutilizables o por razones de portabilidad, el utilizar valores numéricos de longitud prefijado resulta una limitación.

PC004. Evitar asignaciones de código a vectores de tamaño fijo. Categoría: Note.

Para asignaciones en valores de reset, usar preferentemente la macro con tamaño variable.

PC005. Utilizar un estilo consistente de codificación de FSM. Categoría: Error

a. Un diseño de una FSM debe ser definido de una forma consistente sobre un conjunto consistente de estados.
b. La codificación de los estados de una FSM debe permitir ser flexible, no deben ser prefijada usando tipos enumerados.

PC006. Utilizar las transiciones de las FSM de manera segura. Categoría: Error

a. La FSM debe tener definido un estado de reset bien definido.
b. Todos los estados deben transicionar a estados definidos con lo que una condición de erro r debe ser procesada consecuentemente.
c. No deben existir estados no alcanzables o estados que no transiciones a otros.
d. Debe existir una transición por defecto sobre todo, en máquinas incompletamente especificadas.

PC007. Evitar rangos desajustados. Categoría: Warning

Los rangos de los vectores en ambos lados de una asignación, comparación o asociación deben coincidir.

PC008. Asegurar una lista de sensibilidad con todas las señales. Categoría: Warning.

Las listas deben ser sensibles a aquellas señales que forman parte de un proceso.

PC009. Asegurar que la estructura de un subprograma es la correcta. Categoría: Warning

Cada subprograma debe tener:

a. Sólo un punto de salida.
b. No debe ser recursivo
c. Acceso sólo a variables o señales locales

PC010. Asignar un valor antes de usar una señal. Categoría: Warning.

Cada objeto debe tener un valor antes de ser asignado.

PC011. Evitar entradas no conectadas. Categoría: Error

Todas las señales de entrada deben estar conectadas y evitar señales flotantes con efectos impredecibles en síntesis.

PC012. Evitar señales de salidas no conectadas. Categoría: Note

Las señales de salida deben conectarse o declararse intencionadamente como no-conectadas, usando la palabra reservada 'open'.

PC013. Declarar los objetos antes de ser usados. Categoría: Error

Los objetos, (señales constantes,…) han de ser declarados antes de usarse.

PC014. Evitar declaraciones de señales no utilizadas. Categoría: Warning.

Todos los objetos que se declaren deben ser usados, por tanto no declarar objetos que no son utilizados en el código.

PC101. Señales de salida tipo 'buffer' quedan prohibidas. Categoría: Error

Aquella señales de salida que han de leerse en el interior de la entity pueden declararse como 'buffer', sin embargo este tipo se salida queda prohibido, para hacer el código más portable.

PC102. Tipos basados en 'records' no deben formar parte de los puertos de I/O entidades. Categoría: Warning

La conexión de entidades entre si debe ser mediante tipos de conexión naturales y no agrupaciones o tipos no reales.

7.2 Tratamiento de la señal de reloj

Controlar las señales de reloj ante potenciales errores en diseños, conteniendo varios dominios de reloj y transiciones asíncronas.

TR001. Analizar múltiples relojes asíncronos. Categoría: Error

La temporización de un diseño tiene múltiples relojes, asíncronos entre si, o bien se admiten relojes generados internamente, todos los relojes deben ser analizados rigurosamente y en profundidad.

TR002. Evitar utilizar los dos flancos activos de reloj. Categoría: Error.

