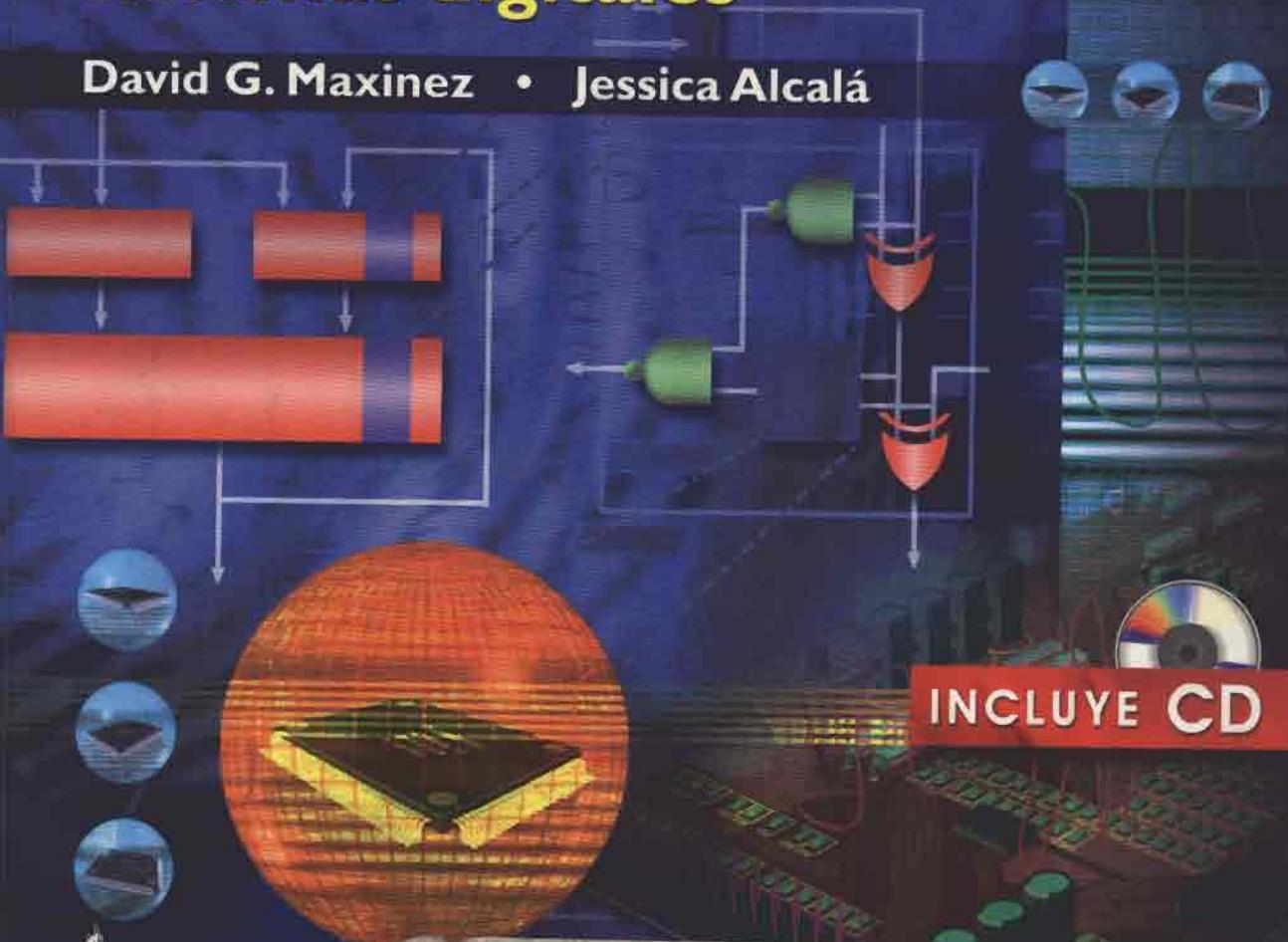


VHDL

**El arte de programar
sistemas digitales**

David G. Maxinez • Jessica Alcalá



INCLUYE CD



Tec
de Monterrey
CAMPUS ESTADO DE MEXICO



CECSA

Contenido

Acerca de los autores	ix
Prólogo	xi
1. Estado actual de la lógica programable	1
1.1 Dispositivos lógicos programables (PLD)	2
1.2 Dispositivos lógicos programables de alto nivel de integración	13
1.3 Ambiente de desarrollo de la lógica programable	18
1.4 Campos de aplicación de la lógica programable	23
1.5 La lógica programable y los lenguajes de descripción en hardware (HDL)	25
1.6 Compañías de soporte en hardware y software	28
Ejercicios	33
2. VHDL: su organización y arquitectura	37
2.1 Unidades básicas de diseño	37
2.2 Entidad	38
2.3 Declaración de entidades	40
2.4 Diseño de entidades utilizando vectores	42
2.5 Arquitecturas (architecture)	46
Ejercicios	56
3. Diseño lógico combinacional mediante VHDL	61
3.1 Programación de estructuras básicas mediante declaraciones concurrentes	61
3.2 Programación de estructuras básicas mediante declaraciones secuenciales	69
Ejercicios	89

✓ 4. Diseño lógico secuencial con VHDL	93
4.1 Diseño lógico secuencial	93
4.2 Flip-Flops	94
4.3 Registros	98
4.4 Contadores	101
4.5 Diseño de sistemas secuenciales síncronos	105
Ejercicios	113
5. Integración de entidades en VHDL	123
5.1 Esquema básico de integración de entidades	123
5.2 Integración de entidades básicas	128
Ejercicios	147
6. Diseño de controladores digitales mediante cartas ASM y VHDL	153
6.1 El algoritmo de la máquina de estado (ASM)	154
6.2 Estructura de una carta ASM	156
6.3 Cartas ASM en comparación con las máquinas de estado	159
6.4 Diseño de controladores mediante cartas ASM	162
6.5 Diseño de cartas ASM mediante VHDL	166
Ejercicios	180
7. Diseño jerárquico en VHDL	197
7.1 Metodología de diseño de estructuras jerárquicas	198
7.2 Análisis del problema y descomposición en bloques individuales de la estructura global	200
7.3 Diseño y programación de componentes o unidades del circuito	201
7.4 Creación de un paquete de componentes	206
7.5 Diseño del programa de alto nivel (Top Level)	207
7.6 Creación de una librería en Warp	208
Ejercicios	225
8. Sistemas embebidos en VHDL	229
8.1 Sistemas embebidos	229
* 8.2 Diseño de un microprocesador	237
8.3 Diseño jerárquico	261
Ejercicios	268
9. Redes neuronales artificiales y VHDL	273
9.1 ¿Qué es una red neuronal artificial?	275
9.2 Aprendizaje en las neuronas artificiales	279
9.3 Aprendizaje por error mínimo	291
9.4 Redes asociativas	294
Ejercicios	308

Apéndices

A. Herramientas de soporte para la programación en VHDL	311
B. Instalación del Software Warp	331
C. Identificadores, tipos y atributos	333
D. Hojas técnicas del CPLD CY7C372i	343
E. Palabras reservadas en VHDL	347
F. Operadores definidos en VHDL según su orden de precedencia	349
Índice analítico	351

Prólogo

Hoy en día, en nuestro ambiente familiar o de trabajo nos encontramos rodeados de sistemas electrónicos muy sofisticados. Teléfonos celulares, computadoras personales, televisores portátiles, equipos de sonido, dispositivos de comunicaciones y estaciones de juego interactivo, entre otros, no son más que algunos ejemplos del desarrollo tecnológico que ha cambiado nuestro estilo de vida haciéndolo cada vez más confortable. Todos estos sistemas tienen algo en común: su tamaño, de dimensiones tan pequeñas que resulta increíble pensar que sean igual o más potentes que los sistemas mucho más grandes que existieron hace algunos años.

Estos avances son posibles gracias al desarrollo de la microelectrónica, la cual ha permitido la miniaturización de los componentes para obtener así mayores beneficios de los chips (circuitos integrados) y para ampliar las posibilidades de aplicación.

La evolución en el desarrollo de los circuitos integrados se ha venido perfeccionando a través de los años. Primero, se desarrollaron los circuitos de baja escala de integración (SSI o Small Scale Integration), después los de mediana escala de integración (MSI o Medium Scale Integration) y posteriormente los de muy alta escala de integración (VLSI o Very Large Scale Integration) hasta llegar a los circuitos integrados de propósito específico (ASIC).

Actualmente, la gente encargada del desarrollo de nueva tecnología perfecciona el diseño de los circuitos integrados orientados a una aplicación y/o solución específica: los ASIC, logrando dispositivos muy potentes y que ocupan un mínimo de espacio. La optimización en el diseño de estos chips tiene dos tendencias en su conceptualización.

La primera tendencia es la técnica de *full custom design* (Diseño totalmente a la medida), la cual consiste en desarrollar un circuito para una aplicación específica mediante la integración de transistor por transistor. En su fabricación se siguen los pasos tradicionales de diseño: preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química.

La segunda tendencia en el diseño de los ASIC proviene de una innovadora propuesta, que sugiere la utilización de celdas programables preestablecidas e insertadas dentro de un circuito integrado. Con base en esta idea surgió la familia de dispositivos lógicos programables (los PLD), cuyo nivel de densidad de integración ha venido evolucionando a través del tiempo. Iniciaron con los PAL (Arreglos Lógicos programables) hasta llegar al uso de los CPLD (Dispositivos Lógicos Programables Complejos) y los FPGA (Arreglos de Compuertas Programables en Campo), los cuales dada su conectividad interna sobre cada una de sus celdas han hecho posible el desarrollo de circuitos integrados de aplicación específica de una forma mucho más fácil y económica, para beneficio de los ingenieros encargados de integrar sistemas.

El contenido de este libro se encuentra orientado hacia este tipo de diseño. Nuestro objetivo es brindar al lector la oportunidad de comprender, manejar y aplicar el lenguaje de programación más poderoso para este tipo de aplicaciones: VHDL.

El lenguaje de descripción en hardware VHDL es considerado como la máxima herramienta de diseño por las industrias y universidades de todo el mundo, pues proporciona a los usuarios muchas ventajas en la planeación y diseño de los sistemas electrónicos digitales.

Esta obra ha sido preparada especialmente para aquellos estudiantes e ingenieros que desean introducirse en el manejo de este lenguaje de programación. El libro se encuentra redactado de una manera muy clara y accesible para guiar al lector paso por paso en los primeros cuatro capítulos. Para aquellas personas que ya tengan conocimientos sobre el tema, hemos incorporado una serie de libros recomendados con mayor profundidad y complejidad.

Uno de los objetivos del libro es proporcionar al estudiante y al ingeniero electrónico o de computación una forma fácil y práctica de integrar aplicaciones digitales utilizando el lenguaje de descripción en hardware VHDL. También esperamos motivar al lector para que comience el desarrollo e integración de sistemas electrónicos a través este lenguaje, con la visión y oportunidad de crecer como microempresario en el desarrollo de sistemas miniaturizados, los cuales pueden ser fácilmente comercializados, y generar así fuentes de empleo en beneficio de la sociedad.

Este libro es recomendable para un curso introductorio de VHDL, tanto a nivel técnico como a nivel universitario dado que para interpretar y entender los ejercicios sobre los que se realiza nuestra explicación sólo se requiere como antecedente un curso básico de diseño lógico que involucre el conocimiento

de los temas de compuertas lógicas, minimización de funciones booleanas, circuitos combinacionales y circuitos secuenciales.

Para nosotros es importante que los conocimientos que se adquieran en la lectura de este libro puedan ser trasladados de manera inmediata hacia la práctica o aplicación. Por ello, hemos incluido algunos datos de orientación para que el lector sepa cómo conseguir las herramientas de diseño y dónde conseguir de manera oportuna los circuitos PLD para integrar sus aplicaciones (véase el Apéndice A). También hemos incluido demostraciones del programa WARP en el CD que acompaña al libro.

Organización del libro

En resumen, esta obra está estructurada en nueve capítulos y cinco apéndices. En el capítulo uno se describe de forma cualitativa el estado actual de la lógica programable: sus antecedentes, ventajas, desventajas y perspectivas, además proporciona la información referente a las compañías que brindan el soporte tanto en software como hardware. Este capítulo es completamente informativo y su finalidad consiste en familiarizar y adentrar al lector en esta área de desarrollo.

En el capítulo dos (*VHDL: su organización y arquitectura*) se presenta la estructura básica del lenguaje de descripción en hardware VHDL y se analizan los módulos básicos de diseño: sus palabras reservadas, los tipos de datos, así como el manejo de las diferentes arquitecturas o estilos de diseño empleados en el desarrollo de un programa.

En el capítulo tres (*Diseño lógico combinacional mediante VHDL*) se describe la forma de emplear declaraciones concurrentes y secuenciales dentro de un programa mediante la solución de circuitos combinacionales individuales tales como los multiplexores, los decodificadores, los codificadores y sumadores, entre otros. Es importante resaltar que en este capítulo y en los subsecuentes no se presenta la solución óptima para un problema dado. Por el contrario, cada problema se aborda de diferente manera con el único fin de presentar al lector el uso cada vez mayor de nuevas palabras reservadas por VHDL en el entendido que la mejor solución de un problema se da por sí solo y llega cuando el lector conoce más y más formas de programar.

En el capítulo cuatro (*Diseño lógico secuencial con VHDL*) se realiza una síntesis de diseño de los principales circuitos secuenciales: flip-flop, registros de corrimiento, contadores, manejo de diagramas de estado etcétera. Todos ellos se analizan por separado haciendo énfasis en las instrucciones encargadas de sincronizar los eventos que se desarrollan en estas entidades de diseño.

El capítulo cinco (*Integración de entidades en VHDL*) detalla la forma en que se unen diferentes bloques lógicos, es decir, se describe cómo se integran dentro de una sola entidad varios circuitos, sean combinacionales y/o se-

cuenciales, manejados en su programación de manera individual o a través exclusivamente de la relación entrada/-salida.

En el capítulo seis (*Diseño de controladores digitales mediante cartas ASM y VHDL*) se describe la programación de algoritmos digitales (controladores), mostrando al lector la forma de como brindar soluciones a un problema dado a través del desarrollo y conceptualización de un algoritmo, cuya descripción se realiza a través de la carta ASM. En este capítulo el lector echa mano de todos los conocimientos adquiridos en los capítulos previos, sorprendiéndose de la facilidad con la cual se programan sistemas digitales complejos.

En el capítulo siete (*Diseño jerárquico en VHDL*) se presenta la forma de jerarquizar o dividir un problema en pequeños subsistemas que pueden analizarse y simularse de manera independiente para posteriormente entrelazarlos mediante un programa de alto nivel denominado Top Level. Esta forma de programación es muy utilizada en VHDL ya que permite crear nuevas librerías de trabajo, que el diseñador puede almacenar para posteriormente incluir en nuevos desarrollos.

En el capítulo ocho (*Sistemas embebidos en VHDL*) se incluye la parte teórica de los sistemas embebidos así como su ciclo de desarrollo mediante el diseño de un microprocesador, describiendo en detalle cada uno de sus módulos y la forma de programarlos mediante el diseño jerárquico o Top Level.

En el capítulo nueve dejamos el contexto de programación básica y abordamos la investigación realizada sobre la programación en VHDL de las redes neuronales artificiales. Esto tiene por objetivo bindar al lector una breve introducción a un nuevo campo de desarrollo e investigación, debido a que actualmente es posible desarrollar sistemas inteligentes a nivel de hardware e integrarlos en un circuito integrado, ya sea un CPLD o un FPGA.

Finalmente, los apéndices del libro se encuentran estructurados de la siguiente manera: en el apéndice A (*Herramientas de soporte para la programación en VHDL*) se describe el uso del software WarpR4 de Cypress Semiconductor, utilizado en la simulación de todos los ejercicios y problemas de este libro, además se incluyen los datos del distribuidor del software y de los circuitos integrados. El apéndice B contiene la información sobre cómo instalar el software en las diferentes plataformas tecnológicas. El apéndice C proporciona los principales identificadores, tipos y atributos que se manejan en VHDL, así como la sintaxis utilizada en cada declaración. En el apéndice D presentamos los datos técnicos del circuito CPLD CY372I-66JC de Cypress Semiconductor, en el cual fueron grabados los diseños presentados en este libro. En el apéndice E incluye una lista de palabras reservadas por VHDL con el fin de que el lector pueda identificarlas fácilmente y, por último, en el apéndice F se incluye una lista de operadores definidos en VHDL según su orden de precedencia.

Capítulo 1

Estado actual de la lógica programable

Introducción

En la actualidad el nivel de integración alcanzado con el desarrollo de la microelectrónica ha hecho posible desarrollar sistemas completos dentro de un solo circuito integrado SOC (*System On Chip*), con lo cual se han mejorado de manera notoria características como velocidad, confiabilidad, consumo de potencia y sobre todo el área de diseño. Esta última característica nos ha permitido observar día a día cómo los sistemas de uso industrial, militar y de consumo han minimizado el tamaño de sus desarrollos; por ejemplo, los teléfonos celulares, computadoras personales, calculadoras de bolsillo, agendas electrónicas, relojes digitales, sistemas de audio, sistemas de telecomunicaciones, etc., no son más que aplicaciones típicas que muestran la evolución de los circuitos integrados también conocidos como chips.

El proceso de miniaturización de los sistemas electrónicos comenzó con la interconexión de elementos discretos como resistencias, capacitores, resistores, bobinas, etc., todos colocados en un chasis reducido y una escasa separación entre ellos. Posteriormente se diseñaron y construyeron los primeros circuitos impresos —aún vigentes—, que relacionan e interconectan los elementos mencionados a través de cintas delgadas de cobre adheridas a un soporte aislante (por lo general baquelita) que permite el montaje de estos elementos [1].

Más tarde, el desarrollo del transistor de difusión planar, construido durante 1947 y 1948, permitió en 1960 la fabricación del primer circuito integrado monolítico. Éste integra cientos de transistores, resistencias, diodos y capacitores, todos fabricados sobre una pastilla de silicio. Como es del conocimiento general, el término monolítico se deriva de las raíces griegas

"mono" y "lithos" que significan uno y piedra, respectivamente; por tanto, dentro de la tecnología de los circuitos integrados un circuito monolítico está construido sobre una piedra única o cristal de silicio que contiene tanto elementos activos (transistores, diodos), como elementos pasivos (resistencias, capacitores), y las conexiones entre ellos [1].

La fabricación de los circuitos monolíticos se basa en los principios de materiales, procesos y diseño que constituyen la tecnología altamente desarrollada de los transistores y diodos individuales. Dicha fabricación incluye la preparación de la oblea o base, el crecimiento epitaxial, la difusión de impurezas, la implantación de iones, la oxidación, la fotolitografía, la metalización y la limpieza química [1].

La integración de sistemas se ha ido superando a medida que surgen nuevas tecnologías de fabricación. Esto ha permitido obtener componentes estándares de mayor complejidad y que brindan mayores beneficios. Sin embargo, el desarrollo de nuevos productos requiere bastante tiempo, por lo cual sólo se emplea cuando se necesita un alto volumen de producción.

Una forma más rápida y directa de integrar aplicaciones es mediante la lógica programable, la cual permite independizar el proceso de fabricación del proceso de diseño fuera de la fábrica de semiconductores. Esta idea fue desarrollada por Hon y Sequin [2] y Conway y Mead [3] a finales de los años sesenta.

1.1 Dispositivos lógicos programables (PLD)

Los dispositivos lógicos programables (o PLD, por sus siglas en inglés) favorecen la integración de aplicaciones y desarrollos lógicos mediante el empaquetamiento de soluciones en un circuito integrado. El resultado es la reducción de espacio físico dentro de la aplicación; es decir, se trata de dispositivos fabricados y revisados que se pueden personalizar desde el exterior mediante diversas técnicas de programación. El diseño se basa en bibliotecas y mecanismos específicos de mapeado de funciones, mientras que su implementación tan sólo requiere una fase de programación del dispositivo que el diseñador suele realizar en unos segundos [4].

En la actualidad, el diseño de ASIC (circuitos integrados desarrollados para aplicaciones específicas) domina las tendencias en el desarrollo de aplicaciones a nivel de microelectrónica. Este diseño presenta varias opciones de desarrollo, como se observa en la tabla 1.1. A nivel de ASIC los desarrollos *full* y *semi custom* ofrecen grandes ventajas en sistemas que emplean circuitos diseñados para una aplicación en particular. Sin embargo, su diseño ahora sólo es adecuado en aplicaciones que requieren un alto volumen de producción; por ejemplo, sistemas de telefonía celular, computadoras portátiles, cámaras de video, etcétera.

Los FPGA (arreglos de compuertas programables en campo) y CPLD (dispositivos lógicos programables complejos) ofrecen las mismas ventajas de un ASIC, sólo que a un menor costo; es decir, el costo por desarrollar un ASIC es mucho más alto que el que precisaría un FPGA o un CPLD, con la ventaja de que ambos son circuitos reprogramables, en los cuales es posible modificar o borrar una función programada sin alterar el funcionamiento del circuito [4].

Categoría	Características
Diseño totalmente a la media (Full-Custom)	<ul style="list-style-type: none"> Total libertad de diseño, pero el desarrollo requiere todas las etapas del proceso de fabricación: preparación de la oblea o base, crecimiento epitaxial, difusión de impurezas, implantación de iones, oxidación, fotolitografía, metalización y limpieza química [1]. <p>Los riesgos y costos son muy elevados; sólo se justifican ante grandes volúmenes o proyectos con restricciones (área, velocidad, consumo de potencia, etcétera) [4].</p>
Matrices de puertas predifundidas (Semi-custom/gate arrays)	<ul style="list-style-type: none"> Existe una estructura regular de dispositivos básicos (transistores) prefabricada que se puede personalizar mediante un conexionado específico que sólo necesita las últimas etapas del proceso tecnológico. <p>El diseño está limitado a las posibilidades de la estructura prefabricada y se realiza con base en una biblioteca de celdas precaracterizadas para cada familia de dispositivos[4].</p>
Celdas estándares precaracterizadas (Semi-custom/standard cells)	<ul style="list-style-type: none"> No se trabaja con alguna estructura fija prefabricada en particular, pero sí con bibliotecas de celdas y módulos precaracterizados y específicos para cada tecnología. <p>Libertad de diseño (en función de las facilidades de la biblioteca); pero el desarrollo exige un proceso de fabricación completo [4].</p>
Lógica programable (FPGA, CPLD).	<ul style="list-style-type: none"> Se trata de dispositivos fabricados y revisados que se pueden personalizar desde el exterior mediante diversas técnicas de programación. <p>El diseño se basa en bibliotecas y mecanismos específicos de mapeado de funciones, mientras que su implementación tan sólo requiere una fase de programación del dispositivo, que por lo general realiza el diseñador en unos pocos segundos[4].</p>

Tabla 1.1 Tecnologías de fabricación de circuitos integrados.

En la actualidad existe una gran variedad de dispositivos lógicos programables, los cuales se usan para reemplazar circuitos SSI (pequeña escala de integración), MSI (mediana escala de integración) e incluso circuitos VLSI (muy alta escala de integración), ya que ahorran espacio y reducen de manera significativa el número y el costo de los diseños. Estos dispositivos, llamados PLD (tabla 1.2), se clasifican por su arquitectura —la forma funcional en que se encuentran ordenados los elementos internos que proporcionan al dispositivo sus características.

Dispositivo	Descripción
PROM	Programmable Read-Only Memory: memoria programable de sólo lectura
PLA	Programmable Logic Array: arreglo lógico programable
PAL	Programmable Array Logic: lógica de arreglos programables
GAL	Generic Logic Array: arreglo lógico genérico
CPLD	Complex PLD: dispositivo lógico programable complejo
FPGA	Field Program Gate Array: arreglos de compuertas programables en campo

Tabla 1.2 Dispositivos lógicos programables.

1.1.1 Estructura interna de un PLD

Los dispositivos PROM, PLA, PAL y GAL están formados por arreglos o matrices que pueden ser fijos o programables, mientras que los CPLD y FPGA se encuentran estructurados mediante bloques lógicos configurables y celdas lógicas de alta densidad, respectivamente.

La arquitectura básica de un PLD está formada por un arreglo de compuertas AND y OR conectadas a las entradas y salidas del dispositivo. La finalidad de cada una de ellas se describe a continuación.

- a) **Arreglo AND.** Está formado por varias compuertas AND interconectadas a través de alambres, los cuales cuentan con un fusible en cada punto de intersección [Fig. 1.1a)]. En esencia, la programación del arreglo

consiste en fundir o apagar los fusibles para eliminar las variables que no serán utilizadas [Fig. 1.1b)]. Obsérvese cómo en cada entrada a las compuertas AND queda intacto el fusible que conecta la variable seleccionada con la entrada a la compuerta. En este caso, una vez que los fusibles se funden no pueden volver a programarse.

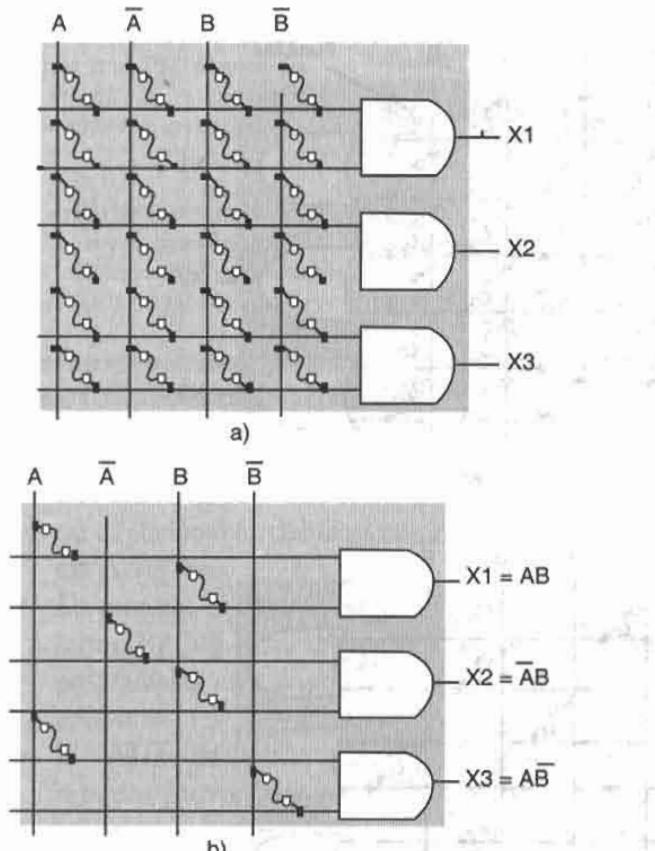


Figura 1.1 Arreglos AND: a) no programado y b) programado.

b) **Arreglo OR.** Está formado por un conjunto de compuertas OR conectadas a un arreglo programable, el cual contiene un fusible en cada punto de intersección. Este tipo de arreglo es similar al de compuertas AND explicado en el punto anterior, ya que de igual manera se programa fundiendo los fusibles para eliminar las variables no utilizadas. En la figura 1.2 se observa el arreglo OR programado y sin programar.

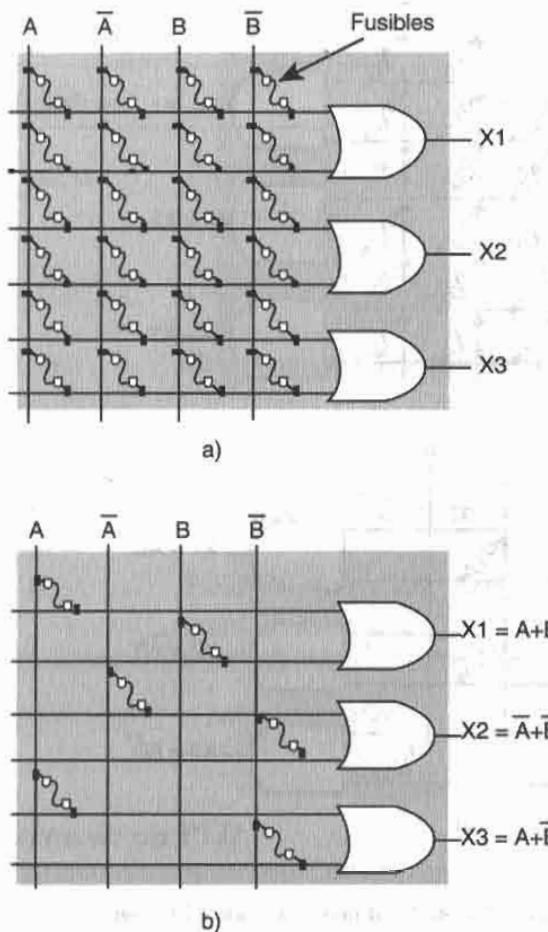


Figura 1.2 Estructura básica de PLD.

De acuerdo con lo anterior, observemos en la tabla 1.3 la estructura de los dispositivos lógicos programables básicos.

Dispositivo	Esquema básico
<p>La memoria programable de sólo lectura (PROM) está formada por un arreglo no programable de compuertas AND conectadas como decodificador y un arreglo programable OR.</p>	
<p>El arreglo lógico programable (PLA) es un PLD formado por un arreglo AND y un arreglo OR, ambos programables.</p>	
<p>El arreglo lógico programable (PAL) se encuentra formado por los arreglos AND programable y OR fijo con lógica de salida.</p>	

Tabla 1.3 Dispositivos lógicos programables.

- La PROM no se utiliza como un dispositivo lógico, sino como una memoria direccionable, debido a las limitaciones que presenta con las compuertas AND fijas.
- En esencia, el PLA se desarrolló para superar las limitaciones de la memoria PROM. Este dispositivo se llama también FPLA (arreglo lógico programable en campo), ya que es el usuario y no el fabricante quien lo programa.
- El PAL se desarrolló para superar algunas limitaciones del PLA, como retardos provocados por la implementación de fusibles adicionales, que resultan de la utilización de dos arreglos programables y de la complejidad del circuito. Un ejemplo típico de estos dispositivos es la familia PAL16R8, la cual fue desarrollada por la compañía AMD (Advanced Micro Devices) e incluye los dispositivos PAL16R4, PAL16R6, PAL16L8, PAL16R8, dispositivos programables por el usuario para reemplazar circuitos combinacionales y secuenciales SSI y MSI en un circuito.

En la figura 1.3 se muestra la arquitectura interna del PAL16L8. Como se puede observar, esta arquitectura está formada básicamente por circuitos combinacionales (compuertas AND, OR, buffers tri-estado e inversores), los cuales permiten la realización de funciones lógicas booleanas de la forma suma de productos (SOP). Cada término producto específico se programa en el arreglo AND, mientras que en el arreglo OR se realiza la suma lógica de los términos que se enviarán a las salidas del dispositivo.

