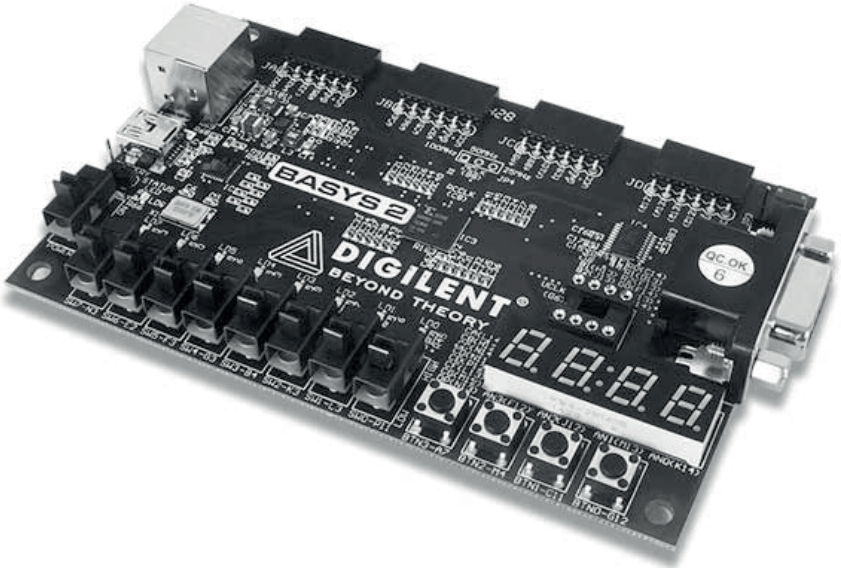
VHDL sintetizable para estudiantes de Ingeniería



Carlos Castro Miguens (Universidad de Vigo)

José B. Castro Miguens (Cesinel)

Universidade de Vigo

Servizo de Publicacións

2018

Edición
Servizo de Publicacións da Universidade de Vigo
Edificio da Biblioteca Central
Campus de Vigo
36310 Vigo

ISBN: 978-84-8158-771-5
D.L.: VG 191-2018

Impresión: Tórculo Comunicación Gráfica, S.A.

Prefacio

Los procedimientos de diseño e implementación de un sistema digital que se utilizan hoy en día no tienen nada que ver con los utilizados en la década de los años 70 y principios de los años 80 del siglo XX. La forma de implementar un sistema digital en la actualidad ya no se basa en interconectar varios circuitos integrados de las familias *TTL* y/o *CMOS*, constituidos internamente por puertas lógicas y bloques funcionales combinacionales y/o secuenciales. Hoy en día se suele implementar un circuito digital utilizando un único circuito integrado consistente en una *CPLD*, en una *FPGA* o en un *ASIC*, dependiendo de la complejidad del circuito, de la frecuencia de las señales que debe manejar y del número de unidades a producir[1].

La forma de diseñar hoy en día un sistema digital mínimamente complejo ya no consiste en dibujar un esquema en un papel en el que se representan puertas lógicas y bloques funcionales combinacionales y/o secuenciales comerciales, indicando las conexiones eléctricas entre ellos. Desde finales de los años 80 del siglo pasado, los sistemas digitales se diseñan básicamente describiendo su comportamiento, utilizando para ello un lenguaje de descripción de hardware (*HDL*: *Hardware Description Language*). Los lenguajes más utilizados actualmente para describir *hardware* digital son el *Verilog* y el *VHDL*. Los fabricantes de *FPGAs* y de *CPLDs* comercializan herramientas *software* que permiten describir, simular e implementar sistemas digitales tanto en *CPLDs* como en *FPGAs*. A modo de ejemplo, cabe decir que en una *FPGA Virtex*-7 2000T de *Xilinx*[2] se puede implementar cualquier circuito o circuitos digitales cuyo diseño a base de puertas lógicas no requiera utilizar más de 20 millones de puertas lógicas.

Este libro se ha escrito para estudiantes de Ingeniería que teniendo unos conocimientos básicos de Electrónica Digital, necesiten aprender a diseñar y a describir sistemas digitales complejos utilizando un lenguaje de descripción de hardware. El lenguaje que se explica en este libro se denomina *VHDL* (***V****ery High Speed Integrated Circuit* ***H****ardware* ***D****escription* ***L****anguage*). En concreto, se explica la parte del lenguaje *VHDL* que permite describir sistemas digitales implementables (sintetizables) en *CPLDs*, en *FPGAs* y en *ASICs*. A la hora de representar los circuitos que se describen a lo largo de este libro se ha seguido la norma ANSI/IEEE std. 91-1984 en la medida de lo posible. Los términos en

[1] CPLD: *Complex Programmable Logic Device*
FPGA: *Field Programmable Gate Array*
ASIC: *Application-specific integrated circuit*

[2] La FPGA Virtex-7 2000T de *Xilinx* tiene 6800 millones de transistores con los que se implementan, entre otras cosas, 2 millones de celdas lógicas. Lo que da lugar a una capacidad equivalente a 20 millones de puertas lógicas para implementar circuitos digitales.

inglés que en opinión de los autores no tienen un término equivalente en castellano, o que no se emplean habitualmente en la literatura técnica se han dejado en inglés y en escritura cursiva.

A lo largo del libro se explican muchas características del lenguaje *VHDL* como si se tratase de un lenguaje de programación, pese a que no lo es. Esto es debido a que resulta mucho más sencillo explicar y entender muchos conceptos del lenguaje *VHDL* realizando una analogía con un lenguaje de programación. En cualquier caso, el lector debe tener siempre presente que las instrucciones de un código en *VHDL* no se ejecutan como si se tratase de las instrucciones de un programa, sino que son interpretadas por un sintetizador. El cual, a partir de las mismas, determina un circuito digital implementable en una *FPGA* o en una *CPLD*, teniendo en cuenta los recursos internos de los que dispone el dispositivo reconfigurable.