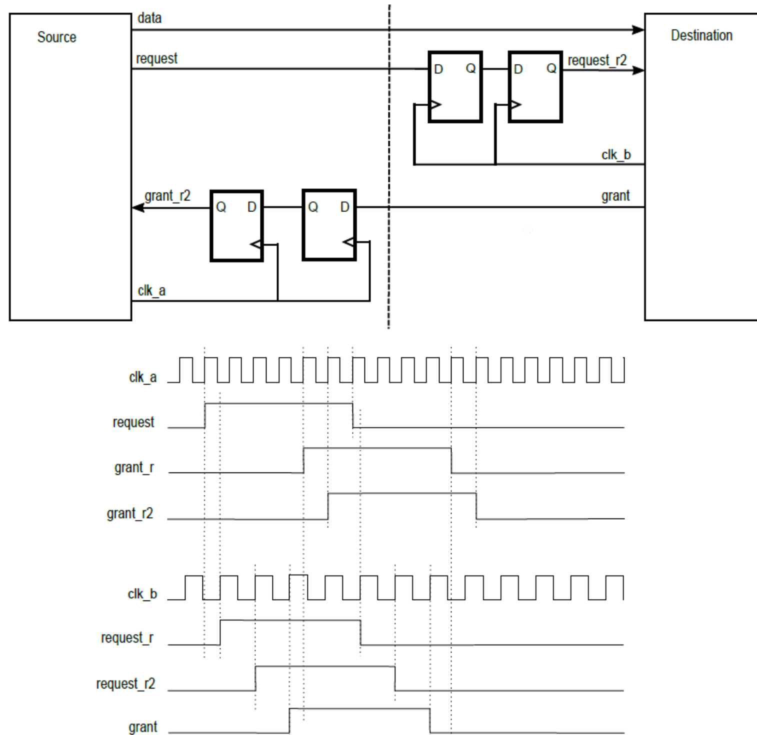
No utilizar partes del circuito activas en flanco de subida y otras activas en flanco de bajada de la misma señal de reloj.

TR101. Diferentes dominios de reloj. Categoría: Warning

La conexión entre diferentes dominios de reloj debe utilizar mecanismo previamente aprobados.

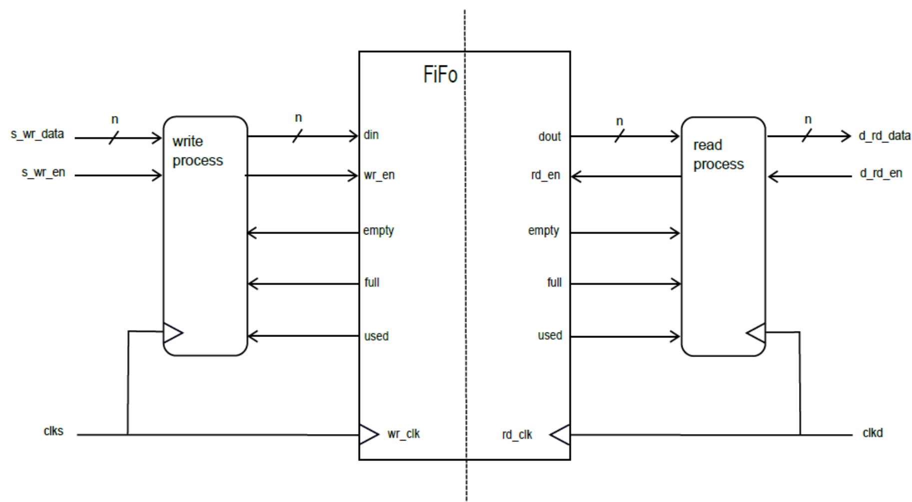
TR102. Dialogar sistemas con dos dominios de reloj, basados en señales de pregunta-respuesta. . Categoría: Warning

Para garantizar una buena comunicación entre sistemas digitales asíncronos entre sí es recomendable usar mecanismos de 'handshake'.



TR103. Dialogar sistemas con dos dominios de reloj, basados en FIFOs de doble reloj.
Categoría: Warning

La sincronización mediante FIFO de doble reloj tiene muchas formas de implementación. El tamaño de la FIFO se escoge para compensar las diferencias de frecuencias entre ambo**s** dominios. La funcionalidad de este tipo de FIFOS se debe determinar a priori.