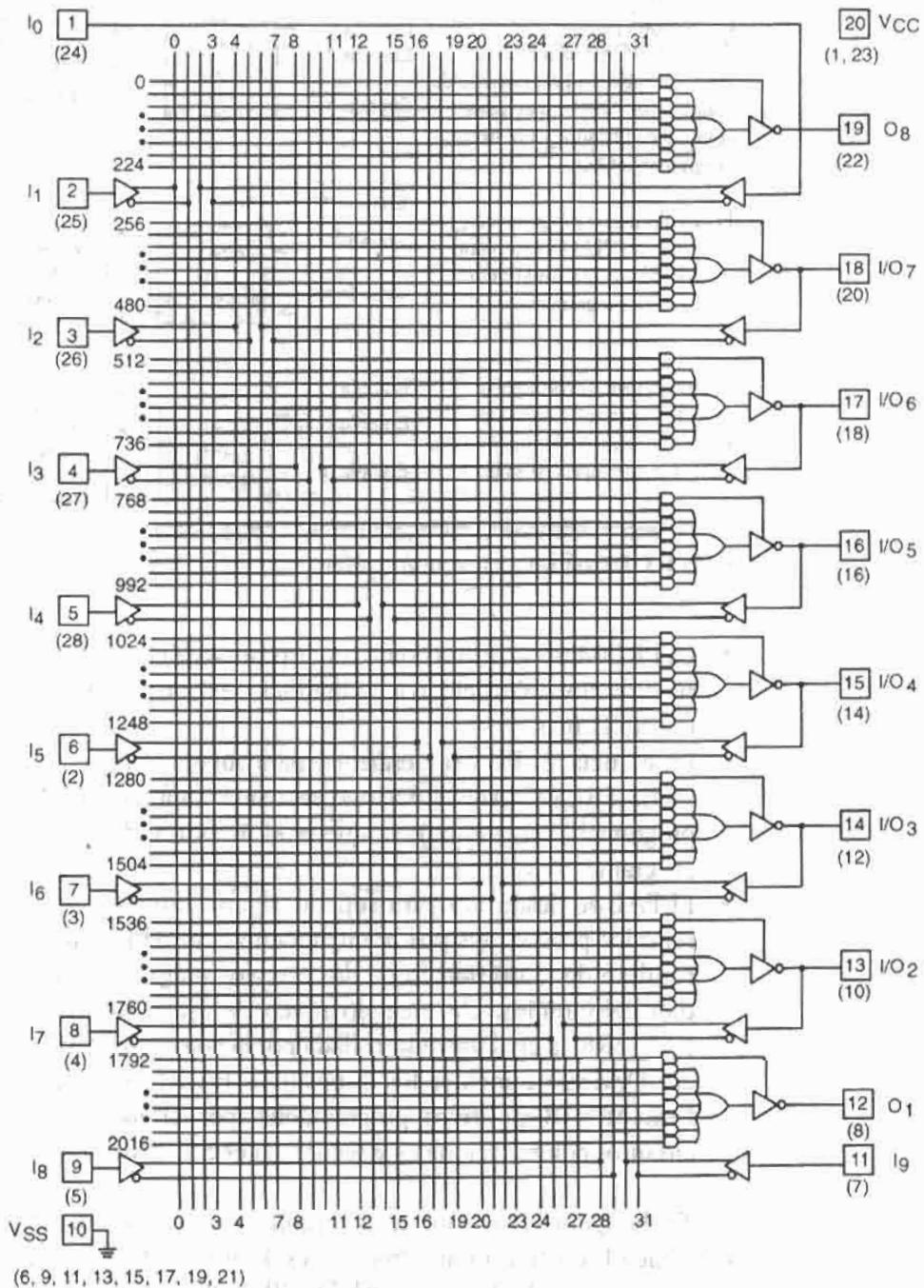


Figura 1.3 Arquitectura interna del PAL16L8.

1.1.2 Arreglo Lógico Genérico (GAL)

El arreglo lógico genérico (GAL) es similar al PAL, ya que se forma con arreglos AND programable y OR fijo, con una salida lógica programable. Las dos principales diferencias entre los dispositivos GAL y PAL radican en que el primero es reprogramable y contiene configuraciones de salida programables. Los dispositivos GAL se pueden programar una y otra vez, ya que usan la tecnología E² CMOS (Electrically Erasable CMOS: CMOS borrable eléctricamente), en lugar de tecnología bipolar y fusibles (Fig. 1.4).

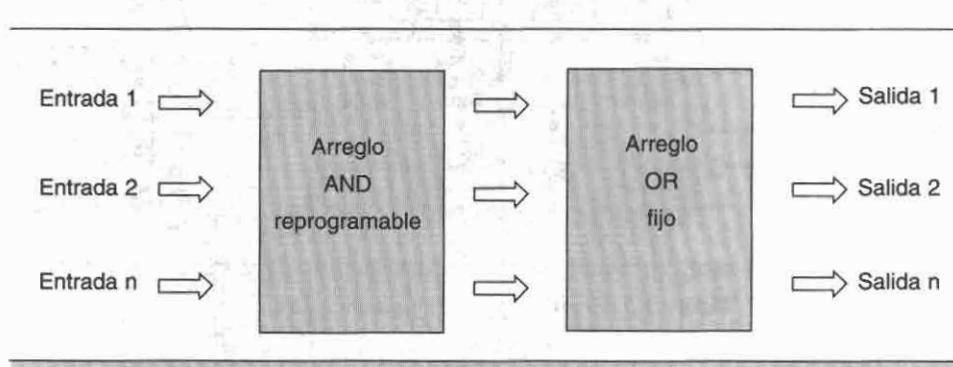


Figura 1.4 Diagrama de bloques del arreglo GAL.

Programación de un arreglo GAL

A diferencia de un PAL, el GAL está formado por celdas programables, las cuales se pueden reprogramar las veces que sea necesario. Como se observa en la figura 1.5, cada fila se conecta a una entrada de la compuerta AND y cada columna a una variable de entrada y sus complementos. Cuando se programa una celda, ésta se activa mediante la aplicación de cualquier combinación de las variables de entrada o sus complementos a la compuerta AND. Esto permite la implementación de cualquier función (producto de términos) requerida.

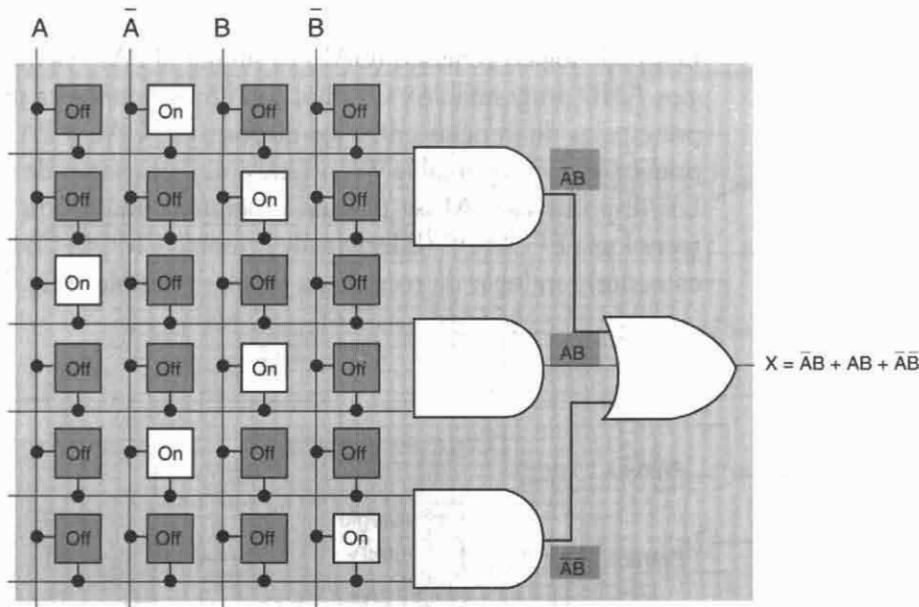


Figura 1.5 Programación del dispositivo GAL.

Arquitectura de un dispositivo GAL

Con el fin de esquematizar una arquitectura GAL, se toma como ejemplo el dispositivo GAL22V10 [Fig. 1.6a)]. Este circuito cuenta con 22 líneas de entrada y sus complementos, lo que da un total de 44 líneas de entrada a cada compuerta AND (estas entradas se encuentran representadas por las líneas verticales en el diagrama). La intersección que forman las líneas de entrada con los términos producto (líneas horizontales), representa cada una de las celdas que se pueden programar para conectar una variable de entrada (o su complemento) a una línea de término producto [Fig. 1.6b)], donde es posible apreciar la forma de obtener la suma de productos.

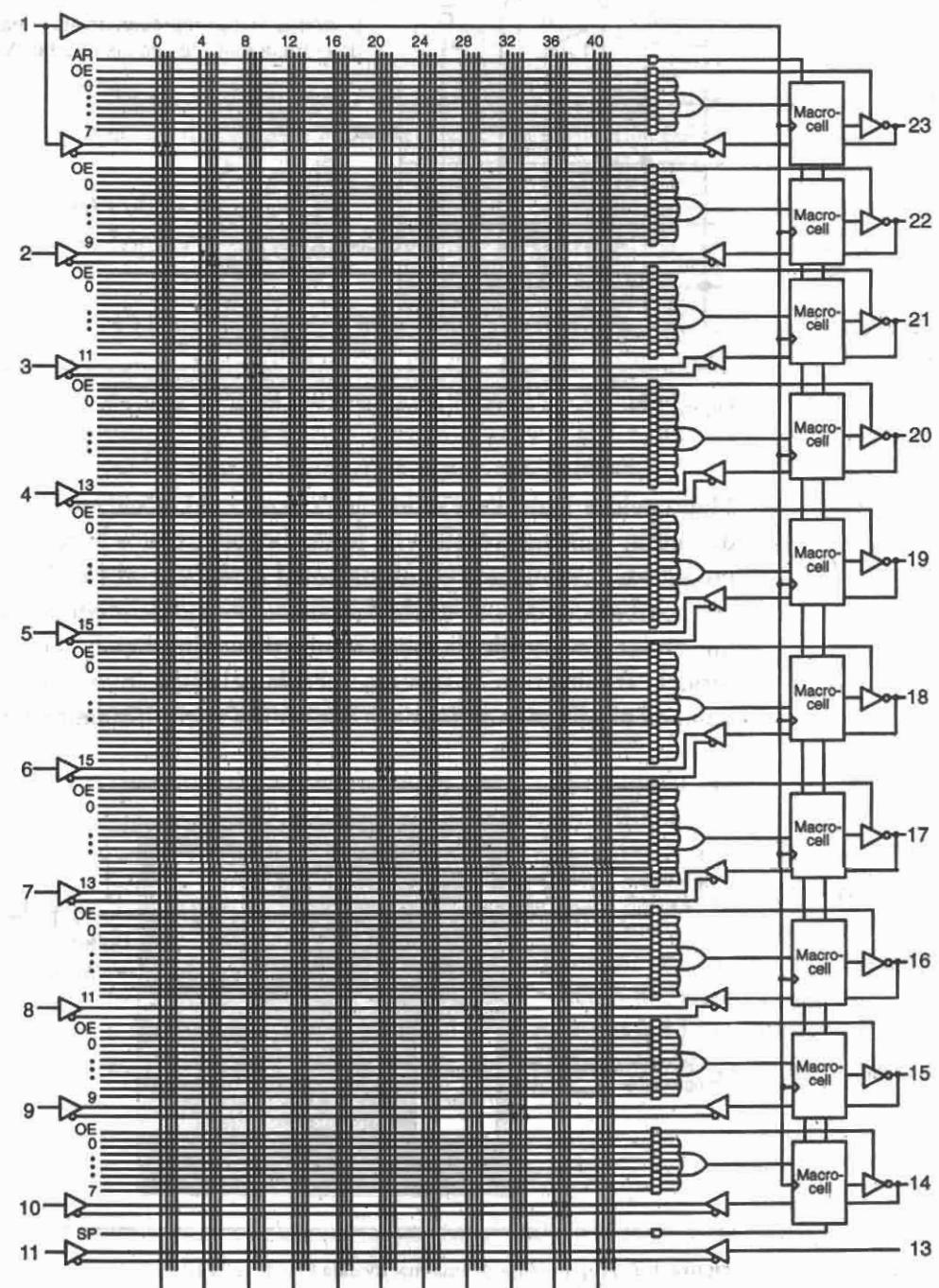


Figura 1.6a Arquitectura del GAL22V10.

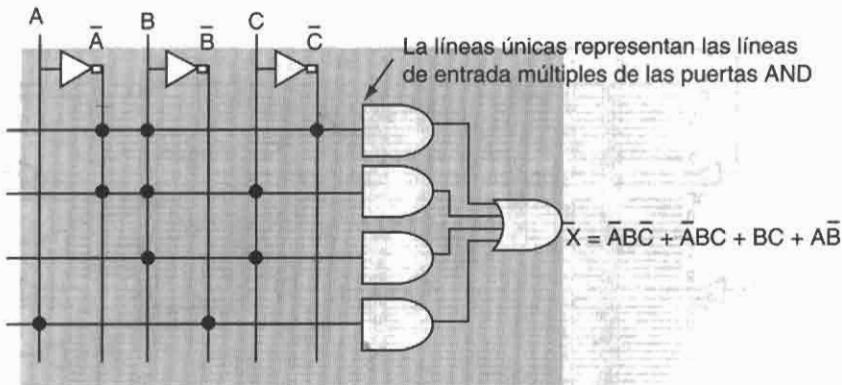


Figura 1.6b Realización de una suma de productos dentro de un GAL.

Macroceldas lógicas de salida. Una macrocelda lógica de salida (u OLMC, de output logic macrocell) está formada por circuitos lógicos que se pueden programar como **lógica combinacional** o **secuencial** [5]. Las configuraciones combinacionales se implementan por medio de programación, mientras que en las secuenciales la salida resulta de un flip-flop. En la figura 1.7 se observa la arquitectura de una macrocelda del dispositivo GAL22V10, la cual de manera general está formada por un flip-flop y dos multiplexores.

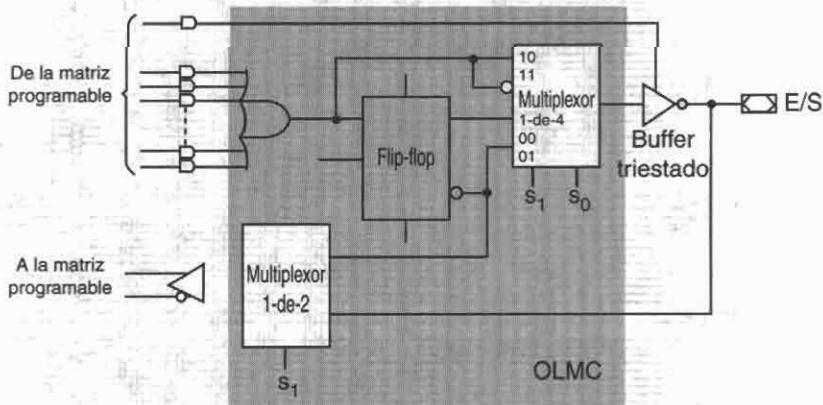


Figura 1.7 Arquitectura de una macrocelda OLMC 22V10.

Puede haber de ocho a dieciséis entradas de las compuertas AND en la compuerta OR. Esto indica las operaciones producto que pueden efectuarse en cada macrocelda. El área punteada está formada por dos multiplexores y

un flip-flop; el multiplexor 1 de 4 conecta una de sus cuatro líneas de entrada al buffer triestado de salida, en función de las líneas de selección S0 y S1. Por otro lado, el multiplexor de 1 de 2 conecta por medio del buffer la salida del flip-flop o la salida del buffer triestado al arreglo AND; esto se determina por medio de S1. Cada una de las líneas de selección se programa mediante un grupo de celdas especiales que se encuentran en el arreglo AND.

1.2 Dispositivos lógicos programables de alto nivel de integración

Los PLD de alto nivel de integración se crearon con el objeto de integrar mayor cantidad de dispositivos en un circuito (sistema en un chip SOC). Se caracterizan por la reducción de espacio y costo, además de ofrecer una mejora sustancial en el diseño de sistemas complejos, dado que incrementan la velocidad y las frecuencias de operación. Además, brindan a los diseñadores la oportunidad de enviar productos al mercado con más rapidez y les permiten realizar cambios en el diseño sin afectar la lógica, agregando periféricos de entrada/salida sin consumir una gran cantidad de tiempo, dado que los circuitos son reprogramables en el campo de trabajo.

1.2.1 Dispositivos lógicos programables complejos (CPLD)

Un circuito CPLD consiste en un arreglo de múltiples PLD agrupados como bloques en un chip. En algunas ocasiones estos dispositivos también se conocen como EPLD (Enhanced PLD: PLD mejorado), Super PAL, Mega PAL, [6] etc. Se califican como de alto nivel de integración, ya que tienen una gran capacidad equivalente a unos 50 PLD sencillos.

En su estructura básica, cada CPLD contiene múltiples bloques lógicos (similares al GAL22V10) conectados por medio de señales canalizadas desde la interconexión programable (PI). Esta unidad PI se encarga de interconectar los bloques lógicos y los bloques de entrada/salida del dispositivo sobre las redes apropiadas (Fig. 1.8).

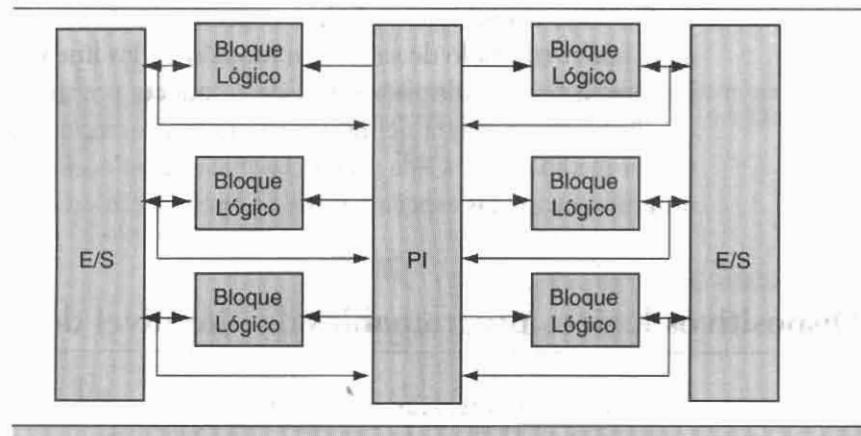


Figura 1.8 Arquitectura básica de un CPLD.

Los bloques lógicos, también conocidos como celdas generadoras de funciones, están formados por un arreglo de productos de términos que implementa los productos efectuados en las compuertas AND, un esquema de distribución de términos que permite crear las sumas de los productos provenientes del arreglo AND y por macroceldas similares a las incorporadas en la GAL22V10 (Fig. 1.9). En ocasiones las celdas de entrada/salida se consideran parte del bloque lógico, aunque la mayoría de los fabricantes coincide en que son externas. Cabe mencionar que el tamaño de los bloques lógicos es importante, ya que determina cuánta lógica se puede implementar dentro del CPLD; esto es, fija la capacidad del dispositivo.

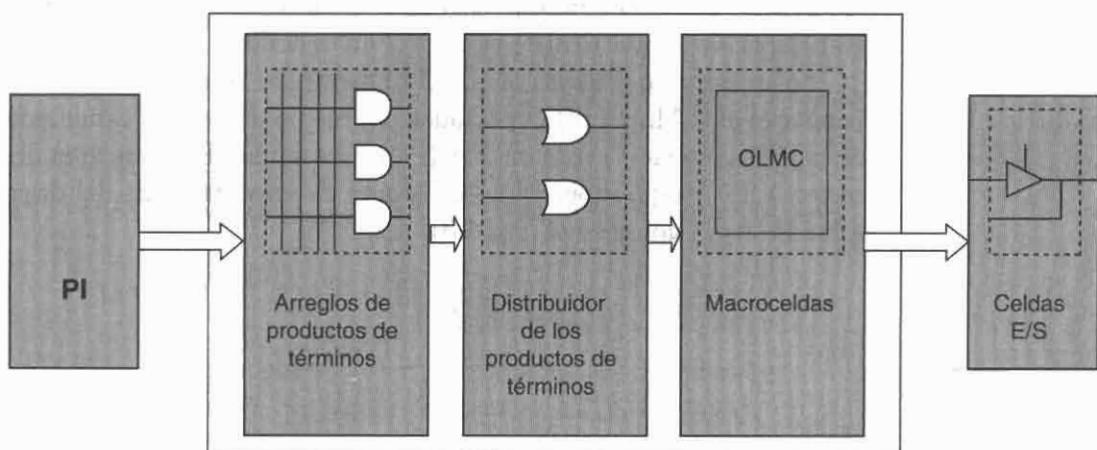


Figura 1.9 Bloque lógico programable.



- a) **Arreglos de productos de términos.** Es la parte del CPLD que identifica el porcentaje de términos implementados por cada macrocelda y el número máximo de productos de términos por bloque lógico. Esto es similar al arreglo de compuertas AND programable de un dispositivo GAL22V10.
- b) **Esquema de distribución de términos.** Es el mecanismo utilizado para distribuir los productos de términos a las macroceldas; esto se realiza mediante el arreglo programable de compuertas OR de un PLD. Los diferentes fabricantes de CPLD implementan la distribución de productos de términos con diferentes esquemas. Mientras algunos como la GAL22V10 usan un esquema de distribución variable (los cuales pueden variar en 8,10,12,14 o 16 productos por macrocelda), los CPLD como la familia MAX de Altera Corporation y Cypress Semiconductor, distribuyen cuatro productos de términos por macrocelda, además de utilizar “productos de términos expandidos”, que se asignan a una o varias macroceldas.
- c) **Macroceldas.** Una macrocelda de un CPLD está configurada internamente por flip-flops y un control de polaridad que habilita cada afirmación o negación de una expresión. Los CPLD suelen tener macroceldas de entrada/salida, de entrada y ocultas, mientras que los PLD sólo tienen macroceldas de entrada/salida.

La cantidad de macroceldas que contiene un CPLD es importante, debido a que cada uno de los bloques lógicos que conforman el dispositivo se expresan en términos del número de macroceldas que contiene. Por lo general, cada bloque lógico puede tener de cuatro a sesenta macroceldas; ahora bien, mientras mayor sea la cantidad, mayor será la complejidad de las funciones que se pueden implementar.

1.2.2 Arreglos de compuertas programables en campo (FPGA)

Los dispositivos FPGA se basan en lo que se conoce como arreglos de compuertas, los cuales consisten en la parte de la arquitectura que contiene tres elementos configurables: bloques lógicos configurables (CLB), bloques de entrada y de salida (IOB) y canales de comunicación [7]. A diferencia de los CPLD, la densidad de los FPGA se establece en cantidades equivalentes a cierto número de compuertas.

Por adentro, un FPGA está formado por arreglos de bloques lógicos configurables (CLB), que se comunican entre ellos y con las terminales de entrada/salida (E/S) por medio de alambrados llamados canales de comunicación. Cada FPGA contiene una matriz de bloques lógicos idénticos, por lo general de forma cuadrada, conectados por medio de líneas metálicas que corren vertical y horizontalmente entre cada bloque (Fig. 1.10).

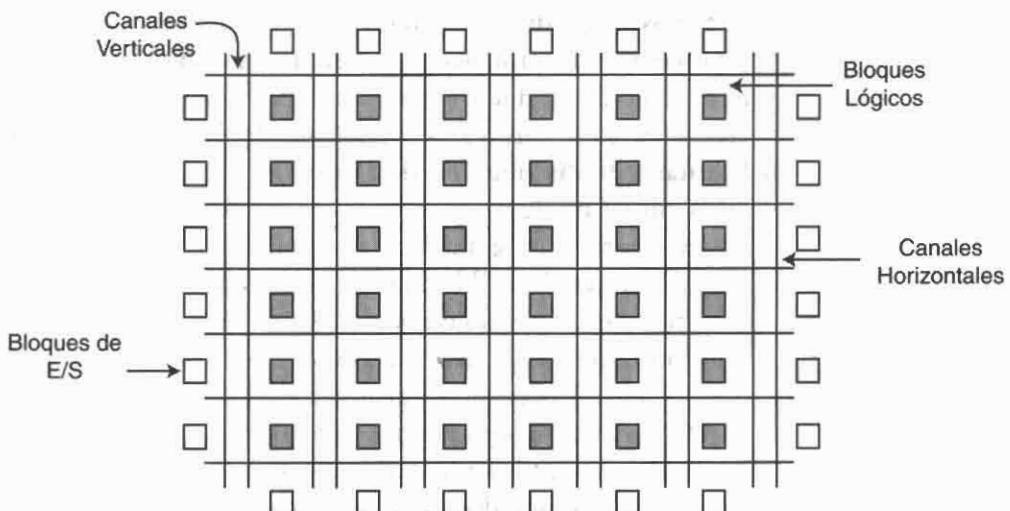


Figura 1.10 Arquitectura básica de un FPGA.

En la figura 1.11 se puede observar una arquitectura FPGA de la familia XC4000 de la compañía Xilinx. Este circuito muestra a detalle la configuración interna de cada uno de los componentes principales que conforman este dispositivo.

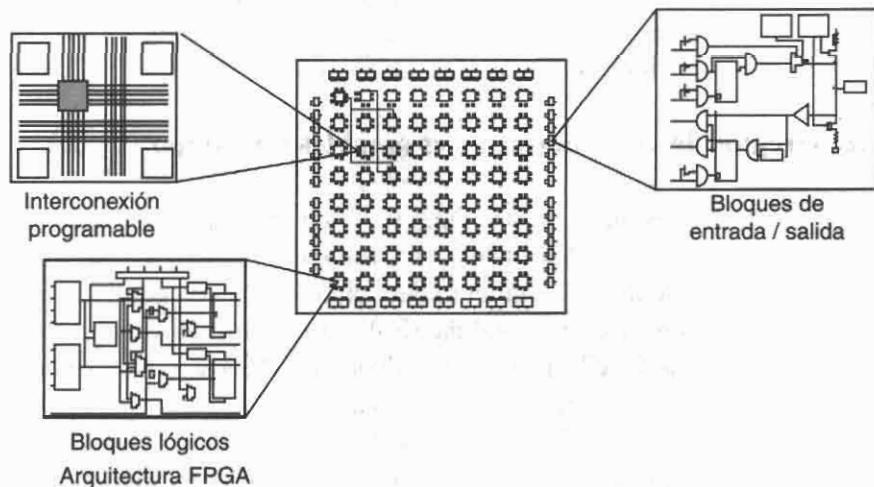


Figura 1.11 Arquitectura del FPGA XC4000 de Xilinx.

Los bloques lógicos (llamados también celdas generadoras de funciones) están configurados para procesar cualquier aplicación lógica. Estos bloques

tienen la característica de ser funcionalmente completos; es decir, permiten la implementación de cualquier función booleana representada en la forma de suma de productos. El diseño lógico se implementa mediante bloques conocidos como generadores de funciones o LUT (Look Up Table: tabla de búsqueda), los cuales permiten almacenar la lógica requerida, ya que cuentan con una pequeña memoria interna —por lo general de 16 bits— [6]. Cuando se aplica alguna combinación en las entradas de la LUT, el circuito la traduce en una dirección de memoria y envía fuera del bloque el dato almacenado en esa dirección. En la figura 1.12 se observan los tres LUT que contiene esta arquitectura, los cuales se encuentran etiquetados con las letras G, F y H.

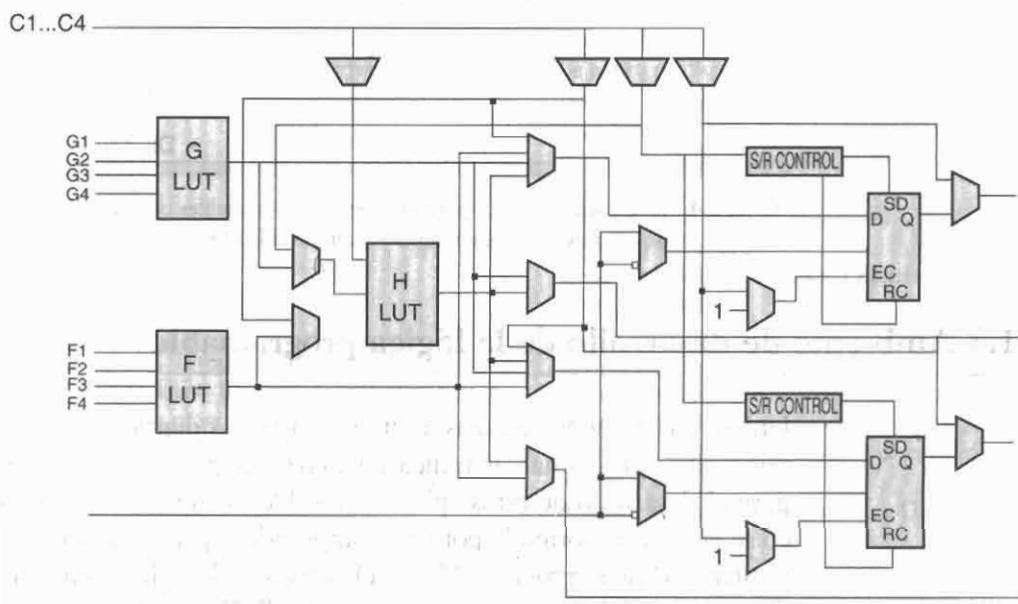


Figura 1.12 Arquitectura de un bloque lógico configurable FPGA.

En un dispositivo FPGA, los CLB están ordenados en arreglos de matrices programables (Programmable Switch Matrix o PSM), la matriz se encarga de dirigir las salidas de un bloque a otro. Las terminales de entrada y salida del FPGA pueden estar conectadas directamente al PSM o CLB, o se pueden conectar por medio de vías o canales de comunicación.

En algunas ocasiones se pueden confundir los dispositivos de FPGA y CPLD, ya que ambos utilizan bloques lógicos en su fabricación. La diferencia entre ellos radica en el número de flip-flops utilizados, mientras la arquitectura del FPGA es rica en registros. La CPLD mantiene una baja densidad. En la tabla 1.4 se presentan algunas otras diferencias entre una y otra arquitectura.

Características	CPLD	FPGA
Arquitectura	<ul style="list-style-type: none"> • Similar a un PLD • Más combinacional 	<ul style="list-style-type: none"> • Similar a los arreglos de compuertas • Más registros + RAM
Densidad	<ul style="list-style-type: none"> • Baja a media 	<ul style="list-style-type: none"> • Media a alta
Funcionalidad	<ul style="list-style-type: none"> • Trabajan a frecuencias superiores a 200 Mhz 	<ul style="list-style-type: none"> • Depende de la aplicación (arriba de los 135Mhz)
Aplicaciones	<ul style="list-style-type: none"> • Contadores rápidos • Máquinas de estado • Lógica combinacional 	<ul style="list-style-type: none"> • Excelentes en aplicaciones para arquitecturas de computadoras
		<ul style="list-style-type: none"> • Procesadores Digitales de Señales (DSP) • Diseños con registros

Tabla 1.4 Diferencias entre dispositivos lógicos programables complejos (CPLD) y los arreglos de compuertas programables en campo (FPGA).

1.3 Ambiente de desarrollo de la lógica programable

Una de las grandes ventajas al diseñar sistemas digitales mediante dispositivos lógicos programables radica en el bajo costo de los recursos requeridos para el desarrollo de estas aplicaciones. De manera general, el soporte básico se encuentra formado por una computadora personal, un grabador de dispositivos lógicos programables y el software de aplicación que soporta las diferentes familias de circuitos integrados PLD (Fig. 1.13).



Figura 1.13 Herramientas necesarias en la programación de PLD.

En la actualidad, diversos programas CAD (diseño asistido por computadora), como PALASM, OPAL, PLP, ABEL, CUPL, entre otros, se encuentran disponibles para la programación de dispositivos lógicos (tabla 1.5).