La necesidad de escribir este libro surge motivada por la considerable reducción de horas de clase en los actuales planes de estudio en las escuelas de Ingeniería. Lo que obliga a los alumnos a tener que recurrir a libros y a manuales para ampliar y profundizar en los conceptos que se imparten en las clases de teoría.

Agradecimientos

Los autores queremos desde aquí darles las gracias a los estudiantes de la escuela de Informática de la Universidad de Vigo por sus comentarios y críticas constructivas durante la redacción de este libro. Y en particular, queremos darle las gracias al profesor Francisco Poza por compartir con nosotros sus conocimientos, por su paciencia… por todo.

Carlos Castro Miguens y José B. Castro Miguens

Índice

Tema 1: **Introducción**

1.1 Breve historia del *VHDL* 11

1.2 Proceso de diseño de un sistema digital utilizando *VHDL* 13

1.3 Características básicas de una *FPGA Spartan*-3E de *Xilinx* y de la placa de entrenamiento *Basys*2 15

1.4 Estructura de un código escrito en *VHDL* 22

1.4.1 Bibliotecas (*Libraries*) 22

1.4.2 Entidad (*Entity*) 24

1.4.3 Arquitectura (*Architecture*) 26

Tema 2: **Tipos de datos y operadores**

2.1 Tipos de datos sintetizables 33

2.1.1 *Std_logic* 34

2.1.2 *Std_logic_vector* 35

2.1.3 *Signed* 38

2.1.4 *Unsigned* 40

2.1.5 *Integer* 41

2.2 Constantes 44

2.3 Genéricos 44

2.4 Tipos de datos definidos por el usuario 46

2.4.1 Tipo de dato enumerado (*enumerated*) 46

2.4.2 Tipo de dato matriz (*array*) 47

2.5 Conversión entre tipos de datos 52

2.6 Operadores 56

2.6.1 Operadores de asignación 56

2.6.2 Operadores lógicos 57

2.6.3 Operador de signo menos 58

2.6.4 Operadores aritméticos 58

2.6.5 Operador de concatenación 65

2.6.6 Operadores de relación o de comparación 66

2.6.7 Operadores y funciones de desplazamiento y de rotación 66

Tema 3: **Código concurrente**

3.1 Introducción 71

3.2 Señales 72

3.3 Operadores 73

3.4 Estructura de asignación condicional *when…else* 76

3.5 Estructura de asignación condicional *with…select…when* 77

3.6 *Generate* 79

3.7 *Block* 82

3.8 Ejemplos de descripción de circuitos combinacionales utilizando código concurrente 84

3.8.1 Decodificadores 84

3.8.2 Codificadores 87

3.8.3 Multiplexores 91

3.8.4 Demultiplexores 93

3.8.5 Comparadores de magnitud 95

3.8.6 Detector de paridad 99

3.8.7 Convertidores de código 100

3.8.8 Desplazador circular 107

3.8.9 Unidad aritmética y lógica 108

3.8.10 Tabla de búsqueda 110

3.8.11 Biestable asíncrono *R-S* (*Latch*) 112

Tema 4: **Código secuencial**

4.1 Introducción 113

4.2 Procesos (*Processes*) 113

4.2.1 Sintaxis de un proceso con lista de sensibilidad 114

a) Lectura y asignación de valores a señales (*signals*) 114

b) Variables 117

c) Estructura *if…then…else* 120

d) Estructura *case…when* 129

e) Estructura *loop* 134

4.2.2 Sintaxis de un proceso sin lista de sensibilidad 137

a) *wait until* 138

b) *wait on* 140

c) *wait for* 140

d) *wait* 140

4.3 Ejemplos de descripción de circuitos combinacionales utilizando código secuencial

4.3.1 Circuitos decodificadores 141

4.3.2 Circuitos multiplexores 143

4.3.3 Circuito demultiplexor de 2^n canales 144

4.3.4 Circuito comparador de magnitud de *n* bits 145

4.3.5 Circuito detector de paridad par e impar de *n* bits 145

4.3.6 Circuitos convertidores de código 146

4.3.7 Circuito que proporciona el mayor de 4 números 147

4.4 Ejemplos de descripción de circuitos secuenciales utilizando código secuencial 148

4.4.1 Biestable *RS* asíncrono (*latch RS*) 148

4.4.2 Flip flop *D* 149

4.4.3 Flip flop *J-K* 151

4.4.4 Flip flop *T* 152

4.4.5 Contador en binario de *n* bits 153

4.4.6 Registro de desplazamiento de *n* bits de entrada en serie y salida en paralelo 155

4.4.7 Circuitos divisores de frecuencia 156

4.4.8 Circuito temporizador I 162

4.4.9 Circuitos detectores de flancos 163

4.4.10 Circuitos sincronizadores y antirrebotes 167

4.4.11 Circuitos contadores en códigos especiales 173

4.4.12 Circuitos convertidores de código 175

4.4.13 Circuitos moduladores *PWM* 178

4.4.14 Circuito que realiza un desplazamiento aritmético hacia la derecha 186

4.4.15 Contador de segundos 187

Tema 5: Componentes

5.1 Conceptos básicos 191

5.2 Ejemplos de descripción de sistemas digitales utilizando componentes 195

5.2.1 Comparadores de magnitud 195

5.2.2 Contador de minutos y de segundos 200

5.2.3 Generador de señales binarias periódicas 208

Tema 6: Sistemas secuenciales síncronos

6.1 Introducción 215

6.2 Ejemplos de descripción de sistemas secuenciales definidos mediante diagramas de estados 228

6.3 Cuestiones a tener en cuenta al describir un sistema secuencial con el entorno *ISE WebPACK* 14.7 de *Xilinx* 258

Apéndices:

A.1 Valor mínimo del periodo de la señal de reloj de un sistema secuencial 261

A.2 Metaestabilidad 263

A.3 *Clock skew* 267

A.4 Problemas de *fan-out* del circuito oscilador que genera la señal de reloj 271

A.5 *Clock jitter* 272

A.6 Circuitos sincronizadores 276

A.6.1 Circuito sincronizador con 1 *flip-flop* 278

A.6.2 Circuito antirrebotes y sincronizador formado por un 1 *flip-flop* 285

A.6.3 Circuito sincronizador formado por 2 *flip-flops* 290

A.6.4 Circuito antirrebotes y sincronizador formado por 2 *flip-flops* 294

A.7 Como crear un proyecto en el entorno *ISE WebPACK* Versión 14.7 de *Xilinx* 302

A.8 Como realizar una simulación funcional en el entorno *ISE WebPACK* de *Xilinx* 307

A.9 Como realizar una simulación temporal en el entorno *ISE WebPACK* de *Xilinx* 312

A.10 Sintaxis de las estructuras concurrentes y secuenciales en VHDL 315

A.11 Procedimientos de conversión entre tipos de datos en VHDL 316

A.12 Palabras reservadas en *VHDL* 317

Bibliografía 319

Lista de ejemplos

Buffer: pag. 83

Decodificadores: pags. 84, 86, 87, 131, 132, 141

Decodificadores de binario a 7 segmentos: pags. 102, 132

Codificadores: pags. 77, 87, 89, 90, 122, 142

Multiplexores: pags. 79, 91, 92, 93, 130, 143

Demultiplexores: pags. 93, 94, 95, 144

Comparadores de Magnitud: pags. 95, 97, 98, 125, 145, 195, 198

Detector de paridad: pag. 99

Generador de paridad impar: pag. 135

Convertidores de código: pags. 100, 101, 102, 175, 176

Desplazador circular: pag. 107

Sumador total: pags. 45, 80, 193

Unidad aritmética y lógica: pag. 108

Biestable asíncrono (*latch*) RS: pags. 112, 148

Biestable asíncrono (*latch*) D: pag. 84

Flip-flop D: pags. 126, 139

Flip-flop J-K: pag. 151

Flip-flop T: pag. 152

Contadores en binario: pags. 127, 128, 153

Contador en código anillo: pag. 173

Contador en código Johnson: pag. 174

Contador de ceros: pag. 134

Contador de unos: pag. 137

Contador de segundos: pag. 187

Contador de minutos y de segundos: pag. 200

Registro paralelo: pag. 139

Registro de desplazamiento de entrada serie y salida serie: pag. 121

Registro de desplazamiento de *n* bits de entrada en serie y salida en paralelo: pag. 154

Tabla de búsqueda (*Lookup table*): pag. 110

Generador de señales binarias: pags. 140, 208

Circuito que proporciona el mayor de 4 números: pag. 147

Divisores de frecuencia: pags. 156, 158, 159

Temporizador: pags. 162, 239

Circuito detector de flancos: pags. 163, 165,

Circuito que cuenta el número de veces que se pulsan dos botones: pag. 171

Moduladores de ancho de pulso (*PWM*): pags. 178, 182

Circuito que realiza un desplazamiento aritmético: pag. 186

Circuitos sincronizadores y antirrebotes: pags. 167, 169

Control del estado de una bombilla utilizando un pulsador: pag. 228

Control del nivel del agua en el interior de un depósito: pag. 234

Detector de la transmisión en serie del valor 101: pag. 241

Control de las luces de un semáforo: pag. 247

Control de la puerta de un garaje: pag. 251

Capítulo 1

Introducción

1.1 Breve historia del VHDL

A principios de los años 80 del siglo pasado se fabricaban circuitos integrados que contenían del orden de 10^5 transistores (VLSI: *very large scale integration*). Si se tiene en cuenta que para implementar una puerta NAND o una puerta NOR sólo se necesitan 4 transistores, resulta evidente que los circuitos digitales que se diseñaban por aquel entonces ya habían alcanzado la suficiente complejidad como para que su diseño a base de dibujar esquemas en un papel, indicando las interconexiones entre puertas lógicas y bloques funcionales (decodificadores, multiplexores, contadores, registros, etc.) resultase inviable desde un punto de vista práctico. La solución a este problema pasó por el desarrollo de lenguajes para describir hardware. Y como ocurre con cualquier nueva tecnología, el uso de una nueva forma de definir y de diseñar circuitos digitales dio lugar a la aparición de nuevos problemas, entre los que cabe destacar los siguientes:

- A principios de los años 80, en la industria de la Electrónica se utilizaban varios lenguajes de descripción de hardware, todos ellos patentados e incompatibles entre si. Lo que hacía que un diseño realizado por una empresa utilizando un lenguaje dado no podía ser utilizado por otra empresa que no conociese dicho lenguaje y/o que no dispusiese de una licencia de uso del mismo.

- No había ninguna garantía de que los distintos lenguajes utilizados por la industria de la Electrónica sobreviviesen a la esperanza de vida del hardware que se describía con ellos. En la práctica, un diseño realizado o descrito en un lenguaje que había dejado de estar en uso no se podía volver a utilizar debido a la incompatibilidad de los lenguajes existentes (no había ninguna garantía de que un diseño se pudiese volver a utilizar en el futuro).