TR104. Dialogar sistemas con dos dominios de reloj, usar un módulo que genere todos los relojes. Categoría: Warning

Los relojes deben ser generados a partir de una frecuencia base, es decir, una única señal de reloj.

TR105. No reasignar la señal de reloj dentro de una entidad. Categoría: Error

La señal de reloj no debe ser nunca reasignada

TR106. Limitación a un dominio de reloj en un mismo diseño. Categoría: Warning

Siempre que sea posible, sólo usar un solo dominio de reloj en un mismo diseño.

TR107. Limitación a un dominio de reloj en un mismo módulo. Categoría: Warning

Siempre que sea posible, sólo usar un solo dominio de reloj en un mismo módulo.

TR108. Limitación siempre el mismo flanco activo en un dominio de reloj. Categoría: Error

El flanco activo en un sólo dominio de reloj debe ser siempre el mismo.

TR109. Detección de flancos de señales mediante máquinas de estado. Categoría: Error.

Una señal externa debe sincronizarse mediante un detector de flanco, debe ser capturada sin que entre en conflicto con la señal de reloj,

TR110. Evitar metastabilidad de registros en las entradas externas. Categoría: Error.

Una señal externa de entrada debe sincronizarse usando al menos dos registros.

TR111. Usar registros para las salidas externas. Categoría: Error.

Una señal externa debe sincronizarse usando el registro del pin de salida, si lo hay. De esta manera evitamos que existan glitches y lógica.

TR112. Claridad en la definición de máquinas de estados FSMs. Categoría: Warning.

El código de una máquina de estados debe ser sencillo y claro, fácilmente interpretable y depurable, con implementación predecible. Se recomienda usar dos procedimientos uno síncrono y otro asíncrono para implementar una FSM.

TR113. Claridad en la definición de otros módulos síncronos. Categoría: Warning.

El código de un módulo síncrono debe ser sencillo y claro, fácilmente interpretable y depurable, con implementación predecible. Se recomienda usar dos procedimientos uno síncrono y otro asíncrono para implementar dicho módulo síncrono.

7.3 Reglas para una síntesis segura

SS001. Evitar ciertas implementaciones de la lógica. Categoría: Warning

Evitar realzar código que implique feed-throughs (conexiones-directas), retardos intencionados de la lógica ó drivers de triestados internos.

SS002. Asegurar selecciones 'case' bien hechas. Categoría: Error

La sentencia "case" debe contener:

a. Ser completa
b. No tener sentencias duplicadas
c. C. No tener selecciones no alcanzables
d. Debe contener la opción "when others" como valor por defecto.

SS003. Evitar bucles combinacionales. Categoría: Error.

No permitir que una señal que se genera en un momento dado mediante una lógica combinacional, siga una cadena lógica también exclusivamente combinacional y se convierta en entrada de la primera lógica.

SS004. Evitar inferencias de latches. Categoría: Error.

Existen múltiple situaciones que pueden generar latches inintencionadamente. Se deben evitar todas ellas. Esta regla no es aplicable a las señales de reloj. Son:

a. Omisión de señales en listas de sensibilidad.
b. Sentencias condicionales que no determinen las salidas en todas las opciones.
c. Sentencias condicionales incompletas.

SS005. Evitar asignaciones de múltiples formas de onda. Categoría: Error

Las asignaciones con formas de onda son no-sintetizables. En general el VHDL no sintetizable debe ser evitado en síntesis.

SS006. Evitar múltiple asignaciones de una señal. Categoría: Error

Una señal debe ser asignada en un único bloque secuencial.

SS007. Evitar constantes sin asignación inmediata. Categoría: Warning

Todas las constantes deben tener un valor que debe asignarse sin que existan demoras.

SS008. Evitar usar relojes como datos. Categoría: Error.

El reloj no debe mezclarse con la lógica de proceso de los datos

SS009. Evitar que los datos se usen como reloj. Categoría: Error.