Compilador lógico	Características
PALASM (PAL Assembler: ensamblador de PAL)	Creado por la compañía Advanced Micro Devices (AMD) Desarrollado únicamente para aplicaciones con dispositivos PAL Acepta el formato de ecuaciones booleanas Utiliza cualquier editor que grabe en formato ASCII
OPAL (Optimal PAL language: lenguaje de optimización para arreglos programables)	Desarrollado por National Semiconductors Se aplica en dispositivos PAL y GAL Formato para usar lenguaje de máquinas de estado, ecuaciones booleanas de distintos niveles, tablas de verdad, o cualquier combinación entre ellas. Disponible en versión estudiantil y profesional (OPAL Jr y OPAL Pro) Genera ecuaciones de diseño partiendo de una tabla de verdad
PLPL (Programable Logic Programming Language: lenguaje de programación de lógica programable)	Creado por AMD Introduce el concepto de jerarquías en sus diseños Formatos múltiples (ecuaciones booleanas, tablas de verdad, diagramas de estado y las combinaciones entre éstos) Aplicaciones en dispositivos PAL y GAL
ABEL (Advanced Boolean Expression Language: lenguaje avanzado de expresiones booleanas)	Creado por Data I/O Corporation Programa cualquier tipo de PLD (Versión 5.0) Proporciona tres diferentes formatos de entrada: ecuaciones booleanas, tablas de verdad y diagramas de estados. Es catalogado como un lenguaje avanzado HDL (lenguaje de descripción en hardware)
CUPL (Compiler Universal Programmable Logic: compilador universal de lógica programable)	Creado por AMD para desarrollo de diseños complejos Presenta una total independencia del dispositivo Programa cualquier tipo de PLD Facilita la generación de descripciones lógicas de alto nivel Al igual que ABEL, también es catalogado como HDL

Tabla 1.5 Descripción de compiladores lógicos para PLD.

Estos programas —conocidos también como **compiladores lógicos**— tienen una función en común: procesar y sintetizar el diseño lógico que se va a introducir en un PLD mediante un método específico (ecuaciones booleanas, diagramas de estado, tablas de verdad) [5].

Método tradicional de diseño con lógica programable

La manera tradicional de diseñar con lógica programable, parte de la representación esquemática del circuito que se requiere realizar y luego define la solución del sistema por el método adecuado (ecuaciones booleanas, tablas de verdad, diagramas de estado, etc.). Por ejemplo, en la figura 1.14 se observa un diagrama que representa a un circuito construido con compuertas lógicas AND y OR. En este caso se eligió el método de ecuaciones booleanas para representar su funcionamiento, aunque se pudo usar también una tabla de verdad.

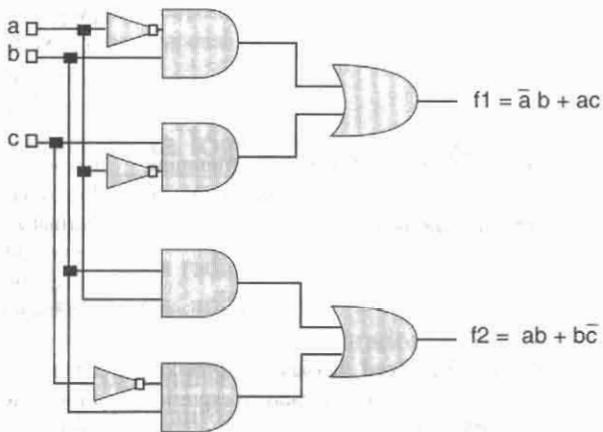


Figura 1.14 Obtención de las ecuaciones booleanas.

Como se puede observar, las ecuaciones que rigen el comportamiento del sistema se encuentran derivadas en función de las salidas f_1 y f_2 del circuito. Una vez que se obtienen estas ecuaciones, el siguiente paso es introducir en la computadora el archivo fuente o de entrada; es decir, el programa que contiene los datos que permitirán al compilador sintetizar la lógica requerida. Típicamente se introduce alguna información preliminar que indique datos como el nombre del diseñador, la empresa, fecha, nombre del diseño, etc. Luego se especifica el tipo de dispositivo PLD que se va a utilizar, la numeración de los pines de entrada y salida, y las variables del diseño. Por último, se define la función lógica en forma de ecuaciones booleanas o cualquier formato que acepte el compilador.

En la figura 1.15 se observa la pantalla principal del programa PALASM, en el cual se compilará el diseño de la figura 1.14 con el fin de exemplificar la metodología que se debe seguir.

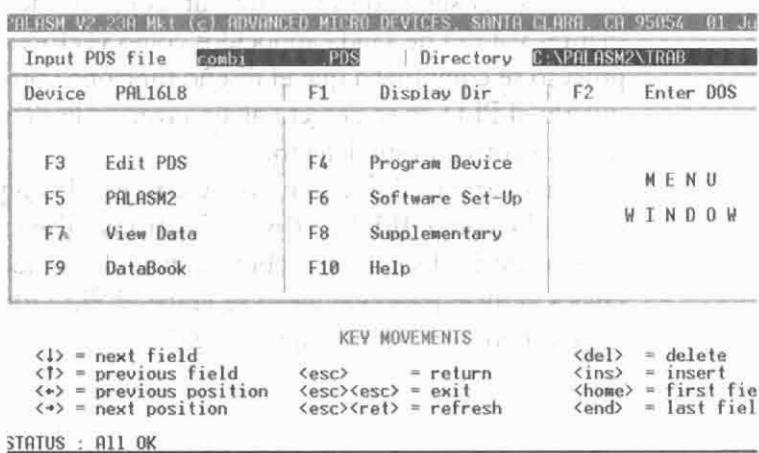


Figura 1.15 Pantalla principal de PALASM.

La forma de introducir el diseño se muestra en el listado 1. Nótese que las palabras reservadas por el compilador se representan con letras negritas.

```

TITLE          EJEMPLO
PATTERN        EJEMPLO.PDS
REVISION       1.0
AUTHOR         JESSICA
COMPANY        UNAM
DATE           00-00-00
CHIP           XX PAL16L8
} Encabezado

;1 2  3  4 5 6 7 8 9 10
NC NC NC NCA B C NC NC GND
;1112 13 14 15 16 17 18 19 20
NC NC F1 F2 NC NC NC NC NC VCC
} Declaración de pines de entrada/salida

EQUATIONS
F1 = /A* B + /A*C
F2 = A* B + B + /C
} Ecuaciones del circuito

SIMULATION
TRACE_ON A B F1 F2
SETF /A /B /C
CHECK /F1 /F2
SETF /A /B C
CHECK F1 /F2
SETF A /B C
CHECK F1 /F2
TRACE_OFF
} Simulación (condiciones e/s)

```

Listado 1.1 Archivo Fuente compilado en formato PALASM.

El siguiente paso consiste en la compilación del diseño, el cual radica básicamente en localizar los errores de sintaxis¹ o de otro tipo, cometidos durante la introducción de los datos en el archivo fuente. El compilador procesa y traduce el archivo fuente y minimiza las ecuaciones. En este paso, el diseño se ha simulado utilizando un conjunto de entradas y sus correspondientes valores de salida conocidos como **vectores de prueba**. Durante este proceso se comprueba que el diseño funcione correctamente antes de introducirlo al PLD. Si se detecta algún error en la simulación, se depura el diseño para corregir este defecto.

Una vez que el diseño no tiene errores, el compilador genera un archivo conocido como **JEDEC** (Joint Electronic Device Engineering Council)² o mapa de fusibles. Este archivo indica al grabador cuáles fusibles fundir y cuáles activar, para que luego se grabe el PLD (de acuerdo con el mapa de fusibles) en un grabador típico (Fig. 1.16).

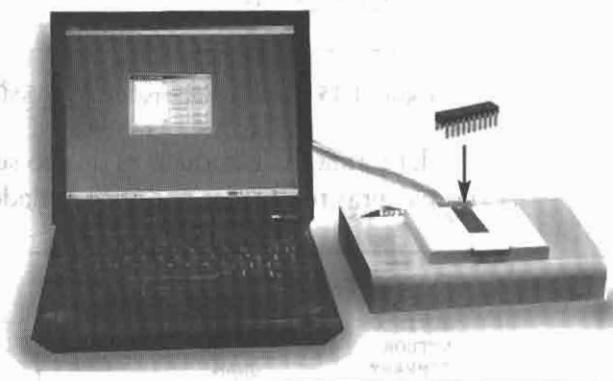


Figura 1.16 Implementación final del diseño en un PLD.

Como se puede observar, ciertos PLD (PROM, PAL, GAL) se programan empleando un grabador de dispositivos lógicos. Algunos otros PLD, como los CPLD y FPGA, presentan la característica de ser programables dentro del sistema (ISP, In-System Programmable); es decir, no hay que introducirlos al grabador, ya que por medio de elementos auxiliares se pueden programar dentro de la tarjeta de circuito integrado.

Como se aprecia, el método de diseño con lógica programable reduce de manera considerable el tiempo de diseño y permite al diseñador mayor control de los errores que se pudieran presentar, ya que la corrección se realiza en el software y no en el diseño físico.

¹ La sintaxis se refiere al formato establecido y la simbología utilizada para describir una categoría de funciones.

² Los archivos JEDEC están estandarizados para todos los compiladores lógicos existentes.

1.4 Campos de aplicación de la lógica programable

La Lógica programable es una herramienta de diseño muy poderosa, que se aplica en el mundo industrial y en proyectos universitarios en todo el mundo. En la actualidad se usan desde los PLD más sencillos (como el GAL, PAL, PLA) como reemplazos de circuitos LSI y MSI, hasta los potentes CPLD y FPGA, que tienen aplicaciones en áreas como telecomunicaciones, computación, redes, medicina, procesamiento digital de señales, multiprocesamiento de datos, microondas, sistemas digitales, telefonía celular, filtros digitales programables, entre otros.

En general, los CPLD son recomendables en aplicaciones donde se requieren muchos ciclos de sumas de productos, ya que pueden introducirse en el dispositivo para ejecutarse al mismo tiempo, lo que conduce a pocos retrasos. En la actualidad, los CPLD son muy utilizados a nivel industrial, ya que resulta fácil convertir diseños compuestos por múltiples PLD sencillos en un circuito CPLD.

Por otro lado, los FPGA son recomendables en aplicaciones secuenciales que no suponen grandes cantidades de términos producto. Por ejemplo, los FPGA desarrollados por la compañía ATMEL ofrecen alta velocidad en cómputo intensivo, aplicaciones en procesadores digitales de señales (DSP) y en otras fases del diseño lógico, debido a la gran cantidad de registros con los que cuentan sus dispositivos (de 1024 a 6400). Esto los hace ideales para su uso en dichas áreas.

Desarrollos recientes

Existen desarrollos realizados por diversas compañías cuyo funcionamiento se basa en un PLD; por ejemplo, la figura 1.17 ilustra una tarjeta basada en un FPGA de la familia XC4000 de Xilinx Corporation. Este desarrollo permite el procesamiento de datos en paralelo a alta velocidad, lo que reduce los problemas de procesamiento de datos intensivo³.

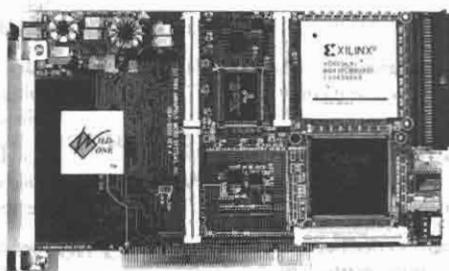


Figura 1.17 Sistema basado en un FPGA.

³ Fuente de información: <http://www.annapmicro.com>

En la figura 1.18 se muestra otra aplicación realizada en un dispositivo CPLD de la familia Flex10K de Altera Corporation (nivel de integración de 7000 compuertas). La función de esta tarjeta es permitir diversas aplicaciones en tiempo real, como el filtrado digital y muchas otras propias del campo del procesamiento digital de señales⁴.

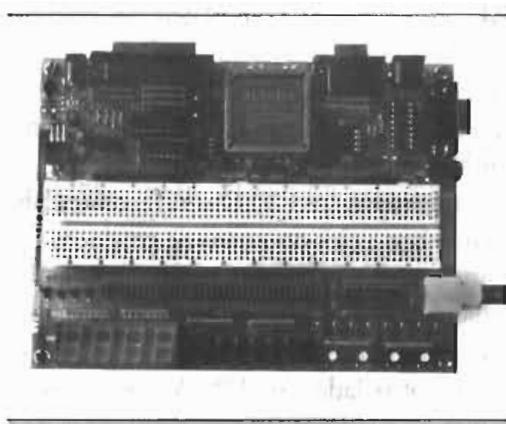


Figura 1.18 Ejemplo de un diseño lógico programable completo.

Como ya se mencionó, el campo de la lógica programable se ha extendido en la industria en los últimos años, ya que compañías de nivel internacional integran o desarrollan lógica programable en sus diseños (vea la tabla 1.6).

Compañía	Productos desarrollados con lógica programable
Andraka Consulting Group http://users.ids.net/~randraka/Inc	Procesadores digitales de señales (DSP) Comunicaciones digitales Procesadores de audio y video
Code Logic http://home.intekom.com/codelogic/	Lógica configurable Control embebido
Boton Line http://www.bltinc.com/	Módems de alta velocidad Audio, video, adquisición de datos y procesamiento de señales en general Aplicaciones militares: Criptografía, seguridad en comunicaciones, proyectos espaciales.
Comit's Services http://www.comit.com/	Redes: aplicaciones en protocolos TCP/IP Multimedia: Compresión de Audio/Video Aplicaciones en tiempo real
New Horizons GU http://www.netcomuk.co.uk/~newhoriz/index.html	Digitalizadores, Cámaras de Video (120Mbytes/sec) Video en tiempo real Puertos paralelos de comunicaciones para PC
Design Service Segments http://www.smartech.fi/	Diseño de microprocesadores complejos Dispositivos para telecomunicaciones, DSP Aplicaciones en diseños para control industrial

Tabla 1.6 Compañías que incorporan lógica programable en sus diseños.

⁴ Fuente de información: <http://www.fgpa.com>

1.5 La lógica programable y los lenguajes de descripción en hardware (HDL)

Como consecuencia de la creciente necesidad de integrar un mayor número de dispositivos en un solo circuito integrado, se desarrollaron nuevas herramientas de diseño que auxilian al ingeniero a integrar sistemas de mayor complejidad. Esto permitió que en la década de los cincuenta aparecieran los lenguajes de descripción en hardware (HDL) como una opción de diseño para el desarrollo de sistemas electrónicos elaborados. Estos lenguajes alcanzaron mayor desarrollo durante los años setenta, lapso en que se desarrollaron varios de ellos como IDL de IBM, TI-HDL de Texas Instruments, ZEUS de General Electric, etc., todos orientados al área industrial, así como los lenguajes en el ámbito universitario (AHPL, DDL, CDL, ISPS, etc.) [8]. Los primeros no estaban disponibles fuera de la empresa que los manejaba, mientras que los segundos carecían de soporte y mantenimiento adecuados que permitieran su utilización industrial. El desarrollo continuó y en la década de los ochenta surgieron lenguajes como VHDL, Verilog, ABEL 5.0, AHDL, etc., considerados lenguajes de descripción en hardware porque permitieron abordar un problema lógico a nivel funcional (descripción de un problema sólo conociendo las entradas y salidas), lo cual facilita la evaluación de soluciones alternativas antes de iniciar un diseño detallado.

Una de las principales características de estos lenguajes radica en su capacidad para describir en distintos niveles de abstracción (funcional, transferencia de registros RTL y lógico o nivel de compuertas) cierto diseño. Los niveles de abstracción se emplean para clasificar modelos HDL según el grado de detalle y precisión de sus descripciones [4].

Los niveles de abstracción descritos desde el punto de vista de simulación y síntesis del circuito pueden definirse como sigue:

- **Algorítmico:** se refiere a la relación funcional entre las entradas y salidas del circuito o sistema, sin hacer referencia a la realización final.
- **Transferencia de registros (RT):** Consiste en la partición del sistema en bloques funcionales sin considerar a detalle la realización final de cada bloque.
- **Lógico o de compuertas:** el circuito se expresa en términos de ecuaciones lógicas o de compuertas.

1.5.1 VHDL, lenguaje de descripción en hardware

En la actualidad, el lenguaje de descripción en hardware más utilizado a nivel industrial es VHDL⁵ (Hardware Description Language), que apareció en

⁵ Fuente de información: <http://www.fgpa.com>

la década de los ochenta como un lenguaje estándar, capaz de soportar el proceso de diseño de sistemas electrónicos complejos, con propiedades para reducir el tiempo de diseño y los recursos tecnológicos requeridos. El Departamento de Defensa de Estados Unidos creó el lenguaje VHDL como parte del programa “Very High Speed Integrated Circuits” (VHSIC), a partir del cual se detectó la necesidad de contar con un medio estándar de comunicación y la documentación para analizar la gran cantidad de datos asociados para el diseño de dispositivos de escala y complejidad deseados [9]; es decir, VHSIC debe entenderse como la rapidez en el diseño de circuitos integrados.

Después de varias versiones revisadas por el gobierno de los Estados Unidos, industrias y universidades, el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) publicó en diciembre de 1987 el estándar IEEEstd 1076-1987. Un año más tarde, surgió la necesidad de describir en VHDL todos los ASIC creados por el Departamento de Defensa, por lo que en 1993 se adoptó el estándar adicional de VHDL IEEE1164.

Hoy en día VHDL se considera como un estándar para la descripción, modelado y síntesis de circuitos digitales y sistemas complejos. Este lenguaje presenta diversas características que lo hacen uno de los HDL más utilizados en la actualidad.

1.5.2 Ventajas del desarrollo de circuitos integrados con VHDL

A continuación se exponen algunas de las ventajas que representan los circuitos integrados con VHDL:

- **Notación formal.** Los circuitos integrados VHDL cuentan con una notación que permite su uso en cualquier diseño electrónico.
- **Disponibilidad pública.** VHDL es un estándar no sometido a patente o marca registrada alguna, por lo que cualquier empresa o institución puede utilizarla sin restricciones. Además, dado que el IEEE lo mantiene y documenta, existe la garantía de estabilidad y soporte.
- **Independencia tecnológica de diseño.** VHDL se diseñó para soportar diversas tecnologías de diseño (PLD, FPGA, ASIC, etc.) con distintas funcionalidades (circuitos combinacionales, secuenciales, síncronos y asíncronos), a fin de satisfacer las distintas necesidades de diseño.
- **Independencia de la tecnología y proceso de fabricación.** VHDL se creó para que fuera independiente de la tecnología y del proceso de fabricación del circuito o del sistema electrónico. El lenguaje funciona de igual manera en circuitos diseñados con tecnología MOS, bipolares, BICMOS, etc., sin necesidad de incluir en el diseño información

concreta de la tecnología utilizada o de sus características (retardos, consumos, temperatura, etc.), aunque esto puede hacerse de manera opcional.

- **Capacidad descriptiva en distintos niveles de abstracción.** El proceso de diseño consta de varios niveles de detalle, desde la especificación hasta la implementación final (niveles de abstracción). VHDL ofrece la ventaja de poder diseñar en cualquiera de estos niveles y combinarlos, con lo cual se genera lo que se conoce como simulación multinivel.
- **Uso como formato de intercambio de información.** VHDL permite el intercambio de información a lo largo de todas las etapas del proceso de diseño, con lo cual favorece el trabajo en equipo.
- **Independencia de los proveedores.** Debido a que VHDL es un lenguaje estándar, permite que las descripciones o modelos generados en un sitio sean accesibles desde cualquier otro, sean cuales sean las herramientas de diseño utilizadas.
- **Reutilización del código.** El uso de VHDL como lenguaje estándar permite reutilizar los códigos en diversos diseños, sin importar que hayan sido generados para una tecnología (CMOS, bipolar, etc.) e implementación (FPGA, ASIC, etc.) en particular.
- **Facilitación de la participación en proyectos internacionales.** En la actualidad VHDL constituye el lenguaje estándar de referencia a nivel internacional. Impulsado en sus inicios por el Departamento de Defensa de Estados Unidos, cualquier programa lanzado por alguna de las dependencias oficiales de ese país vuelve obligatorio su uso para el modelado de los sistemas y la documentación del proceso de diseño [11]. Este hecho ha motivado que diversas empresas y universidades adopten a VHDL como su lenguaje de diseño.

En Europa la situación es similar, ya que en nuestros días la mayoría de las grandes empresas del ramo lo ha definido como el lenguaje de referencia en todas las tareas de diseño, modelado, documentación y mantenimiento de los sistemas electrónicos. De hecho, el número de usuarios de VHDL en Europa es mayor que en Estados Unidos, debido en gran parte a que resulta el lenguaje más común en la mayoría de los consorcios.

1.5.3 Desventajas del desarrollo de circuitos integrados con VHDL

Como se puede observar, VHDL presenta grandes ventajas; sin embargo, es necesario mencionar también algunas desventajas que muchos diseñadores consideran importantes:

- En algunas ocasiones, el uso de una herramienta provista por alguna compañía en especial tiene características adicionales al lenguaje, con lo que se pierde un poco la libertad de diseño. Como método alternativo, se pretende que entre diseñadores que utilizan distintas herramientas exista una compatibilidad en sus diseños, sin que esto requiera un esfuerzo importante en la traducción del código.
- Debido a que VHDL es un lenguaje diseñado por un comité, presenta una alta complejidad, ya que se debe dar gusto a las diversas opiniones de los miembros de éste, por lo que resulta un lenguaje difícil de aprender para un novato.

1.5.4 VHDL en la actualidad

La actividad que se ha generado en torno a VHDL es muy intensa. En muchos países como España se han creado grupos de trabajo alrededor de dicho lenguaje y se realizan periódicamente conferencias, reuniones, etc., donde se presentan trabajos tanto en Estados Unidos (en el VIUF, VHDL International User's Forum) como en Europa (VHDL Forum for CAD in Europe), así como en el congreso EuroVHDL celebrado desde 1992[10].

La participación europea en el esfuerzo de estandarizar el lenguaje se canaliza a través del proyecto ESPRIT, encabezado por SIEMENS-NIXDORF. En el proyecto participan prácticamente todas las grandes compañías europeas del sector electrónico, como ANACAD, ICL, PHILLIPS, TGI y THOMSON-CSF, además de diversas universidades y centros de investigación. Otras empresas dedicadas a la microelectrónica se han ido adaptando poco a poco al lenguaje. Incluso en Japón está teniendo una gran aceptación, no obstante que cuentan con un lenguaje estándar propio llamado UDL/I.

El proceso de estandarización del VHDL no se detuvo con la primera versión del lenguaje (VHDL '87), sino que ha continuado con la nueva versión (VHDL '93) y constantes actualizaciones, mejoras y metodologías de uso. Entre estas adiciones o actualizaciones se encuentra una muy importante: la extensión analógica (1076.1), que permite la utilización de un lenguaje único en todas las tareas de especificación, simulación y síntesis de sistemas electrónicos digitales, analógicos o mixtos.

1.6 Compañías de soporte en hardware y software

Existen diversas compañías internacionales que fabrican o distribuyen dispositivos lógicos programables. Algunas ofrecen productos con características generales y otras introducen innovaciones a sus dispositivos. A continuación se mencionan algunas de las más importantes.

Altera Corporation

Altera es una de las compañías más importantes de producción de dispositivos lógicos programables y también es la que más familias ofrece, ya que tiene en el mercado ocho familias: APEXTM20K, FLEX®10K, FLEX 8000, FLEX 6000, MAX® 9000, MAX7000, MAX5000, y ClassicTM. La capacidad de integración en cada familia varía desde 300 hasta 1 000 000 de compuertas utilizables por dispositivo, además de que todas tienen la capacidad de integrar sistemas complejos.

Las características generales más significativas de los dispositivos Altera son las siguientes:

- Frecuencia de operación del circuito superior a los 175 Mhz y retardos pin a pin de menos de 5 ns.
- La implementación de bloques de arreglos integrados (EAB), que se usan para realizar circuitos que incluyan funciones aritméticas como multiplicadores, ALU, y secuenciadores. También se aplican en microprocesadores, microcontroladores y funciones complejas con DSP (procesadores digitales de señales) [12].
- La programación en sistema (ISP), que permite programar los dispositivos montados en la tarjeta (Fig. 1.19).

En la figura 1.19a observamos la programación en sistema; es decir, no hay que retirar el circuito de la tarjeta para programarlo. En la figura 1.19b se muestra lo contrario: en este caso el tipo de programación es similar a la grabación cotidiana que realizamos, debido a que se debe colocar y quitar el dispositivo todas las veces que se quiera programar.



Figura 1.19 a) Programación en sistema. b) Programación en montaje.

- Más de cuarenta tipos y tamaños de encapsulados, incluyendo el TQFP (thin quad flat pack), el cual es un dispositivo delgado, de forma cuadrangular y plano, que permite ahorrar un espacio considerable en la tarjeta.
- Operación multivoltaje, entre los 5 y 3.3 volts, para máximo funcionamiento y 2.5 V en sistemas híbridos.

- Potentes herramientas de software como MAX+PLUS II, que soporta todas las familias de dispositivos de Altera, así como el software estándar compatible con VHDL.

Cypress semiconductor

La compañía Cypress Semiconductor ofrece una amplia variedad de dispositivos lógicos programables complejos (CPLD), que se encuentran en las familias Ultra37000TM y FLASH370iTM. Cada una de estas familias ofrece la reprogramación en sistema (ISR), la cual permite reprogramar los dispositivos las veces que se quiera dentro de la tarjeta.

Todos los dispositivos de ambas familias trabajan con voltajes de operación de 5 o de 3.3 V y en su interior contienen desde 32 hasta 128 macroceldas.

En lo que respecta a software de soporte, Cypress ofrece su poderoso programa Warp, el cual se basa en VHDL. Este programa permite simular de manera gráfica el circuito programado, generando un archivo de mapa de fusibles (jedec) que puede ser programado directamente en cualquier PLD, CPLD o FPGA de Cypress o de otra compañía que sea compatible.

Clear logic

La compañía Clear Logic introdujo en noviembre de 1998 los dispositivos lógicos procesados por láser (LPDL), tecnología que provee reemplazos de los dispositivos de la Compañía Altera, pero a un costo y tamaño menores. La tecnología LPLD puede disponer de arriba de un millón de transistores para construir alrededor de 512 macroceldas. Sustituye al dispositivo MAX 7512A de Altera y reduce el tamaño más de 60% respecto al chip original. Las primeras familias introducidas por Clear Logic son CL7000 y CL7000E, las cuales tienden a crecer en un futuro.

Motorola

Motorola, empresa líder en comunicaciones y sistemas electrónicos, ofrece también dispositivos **FPGA** y **FPAA** (Field Programmable Array Analog: campos programables de arreglos analógicos). Los FPAA son los primeros campos programables para aplicaciones analógicas, utilizados en las áreas de transporte, redes, computación y telecomunicaciones.

Xilinx

Xilinx es una de las compañías líder en soluciones de lógica programable, incluyendo circuitos integrados avanzados, herramientas en software para diseño, funciones predefinidas y soporte de ingeniería. Xilinx fue la compañía que inventó los FPGA y en la actualidad sus dispositivos ocupan más de la mitad del mercado mundial de los dispositivos lógicos programables.

Los dispositivos de Xilinx reducen de manera significativa el tiempo requerido para desarrollar aplicaciones en las áreas de computación, telecomunicaciones, redes, control industrial, instrumentación, aplicaciones militares y para el consumo general.

Las familias de CPLD XC9500 y XC9500XL proveen una larga variedad de dispositivos programables con características que van desde los 5 a 3.3 volts de operación, 36 a 288 macroceldas, 34 a 192 terminales de entrada y salida, y programación en sistema.

Los dispositivos de las familias XC4000 y XC1700 de FPGA manejan voltajes de operación entre los 5 y 3.3 V, una capacidad de integración arriba de las 40 000 compuertas y programación en sistema.

En lo que se refiere a software, Xilinx desarrolló una importante herramienta llamada Foundation Series, que soporta diseños estándares basados en ABEL-HDL y en VHDL. Esta herramienta se ofrece en versión estudiantil y profesional.

De manera general, existe una amplia y variada gama de dispositivos lógicos programables disponibles en el mercado. La elección de uno u otro depende de los recursos con que cuenta el diseñador y los requerimientos del diseño. En la tabla 1.7 se muestran de forma simplificada algunas de las compañías que ofrecen soluciones de lógica programable, mientras que en la figura 1.20 se presentan sus productos.

Futuro de la lógica programable

Debido al auge actual de la lógica programable, no es difícil suponer que se pretende mejorar a futuro las herramientas existentes con el fin de extender su campo de aplicación a más áreas. Algunas compañías buscan mejorar la funcionalidad e integración de sus circuitos a fin de competir con el mercado de los ASIC. Esto mejoraría el costo por volumen, el ciclo de diseño y se disminuiría el voltaje de consumo.

Otra característica que se pretende mejorar es la reprogramación de los circuitos, debido a que su implementación requiere muchos recursos físicos y tecnológicos. Por esta razón se busca cambiar las metodologías de diseño para incluir sistemas reprogramables por completo.

Algunos desarrollos cuentan con memoria RAM o microprocesadores integrados en la tarjeta de programación. La tendencia de algunos fabricantes es integrar estos recursos en un circuito.

Compañía	Productos de hardware	Herramientas software
Altera	FPGA: Familias APEX 20K, FLEX 10K, FLEX 6000, MAX 9000, MAX 7000, MAX 5000 y CLASSIC	MAX + PLUS II: Soporta VHDL, Verilog y entrada esquemática.
Chip Express	LPGA (Laser Program Gate Array): CX3000, CX2000 y QYH500	QuICk Place&route: Diseños en base a vectores de prueba (CTV , ChipExpress Test Vector) y VHDL
Clear Logic	LPLD (Laser-processed Logic Device): CL7009, CL7000E, CL7000S	Desarrollos basados en herramientas de Altera.
Cypress Semiconductors	PLD: GAL22V10 CPLD: Ultra37000, FLASH370i	WARP: Soporta VHDL, Verilog y esquemáticos.
Motorola	FPAA (Field Programmable Analog Array): MPAA020	Easy Analog: herramienta de diseño interactiva exclusiva para diseño con FPAA.
Vantis	FPGA: Familias MACH4 y MACH5A	MACHXL: VHDL y Verilog
Quick Logic	PAasic (Asic Programable) y la familia QL de FPGA.	Quick Works: herramienta de soporte para VHDL, Verilog y captura esquemática.
Xilinx	CPLD: Familia XC9500 y XC9500XL FPGAs Familia XC400 y XC1700	Xilinx Foundation Series: soporta ABEL-HDL, VHDL y esquemáticos.

Tabla 1.7 Compañías de Soporte de lógica programable.

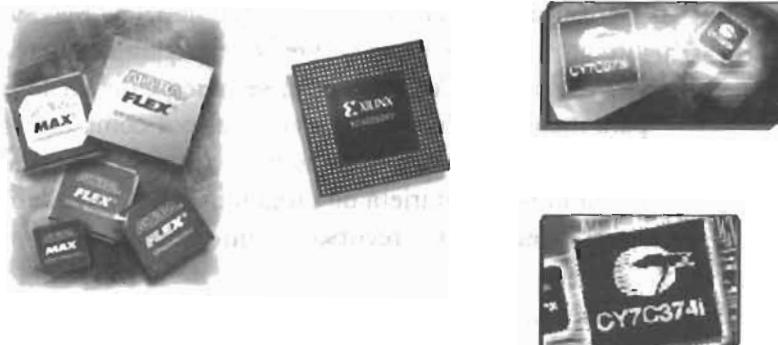


Figura 1.20 Dispositivos lógicos programables.

Ejercicios

- 1.1 ¿Qué significa monolítico?
- 1.2 ¿Cuál es el significado de las siglas ASIC?
- 1.3 ¿Cuáles son las categorías de tecnologías de fabricación de CI?
- 1.4 Describa en qué consiste el diseño Full Custom.
- 1.5 Mencione las características más relevantes del diseño Full Custom.
- 1.6 ¿Cuál es el significado de las siglas PLD?
- 1.7 ¿Qué tienen en común los dispositivos PROM, PLA, PAL, GAL y los CPLD y FPGA?
- 1.8 ¿Qué es OLMC?
- 1.9 ¿Cuál es el significado de las siglas CPLD y FPGA?
- 1.10 Describa cómo se encuentra estructurado un CPLD.
- 1.11 Describa la estructura de un FPGA en términos generales.
- 1.12 ¿Qué es un compilador lógico?
- 1.13 ¿Cuál es el significado de las siglas VHDL?
- 1.14 ¿Qué significado tienen las siglas VHSIC?
- 1.15 Describa tres ventajas de la programación en VHDL.
- 1.16 ¿Cuáles son las compañías más importantes en la fabricación de dispositivos lógicos programables?