La solución a los problemas anteriores consistió en crear un lenguaje de descripción de

hardware estándar[1]. Las especificaciones de dicho lenguaje se establecieron a comienzos de los años ochenta, en un proyecto de investigación del *Departamento de Defensa* de *Estados Unidos*, como parte de un programa denominado *Very High Speed Integrated Circuit* (*VHSIC*). Siendo este el motivo por el que el lenguaje creado se denominó *VHSIC Hardware Description Language* (*VHDL*)[2]. Entre 1883 y 1985 las empresas *IBM*, *TI* e *Intermetrics* colaboraron en el desarrollo del lenguaje *VHDL* (la versión 2.0 se publicó seis meses después de que estas empresas iniciaran la colaboración). En diciembre de 1984 se publicó la versión 6.0. En 1986 se transfirieron los derechos del lenguaje al *IEEE* (*Institute of Electrical and Electronics Engineers*). En 1987 el *IEEE* publicó la norma *IEEE* 1076-1987, conocida como *VHDL*-1987, con la que el lenguaje *VHDL* pasaba a ser un estándar. Desde entonces se han publicado varias actualizaciones [11-14][3]:

_ En 1994 se publicó una actualización de la norma IEEE 1076-1987 que se denominó IEEE 1076-1993 y que en la literatura técnica se suele indicar como VHDL-93. En esta nueva versión se añadieron operadores de desplazamiento, la operación lógica *xnor*, la llamada directa de componentes (*direct instantiation of components*), se mejoró la escritura de los test de prueba (*test benches*), etc.

_ En los años 2000 y 2002 se publicaron dos revisiones menores del estándar (VHDL-2000 y VHDL-2002) que no tienen relevancia en lo que a la descripción de circuitos se refiere.

_ En el año 2007 se publicó una revisión del estándar VHDL-2002 en la que se introdujeron pequeños cambios en la norma VHDL-2002 y se publicó el estándar *VHDL Procedural Language Application Inteface Standard* (VHDL 1076c-2007), también conocido como VHPI. Esta herramienta permite escribir programas en lenguajes de programación como *C* para que interactúen con un simulador de VHDL.

_ En el año 2009 se publicó otra revisión del estándar (VHDL-2008) con la que, entre otras cosas, se amplió la gama de paquetes (*packages*) predefinidos. Una buena parte de las modificaciones introducidas por esta revisión aún no han sido adoptadas por las herramientas de simulación y de síntesis comerciales.

A partir de la publicación del estándar IEEE 1076-1987, diversas empresas que comercializaban simuladores de código VHDL comenzaron a introducir nuevos tipos de datos

[1] Los dos lenguajes de descripción de hardware más utilizados hoy en día son el *VHDL* y el *Verilog*. En este libro se estudia la parte sintetizable del lenguaje *VHDL*.

[2] El lenguaje VHDL está basado en el lenguaje de programación Ada y como consecuencia de ello sus sintaxis se parecen, pero no es un lenguaje de programación. Es un lenguaje creado para describir circuitos digitales (hardware).

[3] El IEEE realiza una revisión de sus normas (estándares) como muy tarde cada 5 años.

no contemplados en el estándar, con el fin de que sus clientes pudiesen simular de forma más precisa el funcionamiento de circuitos complejos. Para evitar el problema creado con la existencia de tipos de datos no estándar, el IEEE publicó el estándar 1164 que define un paquete (*package*) denominado *std_logic*, el cual contiene las definiciones de 9 tipos de datos. Este paquete junto con el estándar IEEE 1076-1993 (VHDL-1993) constituyen el estándar más utilizado hoy en día.

En 1995 se publicó la versión 1076.3 con el fin de reemplazar los paquetes (*packages*) no estándar que muchos desarrolladores de herramientas de síntesis han ido creando y distribuyendo con sus productos a lo largo de los años.

Las principales aplicaciones del lenguaje VHDL son el diseño, la simulación (funcional y temporal) y la implementación de circuitos digitales en CPLDs (*complex programmable logic devices*), en FPGAs (*field programmable gate arrays*) y en ASICs (*application specific integrated circuits*). Algo a tener en cuenta es que no todo lo que se puede describir con el lenguaje VHDL es sintetizable (implementable) en un circuito integrado. En este libro sólo se explica la parte sintetizable del lenguaje VHDL, tomando como referencia el estándar VHDL-1993.

1.2 Proceso de diseño de un sistema digital utilizando *VHDL*

El proceso de diseño y de implementación de un sistema digital utilizando un lenguaje de descripción de hardware consta de los siguientes pasos[4] (ver figura 1):

1º *paso*: Se divide el sistema en bloques o módulos más o menos sencillos que se puedan describir (diseñar) de forma independiente. La elección de los bloques en cuanto a su complejidad depende de la experiencia del diseñador así como de los bloques que ya se tengan diseñados y que se puedan utilizar en la descripción del sistema. En la práctica, a la hora de dividir un sistema complejo en bloques se suele seguir una combinación de las metodologías de diseño conocidas como *top-down* y *bottom-up*[5].

2º *paso*: Se describe en VHDL cada uno de los bloques en los que se haya dividido el sistema y se realiza una comprobación sintáctica de los distintos códigos. A continuación se puede realizar una simulación de su funcionamiento. Dependiendo de la complejidad de

[4] Una descripción detallada del proceso de síntesis y de implementación de un sistema digital en una *CPLD* o en una *FPGA* se sale de los objetivos de este libro. En el caso de utilizar un sintetizador de Xilinx, el lector interesado puede consultar la referencia [17][30].

[5] La metodología *top-down* consiste en dividir un sistema complejo en bloques cuyo diseño (descripción) resulta sencillo. Mientras que la metodología *bottom-up* consiste en realizar el diseño del sistema a partir de bloques ya diseñados.