Se debe evitar el que podríamos producir flancos activos a partir de pulsos imprevistos o glitches cuyo ancho dependa de los retardos de puerta

SS010. Evitar usar el reloj como reset o viceversa. Categoría: Error

La misma señal d inicialización no debe emplearse como señal de evolución e igualmente al contrario

SS011. Evitar puertas lógicas en líneas de reloj. Categoría: Error

Evitar que señales de reloj sean generadas a partir de lógica combinacional, actúen como relojes de registros. Si hubiera algún poderoso motivo para realizar una acción de diseño similar, deberá realizarse utilizando elementos especialmente diseñados para hacerlo con objetivo, siempre a bajo nivel, en cuyo caso cambiamos la categoría a nivel Warning.

SS012. Evitar dominios de relojes generados internamente. Categoría: Warning

Los dominios de relojes generados internos suelen no tener en cuenta los recursos dedicados para generar relojes. Por tanto han de ser generados con extremo cuidado.

SS013. Evitar reset asíncronos generados internamente. Categoría: Warning.

Las señales de inicialización no deben generarse a partir de lógica interna, que contiene riesgos de glitches, y por tanto pulsos de forma incontrolada. Pueden generar señales de reset.

SS014. Evitar mezclar polaridades en el reset. Categoría: Error

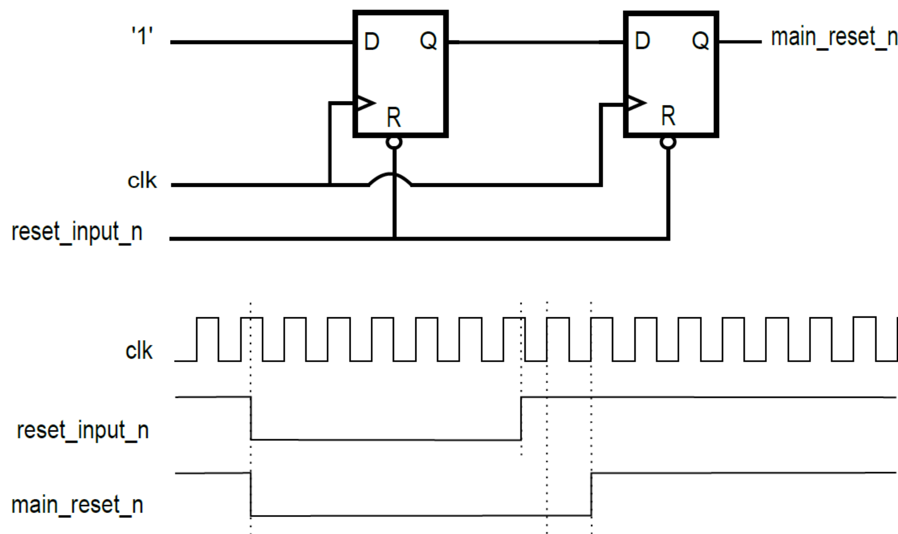
La misma señal de reset no puede ser activa alta y baja en diferentes partes del circuito, ya que significaría que la señal de reset no tendría un estado activo concreto

SS015. Evitar registros sin reset. Categoría: Warning

El control de reset lleva a un estado conocido un biestable con reset, si no lo tiene es imposible conocer el estado inicial de un sistema digital.

SS015. Evitar desactivar el reset de forma asíncrona. Categoría: Error

La señal de reset se debe desactivar de forma síncrona para evitar problemas de simultaneidad con el reloj.



SS016. Evitar usar valores iniciales de señales. Categoría: Error

Una señal toma un valor cuando es asignada, y por tanto los mecanismos de valores iniciales hacen que no funcionen. Distinto es que se asignen tras un reset.