Capítulo 2

VHDL: su organización y arquitectura

Introducción

Tal como lo indican sus siglas, VHDL (**H**ardware **D**escription **L**anguage) es un lenguaje orientado a la descripción o modelado de sistemas digitales; es decir, se trata de un lenguaje mediante el cual se puede describir, analizar y evaluar el comportamiento de un sistema electrónico digital.

VHDL es un lenguaje poderoso que permite la integración de sistemas digitales sencillos, elaborados o ambos en un dispositivo lógico programable, sea de baja capacidad de integración como un GAL, o de mayor capacidad como los CPLD y FPGA.

2.1 Unidades básicas de diseño

La estructura general de un programa en VHDL está formada por **módulos** o **unidades** de diseño, cada uno de ellos compuesto por un conjunto de declaraciones e instrucciones que **definen**, describen, estructuran, analizan y evalúan el comportamiento de un sistema digital.

Existen cinco tipos de unidades de diseño en VHDL: declaración de entidad (**entity declaration**), arquitectura (**architecture**), configuración (**configuration**), declaración del paquete (**package declaration**) y cuerpo del paquete (**package body**). En el desarrollo de programas en VHDL pueden utilizarse o no tres de los cinco módulos, pero dos de ellos (entidad y arquitectura) son indispensables en la estructuración de un programa.

Las declaraciones de entidad, paquete y configuración se consideran unidades de diseño **primarias**, mientras que la arquitectura y el cuerpo del paquete son unidades de diseño **secundarias** porque dependen de una entidad primaria que se debe analizar antes que ellas.

2.2 Entidad

Una entidad (*entity*) es el bloque elemental de diseño en VHDL. Las entidades son todos los elementos electrónicos (sumadores, contadores, compuertas, flip-flops, memorias, multiplexores, etc.) que forman de manera individual o en conjunto un sistema digital. La entidad puede representarse de muy diversas maneras; por ejemplo, la figura 2.1a) muestra la arquitectura de un sumador completo a nivel de compuertas; ahora bien, esta entidad se puede representar a nivel de sistema indicando tan sólo las entradas (Cin, A y B) y salidas (SUMA y Cout) del circuito; figura 2.1b). De igual forma, la integración de varios subsistemas (medio sumador) puede representarse mediante una entidad [Fig. 2.1c)]. Los subsistemas pueden conectarse internamente entre sí; pero la entidad sigue identificando con claridad sus entradas y salidas generales.

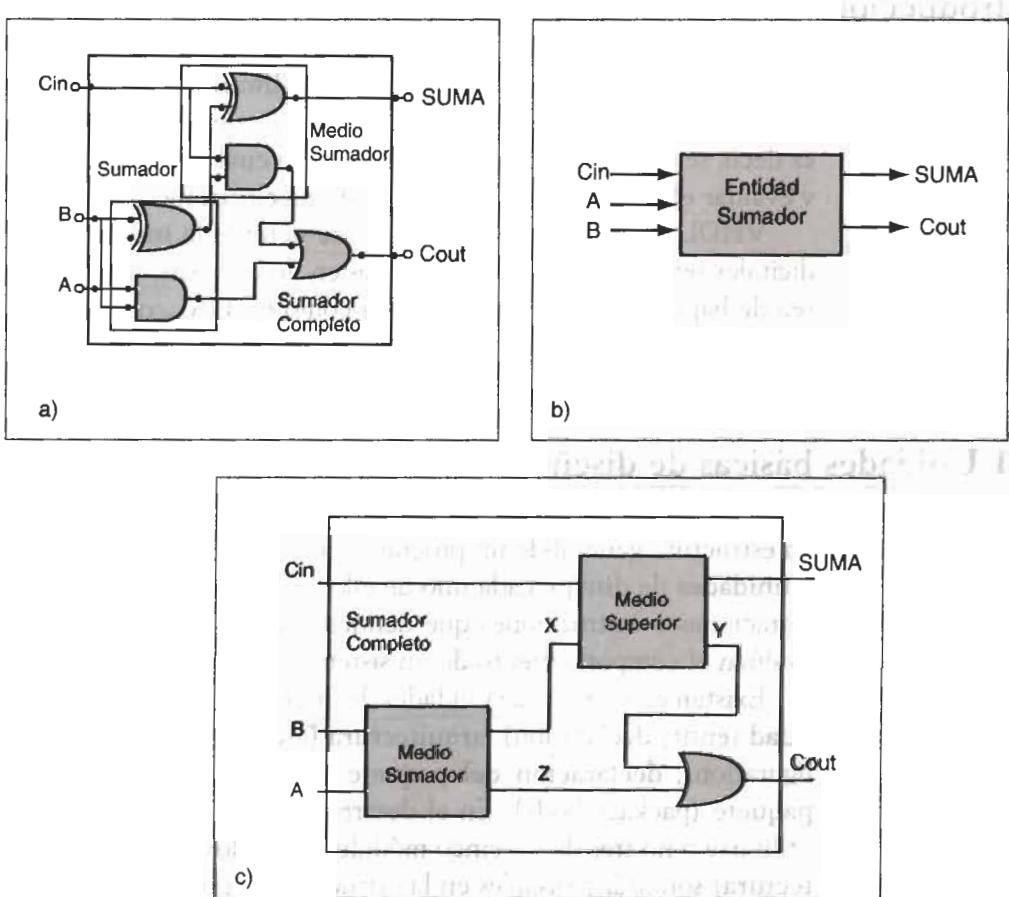


Figura 2.1 a) Descripción a nivel de compuertas. **b)** Símbolo funcional de la entidad; **c)** Diagrama a bloques representativo de la entidad.

2.2.1 Puertos de entrada-salida

Cada una de las señales de entrada y salida en una entidad son referidas como **puerto**, el cual es similar a una terminal (pin) de un símbolo esquemático. **Todos los puertos que son declarados deben tener un nombre, un modo y un tipo de dato.** El nombre se utiliza como una forma de llamar al puerto; el modo permite definir la dirección que tomará la información y el tipo define qué clase de información se transmitirá por el puerto. Por ejemplo, respecto a los puertos de la entidad que representan a un comparador de igualdad (Fig. 2.2), las variables **a** y **b** denotan los puertos de entrada y la variable **c** se refiere al puerto de salida.

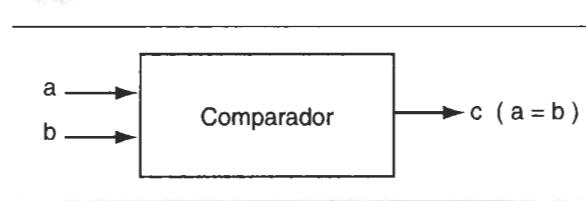


Figura 2.2 Comparador de igualdad.

2.2.2 Modos

Como ya se mencionó, **un modo permite definir la dirección en la cual el dato es transferido a través de un puerto. Un modo puede tener uno de cuatro valores: in (entrada), out (salida), inout (entrada/salida) y buffer (Fig. 2.3).**

- **Modo in.** Se refiere a las señales de entrada a la entidad. Este sólo es unidireccional y nada más permite el flujo de datos hacia dentro de la entidad.
- **Modo out.** Indica las señales de salida de la entidad.
- **Modo inout.** Permite declarar a un puerto de forma bidireccional —es decir, de entrada/salida—; además permite la retroalimentación de señales dentro o fuera de la entidad.
- **Modo buffer.** Permite hacer retroalimentaciones internas dentro de la entidad, pero a diferencia del modo **inout**, el puerto declarado se comporta como una terminal de salida.

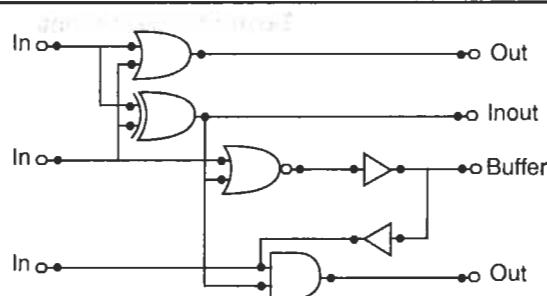


Figura 2.3 Modos y el curso de sus señales.

2.2.3 Tipos de datos

Los tipos son los valores (datos) que el diseñador establece para los puertos de entrada y salida dentro de una entidad; se asignan de acuerdo con las características de un diseño en particular. Algunos de los tipos más utilizados en VHDL son:

- **Bit**, el cual tiene valores de 0 y 1 lógico.
- **Boolean** (booleano) que define valores de verdadero o falso en una expresión
- **Bit_vector** (vectores de bits) que representa un conjunto de bits para cada variable de entrada o salida.
- **Integer** (entero) que representa un número entero.

Los anteriores son sólo algunos de los tipos que maneja VHDL, pero no son los únicos.¹

2.3 Declaración de entidades

Como se mencionó en la sección 2.1(Unidades básicas de diseño), los módulos elementales en el desarrollo de un programa dentro del lenguaje de descripción en hardware (VHDL) son la entidad y la arquitectura.

La declaración de una entidad consiste en la descripción de las entradas y salidas de un circuito de diseño identificado como entity (entidad); es decir, la declaración señala las terminales o pines de entrada y salida con que cuenta la entidad de diseño.

Por ejemplo, la forma de declarar la entidad correspondiente al circuito sumador de la figura 2.1b) se muestra a continuación:



```

1      --Declaración de la entidad de un circuito sumador
2      entity sumador is
3          port  (A,B, Cin:  in bit;
4                  SUMA, Cout: out bit);
5      end sumador;
```

Listado 2.1 Declaración de la entidad sumador de la figura 2.1b).

¹ En el apéndice 6 se listan los tipos de datos existentes en VHDL.

Los números de las líneas (1, 2, 3, 4, 5) no son parte del código; se usan como referencia para explicar alguna sección en particular. Las palabras en **negritas** están reservadas para el lenguaje de programación VHDL; esto es, tienen un significado especial para el programa; el diseñador asigna los otros términos.

Ahora comenzemos a analizar el código línea por línea. Observemos que la línea 1 inicia con **dos guiones (--)**, los cuales indican que el texto que está a la derecha es un **comentario** cuyo objetivo es documentar el programa, ya que el compilador ignora todos los comentarios. En la línea 2 se inicia la declaración de la entidad con la palabra reservada **entity**, seguida del **identificador** o nombre de la entidad (sumador) y la palabra reservada **is**. Los puertos de entrada y salida (**port**) se declaran en las líneas 3 y 4, respectivamente —en este caso los puertos de entrada son A, B y Cin—, mientras que SUMA y Cout representan los puertos de salida. El tipo de dato que cada puerto maneja es del tipo **bit**, lo cual indica que sólo pueden manejarse valores de '0' y '1' lógicos. Por último, en la línea 5 termina la declaración de entidad con la palabra reservada **end**, seguida del nombre de la entidad (sumador).

Debemos notar que como cualquier lenguaje de programación, VHDL sigue una sintaxis y una semántica dentro del código, mismas que hay que respetar. En esta entidad conviene hacer notar el uso de punto y coma (;) al finalizar una declaración y de dos puntos (:) al asignar nombres a las entradas y salidas.

Ejemplo 2.1

Declare la entidad del circuito lógico de la figura E2.1.

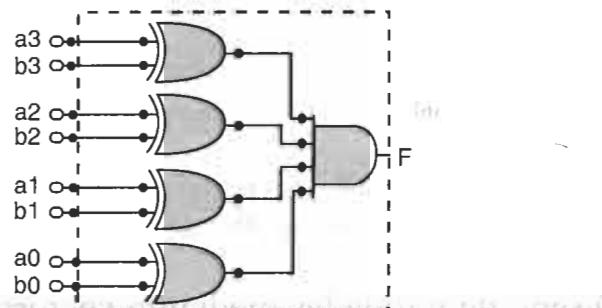


Figura E2.1

Solución

Como puede observarse, las entradas y salidas del circuito se encuentran delimitadas por la línea punteada. En este caso, a3, b3, a2, b2, ..., a0, b0 son las entradas y F es la salida.

La declaración de la entidad sería de la siguiente forma:

```

1 -- Declaración de la entidad
2 Entity circuito is
3     port( a3,b3,a2,b2,a1,b1,a0,b0: in bit;
4                                     F: out bit);
5     end circuito;
```

Identificadores

Los identificadores son simplemente los nombres o etiquetas que se usan para referir variables, constantes, señales, procesos, etc. Pueden ser números, letras del alfabeto y guiones bajos (_) que separan caracteres y no tienen una restricción en cuanto a su longitud. Todos los identificadores deben seguir ciertas especificaciones o reglas para que se puedan compilar sin errores, mismas que aparecen en la tabla 2.1.

Regla	Incorrecto	Correcto
El primer carácter siempre es una letra mayúscula o minúscula.	4suma	Suma4 SUMA4
El segundo carácter no puede ser un guion bajo	S_4bits	S4_bits
Dos guiones juntos no son permitidos	Resta__4	Resta_4_
Un identificador no puede utilizar símbolos	Clear#8	Clear_8

Tabla 2.1 Especificaciones para la escritura de identificadores.

VHDL cuenta con una lista de palabras reservadas que no pueden funcionar como identificadores (vea el Apéndice B).

2.4 Diseño de entidades mediante vectores

La entidad **sumador** realizada en el circuito del listado 2.1, usa bits individuales, los cuales sólo pueden representar dos valores lógicos (0 o 1). De manera general, en la práctica se utilizan conjuntos (palabras) de varios bits; en VHDL las palabras binarias se conocen como **vectores de bits**, los cuales se consideran un grupo y no como bits individuales. Como ejemplo considérense los vectores de 4 bits que se muestran a continuación:

vector_A	= [A3, A2, A1, A0]
vector_B	= [B3, B2, B1, B0]
vector_SUMA	= [S3, S2, S1, S0]

En la figura 2.4 se observa la entidad del sumador analizado antes, sólo que ahora las entradas A, B y la salida SUMA incorporan vectores de 4 bits en sus puertos. Obsérvese cómo la entrada Cin y la salida Cout son de un bit.

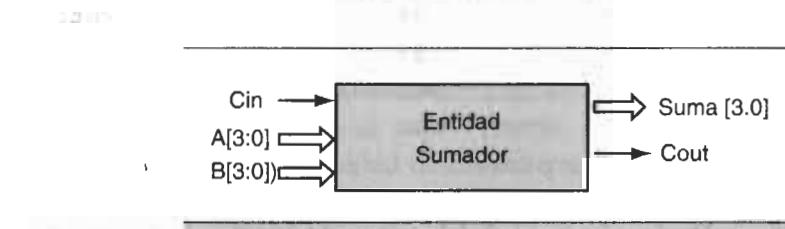


Figura 2.4 Entidad representada por vectores.

La manera de describir en VHDL una configuración que utilice vectores consiste en la utilización de la sentencia **bit_vector**, mediante la cual se especifican los componentes de cada uno de los vectores utilizados. La parte del código que se usa para declarar un vector dentro de los puertos es el siguiente:

```

port (vector_A, vector_B: in bit_vector (3 downto 0);
      vector_SUMA: out bit_vector (3 downto 0));

```

Esta declaración define los vectores (A, B y SUMA) con cuatro componentes distribuidos en orden descendente por medio del comando:

3 downto 0 (3 hacia 0)

los cuales se agruparían de la siguiente manera.

vector_A(3) = A3	vector_B(3) = B3	vector_SUMA(3) = S3
vector_A(2) = A2	vector_B(2) = B2	vector_SUMA(2) = S2
vector_A(1) = A1	vector_B(1) = B1	vector_SUMA(1) = S1
vector_A(0) = A0	vector_B(0) = B0	vector_SUMA(0) = S0

Una vez que se ha establecido el orden en que aparecerán los bits enumerados en cada vector, no se puede modificar, a menos que se utilice el comando **to**:

0 to 3 (0 hasta 3)

que indica el orden de aparición en sentido ascendente.

Ejemplo 2.2

Describa en VHDL la entidad del circuito sumador representado en la figura 2.4. Observe cómo la entrada Cin (Carry in) y la salida Cout (Carry out) se expresan de forma individual.

Solución

```

entity sumador is
port (A,B:  in bit_vector (3 downto 0);
      Cin:   in bit;
      Cout:  out bit;
      SUMA:  out bit_vector(3 downto 0));
end sumador;

```

Ejemplo 2.3

Declare la entidad del circuito lógico mostrado en la figura del ejemplo 2.1, mediante vectores.

Solución

```

1 -Declaración de entidades mediante vectores
2 entity detector is
3 port (a,b: in bit_vector(3 downto 0);
4           F: out bit));
5 end detector;

```

2.4.1 Declaración de entidades mediante librerías y paquetes

Una parte importante en la programación con VHDL radica en el uso de **librerías y paquetes** que permiten declarar y almacenar estructuras lógicas, seccionadas o completas que facilitan el diseño.

Una librería o biblioteca es un lugar al que se tiene acceso para utilizar las unidades de diseño predeterminadas por el fabricante de la herramienta (**paquete**) y su función es agilizar el diseño. En VHDL se encuentran definidas dos librerías llamadas **ieee** y **work** (Fig. 2.5). Como puede observarse, en la librería **ieee** se encuentra el paquete **std_logic_1164**, mientras que en la librería **work** se hallan **numeric_std**, **std_arith** y **gatespkg**.

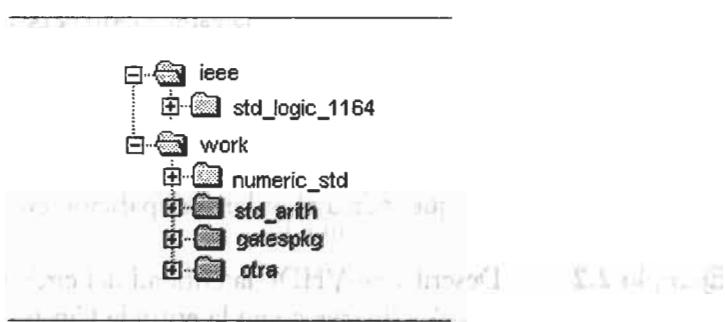


Figura 2.5 Contenido de las librerías ieee y work.

En una librería también se permite almacenar el resultado de la compilación de un diseño, con el fin de utilizar en uno o varios programas. La librería work es el lugar establecido donde se almacenan los programas que el usuario va generando. Esta librería se encuentra siempre presente en la compilación de un diseño y los diseños se guardan en ella mientras no se especifique otra. La carpeta otra mostrada en la figura 2.5 representa esta situación.

Un paquete es una unidad de diseño que permite desarrollar un programa en VHDL de una manera ágil, debido a que contiene algoritmos preestablecidos (sumadores, restadores, contadores, etc.) que ya tienen optimizado su comportamiento. Por esta razón, el diseñador no necesita caracterizar paso a paso una nueva unidad de diseño si ya se encuentra almacenada en algún paquete —en cuyo caso basta con llamarla y especificarla en el programa—. Por lo tanto, un paquete no es más que una unidad de diseño formada por declaraciones, programas, componentes y subprogramas, que incluyen los diversos tipos de datos (bit, booleano, std_logic), empleados en la programación en VHDL y que suelen formar parte de las herramientas en software.

Por último, cuando en el diseño se utiliza algún paquete es necesario llamar a la librería que lo contiene. Para esto se utiliza la siguiente declaración:

library ieee;

Lo anterior permite el uso de todos los componentes incluidos en la librería ieee. En el caso de la librería de trabajo (work), su uso no requiere la declaración library, dado que la carpeta work siempre está presente al desarrollar un diseño.

2.4.2 Paquetes

- El paquete std_logic_1164 (estándar lógico_1164) que se encuentra en la librería ieee contiene todos los tipos de datos que suelen emplearse en VHDL (std_logic_vector, std_logic, entre otros).

El acceso a la información contenida en un paquete es por medio de la sentencia **use**, seguida del nombre de la librería y del paquete, respectivamente:

use nombre_librería.nombre_paquete.all;

por ejemplo:

use ieee.std_logic_1164.all;

En este caso ieee es la librería, std_logic_1164 es el paquete y la palabra reservada **all** indica que se pueden usar todos los componentes almacenados en el paquete.

- El paquete **numeric_std** define funciones para realizar operaciones entre diferentes tipos de datos (sobrecargado); además, los tipos pueden representarse con signo o sin éste (vea el Apéndice A).
- El paquete **numeric_bit** define tipos de datos binarios con signo o sin éste.
- El paquete **std_arith** define funciones y operadores aritméticos, como igual (=), mayor que (>), menor que (<), entre otros (vea el Apéndice A).

En lo sucesivo, se usarán a menudo las librerías y paquetes de los programas desarrollados en el texto.

Ejemplo 2.4

En la figura E2.4 se muestra el bloque representativo de un circuito multiplicador de 2 bits. La multiplicación de (X_1, X_0) y (Y_1, Y_0) producen la salida Z_3, Z_2, Z_1, Z_0 .

Declare la entidad del circuito utilizando la librería **ieee** y el paquete **std_logic_1164.all**.

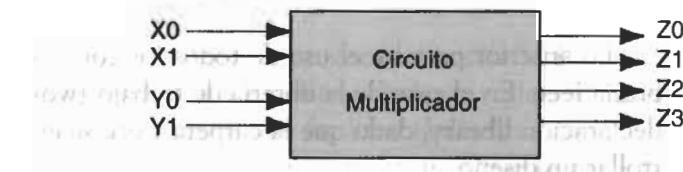


Figura E2.4

Solución

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity multiplica is
4 port (X0,X1,Y0,Y1: in std_logic;
5       Z3,Z2,Z1,Z0: out std_logic);
6 end multiplica;
```

2.5 Arquitecturas

Una **arquitectura** (*architecture*) se define como la estructura que describe el **funcionamiento de una entidad**, de tal forma que permita el desarrollo de los procedimientos que se llevarán a cabo con el fin de que la entidad cumpla las condiciones de funcionamiento deseadas.

La gran ventaja que presenta VHDL para definir una arquitectura radica en la manera en que pueden describirse los diseños; es decir, mediante el **algoritmo de programación** empleado se puede describir desde el nivel de compuertas hasta sistemas complejos.

De manera general, los estilos de programación utilizados en el diseño de arquitecturas se clasifican como:

- Estilo funcional
- Estilo por flujo de datos
- Estilo estructural

El nombre asignado a estos estilos **no es importante**, ya que es tarea del diseñador escribir el comportamiento de un circuito utilizando uno u otro estilo que a su juicio le sea el más acertado.

2.5.1 Descripción funcional

En la figura 2.6 se describe funcionalmente el circuito comparador. Se trata de una descripción funcional porque expone la forma en que trabaja el sistema; es decir, las descripciones consideran la relación que hay entre las entradas y las salidas del circuito, sin importar cómo esté organizado en su interior. Para este caso:

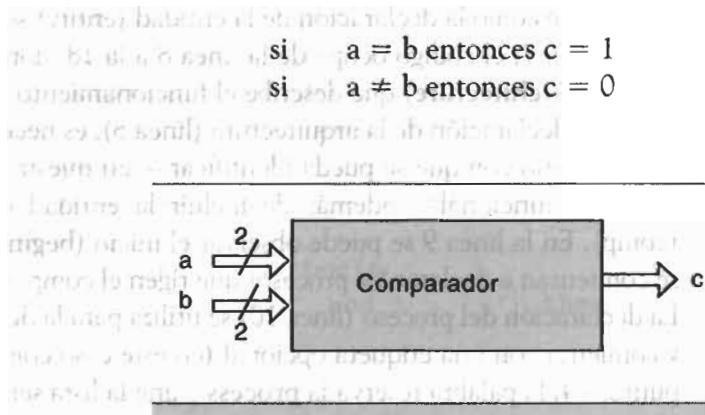


Figura 2.6 Descripción funcional de un comparador de igualdad de dos bits.

El código que representa el circuito de la figura 2.6 se muestra en el listado 2.2:

```

1      -- Ejemplo de una descripción funcional
2      library ieee;
3      use ieee.std_logic_1164.all;
4      entity comp is
5          port (a,b: in bit_vector( 1 downto 0);
6                  c: out bit);
7      end comp;
8      architecture funcional of comp is
9      begin
10         comparam: process (a,b)
11         begin
12             if a = b then
13                 c <='1';
14             else
15                 c<='0';
16             end if;
17         end process comparam;
18     end funcional;

```

Listado 2.2 Arquitectura funcional de un comparador de igualdad de 2 bits.

Nótese cómo la declaración de la entidad (**entity**) se describe en las líneas de la 1 a la 7; el código ocupa de la línea 8 a la 18, donde se desarrolla el algoritmo (**architecture**) que describe el funcionamiento del comparador. Para iniciar la declaración de la arquitectura (línea 8), es necesario definir un nombre arbitrario con que se pueda identificar —en nuestro caso el nombre asignado fue funcional— además de incluir la entidad con que se relaciona (**comp**). En la línea 9 se puede observar el inicio (**begin**) de la sección donde se comienzan a declarar los procesos que rigen el comportamiento del sistema. La declaración del proceso (línea 10) se utiliza para la definición de algoritmos y comienza con una etiqueta opcional (en este caso **comparama**), seguida de dos puntos (:), la palabra reservada **process** y une la lista sensitiva (a y b), que hace referencia a las señales que determinan el funcionamiento del proceso.

Al seguir el análisis, puede notarse que de la línea 12 a la 17 el proceso se ejecuta mediante declaraciones secuenciales del tipo **if-then-else** (si-entonces-si no). Esto se interpreta como sigue (línea 12): si el valor de la señal **a** es igual al valor de la señal **b**, entonces '1' se asigna a **c**, si no (else) se asigna un '0' (el símbolo <= se lee como "se asigna a"). Una vez que se ha definido el proceso, se termina con la palabra reservada **end process** y de manera opcional el nombre del proceso (**comparama**); de forma similar se añade la etiqueta (**funcional**) al terminar la arquitectura en la línea 18.

Como se puede observar, la **descripción funcional** se basa principalmente en el uso de procesos y de declaraciones secuenciales, las cuales permiten modelar la función con rapidez.

Ejemplo 2.5

Describa mediante declaraciones del tipo if-then-else el funcionamiento de la compuerta OR mostrada en la figura E2.5 con base en la tabla de verdad.

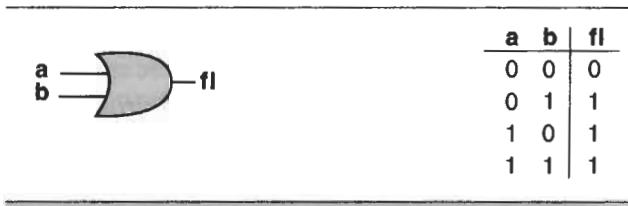


Figura E2.5

Solución

Como puede observarse, la declaración de la librería y el paquete se introducen en las líneas 2 y 3, respectivamente. La declaración de la entidad se define entre las líneas 4 a 7 inclusive. Por último, la arquitectura se describe en las líneas 8 a 17.

```

1 -- Declaración funcional
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity com_or is
5 port( a,b: in std_logic;
6       f1: out std_logic);
7 end com_or;
8 architecture funcional of com_or is
9 begin
10 process (a,b) begin
11   if (a = '0' and b = '0') then
12     f1 <= '0';
13   else
14     f1 <= '1';
15 end if;
16 end process;
17 end funcional;
```

2.5.2 Descripción por flujo de datos

La descripción por flujo de datos indica la forma en que los datos se pueden transferir de una señal a otra sin necesidad de declaraciones secuenciales

(if-then-else). Este tipo de descripciones permite definir el flujo que tomarán los datos entre módulos encargados de realizar operaciones. En este tipo de descripción se pueden utilizar dos formatos: mediante instrucciones when-else (cuando-si no) o por medio de ecuaciones booleanas.

a) Descripción por flujo de datos mediante when-else

A continuación se muestra el código del comparador de igualdad de dos bits descrito antes (Fig. 2.6) Nótese que la diferencia entre los listados 2.2 y 2.3 radica en la eliminación del proceso y en la descripción sin declaraciones secuenciales (if-then-else).

```

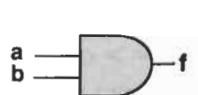
1 --Ejemplo de declaración de la entidad de un comparador
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity comp is
5 port (a,b:  in bit_vector (1 downto 0);
6        c:  out bit);
7 end comp;
8
9 architecture f_datos of comp is
10 begin
11   c <= '1' when (a = b) else '0'; (asigna a C el valor
12   1 de 1 cuando a=b si no vale 0).
13 end f_datos;
```

Listado 2.3 Arquitectura por flujo de datos.

En VHDL se manejan dos tipos de declaraciones: *secuenciales* y *concurrentes*. Una declaración secuencial de la forma if-then-else se halla en el listado 2.2 dentro del proceso, donde su ejecución debe seguir un orden para evitar la pérdida de la lógica escrita. En cambio, en una declaración concurrente esto no es necesario, ya que no importa el orden en que se ejecutan. Tal es el caso del listado 2.3.

Ejemplo 2.6

Con base en la tabla de verdad y mediante la declaración when-else, describa el funcionamiento de la siguiente compuerta AND.



a	b	f1
0	0	0
0	1	0
1	0	0
1	1	1

Figura E2.6

Solución

```

1 --Algoritmo utilizando flujo de datos
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity com_and is
5 port( a,b: in std_logic;
6       f: out std_logic);
7 end com_and;
8 architecture compuerta of com_and is
9 begin
10    f <= '1' when (a = '1' and b = '1') else
11        '0';
12 end compuerta;

```

b) Descripción por flujo de datos mediante ecuaciones booleanas

Otra forma de describir el circuito comparador de dos bits es mediante la obtención de sus ecuaciones booleanas figura 2.7. En el listado 2.4 se observa este desarrollo.

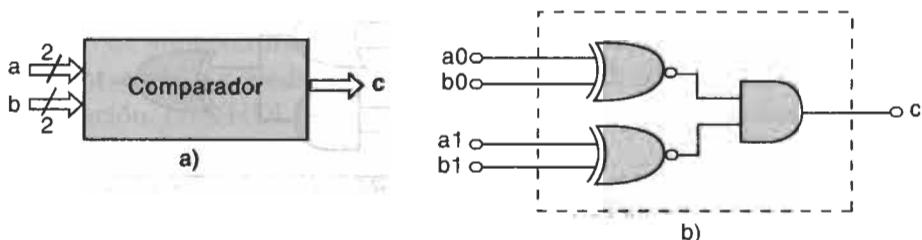


Figura 2.7 a) Entidad del comparador de dos bits. **b)** Comparador de dos bits realizado con compuertas.

El interior del circuito comparador de la figura 2.7a) puede representarse por medio de compuertas básicas [Fig. 2.7b)] y este circuito puede describirse mediante la obtención de sus ecuaciones booleanas.