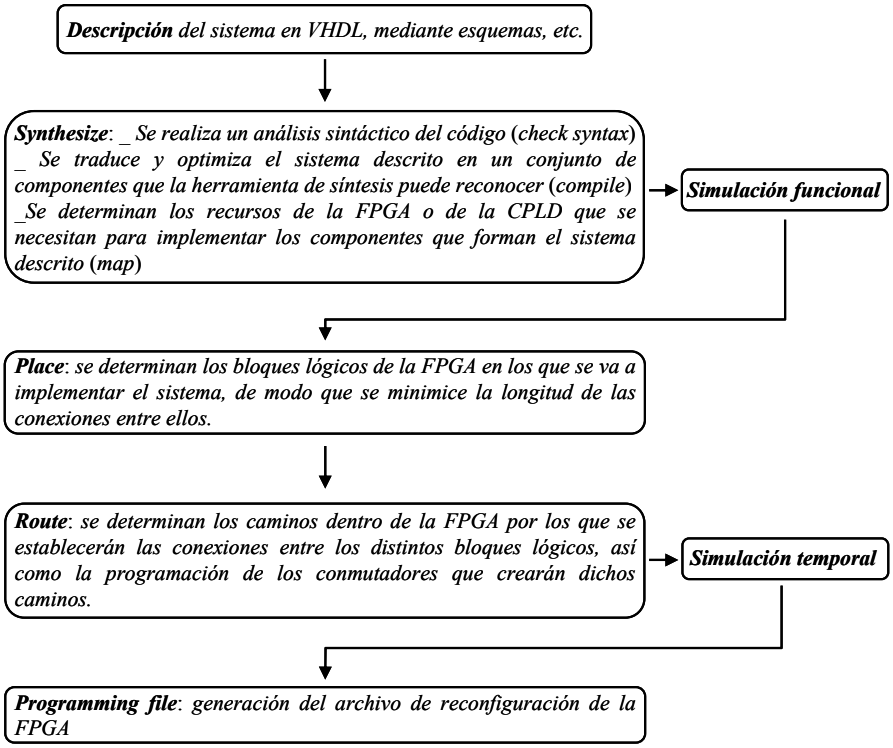


Figura 1.1: Diagrama de flujo del proceso de diseño e implementación de un sistema digital en una *FPGA*.

cada uno de los bloques en los que se ha dividido el sistema y de la experiencia del diseñador, se puede proceder a simular directamente el sistema completo o bien se puede realizar una simulación previa de cada uno de los bloques que lo forman por separado. Con ambos planteamientos se pueden realizar dos tipos de simulaciones:

· Simulación *funcional*: con este tipo de simulación se considera que todos los componentes tienen un comportamiento ideal. Es decir, se realiza una comprobación del funcionamiento del sistema a nivel lógico. En el apéndice A.8 se indica de forma detallada cómo se realiza una simulación funcional en el entrono *ISE WebPACK* 14.7 de *Xilinx*.

· Simulación *temporal*: con este tipo de simulación se tiene en cuenta el comportamiento real de los componentes que se utilizan en la implementación del sistema. Así, por ejemplo, en la simulación se tienen en cuenta los retardos de propagación de los componentes y de los caminos (buses) por los que se propagan las señales digitales, el *fan-out* y el *fan-in* de los componentes, etc. La simulación temporal requiere que se haga un proceso pre-

vio de *Place and Route* [17], con el fin de que el simulador conozca los parámetros temporales de los componentes que debe utilizar en la simulación del sistema, la longitud de las conexiones entre terminales, etc. En el apéndice A.9 se indica de forma detallada cómo se realiza una simulación temporal en el entrono *ISE WebPACK* 14.7 de *Xilinx*.

Para realizar una simulación tanto funcional como temporal es necesario crear previamente un archivo en vhdl en el que se describen las señales de entrada a aplicar al sistema durante un intervalo de tiempo dado. Dicho archivo se denomina *testbench*.

3º *paso*: Se realiza el proceso de síntesis del sistema descrito, en el que el sintetizador traduce el código en *VHDL* en un conjunto de componentes que la herramienta de síntesis reconoce.

4º *paso*: Se realiza la implementación del diseño. En primer lugar se determinan los recursos de la *FPGA* o de la *CPLD* que se necesitan para implementar el sistema. A continuación se eligen los bloques lógicos en los que se va a implementar el diseño así como los caminos por los que se van a establecer las comunicaciones entre ellos y con los pines de la *FPGA*. La elección de los bloques lógicos se puede realizar en base a minimizar el área ocupada en la *FPGA* o bien en base a minimizar la longitud de las conexiones eléctricas entre los bloques lógicos y con los pines de la *FPGA*.

Para indicar los pines de la *FPGA* a los que se van a conectar los terminales de entrada y de salida del circuito a implementar es necesario crear previamente un archivo de restricciones de usuario (*Implementation constraints file*).

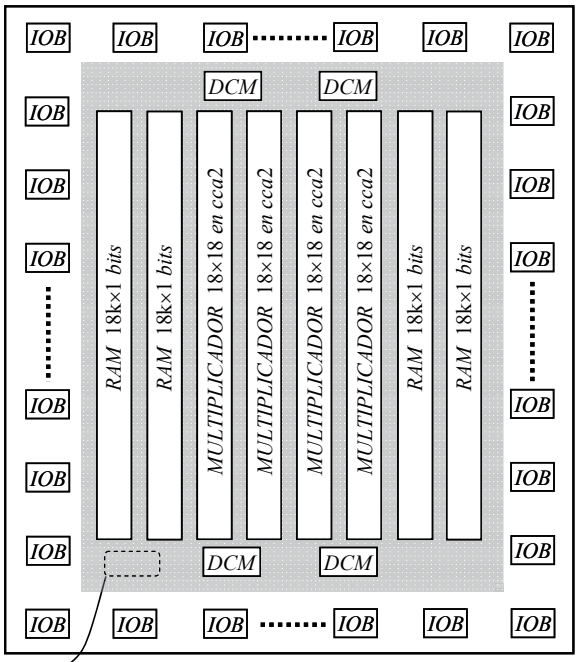
5º *paso*: Se genera el archivo de configuración de la *FPGA*. Dicho archivo se guarda en una memoria que se lee cada vez que se pone en funcionamiento la *FPGA*[6]. Lo que hace que si se modifica el archivo de configuración guardado se modifica el circuito que se implementa.