SS017. Evitar la existencia de sentencias sin asignar o no asignadas. Categoría: Error

La presencia de trozo de código que no se asignan por otras señales o no son asignados a otras señales puede representar código muerto, pero que resulte hardware presente. Las reglar de oro en este caso son:

a. Cada registro o latch instanciados deben ser usados y direccionados en un diseño.
b. Registros y latches que afectan a una lógica sin usar debieran ser examinados.

SS018. Asegurar la controlabilidad de todos los registros. Categoría: Error

Todos los registros han de ser accesibles desde señales de entrada primarias.

SS019. Evitar caminos combinacionales excesivamente largos. Categoría: Warning

Los caminos combinacionales no deben exceder de un determinado valor

SS020. Asegurar límites en los bucles y los anidamientos. Categoría: Warning.

Las sentencias condicionales deben respetar un máximo de sentencias de anidamiento.

SS021. Asegurar consistencia en la definición del orden en los vectores. Categoría: Warning

Usar la misma formulación del orden y rango en los vectores, coherente.

SS101. Utilizar las librerías IEEE para codificar. Categoría: Error

La librería IEEE 'std_logic_1164' y la 'numeric_std' son las librerías correctas.

SS102. No usar variables en código RTL. Categoría: Warning.

Las variables no dejar traza cuando se sintetizan y hay que evitar usarlas, si no se tiene esto en cuenta.

SS103. No usar comparaciones con igualdades estrictas. Categoría: Warning

Las síntesis de comparaciones con igualdades dan resultados de síntesis peores que las comparaciones simples: (< ó > en vez de <= ó >=)

SS104. Usar signed o unsigned para hacer comparaciones y controlar tipo y rango. Categoría Error

El uso de std_logic_vector para hacer comparaciones puede llevar a errores impredecibles, mientras que signed o unsigned son tipos mejor controlados.

7.4 Facilitar la revisión de un código

El objetivo es redactar un código simple, fácil de leer y portable. Que pueda fácilmente reutilizarse y analizarse por los miembros del equipo de desarrollo.

RC001. Utilizar etiquetas siempre. Categoría: Note

Todas las sentencias, procesos, Bloques… pueden ser etiquetadas para mejorar la claridad del mismo. La síntesis posterior mantiene los nombres asignados en dichas etiquetas.

RC002. Evitar diferenciar nombres usando mayúsculas y minúsculas. Categoría: Error

En VHDL no es sensible a mayúsculas y minúsculas.

RC003. Evitar usar el mismo nombre para señales en diferentes espacios de un diseño. Categoría: Warning,

El mismo nombre, usado en diferentes partes del diseño, aunque tengan diferentes instancias que hagan la diferenciación, puede llevar a confusión.

RC004. Usar declaraciones de señales fácilmente reconocibles. Categoría: Note

Realizar declaraciones de las señales usando una declaración por línea

RC005. Usar asignaciones de señales fácilmente reconocibles. Categoría: Note

Realizar asignaciones de las señales usando una declaración por línea

RC006. Usar indentaciones consistentes. Categoría: Note

Mejorar la legibilidad de las señales realizando indentaciones consistentes.

RC007. Evitar indentaciones usando tabuladores. Categoría: Warning

Los tabuladores no deben utilizarse, debido a diferentes tratamientos por distintas plataformas.

RC008. Evitar ficheros de los diseño excesivamente largos. Categoría: Warning

Un fichero excesivamente largo resuelve muchos aspectos de un diseño. Reduce, en general, la portabilidad.

RC009. Utilizar nombres significativos para las señales relacionados con su comportamiento. Categoría: Warning

Para poder hacer un seguimiento de la funcionalidad

RC010. Utilizar cabeceras de los ficheros consistentes. Categoría: Warning

Hacer una buena descripción del principio de funcionamiento y etiquetar bien diseño. Detalles como autor, fecha, versión, descripción, histórico, revisión,… facilitan la trazabilidad del mismo.

RC011. Utilizar suficientes comandos significativos en un solo fichero. Categoría: Warning

Un fichero ha de contener un número significativo de sentencias funcionalmente relevantes.