```

1 -- Ejemplo de declaración de la entidad de un comparador
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity comp is
5 port      ( a,b:  in bit_vector (1 downto 0) ;
6             c:    out bit) ;
7 end comp;
8 architecture booleana of comp is
9 begin
10    c <= (a(1)  xnor b(1)
11        and   a(0) xnor b(0));
12 end booleana;

```

Listado 2.4 Arquitectura de forma de flujo de datos construido por medio de ecuaciones booleanas.

La forma de **flujo de datos** en cualquiera de sus representaciones describe el camino que los datos siguen al ser transferidos de las operaciones efectuadas entre las entradas a y b a la señal de salida c.

Ejemplo 2.7

Describa mediante ecuaciones booleanas el circuito mostrado a continuación.

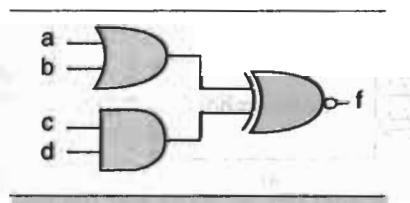


Figura E2.7

Solución

```

--Declaración mediante ecuaciones booleanas
library ieee;
use ieee.std_logic_1164.all;
entity ejemplo is
port ( a,b,c,d: in std_logic;
       f: out std_logic);
end ejemplo;
architecture compuertas of ejemplo is
begin
  f <= ((a or b) xnor (c and b));
end compuertas;

```

2.5.3 Descripción estructural

Como su nombre indica, una **descripción estructural** basa su comportamiento en modelos lógicos establecidos (compuertas, sumadores, contadores, etc.). Según veremos más adelante, el usuario puede diseñar estas estructuras y guardarlas para su uso posterior o tomarlas de los paquetes contenidos en las librerías de diseño del software que se esté utilizando.

En la figura 2.8 se encuentra un esquema del circuito comparador de igualdad de 2 bits, el cual está formado por compuertas nor-exclusivas y una compuerta AND.

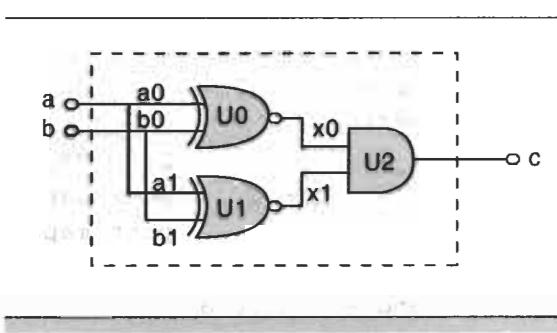


Figura 2.8 Representación esquemática de un comparador de 2 bits.

En nuestro caso, cada compuerta (modelo lógico) se encuentra dentro del paquete **gatespkg**,⁶ del cual se toman para estructurar el diseño. A su vez, este tipo de arquitecturas estándares se conoce como **componentes**, que al interconectarse por medio de señales internas (x_0 , x_1) permiten proponer una solución. En VHDL esta conectividad se conoce como **netlist**⁷ o **listado de componentes**.

Para iniciar la programación de una entidad de manera estructural, es necesario la descomposición lógica del diseño en pequeños submódulos (jerarquizar), los cuales permiten analizar de manera práctica el circuito, ya que la función de entrada/salida es conocida. En nuestro ejemplo se conoce la función de salida de las dos compuertas **xnor**, por lo que al unirlas a la compuerta **and**, la salida **c** es el resultado de la operación **and** efectuada en el interior a través de las señales x_0 y x_1 (Fig. 2.8).

Es importante resaltar que una **jerarquía** en VHDL se refiere al procedimiento de dividir en bloques y no a que un bloque tenga mayor jerarquía (peso) que otro. Esta forma de dividir el problema hace de la descripción estructural una forma sencilla de programar. En el contexto del diseño lógico esto es observable cuando se analiza por separado alguna sección de un sistema integral.

⁶ El paquete *compuerta* fue programado para este ejemplo. En el capítulo 8 se verá a detalle su desarrollo.

⁷ Un netlist se refiere a la forma en como se encuentran conectados los componentes dentro de una estructura y las señales que propicia esta interconexión.

En el listado 2.5 se muestra el código del programa que representa al esquema de la figura 2.8.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity comp is port(
4     a,b: in bit_vector (0 to 1);
5     c: out bit);
6 end comp;
7 use work.compuerta.all;
8 architecture estructural of comp is
9 signal x: bit_vector (0 to 1);
10 begin
11     U0: xnor2    port map (a(0), b(0), x(0));
12     U1: xnor2    port map (a(1), b(1), x(1));
13     U2: and2      port map (x(0), x(1), c);
14 end estructural;

```

Listado 2.5 Descripción estructural de un comparador de igualdad de 2 bits.

En el código se puede ver que en la entidad nada más se describen las entradas y salidas del circuito (a, b y c), según se ha venido haciendo (líneas 3 a la 6). Los componentes **xnor** y **and** no se declaran debido a que se encuentran en el paquete de compuertas (**gatespkg**), el cual a su vez está dentro de la librería de trabajo (**work**), línea 7.

En la línea 8 se inicia la declaración de la arquitectura *estructural*. El algoritmo propuesto (líneas 11 a 13) describe la estructura de la siguiente forma: cada compuerta se maneja como un bloque lógico independiente (componente) del diseño original, al cual se le asigna una variable temporal (U0, U1 y U2); la salida de cada uno de estos bloques se maneja como una señal línea 9, **signal** x (x0 y x1), las cuales se declaran dentro de la arquitectura y no en la entidad, debido a que no representan a una terminal (**pin**) y sólo se utilizan para conectar bloques de manera interna a la entidad.

Por último, podemos observar que la compuerta **and** recibe las dos señales provenientes de x (x0 y x1), ejecuta la operación y asigna el resultado a la salida c del circuito.

Ejemplo 2.8

Realice el programa correspondiente en VHDL para el circuito mostrado en la figura E2.8. Utilice descripción estructural.

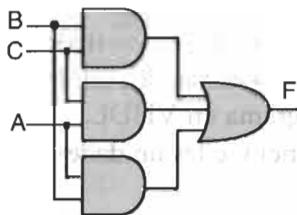


Figura E2.8

Solución

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity comp is
4 port( A,B,C : in std_logic;
5       F: out std_logic);
6 end comp;
7 use work.compuerta.all;
8 architecture estructura of comp is
9 signal x: bit_vector (0 to 2);
10 begin
11 U0: and2 port map (B, C, x(0));
12 U1: and2 port map (C, A, x(1));
13 U2: and2 port map (A, B, x(2));
14 U3: or3 port map (x(0), x(1), x(2), F);
15 end estructura;

```

Estructural

Comparación entre los estilos de diseño

El estilo de diseño utilizado en la programación del circuito depende del diseñador y de la complejidad del proyecto. Por ejemplo, un diseño puede describirse por medio de ecuaciones booleanas, pero si es muy extenso quizás sea más apropiado emplear estructuras jerárquicas para dividirlo; ahora bien, si se requiere diseñar un sistema cuyo funcionamiento dependa sólo de sus entradas y salidas, es conveniente utilizar la descripción funcional, la cual presenta la ventaja de requerir menos instrucciones y el diseñador no necesita un conocimiento previo de cada componente del circuito.

Ejercicios

Unidades básicas de diseño

- 2.1 Describa los cinco tipos de unidades de diseño en VHDL.
- 2.2 Determine cuáles son las unidades de diseño necesarias para realizar un programa en VHDL.
- 2.3 Mencione las unidades de diseño primarias y secundarias.

Declaración de entidades

- 2.4 Describa el significado de una entidad y cuál es su palabra reservada.
- 2.5 En la siguiente declaración de entidad indique:

```
library ieee;
use ieee.std_logic_1164.all;
entity seleccion is port (
    x: in std_logic_vector(0 to 3);
    f: out std_logic);
end seleccion;
```

- a) El nombre de la entidad _____
 - b) Los puertos de entrada _____
 - c) Los puertos de salida _____
 - d) El tipo de dato _____
- 2.6 Señale cuáles de los siguientes identificadores son correctos o incorrectos, colocando en las líneas de respuesta la letra 'C' o 'T', respectivamente.

lógico _____
 con_trol _____
 Página _____
 Registro _____
 2Suma _____

Desp_laza _____
 N_ivel _____
 architecture _____
 S_uma# _____
 Res_ta _____

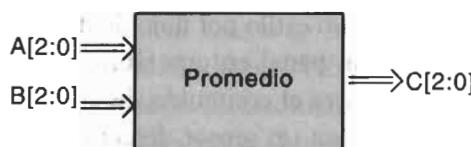
- 2.7 Declare la entidad para la compuerta AND del ejercicio 2.7:



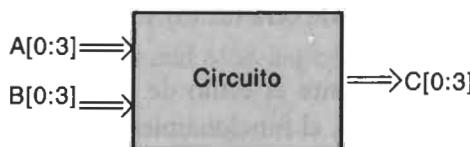
2.8 Declare la entidad para el siguiente circuito.



2.9 Declare la entidad para el circuito que se muestra en la figura. Utilice vectores.



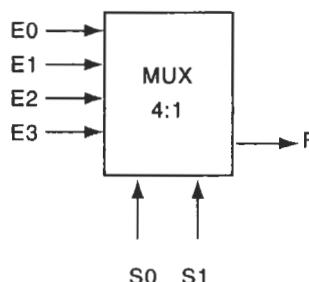
2.10 Declare la entidad para el siguiente circuito que utiliza vectores.



2.11 Describa qué es una librería en VHDL.

2.12 Indique el significado de la siguiente expresión:
`use ieee.std_logic_1164.all;`

2.13 Declare la entidad del circuito multiplexor de 4:1 mostrado en la figura del ejercicio 2.11 utilizando la librería: `ieee.std_logic_1164.all;`



2.14 Declare la entidad del multiplexor de 4:1 mostrado en la figura del ejercicio 2.11, si cada entrada esta formada por un vector de 4 bits.

2.15 Declare la entidad del multiplicador mostrado en el ejercicio 2.2 utilizando vectores y el paquete `std_logic_1164`.

Arquitecturas

- 2.16 Mediante un estilo funcional, programe en VHDL el funcionamiento de una lámpara para código Morse que encienda la luz al presionar un botón y la apague al soltarlo.
- 2.17 Con un estilo funcional, programe en VHDL el funcionamiento del motor de un ventilador en que el motor gire en un sentido al presionar el botón 'a' y en dirección contraria al oprimir el botón 'b'.
- 2.18 Con un estilo por flujo de datos, programe en VHDL el funcionamiento de un panel en una fábrica de empaquetamiento de arroz. Este panel muestra el contenido de 2 silos (a, b) que tiene la fábrica para guardar el arroz; un sensor detecta cuán llenos están, cuando se encuentran al 100% de su capacidad, envía un '1 lógico', y cuando tienen 25% o menos envía un '0 lógico'; si en uno de estos silos disminuye el contenido a 25% o menos, se prende una luz (c), si los dos sobrepasan ese límite se enciende otra luz (d) y suena una alarma (e).
- 2.19 Mediante el estilo de programación por flujo de datos, programe en VHDL el funcionamiento de un robot en una planta que espera a que se llene una tarima con cuatro cajas antes de llevarla a la bodega de almacenamiento; para saber si la tarima está llena cuenta con cuatro sensores, cada uno apunta a sendas cajas; si hay una caja marca un '1 lógico'; si falta, marca un '0 lógico'. Si falta alguna caja el robot no se puede ir, cuando están las cuatro cajas el robot se lleva la tarima.
- 2.20 Con el estilo de programación por flujo de datos, programe en VHDL el funcionamiento de una caja de seguridad cuya apertura requiere la presión simultánea de tres de cuatro botones ('a', 'b', 'c' y 'd'). Los botones que se deben oprimir son: 'a', 'c' y 'd'.
- 2.21 Mediante el estilo de programación estructural, programe en VHDL el problema del apagador de escalera. La función para este problema es $c = a \bar{b} + \bar{a} b$, donde a es el interruptor inferior, b es el interruptor superior y c es el foco.
- 2.22 Con un estilo estructural, programe en VHDL el funcionamiento de un motor que se enciende con la siguiente ecuación:
$$y = a \bar{b} + c \bar{b} + a c.$$

Capítulo 3

Diseño lógico combinacional mediante VHDL

Introducción

En este capítulo se diseñan los circuitos combinacionales más utilizados en el diseño lógico a través del lenguaje de descripción en hardware. Esto permite introducir nuevos conceptos, palabras reservadas, reglas, algoritmos, etc., que muestran la potencia y profundidad del lenguaje VHDL.

El desarrollo de cada una de las entidades de diseño descritas en este capítulo se puede optimizar mediante el uso adecuado de las declaraciones secuenciales, concurrentes o ambas, utilizando en esta descripción cualquiera de los tres tipos de arquitectura —funcional, por flujo de datos y estructural— vistos en el capítulo anterior. Sin embargo y dada la filosofía que queremos manejar en este texto, nos parece conveniente presentar soluciones que incluyan nuevas declaraciones, nuevos tipos de datos y nuevos algoritmos de análisis; es decir, no se pretende presentar la mejor opción de diseño para un problema; por el contrario, se propone brindar la mayor cantidad de soluciones (**modelos**) que le permitan deducir y construir sus estrategias de diseño para optimizar sus resultados.

3.1 Programación de estructuras básicas mediante declaraciones concurrentes

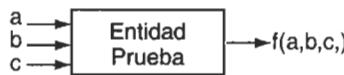
Como se mencionó antes, las declaraciones concurrentes se encuentran fuera de la declaración de un proceso y suelen usarse en las descripciones de flujo de datos y estructural. Esto se debe a que en una declaración concurrente no importa el orden en que se escriban las señales, ya que el resultado para determinada función sería el mismo.

En VHDL existen tres tipos de declaraciones concurrentes:

- Declaraciones condicionales asignadas a una señal (**when-else**)
- Declaraciones concurrentes asignadas a señales
- Selección de una señal (**with-select-when**)

3.1.1 Declaraciones condicionales asignadas a una señal (**when-else**)

La declaración **when-else** se utiliza para asignar valores a una señal, determinando así la ejecución de una condición propia del diseño. Para exemplificar, consideremos la entidad mostrada en la figura 3.1, cuyo funcionamiento se define en la tabla de verdad.



a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figura 3.1 Declaraciones **when-else**.

La entidad se puede programar mediante **declaraciones condicionales (when-else)**, debido a que este modelo permite definir paso a paso el comportamiento del sistema, según se muestra en el listado 3.1.

```

1 - Ejemplo combinacional básico
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity tabla is port(
5     a,b,c: in std_logic;
6     f: out std_logic);
7 end tabla;
8 architecture ejemplo of tabla is
9 begin
10    f <= '1' when (a='0' and b='0' and c='0') else
11        '1' when (a='0' and b='1' and c='1') else
12        '1' when (a='1' and b='1' and c='0') else
13        '1' when (a='1' and b='1' and c='1') else
14        '0';
15 end ejemplo;
  
```

Listado 3.1 Descripción de la entidad mostrada en la tabla de la figura 3.1.

Nótese que la función de salida f (línea 10) depende directamente de las condiciones que presentan las variables de entrada, además y dado que la ejecución inicial de una u otra condición no afecta la lógica del programa, el resultado es el mismo; es decir, la condición de entrada “111”, visualizada en la tabla de verdad, puede ejecutarse antes que la condición “000” sin alterar el resultado final.

La ventaja de la programación en VHDL en comparación con el diseño lógico puede intuirse considerando que la función de salida f mediante álgebra booleana se representa con:

$$f = \bar{a} \bar{b} \bar{c} + \bar{a} b c + a b \bar{c} + a b c$$

en el diseño convencional se utilizarían inversores, compuertas OR y compuertas AND; en VHDL la solución es directa utilizando la función lógica *and*. Como ejemplo, observemos que de la línea 10 a la 14 las instrucciones se interpretarán de la siguiente manera:

- 10 asigna a “ f ” el valor de 1 cuando $a = 0$ y $b = 0$ y $c = 0$ si no
- 11 asigna a “ f ” el valor de 1 cuando $a = 0$ y $b = 1$ y $c = 1$ si no
- 12 asigna a “ f ” el valor de 1 cuando $a = 1$ y $b = 1$ y $c = 0$ si no
- 13 asigna a “ f ” el valor de 1 cuando $a = 0$ y $b = 1$ y $c = 1$ si no
- 14 asigna a “ f ” el valor de 0.

Operadores lógicos

Los operadores lógicos más utilizados en la descripción de funciones booleanas, y definidos en los diferentes tipos de datos bit, son los operadores *and*, *or*, *nand*, *xor*, *xnor* y *not*. Las operaciones que se efectúen entre ellos (excepto *not*) deben realizarse con datos que tengan la misma longitud o palabra de bits.

En el momento de ser compilados los operadores lógicos presentan el siguiente orden y prioridad:

- 1) Expresiones entre paréntesis
- 2) Complementos
- 3) Función AND
- 4) Función OR

Las operaciones *xor* y *xnor* son transparentes al compilador y las interpreta mediante la suma de productos correspondiente a su función.

Como ejemplo del uso de operadores lógicos en VHDL, observemos la siguiente comparación:

Ecuación	En VHDL
$q = a + x \cdot y$	$q = a \text{ or } (x \text{ and } y)$
$y = a + \bar{b} \cdot \bar{c} + d$	$y = \text{not } (a \text{ or } (b \text{ and not } c) \text{ or } d)$

Ejemplo 3.1

Una función F depende de cuatro variables D, C, B, A, que representan un número binario, donde A es la variable menos significativa. La función F adopta el valor de uno si el número formado por las cuatro variables es inferior o igual a 7 y superior a 3. En caso contrario la función F es cero.

- Obtenga la tabla de verdad de la función F y realice el programa correspondiente en VHDL (utilice estructuras del tipo **when-else** y operadores lógicos).

Solución

Primero analizamos el enunciado y estructuramos la siguiente tabla de verdad según las especificaciones indicadas.

D	C	B	A	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

A partir de la tabla anterior, se puede programar la función F utilizando declaraciones condicionales **when-else**. El código VHDL es el siguiente:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity funcion is port(
4     D,C,B,A: in std_logic;
5     F: out std_logic);
6 end funcion;
7 architecture a_func of funcion is
8 begin
9     F <= '1' when (A = '0' and B = '0' and C = '1' and D = '0') else
10        '1' when (A = '1' and B = '0' and C = '1' and D = '0') else
11        '1' when (A = '0' and B = '1' and C = '1' and D = '0') else
12        '1' when (A = '1' and B = '1' and C = '1' and D = '0') else
13        '0';
14 end a_func;

```

3.1.2 Declaraciones concurrentes asignadas a señales

En este tipo de declaración encontraremos las funciones de salida mediante la ecuación booleana que describe el comportamiento de cada una de las compuertas. Obsérvese que ahora el circuito de la figura 3.2 cuenta con tres salidas (x_1 , x_2 y x_3) en lugar de una.

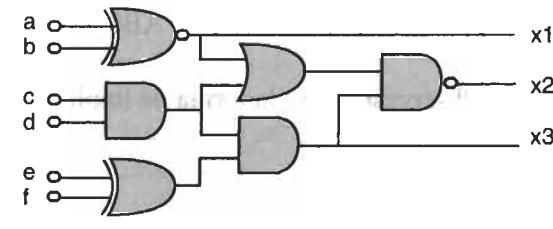


Figura 3.2 Circuito lógico realizado con compuertas.

El programa correspondiente al circuito de la figura 3.2 se muestra en el listado 3.2.

```

library ieee;
use ieee.std_logic_1164.all;
entity logic is port(
    a,b,c,d,e,f: in std_logic;
    x1,x2,x3:      out std_logic);
end logic;
architecture booleana of logic is
begin
    x1 <= a xnor b;
    x2 <= (((c and d)or(a xnor b)) nand
            ((e xor f)and(c and d)));
    x3 <= (e xnor f) and (c and d);
end booleana;

```

Listado 3.1 Declaraciones concurrentes asignadas a señales.

Ejemplo 3.2

Dada la tabla de verdad mostrada a continuación, halle las ecuaciones X, Y, Z, de la forma suma de productos y prográmelas en VHDL, utilizando declaraciones concurrentes asignadas a señales.

A	B	C	X	Y	Z
0	0	0	1	0	1
0	0	1	1	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	0	0

Solución

Las ecuaciones de la forma suma de productos para X, Y y Z se muestran a continuación:

- 1) $X = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + ABC$
- 2) $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$
- 3) $Z = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C$

Obsérvese ahora la forma de implementar estas ecuaciones por medio de operadores lógicos.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity concurrente is port (
4     A,B,C: in std_logic;
5     X,Y,Z: out std_logic);
6 end concurrente;
7 architecture a_conc of concurrente is
8 begin
9     X <= (not A and not B and not C) or (not A and not B and C);
10    or (not A and B and C ) or (A and B and C);
11    Y <= (not A and not B and C) or (A and not B and C)
12    or (A and B and not C);
13    Z <= (not A and not B and not C) or (not A and B and not C);
14    or (not A and B and C);
15 end a_conc;
```

3.1.3 Selección de una señal (with-select-when)

La declaración **with-select-when** se utiliza para asignar un valor a una señal con base en el valor de otra señal previamente seleccionada. Por ejemplo, en el listado correspondiente a la figura 3.3 se muestra el código que representa a este tipo de declaración. Como puede observarse, el valor de la salida C depende de las señales de entrada seleccionadas a(0) y a(1), de acuerdo con la tabla de verdad correspondiente.

a(0)	a(1)	C
0	0	1
0	1	0
1	0	1
1	1	0

Figura 3.3 Tabla de verdad.

```

library ieee;
use ieee.std_logic_1164.all;
entity circuito is port(
    a: in std_logic_vector (1 downto 0);
    c: out std_logic);
end circuito;
architecture arg_cir of circuito is
begin
    with a select
        c <= '1' when "00",
        '0' when "01",
        '1' when "10",
        '0' when others1;
end arg_cir;

```

Listado 3.3 Código VHDL correspondiente a la tabla de verdad de la figura 3.3.

Ejemplo 3.3

Se requiere diseñar un circuito combinacional que detecte números primos de 4 bits. Realice la tabla de verdad y elabore un programa que describa su función. Utilice instrucciones del tipo **with – select – when**.

¹ El uso de la palabra reservada **others** se explica a detalle en la sección multiplexores de este capítulo.

Solución

La tabla de verdad que resuelve la función es la siguiente:

X0	X1	X2	X3	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Si se considera que la entrada X es un vector de 4 bits y que F es la función de salida, el programa en VHDL quedaría de la siguiente manera.

3.2 Programación de estructuras básicas mediante declaraciones secuenciales

Como ya se mencionó, las declaraciones secuenciales son aquellas en las que el orden que llevan puede tener un efecto significativo en la lógica descrita. A diferencia de una declaración concurrente, una secuencial debe ejecutarse en el orden en que aparece y formar parte de un proceso (**process**).

Declaración if-then-else (si-entonces-si no). Esta declaración sirve para seleccionar una condición o condiciones basadas en el resultado de evaluaciones lógicas (falso o verdadero). Por ejemplo, observemos que en la instrucción:

```

if la condición es cierta then
    realiza la operación 1;
else
    realiza la operación 2;
end if;
  
```

si (**if**) condición se evalúa como verdadera, entonces (**then**) la instrucción indica que se ejecutará la *operación 1*. Por el contrario, si la condición se evalúa como falsa (**else**) correrá la *operación 2*. La instrucción que indica el fin de la declaración es **end if** (fin del si). Un ejemplo que ilustra este tipo de declaración se encuentra en la figura 2.6 y por comodidad se repite en la figura 3.4.

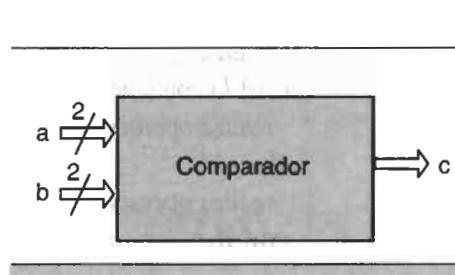


Figura 3.4 Comparador de igualdad de dos bits.

El código correspondiente a esta entidad de diseño se muestra en el listado 3.4.

```

1  --Ejemplo de declaración de la entidad comparador
2  entity comp is
3      port (a,b: in bit_vector( 1 downto 0 );
4          c: out bit);
5  end comp;
6  architecture funcional of comp is
7  begin
8      comienza; process (a,b)
9      begin
10         if a = b then
11             c <='1';
12         else
13             c <='0';
14         end if;
15     end process comienza;
16 end funcional;

```

Listado 3.4 Declaración secuencial de un comparador de igualdad de dos bits.

Notemos cómo en este tipo de ejemplos sólo son necesarias dos condiciones por evaluar, pero no en todos los diseños es así. Por tanto, cuando se requieren más condiciones de control, se utiliza una nueva estructura llamada **elsif** (**si no-si**), la cual permite expandir y especificar prioridades dentro del proceso. La sintaxis para esta operación es:

```

if la condición 1 se cumple then
    realiza operación 1;
elsif la condición 2 se cumple then
    realiza operación 2;
else
    realiza operación 3;
end if;

```

la cual se interpreta como sigue: Si (**if**) la *condición 1* es verdadera, entonces (**then**) se ejecuta la *operación 1*, si no-si (**elsif**) se evalúa la *condición 2* y si es verdadera entonces (**then**) se ejecuta la *operación 2*, si no (**else**) se ejecuta la *operación 3*.

3.2.1 Comparador de magnitud de 4 bits

La forma de utilizar las declaraciones secuenciales se ilustra en el diseño de un comparador de dos números de 4 bits, figura 3.5a). En este caso el sistema tiene tres salidas que indican cuando uno de los números es mayor, igual

o menor que el otro. La figura 3.5b) representa el mismo comparador de manera simplificada utilizando la notación vectorial.

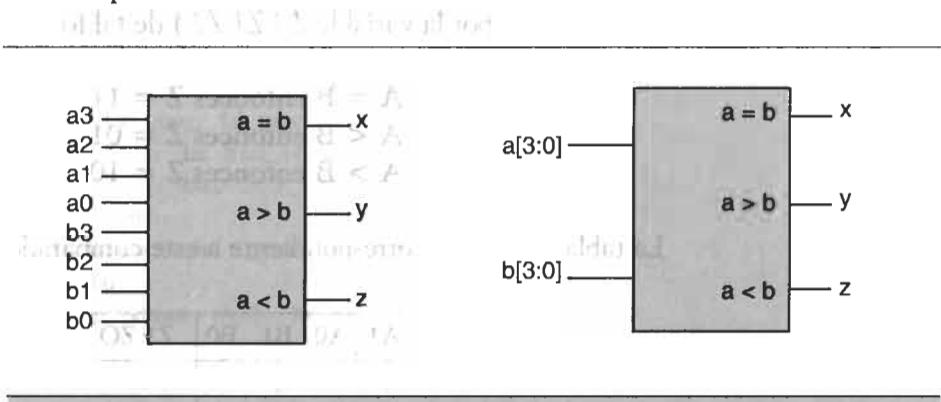


Figura 3.5 a) Comparador de 4 bits. b) Comparador expresado con vectores de 4 bits.

En el listado 3.5 se observa el algoritmo en VHDL que describe el funcionamiento del comparador. En la línea 9 se muestra la lista sensitiva (*a* y *b*) del proceso (*process*). En las líneas 10 a 17 el proceso se desenvuelve mediante el análisis de las variables de la lista sensitiva. Sin mucho esfuerzo puede verse que si *a* = *b*, entonces *x* toma el valor de 1, de forma similar se intuye el comportamiento para *a* > *b* y *a* < *b*, incluyendo la declaración *elsif*.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity comp4 is port(
4   a,b:  in std_logic_vector(3 downto 0);
5   x,y,z: out std_logic);
6 end comp4;
7 architecture arq_comp4 of comp4 is
8 begin
9   process (a,b)
10  begin
11    if (a = b) then
12      x <= '1';
13    elsif (a > b) then
14      y <= '1';
15    else
16      z <= '1';
17    end if;
18  end process;
19 end arq_comp4;

```

Listado 3.5 Descripción del comparador de 4 bits utilizando el estilo funcional.

Ejemplo 3.4

Diseñe un comparador de dos números A y B, cada número formado por dos bits (A1 A0) y (B1 B0) la salida del comparador también es de dos bits y está representada por la variable Z (Z1 Z0) de tal forma que si:

$$A = B \text{ entonces } Z = 11$$

$$A < B \text{ entonces } Z = 01$$

$$A > B \text{ entonces } Z = 10$$

La tabla de verdad correspondiente a este comparador es la siguiente.

A1	A0	B1	B0	Z1	Z0
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

Las ecuaciones lógicas reducidas mediante un mapa de Karnaugh para Z1 y Z0 son las siguientes:

$$Z1 = A0 A1 + \bar{B}0 \bar{B}1 + A1 \bar{B}0 + A0 \bar{B}0 + A0 \bar{B}1$$

$$Z0 = \bar{A}0 \bar{A}1 + B0 B1 + \bar{A}0 B1 + \bar{A}0 B0 + \bar{A}1 B0$$

El circuito representativo de este comparador es el siguiente, figura E3.4.

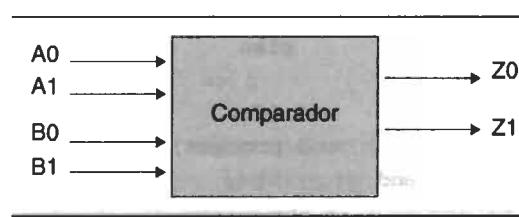


Figura E3.4 Descripción funcional de un comparador de igualdad de dos bits.

La programación de este comparador se muestra en el siguiente listado.

```

library ieee;
use ieee.std_logic_1164.all;
entity comp is port (
    A,B: in std_logic_vector(1 downto 0);
    Z: out std_logic_vector (1 downto 0));
end comp;
architecture a_comp of comp is
begin
    process (A,B) begin
        if A = B then
            Z <= "11";
        elsif A < B then
            Z <= "01";
        else
            Z <= "10";
        end if;
    end process;
end a_comp;

```

Operadores relacionales. Los operadores relacionales se usan para evaluar la igualdad, desigualdad o la magnitud en una expresión. Los operadores de igualdad y desigualdad ($=$ y $/=$) se definen en todos los tipos de datos. Los operadores de magnitud ($<$, $<=$, $>$ y $>=$) lo están sólo dentro del tipo escalar. En ambos casos debe considerarse que el tamaño de los vectores en que se aplicarán dichos operadores debe ser igual. En la tabla 3.1 se muestran estos operadores y su significado.

Operador	Significado
$=$	Igual
$/=$	Diferente
$<$	Menor
$<=$	Menor o igual
$>$	Mayor
$>=$	Mayor o igual

Tabla 3.1 Operadores relacionales.

3.2.2 Buffers tri-estado

Los registros de tres estados (buffers tri-estado) tienen diversas aplicaciones, ya sea como salidas de sistemas (**modo buffer**) o como parte integral de un circuito. En VHDL estos dispositivos son definidos a través de los valores que manejan (0,1 y alta impedancia 'Z'). En la figura 3.6 se observa el diagrama correspondiente a este circuito, y en el listado 3.6 el código que describe su funcionamiento.

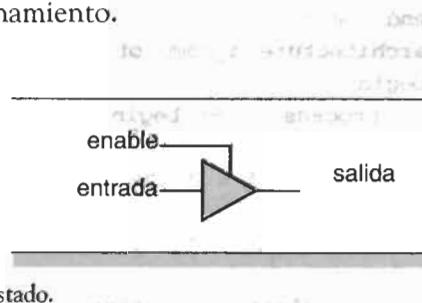


Figura 3.6 Buffer tri-estado.

```

library ieee;
use ieee.std_logic_1164.all;
entity tri_est is port(
    enable, entrada: in std_logic;
    salida: out std_logic);
end tri_est;
architecture arq_buffer of tri_est is
begin
    process (enable, entrada) begin
        if enable = '0' then
            salida <= 'Z';
        else
            salida <= entrada;
        end if;
    end process;
end arq_buffer;

```

Listado 3.6 Descripción mediante valores de alta impedancia.