1.3 Características básicas de una *FPGA Spartan*-3E y de la placa de entrenamiento *Basys* 2

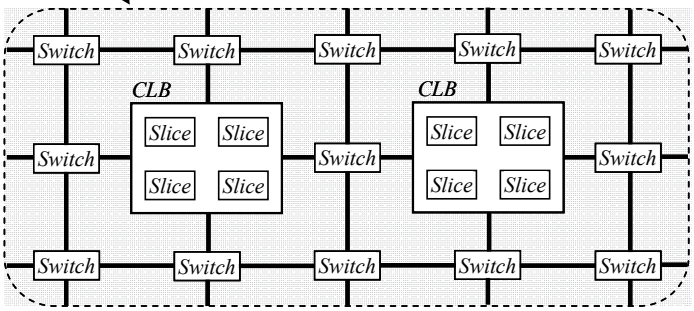
A mediados de los años 80 se empezaron a comercializar unos dispositivos reconfigurables que se denominan genéricamente como *FPGAs* (*Field Programmable Gate Arrays*). Dichos dispositivos presentan una arquitectura interna basada en el uso de tablas de búsqueda (*LUTs*: *look-up tables*), caracterizándose por contener los siguientes elementos (ver figura 1.2):

[6] Hay FPGAs de tipo *OTP* (*One Time Programming*) que sólo se pueden reconfigurar una vez.

a)



b)



c)

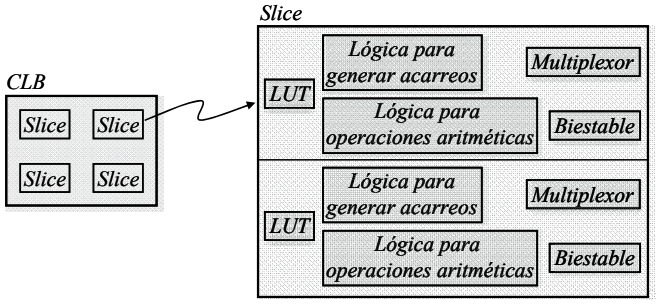


Figura 1.2 ***a***) Arquitectura interna de una *FPGA Spartan*-3*E*-100 de Xilinx ***b***) Matriz de bloques lógicos configurables ( ) ***c***) Estructura de un *bloque lógico configurable* (*CLB*)

- Una matriz de *bloques lógicos configurables* (*CLBs*: *configurable logic blocks*). Los *CLBs* son el principal recurso de que disponen las *FPGAs* para implementar tanto circuitos combinacionales como circuitos secuenciales síncronos. Si bien es cierto que en un *CLB* sólo se pueden implementar circuitos relativamente 'sencillos', hay que tener en cuenta que por medio de conmutadores (*switches*) programables se pueden interconectar varios *CLBs* (ver figura 1.2*b*)). Lo que hace que utilizando el número suficiente de *CLBs* se pueda implementar prácticamente cualquier circuito, por muy complejo que sea.

- Una serie de bloques de entrada/salida (*IOBs*: *Input/Output Blocks*) que rodean a la matriz de *CLBs*. Estos bloques se utilizan para controlar el flujo de datos entre los pines de la *FPGA* y el circuito o circuitos implementados en los *CLBs*.

- Bloques de memoria *RAM* para guardar datos (ver figura 1.2*a*))

- Circuitos multiplicadores de $n \times n$ bits (ver figura 1.2*a*))

- Circuitos para gestionar la señal de reloj (*DCM*: *digital clock manager*). Permiten distribuir la señal de reloj por el interior de la *FPGA*, introducir retardos así como dividir y multiplicar su frecuencia.

- Hay *FPGAs* que también contienen procesadores.

Para comprobar los ejemplos que aparecen en este libro a nivel de hardware se ha utilizado una placa de entrenamiento *Basys* 2, que tiene una *FPGA Spartan*-3E-100 TQ144 de Xilinx [7] [21]. Dicha *FPGA* se caracteriza porque contiene 240 *CLBs* organizados en forma de una matriz de 22 filas y 16 columnas[7]. Cada *CLB* contiene 4 *slices* y cada *slice* contiene:

- 2 tablas de búsqueda (*LUTs*) de 4 entradas, de modo que cada *LUT* se puede utilizar para implementar una función lógica dependiente de hasta 4 variables. Las *LUTs* de algunos *slices* también se pueden utilizar como memorias *RAM* de capacidad 16×1 bits o como registros de desplazamiento de 16 bits.

- 2 elementos de memoria que se pueden utilizar como *flip-flops* de tipo *D* o bien como *latches* de tipo *D*, con entradas de *set* y de *reset* asíncronas.

- 2 multiplexores.

- puertas lógicas para generar acarreos y realizar determinadas operaciones matemáticas de forma rápida.

[7] El número de *CLBs* es inferior al producto del número de filas por el número de columnas. Esto es debido a que los bloques de memoria RAM, los circuitos multiplicadores y los circuitos DCM ocupan el espacio de 112 *CLBs*.