RC012. Utilizar suficientes comentarios en el código. Categoría: Note

Usar una buena política de comentarios

RC013. Utilizar convenciones internas en los nombre de las señales. Categoría: Note

Una misma institución debe mantener una misma política de nombres de las señales, que sean fácilmente reconocibles. Por ejemplo, una señal activa baja puede terminar en 'z', las señales de entrada o salida primaria deben pueden acabar en 'I' y en 'O' respectivamente.

RC101. Llamar 'clock' o 'clk' a la señal de reloj. Categoría: Note

De esta forma es fácilmente identificable en el diseño.

RC102. Llamar 'reset' o 'rst' a la señal de inicializacón asíncrona. Categoría: Note

De esta forma es fácilmente identificable en el diseño.

RC103. Utilizar la extensión vhd en los ficheros. Categoría: Note

Los ficheros que contienen código VHDL deben ser reconocibles.

RC104. Utilizar nombres para los ficheros significativos. Categoría: Note

Los ficheros que contienen código deben hacer referencia a su contenido.

RC105. Nombrar las entidades como los ficheros. Categoría: Warning

Los ficheros han de nombrarse como las entidades que contienen hacer referencia a su comportamiento.

RC106. Usar una entidad por fichero. Categoría: Warning

Las entidades deben encapsularse con una entidad por fichero.

RC107. Usar una arquitectura por fichero. Categoría: Warning

Las arquitecturas deben encapsularse con una arquitectura por fichero.

RC108. Número de puertos Ì/O por línea. Categoría: Note

Por claridad, los I/O deben ocupar una por línea

RC109. Instanciación de componentes explícitamente, por nombre. Categoría: Error

Los puertos de los componentes deben declararse explícitamente, por nombre, nunca por referencia.

RC110. Los tipos de puertos de las entidades deben seguir una organización. Categoría: Note

Seguir una ordenación coherente y previsible de los puertos de cada entidad. Por ejemplo por conectividad a otras entidades.

RC111. Puertos especiales de una entidad. Categoría: Note

Declara primeramente los puerto especiales como reloj o reset.

RC112. Encapsular las instancias primitivas en entidades aparte. Categoría: Warning

Las primitivas hacen referencia a unidades funcionales que dependen de la tecnología de un fabricante u otro. (p. ej. memorias, PLL, multiplicadores,…) Es buena práctica escribir una entidad genérica que las determine.

RC113. Escribir los comentarios en inglés. Categoría: Note

Unificar el lenguaje para realizar los comentarios, que deben estar preferentemente en inglés.

RC114. Las simulaciones deben tener una extensión finita. Categoría: Error

La extensión de las simulaciones funcionales tienen está definida, por tanto tienen un vector final.

RC115. Uso de procedures y functions en testbech. Categoría: Error

Se recomiendan el uso de funciones y procedimientos en las sentencias de los Test_Benches, donde no se sintetiza un circuito.

RC116. Uso de 'wait' y 'after' en testbech. Categoría: Error

Se recomienda el uso de las sentencias wait y after en Test_Benches para la temporización de señales.

7.5 HDL en estándares

Chequear HDL bajo estándares vía una simple revisión puede ser considerado una manera de revisar para el cumplimiento de un estándar. Pero si se realiza mediante una herramienta automática, se debe calificar la propia herramienta.

Se debe realizar una 'evaluación absolutamente independiente' de la misma, pero dado que una vez sea la calificación de la herramienta, la propia herramienta no sustituye la revisión de ciertos aspectos del propio código. La revisión manual para entender el funcionamiento del código no puede ser sustituida por una revisión automática.

Bibliografía

[1] Lange M. et al. "Best Practice VHDL Coding Stanr¡dard for D0-254 Programs.(rev 1a)"

[2] "Design and VHDL Handbook for VLSI and development" CNES