El listado anterior se basa en un proceso, el cual se utiliza para describir los valores que tomará la salida del registro (buffer). En este proceso se indica que cuando se confirma el habilitador del circuito (*enable*), el valor que se

encuentra a la entrada del circuito se asigna a la salida; si por el contrario no se confirma *enable*, la salida del buffer tomará un valor de alta impedancia (Z). El tipo std_logic soporta este valor —al igual que 0 y 1—. A esto se debe que en el diseño se prefiera utilizar el estándar std_logic_1164 y no el tipo bit, ya que el primero es más versátil al proveer valores de *alta impedancia* y condiciones de *no importa* (-), los cuales no están considerados en el tipo bit.

3.2.3 Multiplexores

Los multiplexores se diseñan describiendo su comportamiento mediante la declaración **with-select-when** o ecuaciones booleanas.

En la figura 3.7a) se observa que el multiplexor dual tiene como entrada de datos las variables a, b, c y d, cada una de ellas representadas por dos bits (a1, a0), (b1, b0), etc., las líneas de selección (s) de dos bits (s1 y s0) y la línea de salida z (z1 y z0).

En la figura 3.7b) se muestra un diagrama simplificado que resalta la representación mediante vectores de bits.

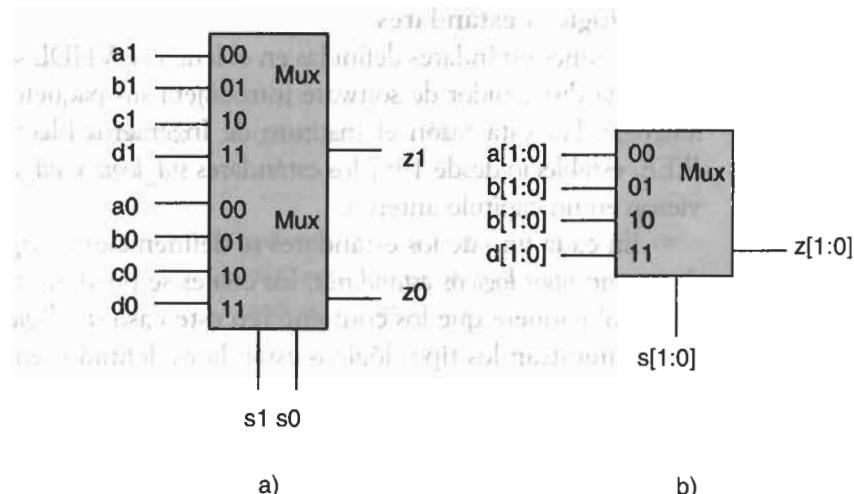


Figura 3.7 a) Multiplexor de 4 bits. **b)** Multiplexor con vectores.

En el listado 3.7 se muestra la descripción mediante **with-select-when** del multiplexor dual de 2×4 . En este caso la señal s determina cuál de las cuatro señales se asigna a la salida z. Los valores de s están dados como "00", "01" y "10"; el término **others** (otros) especifica cualquier combinación adicional que pudiera presentarse (que incluye el "11"), ya que esta variable se encuentra definida dentro del tipo std_logic_vector, el cual contiene nueve

valores posibles que la herramienta de síntesis² reconoce como **tipos lógicos estándares**.

```

library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
    a,b,c,d: in std_logic_vector(1 downto 0);
    s: in std_logic_vector(1 downto 0);
    z: out std_logic_vector(1 downto 0));
end mux;

architecture argmux4 of mux is
begin
with s select
    Z <=  a when "00",
            b when "01",
            c when "10",
            d when others;
end argmux4;

```

Listado 3.7 Multiplexor descrito con declaraciones with-select-when.

Tipos lógicos estándares

Las funciones estándares definidas en el lenguaje VHDL se crearon para evitar que cada distribuidor de software introdujera sus paquetes y tipos de datos al lenguaje. Por esta razón el Instituto de Ingenieros Eléctricos y Electrónicos, IEEE, estableció desde 1987 los estándares *std_logic* y *std_logic_vector*, que ya se vieron en un capítulo anterior.

En cada uno de los estándares se definen ciertos tipos de datos conocidos como *tipos lógicos estándares*, los cuales se pueden utilizar haciendo referencia al paquete que los contiene (en este caso *std_logic_1164*). En la tabla 3.2 se muestran los tipos lógicos estándares definidos en VHDL.

'U'	- Valor no inicializado
'X'	- Valor fuerte desconocido
'0'	- 0 Fuerte
'1'	- 1 Fuerte
'Z'	- Alta impedancia
'W'	- Valor débil desconocido
'L'	- 0 débil
'H'	- 1 débil
'.'	- No importa (don't care));

Tabla 3.2 Tipos lógicos **estándares definidos en VHDL**.

² La herramienta de síntesis es el software incluido en VHDL, que genera las ecuaciones de diseño de cualquier circuito. Esta herramienta permite implementar diseños sin el formato de ecuaciones booleanas que utilizan otros programas.

Como se puede apreciar, estos tipos de datos no tienen un significado evidente para el diseñador, pero sí lo tienen en la compilación del programa. Por ejemplo, el estándar no especifica alguna interpretación de L y H, debido a que la mayoría de las herramientas no los soportan. Los valores metalógicos³ ('U', 'W', 'X', 'Z') carecen de sentido en la síntesis, pero la herramienta los usa en la simulación del código. Como se mencionó en la sección anterior, el uso de 'Z' entraña un valor de alta impedancia.

3.2.4 Descripción de multiplexores mediante ecuaciones booleanas

En el listado 3.8 se muestra una solución con ecuaciones booleanas para el multiplexor dual de la figura 3.7.

```

library ieee;
use ieee.std_logic_1164.all;
entity mux is port(
    a,b,c,d: in std_logic_vector(1 downto 0);
    s:         in std_logic_vector(1 downto 0);
    z:         out std_logic_vector(1 downto 0));
end mux;

architecture arqmux of mux is
begin

    z(1) <=(a(1) and not(s(1)) and not(s(0))) or
              (b(1) and not(s(1)) and s(0)) or
              (c(1) and s(1) and not(s(0))) or
              (d(1) and s(1) and s(0));

    z(0) <=(a(0) and not(s(1)) and not(s(0))) or
              (b(0) and not(s(1)) and s(0)) or
              (c(0) and s(1) and not(s(0))) or
              (d(0) and s(1) and s(0));

end arqmux;

```

Listado 3.8 Multiplexor descrito con ecuaciones booleanas.

³ Un dato metalógico consiste en los valores definidos para el tipo std_logic, los cuales están contenidos en el paquete estándar IEEE 1164.

3.2.5 Sumadores

Diseño de un circuito medio sumador

Para describir el funcionamiento de los circuitos sumadores es importante recordar las reglas básicas de la adición.

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

En el caso de la suma $1 + 1 = 10$, el resultado “0” representa el valor de la suma, mientras que el “1” el valor del acarreo.

Para observar en detalle el funcionamiento de un circuito medio sumador, considere la suma de los números A y B mostrados en la tabla 3.3.

A	B	Suma	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	(1 + 1 = 10)

Tabla 3.3 Tabla de verdad de un circuito sumador.

La ecuación lógica que corresponde a la expresión $\text{Suma} = \bar{A}B + A\bar{B}$ es la función lógica or-exclusiva $A \oplus B$, mientras que la ecuación lógica del acarreo de salida es $\text{Cout} = AB$ que corresponde a la compuerta lógica *and*. La realización física de estas ecuaciones se muestra en la figura 3.8a, en ella se presenta el bloque lógico del medio sumador (MS) y la figura 3.8b representa su implantación mediante compuertas lógicas.

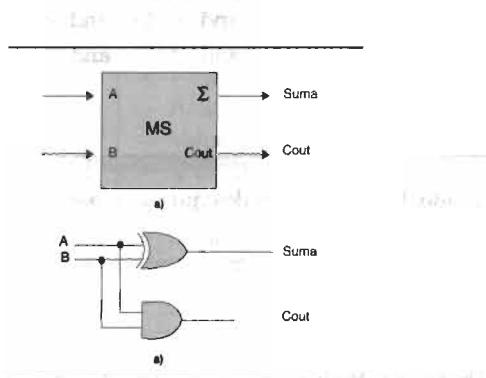


Figura 3.8 a) Diagrama a bloques de un medio sumador; b) Medio sumador lógico.

El programa en VHDL que representa este medio sumador se muestra en el listado 3.9.

```

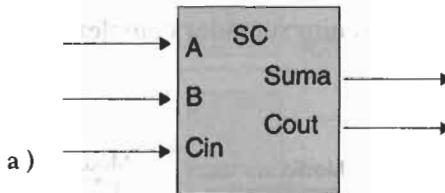
library ieee;
use ieee.std_logic_1164.all;
entity m_sum is port (
    A,B: in std_logic;
    SUMA, Cout: out std_logic);
end m_sum;
architecture am_sum of m_sum is
begin
    SUMA <= A XOR B;
    Cout <= A AND B;
end am_sum;

```

Listado 3.9 Código de un medio sumador.

Diseño de un sumador completo

Un sumador completo (SC) a diferencia del circuito medio sumador considera un acarreo de entrada (Cin) tal y como se muestra en la figura 3.9a, el comportamiento de este sumador se describe a través de su tabla de verdad.



A	B	Cin	Suma	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

b)

Listado 3.9 a) Sumador completo; b) Tabla de verdad del medio sumador.

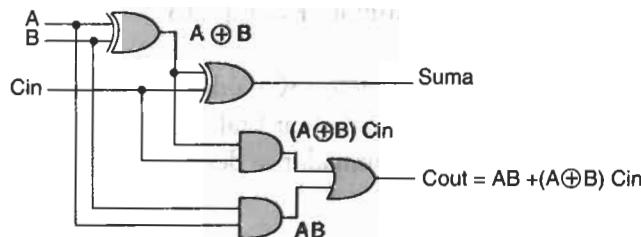
Las ecuaciones reducidas mediante un mapa de Karnaugh correspondientes a la salida Suma y Cout se muestran a continuación

$$\begin{aligned}\text{Suma} &= \bar{A} \bar{B} \text{Cin} + \bar{A} B \bar{\text{Cin}} + A \bar{B} \bar{\text{Cin}} + A B \text{Cin} \\ \text{Cout} &= A B + B \text{Cin} + A \text{Cin}\end{aligned}$$

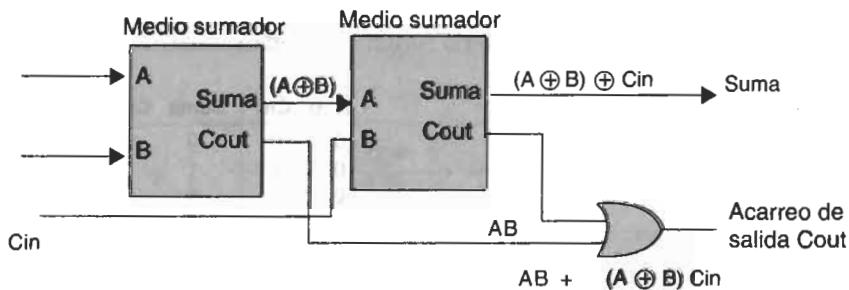
Si se manipulan las ecuaciones anteriores mediante álgebra booleana obtenemos que la función de Suma y Cout puede expresarse como:

$$\begin{aligned}\text{Suma} &= A \oplus B \oplus \text{Cin} \\ \text{Cout} &= AB + (A \oplus B) \text{Cin}\end{aligned}$$

La realización física del circuito se basa en la utilización de compuertas or-exclusiva como se muestra en la figura 3.10 a). Como puede observarse en la figura 3.10 b), dos circuitos medio sumadores pueden implementar un sumador completo.



a) Circuito sumador completo implementado por compuertas.



b) Circuito sumador completo implementado por medio sumadores

Figura 3.10 Circuitos sumadores.

La programación en VHDL del sumador completo se presenta en el listado 3.10.

```

library ieee;
use ieee.std_logic_1164.all;
entity sum is port(
    A,B,Cin: in std_logic;
    Suma, Cout: out std_logic);
end sum;
architecture a_sum of sum is
begin
    Suma <= A xor B xor Cin;
    Cout <= (A and B) or (A xor B) and Cin;
end a_sum;

```

Listado 3.10 Sumador Completo.

Sumador Paralelo de 4 bits

Según lo anterior, para realizar un sumador paralelo de 4 bits sólo se requiere conectar en cascada un circuito medio sumador y tres sumadores completos como se muestra en la figura 3.11

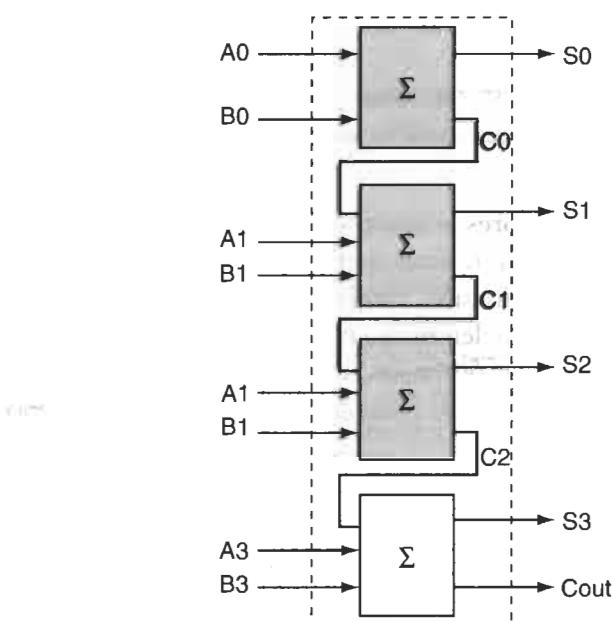


Figura 3.11 Sumador de 4 bits.

A su vez, éste es un buen ejemplo para reafirmar el manejo de señales (**signal**). En la figura 3.11 se observa cómo los acarreos de salida (C_0 , C_1 y C_2) se encuentran retroalimentados dentro del circuito, por lo que no tienen un pin externo asignado. En el listado 3.11 se ilustra la programación en VHDL.

Como puede apreciarse, el intervalo utilizado dentro de **signal** es (0 to 2), debido a que sólo se retroalimentan los acarreos C_0 , C_1 y C_2 . Por otro lado, con un poco de esfuerzo puede intuirse que cada uno de los diferentes bloques que forma el sumador se caracteriza usando compuertas **xor** (or exclusiva).

```

library ieee;
use ieee.std_logic_1164.all;
entity suma is port(
    A,B: in std_logic_vector (0 to 3);
    S: out std_logic_vector (0 to 3);
    Cout: out std_logic);
end suma;
architecture arqsuma of suma is
    signal C: std_logic_vector(0 to 2);
begin
    S(0) <= A(0) xor B(0);
    C(0) <= A(0) and B(0);
    S(1) <= (A(1) xor B(1)) xor C(0);
    C(1) <= (A(1) and B(1)) or (C(0) and (A(1)xor B(1)));
    S(2) <= (A(2) xor B(2)) xor C(1);
    C(2) <= (A(2) and B(2)) or (C(1) and (A(2)xor B(2)));
    S(3) <= (A(3) xor B(3)) xor C(2);
    Cout <= (A(3) and B(3)) or (C(2) and (A(3)xor B(3)));
end arqsuma;

```

Listado 3.11 Descripción de un sumador de 4 bits.

Operadores aritméticos. Como su nombre indica, los operadores aritméticos permiten realizar operaciones del tipo aritmético, como suma, resta, multiplicación, división, cambios de signo, valor absoluto y concatenación. Estos operadores suelen usarse en el diseño lógico para describir sumadores y restadores o en las operaciones de incremento y decremento de datos. En la tabla 3.4 se muestran los operadores aritméticos predefinidos en VHDL.

Operador	Descripción
+	Suma
-	Resta
/	División
*	Multiplicación
**	Potencia

Tabla 3.4 Operadores aritméticos utilizados en VHDL

Como ejemplo, analicemos el diseño de un circuito sumador de 4 bits que no considera el acarreo de salida (listado 3.12).

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity sum is port(
    A,B:  in std_logic_vector(0 to 3);
    Suma: out std_logic_vector(0 to 3));
end sum;
architecture arqsum of sum is
begin
    Suma <= A + B;
end arqsum;
```

Listado 3.12 Descripción de un sumador mediante `std_logic_vector` y el paquete `std_arith`.

En el listado 3.12 se introdujo el paquete `std_arith`, el cual — como ya se mencionó — se encuentra en la librería de trabajo `work`. Este paquete permite el uso de los operadores aritméticos con operaciones realizadas entre arreglos del tipo `std_logic_vector`; es decir, dado que dentro del paquete estándar (`std_logic_1164`) no están definidos los operadores aritméticos, es necesario usar el paquete `std_arith`.

El uso de los operadores existentes en VHDL, así como los tipos de datos para los cuales se encuentran definidos, se incluye en el apéndice B.

3.2.6 Decodificadores

La programación de circuitos decodificadores se basa en el uso de declaraciones que permiten establecer la relación entre **un código binario aplicado a las entradas del dispositivo y el nivel de salida obtenido**.

En esta sección se **presentan dos tipos de decodificadores**: el decodificador BCD decimal y el decodificador de BCD a siete segmentos, ya que consideramos que son dos de los más utilizados en el diseño lógico combinacional.

Decodificador BCD a decimal

En la figura 3.12 podemos observar la entidad de diseño correspondiente a un circuito decodificador, el cual convierte código BCD (código de binario a decimal) en uno de los diez dígitos decimales. Por lo general estos dispositivos

se conocen como decodificadores de 4 a 10 líneas, ya que contienen cuatro líneas de entrada y 10 de salida.

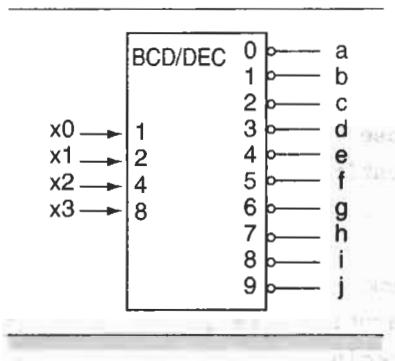


Figura 3.12 Decodificador de BCD a decimal.

El programa que describe el comportamiento de la entidad de la figura 3.12 se encuentra en el listado 3.13.

Como se puede apreciar, el código correspondiente a este circuito se basa en la ejecución de un proceso en que se establecen las condiciones que se evalúan para activar cada salida de acuerdo con el valor binario correspondiente. Para fines prácticos se asignó a cada salida un nombre a fin de facilitar su identificación.

A manera de ejemplo consideremos el valor de la entrada $x = 0010$, la cual corresponde al dígito decimal 2. Nótese cómo la condición que determina la asignación del valor evalúa primero la condición de x y si la confirma, asigna a la salida c el valor correspondiente al dígito decimal 2.

Por otro lado, se puede ver que al inicio del proceso se declararon todas las salidas con un valor inicial de '1', esto fue con el fin de asegurar que permanecieran desactivadas cuando no estuvieran en evaluación.

```

--Decodificador de BCD a decimal
library ieee;
use ieee.std_logic_1164.all;
entity deco is port (
    x: in std_logic_vector(3 downto 0);
    a,b,c,d,e,f,g,h,i,j: out std_logic);
end deco;
architecture arqdeco of deco is
begin
    process (x) begin
        a <= '1';
        b <= '1';
        c <= '1';
        d <= '1';

```

```

e <= '1';
f <= '1';
g <= '1';
h <= '1';
i <= '1';
j <= '1';

if      x = "0000" then
    a <= '0';
elsif x = "0001" then
    b <= '0';
elsif x = "0010" then
    c <= '0';
elsif x = "0011" then
    d <= '0';
elsif x = "0100" then
    e <= '0';
elsif x = "0101" then
    f <= '0';
elsif x = "0110" then
    g <= '0';
elsif x = "0111" then
    h <= '0';
elsif x = "1000" then
    i <= '0';
else
    j <= '0';
end if;
end process;
end arqdeco;

```

Listado 3.13 Descripción de un decodificador de BCD a decimal.

Decodificador de BCD a display de siete segmentos

En la figura 3.13a) se muestra un circuito decodificador, el cual acepta código BCD en sus entradas y proporciona salidas capaces de excitar un display de siete segmentos que indica el dígito decimal seleccionado. En la figura 3.13b) se observa la distribución de los segmentos dentro del display.

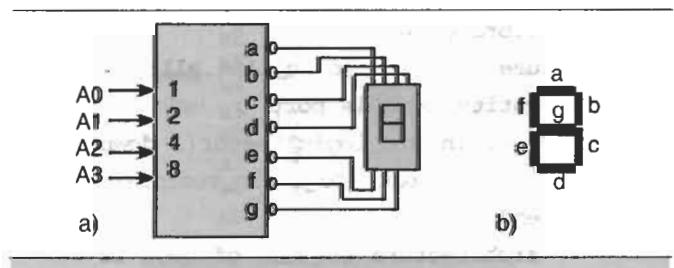


Figura 3.13 a) Decodificador BCD a siete segmentos. **b)** Configuración del display de siete segmentos.

Como se puede apreciar, la entidad del decodificador cuenta con una entrada llamada A, formada por cuatro bits (A_0, A_1, A_2, A_3), y siete salidas (a, b, c, d, e, f, g) activas en nivel bajo, las cuales corresponden a los segmentos del display. En la tabla 3.5 se indican los valores lógicos de salida correspondientes a cada segmento.

"A" Código BCD				Segmento del display "d"						
A0	A1	A2	A3	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	0
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0

Tabla 3.5 Valores lógicos correspondientes a cada segmento del display.

La función del programa cuyo código se exhibe en el listado 3.14 utiliza declaraciones secuenciales del tipo **case-when** que, como se puede apreciar, ejecutan un conjunto de instrucciones basadas en el valor que pueda tomar una señal. En nuestro ejemplo, se describe de qué manera se maneja el decodificador de acuerdo con el valor que toma la señal A. Para fines prácticos se declararon todas las salidas como un solo vector de bits (identificado como d); de esta forma se entiende que la salida *a* corresponde al valor *d*₀, la *b* al valor *d*₁, etc. Por otro lado, la palabra reservada **others**, como ya se indicó, define los valores metalógicos que puede tomar en la síntesis la salida *d*.

```

library ieee;
use ieee.std_logic_1164.all;
entity deco is port (
  A: in std_logic_vector(3 downto 0);
  d: out std_logic_vector(6 downto 0));
end deco;
architecture arqdeco of deco is
begin

```

```

process (A) begin
    case A is
        when "0000" => d <= "00000001";
        when "0001" => d <= "1001111";
        when "0010" => d <= "0010010";
        when "0011" => d <= "0000110";
        when "0100" => d <= "1001100";
        when "0101" => d <= "0100100";
        when "0110" => d <= "0100000";
        when "0111" => d <= "0001110";
        when "1000" => d <= "0000000";
        when "1001" => d <= "0000100";
        when others => d <= "1111111";
    end case;
end process;
end arqdeco;

```

Listado 3.14 Uso de declaraciones case-when.

En la instrucción **case-when** podemos observar el uso de *asignaciones dobles* ($=>$ $d \leq$), las cuales permiten que una señal adopte un determinado valor de acuerdo con el cumplimiento de una condición especificada. Por ejemplo, en nuestro código (listado 3.14) estas instrucciones se interpretan de la siguiente manera: cuando la señal A sea “0000”, asigna a la señal d el valor “0000001”; cuando A = “0001”, asigna a d el valor “1001111”, etc. Cabe mencionar que en la simulación del programa esta asignación se realiza simultánea.

3.2.7 Codificadores

En esta sección se presenta la forma de programar un circuito codificador, el cual como se observa en la figura 3.14, posee 10 entradas (cada una correspondiente a un dígito decimal) y cuatro salidas para el código binario de 4 bits BCD.

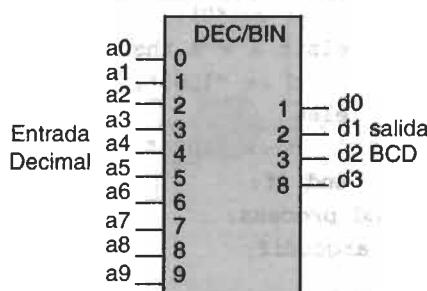


Figura 3.14 Codificador de decimal a BCD.

El programa que describe al circuito de la figura 3.14, se muestra en el listado 3.15. Como se puede ver, el puerto de entrada "a" se declara como un vector para indicar la numeración decimal del 0 al 9, mientras que la salida binaria se realiza a través del vector "d". Asimismo, observe que la programación se realizó utilizando declaraciones del tipo **if-then-elsif**.

- Codificador de decimal a BCD

```

library ieee;
use ieee.std_logic_1164.all;
entity codif is port (
    a: in integer range 0 to 9;
    d: out std_logic_vector(3 downto 0));
end codif;

architecture arqcodif of codif is
begin
    process (a)
    begin
        If a = 0 then
            d <= "0000";
        elsif a = 1 then
            d <= "0001";
        elsif a = 2 then
            d <= "0010";
        elsif a = 3 then
            d <= "0011";
        elsif a = 4 then
            d <= "0100";
        elsif a = 5 then
            d <= "0101";
        elsif a = 6 then
            d <= "0110";
        elsif a = 7 then
            d <= "0111";
        elsif a = 8 then
            d <= "1000";
        else
            d <= "1001";
        end if;
    end process;
end arqcodif;

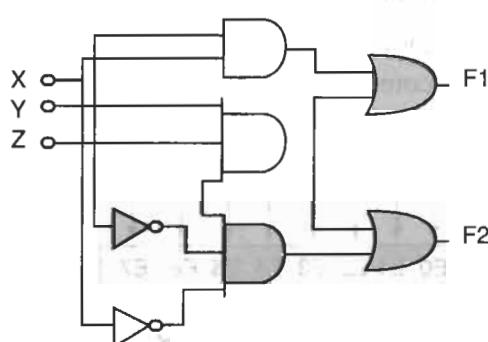
```

Listado 3.15 Diseño de un codificador de decimal a binario.

Ejercicios

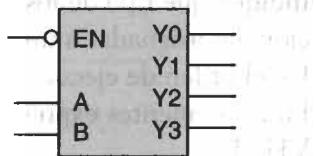
Declaraciones concurrentes

- 3.1 Mencione los tres tipos de declaraciones concurrentes en VHDL.
- 3.2 Indique qué tipo de instrucciones se usan en las declaraciones condicionales asignadas a una señal.
- 3.3 Dé el orden de ejecución de los operadores lógicos en VHDL.
- 3.4 En las siguientes expresiones proporcione la expresión equivalente en VHDL.
 - a) $X = (a + b)(c \text{ xor } d)$
 - b) $F = (a + c + d) + (a \cdot d \cdot c) \cdot (a + b)$
 - c) $Z = (w \cdot x \cdot y) + (x \text{ xnor } y)$
- 3.5 Mencione la principal diferencia entre las declaraciones secuenciales y las declaraciones concurrentes.
- 3.6 Mencione qué tipo de instrucciones utilizan las declaraciones secuenciales.
- 3.7 Indique qué instrucción se utiliza cuando se requieren más condiciones de evaluación en un proceso.
- 3.8 Mencione los operadores aritméticos que se utilizan en VHDL.
- 3.9 Indique en cuál librería y en qué paquete se encuentran los operadores de la pregunta 3.8.
- 3.10 Un circuito comparador de 3 bits recibe dos números de 3 bits $X = X_2, X_1, X_0$ y $Z = Z_2, Z_1, Z_0$. Diseñe un programa en VHDL que produzca una salida $F = 1$ si y sólo si $X < Z$.
- 3.11 Elabore un programa en VHDL que describa el funcionamiento del circuito mostrado en la figura siguiente.



Decodificadores

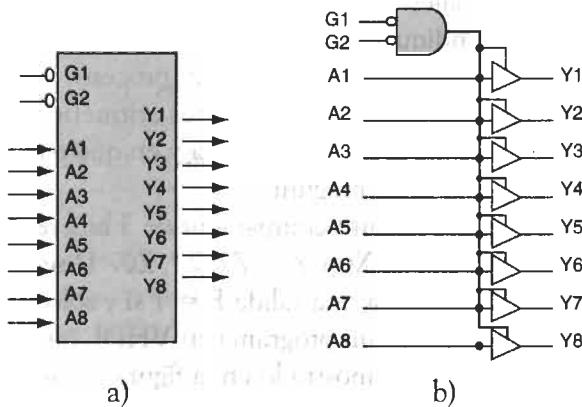
- 3.12 Se requiere un programa en VHDL de un circuito decodificador de 2 a 4, según se muestra en el siguiente diagrama. Utilice estructuras del tipo **if-then-elsif**.



Donde:

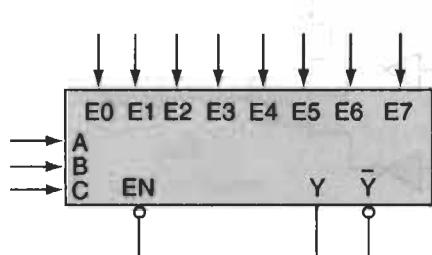
- EN = Entrada de habilitación del circuito (se activa en baja)
- A, B = Entradas del circuito
- $Y[0:3]$ = Salidas del circuito

- 3.13 Con base en el programa del listado 3.6 (Sec. 3.1.2.2) que describe un buffer triestado, elabore un programa de un circuito triestado octal como el de las figuras 3.13a) y 3.13b).



Multiplexores

- 3.14 Diseñe un programa de un multiplexor de 1 bit con ocho entradas como el que se ilustra en la figura siguiente. Implemente el algoritmo con base en la tabla de verdad adjunta.



a) Multiplexor

EN	C	B	A	Y	\bar{Y}
1	X	X	X	0	1
0	0	0	0	E_0	E_0'
0	0	0	1	E_1	E_1'
0	0	0	0	E_2	E_2'
0	0	0	1	E_3	E_3'
0	1	1	0	E_4	E_4'
0	1	1	1	E_5	E_5'
0	1	1	0	E_6	E_6'
0	1	1	1	E_7	E_7'

b) Tabla de verdad

3.15 Diseñe un programa en VHDL que produzca un decodificador binario de 2×4 como un circuito demultiplexor de cuatro salidas y un bit. Considere las siguientes especificaciones en el diseño:

- El circuito debe tener dos señales para seleccionar una salida.
- El circuito sólo tendrá una señal de entrada.
- Implemente una señal ENABLE para la habilitación del circuito.

Capítulo 4

Diseño lógico secuencial con VHDL

Introducción

Los circuitos digitales que hemos manejado con anterioridad han sido de tipo combinacional; es decir, son circuitos que dependen por completo de los valores que se encuentran en sus entradas, en determinado tiempo. Si bien un sistema secuencial puede tener también uno o más elementos combinacionales, la mayoría de los sistemas que se encuentran en la práctica incluyen elementos de memoria, los cuales requieren que el sistema se describa en términos de *lógica secuencial*.

En este capítulo se describen algunos de los circuitos secuenciales más utilizados en la práctica, como flip-flops, contadores, registros, etc., además se desarrollan ejercicios para aprender la programación de circuitos secuenciales en los que se integran los conceptos adquiridos en los capítulos anteriores.

4.1 Diseño lógico secuencial

Un sistema secuencial está formado por un circuito combinacional y un elemento de memoria encargado de almacenar de forma temporal la historia del sistema.

En esencia, la salida de un sistema secuencial no sólo depende del valor presente de las entradas, sino también de la historia del sistema, según se observa en la figura 4.1.

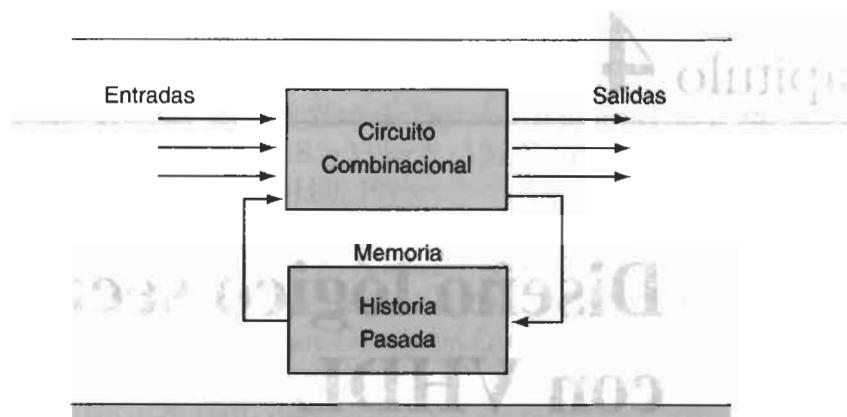


Figura 4.1 Estructura de un sistema secuencial.

Básicamente hay dos tipos de sistemas secuenciales: síncronos y asíncronos; el comportamiento de los primeros se encuentra sincronizado mediante el pulso de reloj del sistema, mientras que el funcionamiento de los sistemas asíncronos depende del orden y momento en el cual se aplican sus señales de entrada, por lo que no requieren un pulso de reloj para sincronizar sus acciones.

4.2 Flip-flops

El elemento de memoria utilizado indistintamente en el diseño de los sistemas síncronos o asíncronos se conoce como **flip-flop** o **celda binaria**.

La característica principal de un flip-flop es mantener o almacenar un bit de manera **indefinida hasta que** a través de un pulso o una señal cambie de estado. Los flip-flops más conocidos son los tipos SR, JK, T y D. En la figura 4.2 se presenta cada uno de estos elementos y la tabla de verdad que describe su comportamiento.

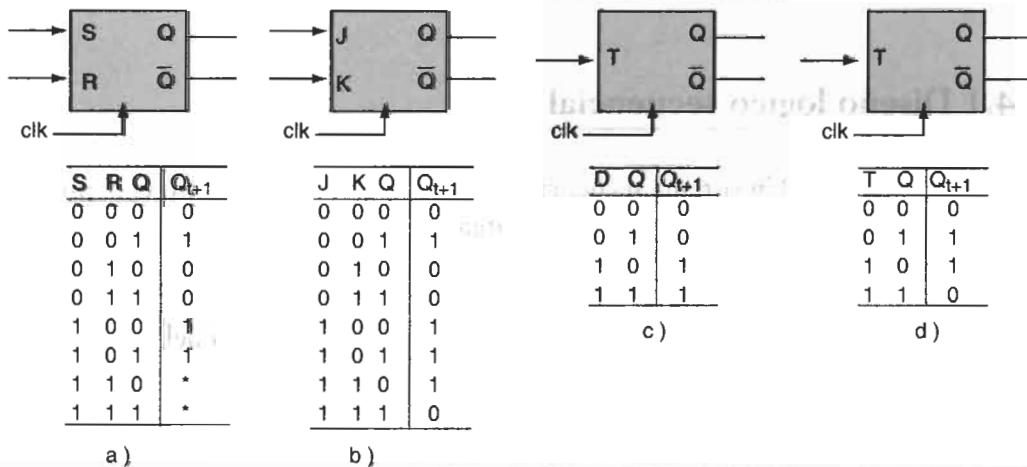


Figura 4.2 Flip-flops y tablas de verdad características.

Es importante recordar el significado de la notación Q y $Q_{(t+1)}$:

$$Q = \text{estado presente o actual}$$

$$Q_{t+1} = \text{estado futuro o siguiente}$$

Por ejemplo, consideremos la tabla de verdad que describe el funcionamiento del flip-flop tipo D, mostrado en la figura 4.2c) y que se muestra de nuevo en la figura 4.3a).

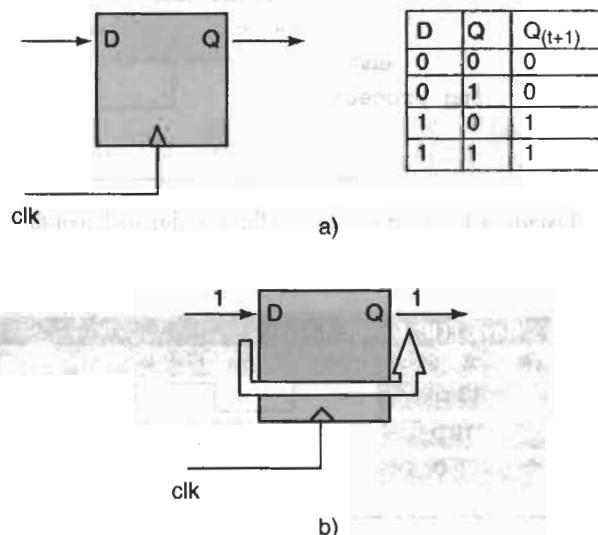


Figura 4.3 a) Diagrama y tabla de verdad del flip-flop D, b) Flujo de información en el dispositivo.

Cuando el valor de la entrada D es igual a 1, figura 4.3b), la salida Q_{t+1} adopta el valor de 1: $Q_{t+1} = 1$ siempre y cuando se genere un pulso de reloj. Es importante resaltar que el valor actual en la entrada D es transferido a la salida Q_{t+1} sin importar cuál sea el valor previo que haya tenido la salida Q en el estado presente.

En el diseño secuencial con VHDL las declaraciones If-then-else son las más utilizadas; por ejemplo, el programa del listado 4.1 usa estas declaraciones.

La ejecución del proceso es sensible a los cambios en clk (pulso de reloj); esto es, cuando clk cambia de valor de una transición de 0 a 1 ($clk = 1$), el valor de D se asigna a Q y se conserva hasta que se genera un nuevo pulso. A la inversa, si clk no presenta dicha transición, el valor de Q se mantiene igual. Esto puede observarse con claridad en la simulación del circuito, fig. 4.4.