Se denomina *logic cell* a la combinación de una *LUT* y de un biestable (ver figura 1.3). Los elementos adicionales existentes en un *slice* ayudan a implementar algunas funciones matemáticas que de no existir sólo se podrían implementar a base de utilizar más *LUTs*. La eficiencia de los elementos adicionales que tienen los *slices* de una *FPGA Spartan*-3E-100 se ha medido con *benchmarks*[8] obteniéndose que los *slices* de esta *FPGA* proporcionan unas prestaciones equivalentes a 2,25 *logic cells*.

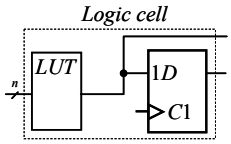


Figura 1.3 Esquema de una *logic cell*

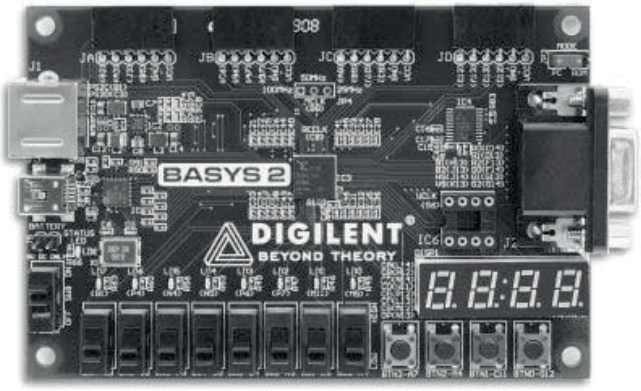
A modo de resumen se puede decir que una *FPGA Spartan*-3E-100 tiene 240 *CLBs*, 960 *slices*, 1920 *LUTs* (960 de las cuales también se pueden utilizar como memorias *RAM* de capacidad 16×1 o como registros de desplazamiento de 16 bits) y 1920 biestables. Dispone de 4 bloques de memoria RAM de doble puerto con una capacidad de 18k bits cada uno, lo que corresponde a una capacidad total de 73.728 bits. Dispone de cuatro circuitos multiplicadores de 18×18 bits cada uno, que realizan los productos en el código *ca*2. El resultado de las multiplicaciones se representa con 32 bits.

La empresa *Digilent Inc* comercializa una placa de entrenamiento denominada *Basys* 2 que contiene lo siguiente (ver figura 1.4):

- Una *FPGA Spartan*-3E-100, en la que se puede implementar cualquier circuito digital que no requiera más de 100.000 puertas lógicas.
- Dispone de una memoria flash para guardar el archivo de reconfiguración
- Tiene un oscilador con el que se genera la señal de reloj a utilizar en la FPGA. El jumper JP4 permite seleccionar una frecuencia de 25MHz, 50MHz o 100MHz (la frecuencia por defecto es de 50MHz).
- Tiene 8 leds, 4 botones, 8 conmutadores y 4 *displays* de 7 segmentos de ánodo común conectados a pines de la FPGA (ver figura 1.5)
- Tiene un puerto PS/2 y un puerto VGA de 8 bits

[8] Un *benchmark* es un procedimiento para evaluar el rendimiento de un sistema o componente.

a)



b)

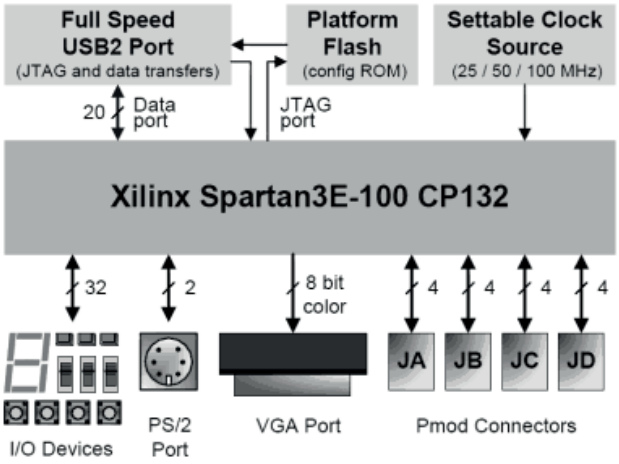


Figura 1.4 Placa de entrenamiento *Basys* 2: ***a***) Fotografía ***b***) Diagrama de bloques

- Tiene 4 terminales (JA, JB, JC y JD) de 6 pines cada uno conectados a la *FPGA*.
- Para cargar el archivo de reconfiguración de la *FPGA* hay que utilizar el programa Adept 2.0 o posterior.

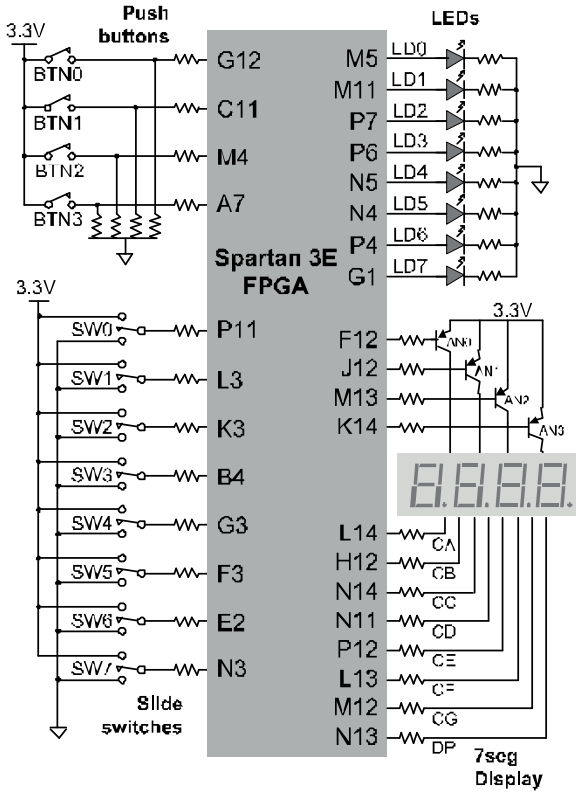


Figura 1.5 Conexión de los botones, de los conmutadores, de los *leds* y de los *displays* de 7 segmentos a la *FPGA* existente en una placa de entrenamiento *Basys* 2.