```

library ieee;
use ieee.std_logic_1164.all;
entity ffd is port(
    D,clk: in std_logic;
    Q:      out std_logic);
end ffd;
architecture arq_ffd of ffd is
begin
    process (clk) begin
        if (clk'event and clk='1') then
            Q <= D;
        end if;
    end process;
end arq_ffd;

```

Listado 4.1 Descripción de un flip-flop disparado por flanko positivo.

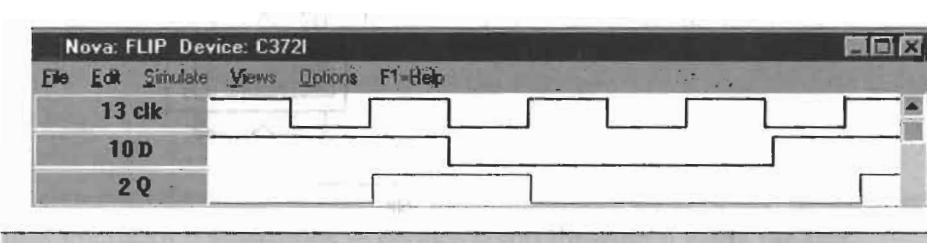


Figura 4.4 Simulación del flip-flop D.

Notemos que la salida Q toma el valor de la entrada D sólo cuando la transición del pulso de reloj es de 0 a 1 y se mantiene hasta que se ejecuta de nuevo el cambio de valor de la entrada clk.

Atributo 'event'

En el lenguaje VHDL los atributos sirven para definir características que se pueden asociar con cualquier tipo de datos, objeto o entidades. El atributo 'event'¹ (evento) se utiliza para describir un hecho u ocurrencia de una señal en particular.

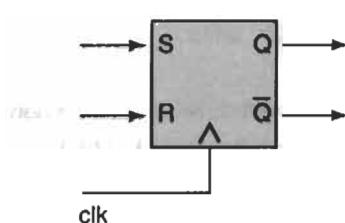
En el código del listado 4.1 podemos observar que la condición if clk'event es cierta sólo cuando ocurre un cambio de valor; es decir, un suceso (event) de la señal clk. Como se puede apreciar, la declaración (if-then) no maneja la condición else, debido a que el compilador mantiene el valor de Q hasta que no existe un cambio de valor en la señal clk.

¹ El apóstrofo ' indica que se trata de un atributo.

Para mayor información de los atributos predefinidos en VHDL consulte el apéndice C.

Ejemplo 4.1

Escriba un programa que describa el funcionamiento de un flip-flop SR con base en la siguiente tabla de verdad.



S	R	Q	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

Figura E4.1 Tabla de funcionamiento.

Solución

La tabla de verdad del flip-flop SR muestra que cuando la entrada S es igual a 1 y la entrada R es igual a 0, la salida Q_{t+1} toma valores lógicos de 1. Por otro lado, cuando $S = 0$ y $R=1$, la salida $Q_{t+1} = 0$; en el caso de que S y R sean ambas igual a 1 lógico, la salida Q_{t+1} queda indeterminada; es decir, no es posible precisar su valor y éste puede adoptar el 0 o 1 lógico.

Por último, cuando no existe cambio en las entradas S y R —es decir, son igual a 0—, el valor de Q_{t+1} mantiene su estado actual Q.

Con base en el análisis anterior, el programa en VHDL puede realizarse utilizando instrucciones condicionales y un nuevo tipo de datos: *valores no importa* (''), los cuales permiten adoptar un valor de 0 o 1 lógico de manera indistinta (Listado 4.2).



```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity ffsr is port (
4     S,R,clk: in std_logic;
5     Q, Qn: inout std_logic);
6 end ffsr;
7 architecture a_ffsr of ffsr is
8 begin
9 process (clk, S, R)
11 begin
10    if (clk'event and clk = '1') then
11        if (S = '0'and R = '1') then
12            Q <= '0';
13            Qn <= '1';
14        elsif (S = '1' and R = '0') then
15            Q <= '1';
16            Qn <= '0';
17        elsif (S = '0' and R = '0') then
18            Q <= Q;
19            Qn <= Qn;
20        else
21            Q <= '-';
22            Qn <= '-';
23        end if;
24    end if;
25 end process;
26 end a_ffsr;

```

Listado 4.2 Código VHDL de un registro de 8 bits.

4.3 Registros

En la figura 4.5 se presenta la estructura de un registro de 8 bits con entrada y salida de datos en paralelo. El diseño es muy similar al flip-flop anterior, la diferencia radica en la utilización de vectores de bits en lugar de un solo bit, como se observa en el listado 4.2.

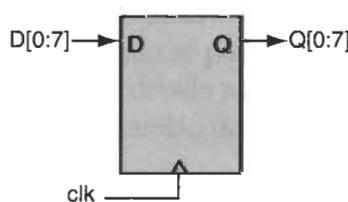


Figura 4.5 Registro paralelo de 8 bits.

```

library ieee;
use ieee.std_logic_1164.all;
entity reg is port(
    D:  in std_logic_vector(0 to 7);
    clk: in std_logic;
    Q:  out std_logic_vector(0 to 7));
end reg;
architecture arqreg of reg is
begin
    process (clk) begin
        if (clk'event and clk='1') then
            Q <= D;
        end if;
    end process;
end arqreg;

```

Listado 4.2 Código VHDL de un registro de 8 bits.

Ejemplo 4.2

Escriba un programa de un registro de 4 bits, como el que se muestra en la figura E4.2. Realice el diseño utilizando instrucciones **if-then-else** y procesos.

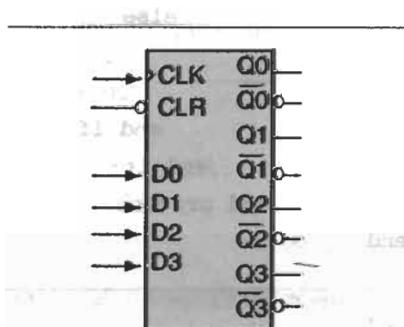


Figura E4.2

Solución

Como puede observarse en la tabla que describe el comportamiento del circuito, si CLR = 0, las salidas Q adoptan el valor de 0; pero si CLR = 1, toman el valor de las entradas D0, D1, D2 y D3.

CLR	D	Q	QN
0	*	0	1
1	Dn	Dn	Dn

El código del programa se observa en el siguiente listado.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity reg4 is port(
4     D: in std_logic_vector(3 downto 0);
5     CLK,CLR: in std_logic;
6     Q,Qn: inout std_logic_vector(3 downto 0));
7 end reg4;
8 architecture a_reg4 of reg4 is
9 begin
10    process (CLK,CLR) begin
11        if (CLK'event and CLK = '1') then
12            if (CLR = '1') then
13                Q <= D;
14                Qn <= not Q;
15            else
16                Q <= "0000";
17                Qn <= "1111";
18            end if;
19        end if;
20    end process;
21 end a_reg4;

```

Listado E4.2

Las variables sensitivas que determinan el comportamiento del circuito se encuentran dentro del proceso (CLR y CLK). Para transferir los datos de entrada hacia la salida es necesario, según se expuso, generar un pulso de reloj a través del atributo **event** (**CLK'event and CLK = '1'**)

4.4 Contadores

Los contadores son entidades muy utilizadas en el diseño lógico. La forma usual para describirlos en VHDL es mediante operaciones de incremento, decremento de datos o ambas.

Como ejemplo veamos la figura 4.6 que representa un contador ascendente de 4 bits, así como el diagrama de tiempos que muestra su funcionamiento.

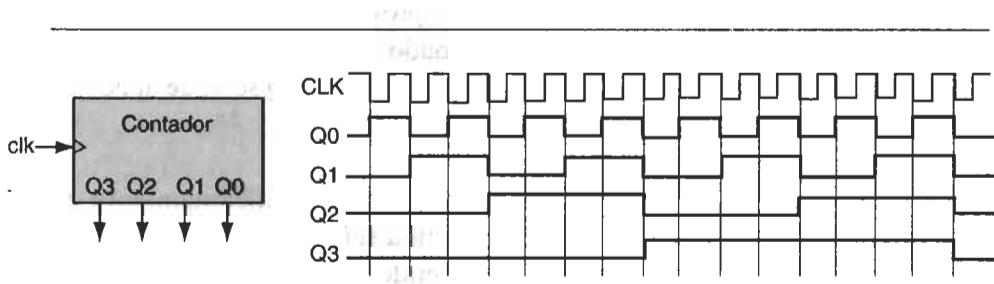


Figura 4.6 Contador binario de cuatro bits.

Cabe mencionar que la presentación del diagrama de tiempos de este circuito tiene la finalidad de ilustrar el procedimiento que se sigue en la programación, ya que puede observarse con claridad el incremento que presentan las salidas cuando se aplica un pulso de reloj a la entrada (listado 4.3).

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cont4 is port(
    clk: in std_logic;
    Q: inout std_logic_vector(3 downto 0));
end cont4;
architecture argcont of cont4 is
begin
    process (clk)
        begin
            If (clk'event and clk = '1') then
                Q <= Q + 1;
            end if;
        end process;
    end argcont;
```

Listado 4.3 Código que describe un contador de 4 bits.

Cuando requerimos la retroalimentación de una señal ($Q \leq Q+1$), ya sea dentro o fuera de la entidad, utilizamos el modo **inout** (Vea sección de Modos en Capítulo 2). En nuestro caso el puerto correspondiente a Q se maneja como tal, debido a que la señal se retroalimenta en cada pulso de reloj. Notemos también el uso del paquete **std_arith** que, como ya se mencionó, permite usar el operador **+** con el tipo **std_logic_vector**.

El funcionamiento del contador se define básicamente en un proceso, en el cual se llevan a cabo los eventos que determinan el comportamiento del circuito. Al igual que en los otros programas, una transición de 0 a 1 efectuada por el pulso de reloj provoca que se ejecute el proceso, lo cual incrementa en 1 el valor asignado a la variable Q . Cuando esta salida tiene el valor de 15 ("1111") y si el pulso de reloj se sigue aplicando, el programa empieza a contar nuevamente de 0.

Ejemplo 4.3

Elabore un programa que describa el funcionamiento de un contador de 4 bits. Realice en el diseño una señal de control (Up/Down) que determine el sentido del conteo: ascendente o descendente (Fig. E4.3).

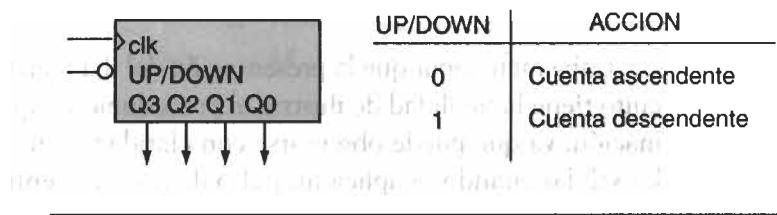


Figura E4.3 Contador binario de cuatro bits.

Solución

La señal de control Up/Down permite definir si el conteo se realiza en sentido ascendente o descendente. En nuestro caso, un cero aplicado a esta señal determina una cuenta ascendente: del 0 al 15. De esta forma, el funcionamiento del circuito queda determinado por dos señales: el pulso de reloj (clk) y la señal Up/Down, como se ve en el siguiente programa.

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity contador is port (
    clk: in std_logic;
    UP: in std_logic;
    Q: inout std_logic_vector(3 downto 0));
end contador;
architecture a_contador of contador is
begin
    process (UP, clk) begin
        if (clk'event and clk = '1') then
            if (UP = '0') then
                Q <= Q+1;
            else
                Q <= Q-1;
            end if;
        end if;
    end process;
end a_contador;

```

Listado E4.3.

Contador con reset y carga en paralelo (load)

La entidad de diseño que aparece en la figura 4.7a) es un ejemplo de un circuito contador síncrono de 4 bits. Este contador tiene varias características adicionales respecto al anterior. Su funcionamiento se encuentra predeterminado por los valores que se hallan en la tabla de la figura 4.7b). Como se puede observar, este contador posee las entradas de control Enp y Load y según el valor lógico que tengan en sus terminales realizarán cualesquiera de las operaciones reseñadas en la tabla. Note que la señal de Reset se activa en nivel alto de forma asíncrona; por último, el circuito tiene cuatro entradas en paralelo declaradas como P3, P2, P1 y P0.

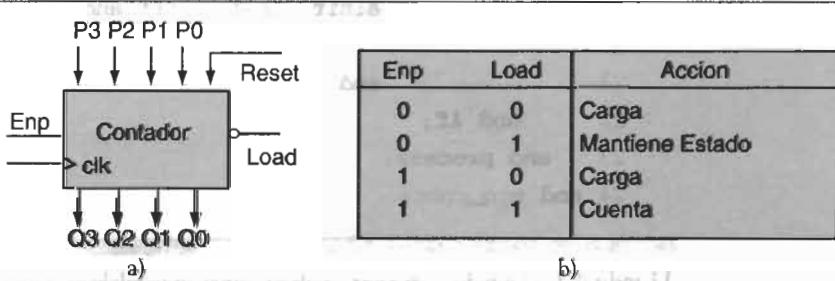


Figura 4.7 a) Contador binario de cuatro bits. b) Tabla de funcionamiento.

El propósito de describir este ejemplo radica en subrayar el uso de varias condiciones, de tal manera que no sea posible evaluar una instrucción si la(s) condición(es) predeterminada(s) no se cumple(n). Por ejemplo, observemos el listado 4.4, donde la línea 11 presenta la lista sensitiva de las variables que intervienen en el proceso (clk, ENP, LOAD, RESET). Nótese que no importa el orden en que se declaran. En la línea 12 se indica que si RESET = 1, las salidas Q toman el valor de 0; si no es igual a 1, se pueden evaluar las siguientes condiciones (es importante resaltar que la primera condición debe ser RESET). En la línea 15 se observa el proceso de habilitar una carga en paralelo, por lo que si LOAD = 0 y ENP es una condición de no importa ('-) que puede tomar el valor de 0 o 1 (según la tabla 4.7b), las salidas Q adoptarán el valor que se halle en las entradas P (P3, P2, P1, P0). Este ejemplo muestra de forma didáctica el uso de la declaración **elsif** (si no-si).

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use work.std_arith.all;
4 entity cont is port(
5     P: in std_logic_vector(3 downto 0);
6     clk,LOAD,ENP,RESET: in std_logic;
7     Q: inout std_logic_vector(3 downto 0));
8 end cont;
9 architecture arq_cont of cont is
10 begin
11     process (clk, RESET,LOAD, ENP) begin
12         if (RESET = '1') then
13             Q <= "0000";
14         elsif (clk'event and clk = '1') then
15             if (LOAD = '0' and ENP = '-') then
16                 Q <= P;
17             elsif (LOAD = '1' and ENP = '0') then
18                 Q <= Q;
19             elsif (LOAD = '1' and ENP = '1') then
20                 Q <= Q + 1;
21         end if;
22     end if;
23 end process;
24 end arq_cont;
```

Listado 4.4 Contador con reset, enable y carga en paralelo.

4.5 Diseño de sistemas secuenciales síncronos

Como ya se mencionó, la estructura de los sistemas secuenciales síncronos basa su funcionamiento en los elementos de memoria conocidos como flip-flops. La palabra **síncronía** se refiere a que cada uno de estos elementos de memoria que interactúan en un sistema se encuentran conectados a la misma señal de reloj, de forma tal que sólo se producirá un cambio de estado en el sistema cuando ocurra un flanco de disparo o un pulso en la señal de reloj.

Existe una división en el diseño de los sistemas secuenciales que se refiere al momento en que se producirá la salida del sistema.

- En la estructura de Mealy (Fig. 4.8) las señales de salida dependen tanto del estado en que se encuentra el sistema, como de la entrada que se aplica en determinado momento.
- En la estructura de Moore (Fig. 4.9) la señal de salida sólo depende del estado en que se encuentra.

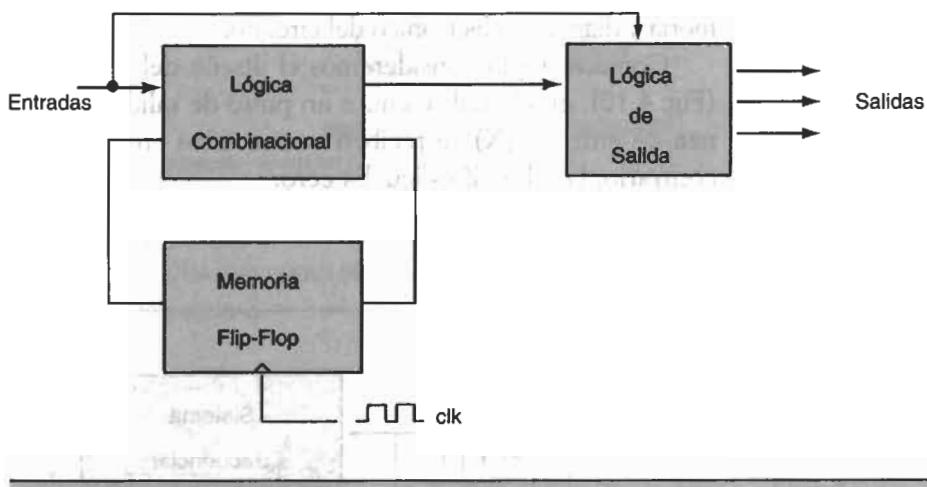


Figura 4.8 Arquitectura secuencial de Mealy.

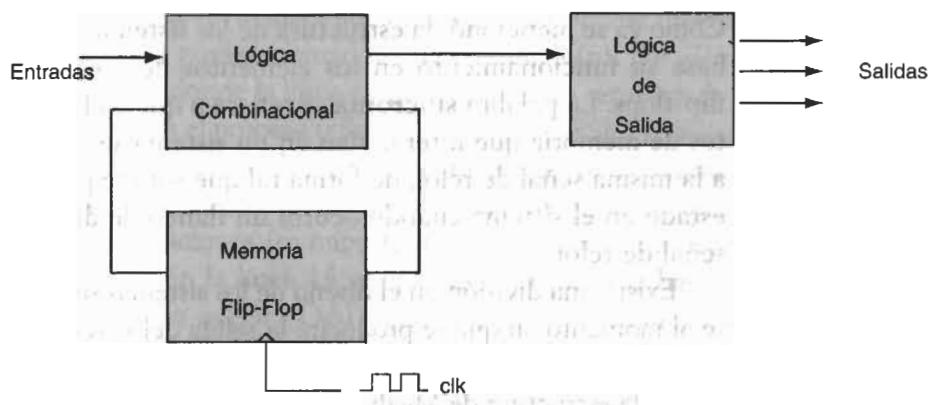


Figura 4.9 Arquitectura secuencial de Moore.

Un sistema secuencial se desarrolla a través de una serie de pasos generalizados que comprenden el enunciado del problema, diagrama de estados, tabla de estados, asignación de estados, ecuaciones de entrada a los elementos de memoria y diagrama electrónico del circuito.

Como ejemplo consideremos el diseño del siguiente sistema secuencial (Fig. 4.10), en el cual se emite un pulso de salida Z ($Z=1$) cuando en la línea de entrada (X) se reciben cuatro unos en forma consecutiva; en caso contrario, la salida Z es igual a cero.

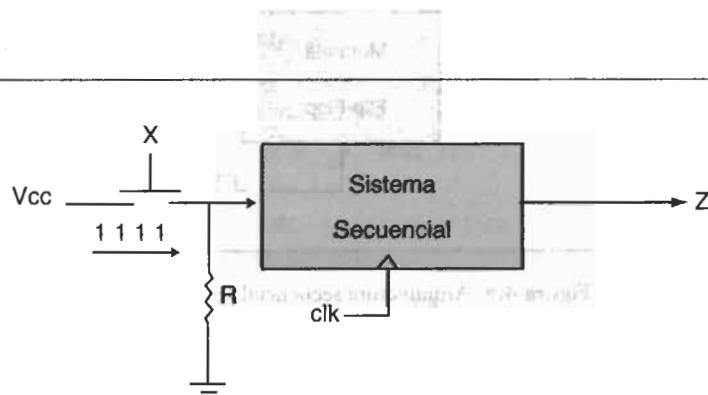


Figura 4.10 Detector de secuencia.

Diagramas de estado

El uso de diagramas de estados en la lógica programable facilita de manera significativa la descripción de un diseño secuencial, ya que no es necesario seguir la metodología tradicional de diseño. En VHDL se puede utilizar un modelo funcional en que sólo se indica la transición que siguen los estados y las condiciones que controlarán el proceso.

De acuerdo con nuestro ejemplo (Fig. 4.12), vemos que el sistema secuencial se puede representar por medio del diagrama de estados de la figura 4.12a): arquitectura Mealy. En este diagrama se advierte que el sistema cuenta con una señal de entrada denominada X y una señal de salida Z . En la figura 4.12b) se muestra la tabla de estados que describe el comportamiento del circuito.

Cuando se está en el estado d_0 y la señal de entrada X es igual a uno, se avanza al estado d_1 y la salida Z durante esta transición es igual a cero; en caso contrario, cuando la entrada X es igual a cero, el circuito se mantiene en el estado d_0 y la salida también es cero (Fig. 4.11).

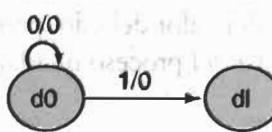


Figura 4.11 Transición de un estado a otro.

Con un poco de lógica se puede intuir el comportamiento del diagrama de estados. Observe como sólo la secuencia de cuatro unos consecutivos provoca que la salida $Z = 1$.

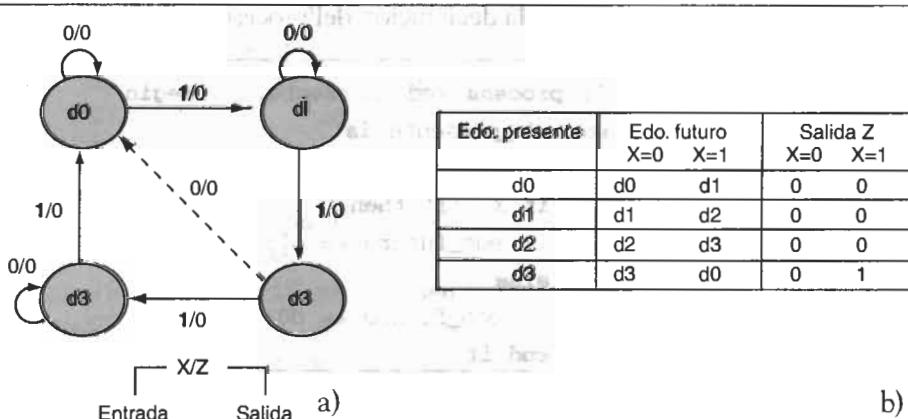


Figura 4.12 a) Diagrama de Estados. b) Tabla de estados.

Este diagrama se puede codificar con facilidad mediante una descripción de alto nivel en VHDL. Esta descripción supone el uso de declaraciones **case-when** las cuales determinan, en un caso particular, el valor que tomará el siguiente estado. Por otro lado, la transición entre estados se realiza por medio de declaraciones **if-then-else**, de tal forma que éstas se encargan de establecer la lógica que seguirá el programa para realizar la asignación del estado.

Como primer paso en nuestro diseño, consideremos los estados d0, d1, d2 y d3. Para poder representarlos en código VHDL, hay que definirlos dentro de un tipo de datos enumerado² (apéndice C) mediante la declaración **type**. Observemos la forma en que se listan los identificadores de los estados, así como las señales utilizadas para el estado actual (**edo_presente**) y siguiente (**edo_futuro**):

*definición
estados*

```

type estados is (d0, d1, d2, d3);
signal edo_presente, edo_futuro : estados;
  
```

El siguiente paso consiste en la declaración del proceso que definirá el comportamiento del sistema. En éste debe considerarse que el **edo_futuro** depende del valor del **edo_presente** y de la entrada X. De esta manera la lista sensitiva del proceso quedaría de la siguiente forma:

```
procesol: process (edo_presente, X)
```

Dentro del proceso se describe la transición del **edo_presente** al **edo_futuro**. Primero se inicia con la declaración **case** que especifica el primer estado que se va a evaluar —en nuestro caso consideremos que el análisis comienza en el estado d0 (**when d0**), donde la salida Z siempre es cero sin importar el valor de X. Si la entrada X es igual a 1 el estado futuro es d1; en caso contrario, es d0.

De este modo, la declaración del proceso quedaría de la siguiente manera:

```

procesol: process (edo_presente, X) begin
    case edo_presente is
        when d0 => Z<= '0';
        if X = '1' then
            edo_futuro <= d1;
        else
            edo_futuro <= d0;
        end if;
    end case;
end process procesol;
  
```

² Se llaman tipos enumerados porque en ellos se agrupan o enumeran elementos que pertenecen a un mismo género.

Nótese que en cada estado debe indicarse el valor de la salida ($Z \leq 0$) después de la condición **when**, siempre y cuando la variable Z no cambie de valor.

En el listado 4.5 se muestra la definición completa del código explicado. Como podemos observar, en el programa se utilizan dos procesos. En el primero, proceso1 (línea 11) se describe la transición que sufren los estados y las condiciones necesarias que determinan dicha transición. En el segundo, proceso2 (línea 42) se lleva a cabo de manera síncrona la asignación del estado futuro al estado presente, de suerte que cuando se aplica un pulso de reloj, el proceso se ejecuta.

En la línea 31 se describe la forma de programar la salida Z en el estado d3 cuando ésta obtiene el valor de 0 o 1, según el valor de la entrada X.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity diagrama is port(
4   clk,x:  in std_logic;
5   z:  out std_logic);
6 end diagrama;
7 architecture arq_diagrama of diagrama is
8 type estados is (d0, d1, d2, d3);
9 signal edo_presente, edo_futuro: estados;
10 begin
11   proceso1: process (edo_presente, x) begin
12     case edo_presente is
13       when d0 => z <= '0';
14         if x ='1' then
15           edo_futuro <= d1;
16         else
17           edo_futuro <= d0;
18         end if;
19       when d1 => z <='0';
20         if x='1' then
21           edo_futuro <= d2;
22         else
23           edo_futuro <= d1;
24         end if;
25       when d2 => z <='0';
26         if x='1' then
27           edo_futuro <= d3;
28         else
29           edo_futuro <= d0;
30         end if;

```

```

31      when d3 =>
32          if x='1' then
33             edo_futuro <= d0;
34              z <='1';
35      else
36         edo_futuro <= d3;
37              z <= '0';
38      end if;
39  end case;
40 end process proceso1;
41
42 proceso2: process(clk) begin
43     if (clk'event and clk='1') then
44        edo_presente <= edo_futuro;
45     end if;
46 end process proceso2;
47 end arq_diagrama;

```

Listado 4.5 Diseño de un diagrama de estados.

Ejemplo 4.4

Se requiere programar el diagrama de estados de la figura E4.4.

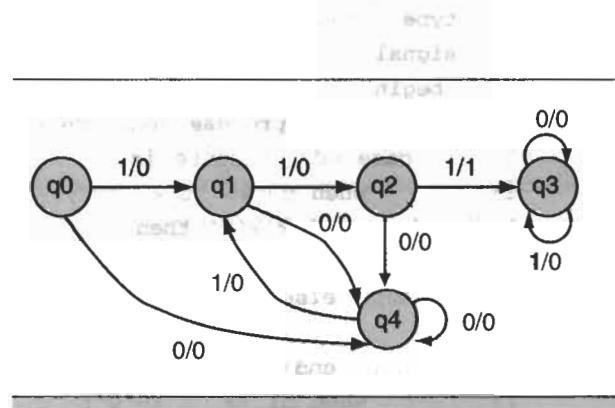


Figura E4.4 Diagrama de estados.

Solución

Por ejemplo, en la figura E4.4 notamos que el circuito pasa al estado q1 con el primer 1 de entrada y al estado q2 con el segundo. Las salidas asociadas con estas dos entradas son 0, según se señala. La tercera entrada consecutiva de 1 genera un 1 en la salida y hace que el circuito pase al estado q3. Una vez que se encuentra en q3, el circuito permanecerá en este estado, emitiendo salidas 0.

En lo que respecta a la programación en VHDL, este ejemplo sigue la misma metodología que el ejercicio anterior (Fig. 4.12), la cual consiste en usar estructuras **case – when** y tipo de datos enumerado, que en este caso contiene los cinco estados (q_0, q_1, q_2, q_3 y q_4) que componen el diagrama.

El listado correspondiente al programa se encuentra a continuación listado E4.4.

```

library ieee;
use ieee.std_logic_1164.all;
entity diag is port(
    clk,x: in std_logic;
    z: out std_logic);
end diag;
architecture arq_diag of diag is
type estados is (q0,q1,q2,q3,q4);
signal edo_pres,edo_fut: estados;
begin
procesol: process (edo_pres,x) begin
    case edo_pres is
        when q0 => z <= '0';
            if x = '0' then
                edo_fut <= q4;
            else
                edo_fut <= q1;
            end if;
        when q1 => z <= '0';
            if x = '0' then
                edo_fut <= q4;
            else
                edo_fut <= q2;
            end if;
        when q2 =>
            if x = '0' then
                edo_fut <= q4;
                z <= '0';
            else
                edo_fut <= q3;
                z <= '1';
            end if;
        when q3 => z <= '0';
            if x = '0' then

```

```
       edo_fut <= q3;
      else
       edo_fut <= q3;
      end if;
    when q4 => z <= '0';
      if x = '0' then
       edo_fut <= q4;
      else
       edo_fut <= q1;
      end if;
    end case;
  end process procesol;
proceso2: process (clk) begin
  if (clk'event and clk='1')then
    edo_pres <= edo_fut;
  end if;
  end process proceso2;
end arq_diag;
```

Listado E4.4.

Ejercicios

Flip-Flop

- 4.1 Realice un programa en VHDL que describa el funcionamiento del flip-flop tipo JK. Auxíliese para su descripción con la tabla característica del Flip-Flop.

J	K	Q	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Tabla característica del Flip-Flop JK

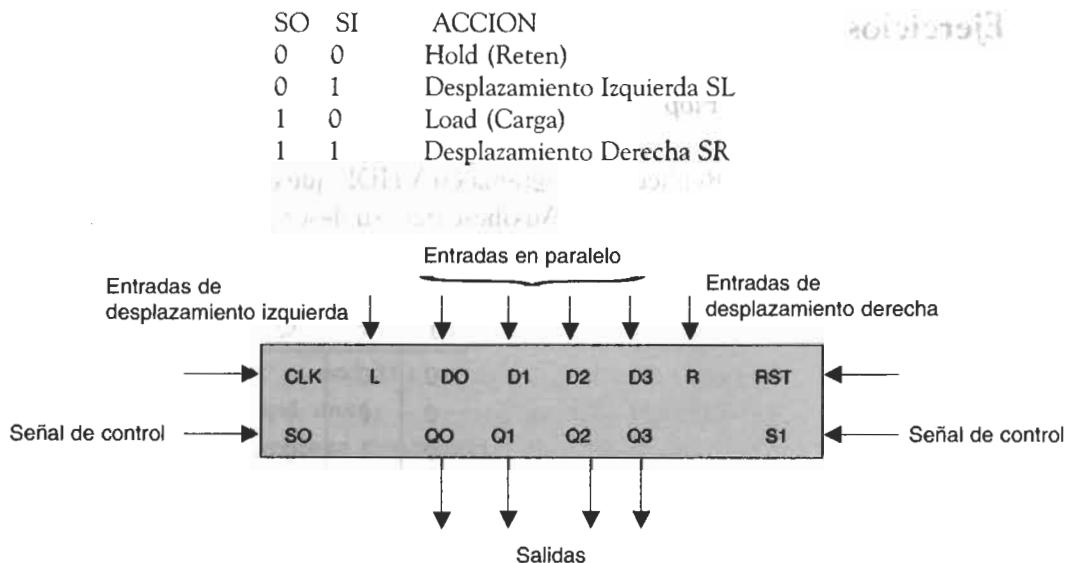
- 4.2. Realice un programa en VHDL que describa el funcionamiento del flip-flop tipo T. Auxíliese para su descripción con la tabla característica del Flip-Flop.

T	Q	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

Tabla característica del Flip-Flop T

Registros

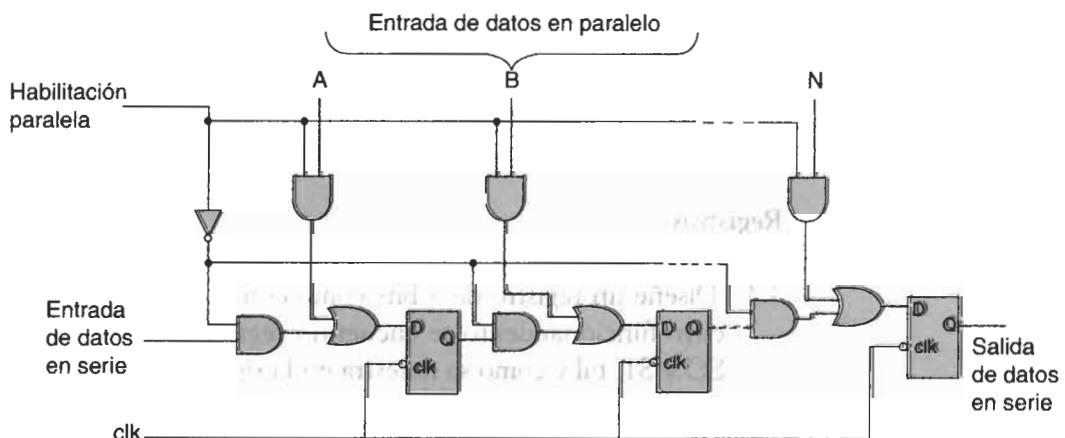
- 4.3. Diseñe un registro de 4 bits como el mostrado en la siguiente figura y cuyo funcionamiento se encuentra regulado por las señales de control SO y S1, tal y como se muestra en la siguiente tabla:



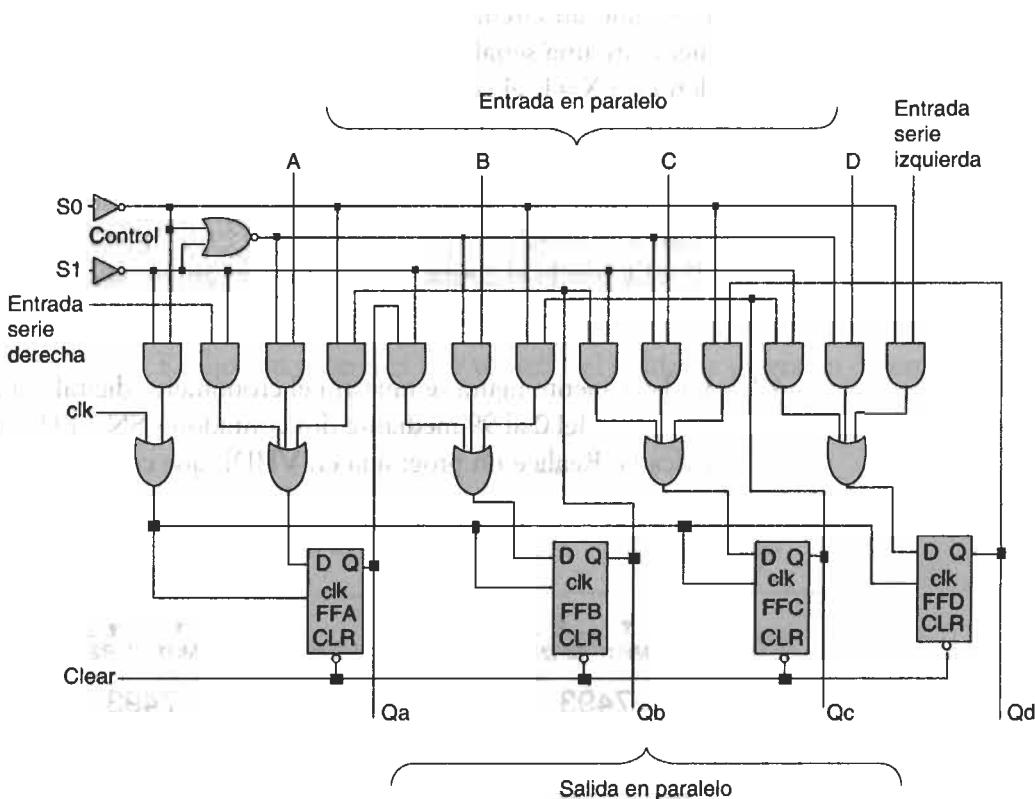
CLK = Reloj
 SO, S1 = Señales de control
 D0...D3 = Entradas de datos
 RST = Reset,
 Q0...Q3 = Salida de datos

L = Entrada serie desplazamiento a la izquierda
 R = Entrada serie desplazamiento a la derecha.

- 4.4 En la figura siguiente se muestra el esquema lógico de un registro de desplazamiento con entrada serie/ paralelo y salida serie. Realice un programa en VHDL que realice la misma función.

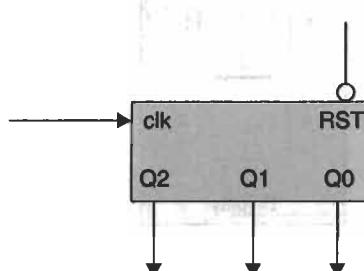


- 4.5 En la figura siguiente se muestra el esquema lógico de un registro de desplazamiento con salida en paralelo. La tabla de funcionamiento correspondiente se muestra a continuación. Realice un programa en VHDL que realice la función del circuito.



Contadores

- 4.6 Diseñe y programe un contador que realice la secuencia 0,1,3, 5 y repita el ciclo. El circuito debe contar con una señal de reset activo en bajo, que coloca las salidas Q en estado bajo.



- 4.7 Diseñe y programe un contador que realice la secuencia 0,1,2,3,4,5,6,7 y repita el ciclo. El circuito debe contar con una señal de reset activo en bajo, que coloca las salidas Q en estado bajo.

- 4.8 Programe un circuito contador ascendente / descendente del 0 al 3 mediante una señal de control X. Si $X=0$, el contador cuenta ascendente, si $X=1$, el contador cuenta descendente.

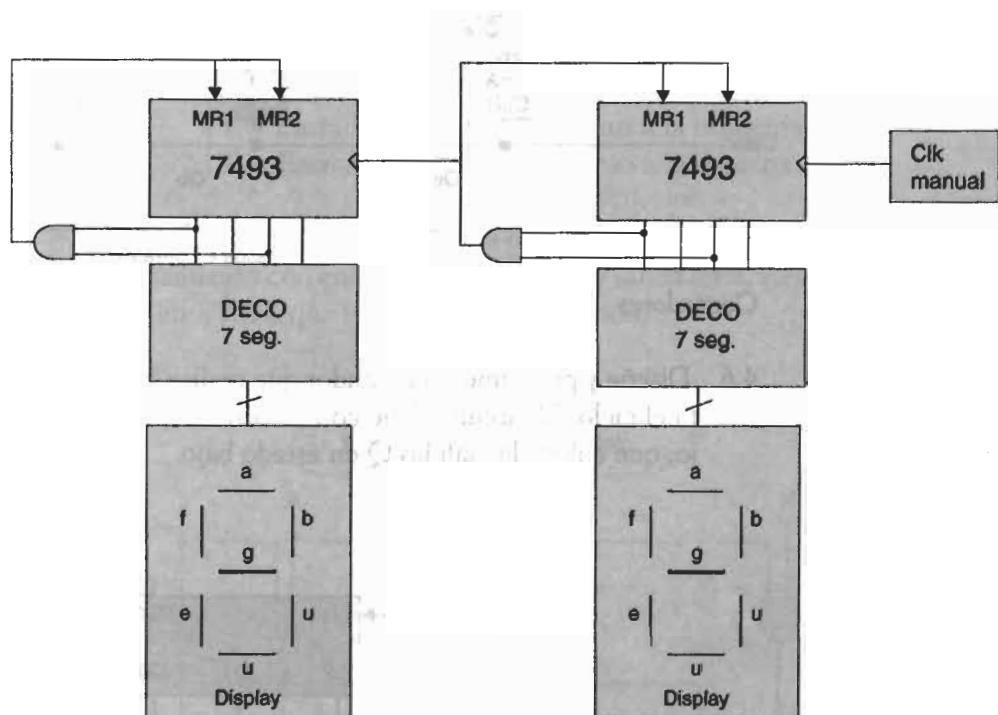
- 4.9 Programe un circuito contador ascendente / descendente del 0 al 15 mediante una señal de control X. Si $X=0$, el contador cuenta ascendente; si $X=1$, el contador cuenta descendente.

Existen dos señales de salida denominadas Z1 y Z2 que se activan de la siguiente forma:

Z1=1 Cuando el contador se encuentra en los estados pares, en caso contrario Z1=0.

Z2=1 Cuando el contador se encuentra en los estados impares, en caso contrario Z2=0

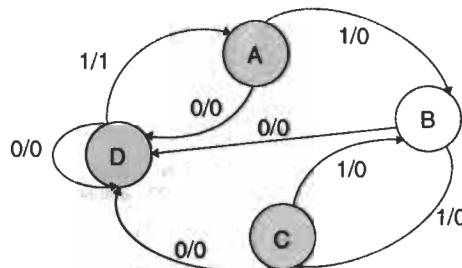
- 4.10 En la siguiente figura se muestra el cronómetro digital configurado para contar del 0 al 99 mediante dos contadores SN 7493 conectados en cascada. Realice un programa en VHDL que cuente del 0 al 99.



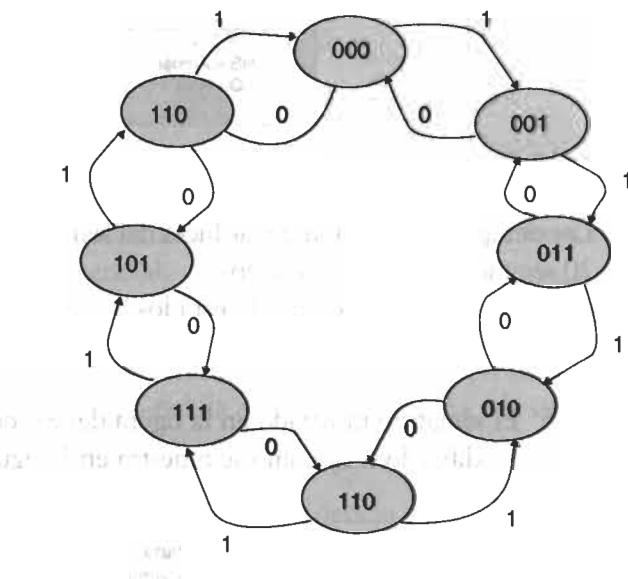
- 4.11 Realice un programa para un cronómetro que debe contar del 0 al 245 y repita el ciclo.

Sistemas secuenciales

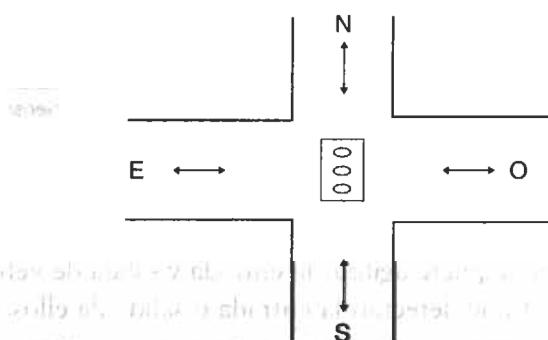
4.12 Realice un programa que resuelva el siguiente diagrama de estados:



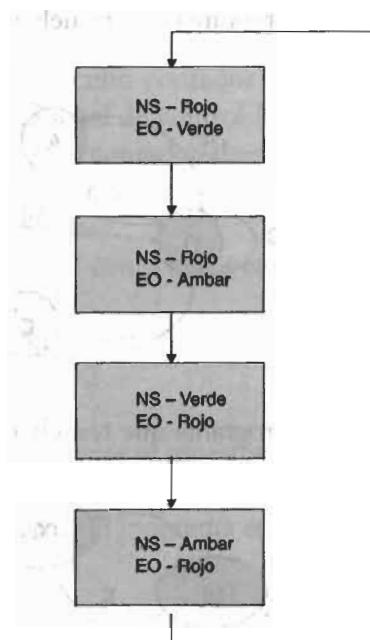
4.13 Realice un programa que resuelva el siguiente diagrama de estados:



4.14 En la figura se muestra el crucero de una avenida controlada a través de un semáforo.



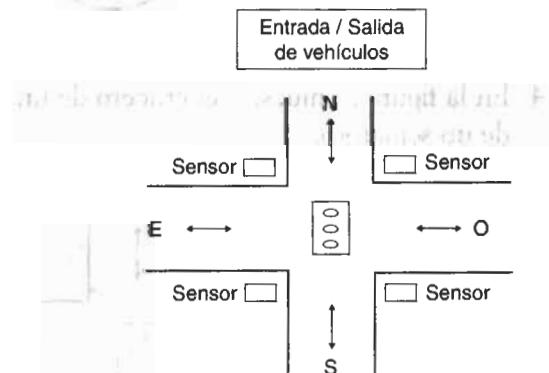
Los automóviles pueden circular en la dirección NS o EO mediante la siguiente secuencia:



Los tiempos de duración de las luces del semáforo son: rojo 20 segundos, verde 20 segundos, ámbar 5 segundos.

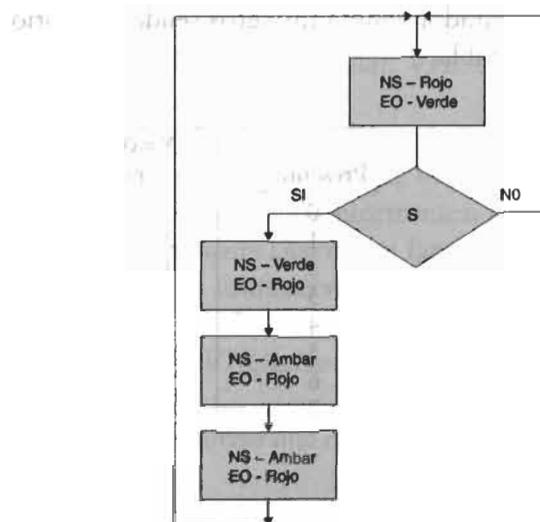
Proponga el diagrama de estados correspondiente y realice el programa del sistema secuencial.

- 4.15 El semáforo mostrado en la figura del ejercicio de la página 117 ha sido modificado tal y como se muestra en la siguiente figura.



Se requiere agilizar la entrada y salida de vehículos por medio de sensores (S) que detectan la entrada o salida de ellos. Generalmente el semáforo EO siempre se encuentra en verde y en contraparte el semáforo NS está en

rojo; cuando alguno de los sensores detecta la entrada o salida del vehículo envía una señal al sistema que gobierna el semáforo para que éste realice la secuencia siguiente:



Nuevamente la duración de encendido de los focos del semáforo son: roja 20 segundos, verde 20 segundos, ámbar 5 segundos.

Proponga el diagrama de estados correspondiente y realice el programa del sistema secuencial.

4.16 Programe un detector de secuencia que produce una salida $Z=1$ sólo cuando en la señal de entrada X se produce la siguiente secuencia $X = 110011$.

4.17 Programe un detector de secuencia cuya salida $Z = 1101111$ cuando aplicamos a la entrada la secuencia $X = 01101010$.

4.18 Realice el programa correspondiente al sistema secuencial especificado en la siguiente tabla.

Presente	$X=0$		$X=1$
			Futuro
A	B/0	E/0	
B	A/1	C/1	
C	B/0	C/1	
D	C/0	E/0	
E	D/1	A/0	

Considere la siguiente asignación de estados:

A 000 B 001 C 010 D 011 E 100

- 4.19 Realice el programa de un contador ascendente /descendente de números seudo aleatorios de 3 bits. El circuito tiene una entrada de control X .Cuando X=0 , el contador cuenta ascendente , si X=1 el contador genera números seudo aleatorios, tal y como se muestra en la tabla.

Presente	X=0 X=1	
	Futuro	
0	1	0
1	2	4
2	3	5
3	4	1
4	5	2
5	6	6
6	7	7
7	0	3

- 4.20 Programe el sistema secuencial correspondiente a una máquina deschadora de refrescos, el valor del refresco es de \$ 5.00 pesos y la máquina acepta monedas de \$5.00, \$10.00 y \$ 20.00. Cuando se introduce una moneda de diez o veinte pesos, la máquina debe dar cambio en monedas de \$5.00 pesos hasta completar el cambio correspondiente.

Para realizar este programa no considere el sistema detector de monedas ni el sistema de servicio encargado de dar el refresco.

Programe exclusivamente el sistema secuencial encargado de controlar la secuencia descrita anteriormente.

Capítulo 5

Integración de entidades en VHDL

Introducción

Hasta este momento hemos utilizado la programación en VHDL para diseñar entidades individuales (bloques lógicos mínimos), con el único objeto de familiarizarlo con los diversos estilos de diseño y programación o ambos, así como con el uso y aplicación de las palabras reservadas en VHDL.

Sin embargo, es obvio que esta herramienta de diseño no fue creada para este fin. Como vimos en el capítulo 1, existen varias razones para su utilización; pero quizás su verdadera fortaleza radica en que permite integrar "sistemas digitales" que contienen una gran cantidad de subsistemas electrónicos con el fin de minimizar el tamaño de la aplicación. En primera instancia, en un solo circuito integrado y si el problema es muy complejo, a través de una serie sucesiva de circuitos programables, sea que se llamen CPLD (dispositivo lógico programable complejo) o FPGA (arreglos de compuertas programables en campo).

5.1 Esquema básico de integración de entidades

La integración de entidades puede realizarse mediante el diseño individual de cada bloque lógico a través de varios procesos internos que posteriormente pueden unirse mediante un programa común. Otra posibilidad es observar y analizar de manera global todo el sistema evaluando su comportamiento sólo a través de sus entradas y salidas. En ambos casos el resultado es satisfactorio; más bien, nuestra tarea consiste en analizar las ventajas y desventajas que existen en ambas alternativas de solución. En el primer caso, el inconveniente principal es el número excesivo de terminales utilizadas en el dispositivo, debido a que al diseñar entidades individuales, se tendrían que declarar las terminales de entrada-salida de cada entidad (Fig 5.1a).

Apéndice E

Palabras reservadas en VHDL

A continuación se muestra una lista de las palabras reservadas en VHDL. Como se mencionó en el apéndice C, ninguna palabra reservada se puede usar como identificador de señales.

Abs	Exit	Not	Signal
Access	File	Null	Shared
After	For	Of	Sla
Alias	Function	On	Sll
All	Generate	Open	Sra
And	Generic	Or	Srl
Architecture	Group	Others	Subtype
Array	Guarded	Out	Then
Assert	If	Package	To
Atribute	Impure	Port	Transport
Begin	In	Postponed	Type
Block	Inertial	Procedure	Unaffected
Body	Inout	Process	Units
Buffer	Is	Pure	Until
Bus	Label	Range	Use
Case	Library	Record	Variable
Compoennt	Lindage	Register	Wait
Configutation	Literal	Reject	When
Constant	Loop	Rem	While
Disconnect	Map	Report	With
Downto	Mod	Return	Xnor
Else	Nand	Rol	Xor
Elsif	New	Ror	
End	Next	Select	
Entity	Nor	Severity	

Apéndice F

Operadores definidos en VHDL según su orden de precedencia

Operador	Descripción	Tipos de operandos	Resultado
**	potencia	Entero operador entero Real operador entero	Entero Real
Abs	Valor absoluto	Numérico	Ídem operando
not	negación	Bit, booleano, vectores de bits	Ídem operando
*	multiplicación	Entero operador entero Real op real Físico op real Físico op entero Entero op físico Real op físico	Entero Real Físico Físico Físico Físico
/	División	Entero op entero Real op real Físico op entero Físico op real Físico op físico	Entero Real Físico Físico Físico
Mod	módulo	Entero op entero	Entero
+	suma	Numérico op numérico	Ídem operandos
-	resta	Numérico op numérico	Ídem operandos
&	concatenación	Vector op vector Vector op elemento Elemento op vector Elemento op elemento	Vector Vector Vector Vector

Operador	Descripción	Tipos de operandos	Resultado
sll	Desp. Lógico izquierdo	Vectores de bits op entero	Vector de bits
srl	Despl. Lógico derecho	Vector de bits op entero	Vector de bits
sla	Despl. Arit. izquierdo	Vector de bits op entero	Vector de bits
rol	Rotación izquierda	Vector de bits op entero	Vector de bits
ror	Rotación derecha	Vector de bits op entero	Vector de bits
=	Igual que	No archivo op no archivo	Booleano
/=	Diferente que	No archivo op no archivo	Booleano
<	Menor que	No archivo op no archivo	Booleano
>	Mayor que	No archivo op no archivo	Booleano
<=	Menor o igual que	No archivo op no archivo	Booleano
>=	Mayor o igual que	No archivo op no archivo	Booleano
and	y lógica	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
or	o lógica	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
nand	y lógica negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
nor	o lógica negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
xor	or exclusiva	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos
xnor	or exclusiva negada	Bit,booleano, bit_vector op bit, booleano, bit_vector	Ídem operandos

Fuente de información: The IEEE Estándar VHDL Language Reference Manual, IEEE Std 1076-1987, 1988.

Índice analítico

- Archivos xxiv, 339-341
Arquitectura (*architecture*), 46-55
 descripción
 estructural, 53
 funcional, 47-49
 por flujo de datos, 49-52
 estructural, 53
Arreglo(s)
 (*array*), 298-299
 de compuertas programables, 3-4,15-18,23
 lógico genérico (*GAL*), 4,9-19
Asignaciones dobles, 87
Buffer tri-estado, 74
Carta ASM
 descripción, 157
 diseño mediante VHDL, 166-173
 en comparación con máquina de estado, 159
 estructura, 156
Circuito AMD 2909
 descripción, 199-200
 diseño y programación de componentes, 201-208
Codificadores, 87-88
Comparador de magnitud, 70-71
Compilación de un diseño, 316-319
Compiladores lógicos, 20
Componente (*component*), 53,201,206,262-264
Configuración (*configuration*, 37
Contadores, 101-104
 con reset y carga en paralelo, 103
Controladores, 153-154
 algoritmos de, 165
 diseño, 162-166
CPLD véase dispositivos lógicos programables
 complejos
 secuenciales, 46, 50, 69-75
Decodificadores, 83-87
 BCD a decimal, 83-85
 BCD a display de siete segmentos, 85-87
Diseño jerárquico, 53,197-208, 261-270
 Metodología de diseño, 198
Dispositivos lógicos programables, 2-8,13
Dispositivos lógicos programables
 complejos, 2-4,13-15,23
Ecuaciones booleanas, 51
Entidad (*entity*), 37-46
 declaración de, 40
 diseño utilizando vectores, 42
 diseño utilizando librerías y paquetes, 44
 integración de, 123-127
Eventos (*event*), 96, 100
Flip-flop, 94-98
FPGA véase arreglos de compuertas programables
Full-custom, 3
GAL véase arreglo lógico genérico
Galaxy, 312-315
Identificadores, 42, 333
ISR programador, 327-329
Lenguaje de descripción en hardware
 VHDL, 25-28,37
 desventajas, 27-28
 en la actualidad, 28
 ventajas, 26-27
 HDL, 25
Librerías
 creación de, 208-210
 declaración de, 45
 ieee, 44
 library, 44
 work, 44
Lógica programable
 ambiente de desarrollo, 18-19
 campos de aplicación, 23-24
 compañías de soporte, 28-31,32
 futuro, 31
 método tradicional de diseño, 20-22
Macroceldas, 12,14
Máquina de estado ASM, 154-159
 bloque de estado, 154
 bloque de decisión, 155
 diseño con VHDL, 166

- Microprocesadores, 230-31
programación de, 211-224, 236-261
- Modos, 39
buffer, 39
in, 39
inout, 39
out, 39
- Multiplexores, 75
otros (*others*), 75
tipos lógicos estándar, 76
- Netlist*, 53
- No importa (*don't care*) véase valores no importa,
- Nova simulador, 320
- Objetos de datos, 333-335
archivos, 335
constantes, 333
variables, 335
- Operadores
aritméticos, 82
lógicos, 63
relacionales, 73
- Optimización, 323
- PALASM, 21
- Paquete (*package*), 37, 44, 122
cuerpo del, 37
declaración, 44
numeric_std, 46
numeric_bit, 46
std_arith, 46, 83
- PLD véase dispositivos lógicos programables
- Proceso (*process*), 48, 69
- Programa de alto nivel (*top level*), 197, 200, 207
diseño del 198, 223, 264-266
- Project*, 313
- Puertos, 39
- Redes neuronales artificiales, 273-279
aprendizaje en las,
- Redes asociativas, 294-297
- Registros, 98-100
- Reporte, archivo de, 322-327
- Semi-custom, 3
- Señal (*signal*), 54
- Síntesis, 323
- Sistema secuencial, 93-94
síncrono, 105
- Sistema embebido, 229-237
clasificación, 235-236
diseño, 231-235
- Sumadores 78
- Tecnologías de fabricación
de circuitos integrados, 3
- Tipo de datos, 36, 39, 40
archivo. (*record*),
arreglo, 338
bit, 40
bit_vector, 40, 42, 43
boolean, 40
compuestos, 336
enumerado, 108, 336
escalar, 336
físicos, 337
integer, 40
reales, 337
- Unidades de diseño, 37
primarias, 37
secundarias, 37
- Valores no importa (*don't care*), 97
- Variables, 335
- VHDL véase lenguaje de descripción en hardware
- WarpR4, 311-312
Instalación, 331-332

Jramonse@uis.edu.co