Para que un sintetizador pueda generar el archivo de reconfiguración de una *FPGA* es necesario indicarle los terminales de la *FPGA* a los que se quiere que se conecten los terminales de entrada y de salida del circuito implementado en la *FPGA*. En el caso del sintetizador *ISE* 14.7 de *Xilinx* esto se hace mediante un archivo denominado *Implementation Constraints File*. A continuación se muestra como se indica la conexión de los terminales de un circuito implementado en la *FPGA* de una placa de entrenamiento *Basys* 2 a los leds, a los conmutadores, a los botones, a los *displays* de 7 segmentos y a la señal de reloj existentes en una *Basys* 2[9] (ver figura 1.5).

· Asignación de los terminales de una señal de tipo *std_logic_vector*, denominada *Led*, a los pines de la *FPGA* a los que están conectados los *leds* de la *Basys* 2 (ver figura 1.5)

```
net "Led<0>" loc = "M5";
```

[9] En el archivo *Implementation Constraints File* el carácter # sirve para indicar un comentario.

```
net "Led<1>" loc = "M11";
net "Led<2>" loc = "P7";
net "Led<3>" loc = "P6";
net "Led<4>" loc = "N5";
net "Led<5>" loc = "N4";
net "Led<6>" loc = "P4";
net "Led<7>" loc = "G1";
```

· Asignación de los terminales de una señal de tipo *std_logic_vector*, denominada *Btn*, a los pines de la *FPGA* a los que están conectados los botones de la *Basys* 2 (ver figura 1.5)

```
net "Btn<0>" loc = "G12";
net "Btn<1>" loc = "C11";
net "Btn<2>" loc = "M4";
net "Btn<3>" loc = "A7";
```

· Asignación de los terminales de una señal de tipo *std_logic_vector*, denominada *Sw*, a los pines de la *FPGA* a los que están conectados los conmutadores de la *Basys* 2 (ver figura 1.5)

```
net "Sw<0>" loc = "P11";
net "Sw<1>" loc = "L3";
net "Sw<2>" loc = "K3";
net "Sw<3>" loc = "B4";
net "Sw<4>" loc = "G3";
net "Sw<5>" loc = "F3";
net "Sw<6>" loc = "E2";
net "Sw<7>" loc = "N3";
```

· Asignación de los terminales de una señal de tipo *std_logic_vector*, denominada *Segmentos*, a los pines de la *FPGA* a los que están conectados los *leds* de los *displays* de 7 segmentos de la *Basys* 2 (ver figura 1.5)

```
net "Segmentos<0>" loc = "L14";  # segmento a
net "segmentos<1>" loc = "H12";  # segmento b
net "segmentos<2>" loc = "N14";  # segmento c
net "segmentos<3>" loc = "N11";  # segmento d
net "segmentos<4>" loc = "P12";  # segmento e
net "segmentos<5>" loc = "L13";  # segmento f
net "segmentos<6>" loc = "M12";  # segmento g
net "dp" loc = "N13";  # punto decimal
```

· Asignación de los terminales de una señal de tipo *std_logic_vector*, denominada *AN*, a los pines de la *FPGA* a los que están conectados los terminales con los que se controla el encendido/apagado de los *displays* de 7 segmentos de la *Basys* 2 (ver figura 1.5)

```
net "AN<0>" loc = "F12";  # ánodo común display 0 (AN0)
net "AN<1>" loc = "J12";  # ánodo común display 1 (AN1)
net "AN<2>" loc = "M13";  # ánodo común display 2 (AN2)
net "AN<3>" loc = "K14";  # ánodo común display 3 (AN3)
```

· En la placa de entrenamiento *Basys* 2 hay un oscilador que genera una señal de reloj con una frecuencia seleccionable entre 25, 50 y 100MHz. Dicha señal de reloj está conectada al pin B8 de la *FPGA*. La forma de indicar la conexión de una señal de entrada denominada *clk* al pin B8 de la *FPGA* es la siguiente:

```
net "clk" loc = "B8";
```

1.4 Estructura de un código escrito en *VHDL*[10]

La descripción en VHDL de un circuito digital consta de tres partes: declaración de bibliotecas y de paquetes, una entidad y una arquitectura. A continuación se describen de forma detallada las características y la función de cada una de estas partes:

1.4.1 Bibliotecas (*Libraries*)

Una biblioteca (*library*) puede guardar información muy variada como, por ejemplo, las definiciones de tipos de datos, de operadores aritméticos y lógicos, la descripción de diversos circuitos que resultan útiles a la hora de describir otros circuitos, etc. Una biblioteca también puede contener paquetes (*packages*) los cuales, a su vez, pueden contener funciones (*functions*) y/o procedimientos (*procedures*) que se denominan genéricamente como subprogramas.

Para hacer visible y por lo tanto utilizable en un diseño el contenido de una biblioteca es necesario declararla previamente. Para declarar una biblioteca es necesario escribir, al menos, dos líneas de código. En la primera línea se indica su nombre, lo que hace que dicha biblioteca esté disponible en la entidad y en la arquitectura que se indican más adelante. En la siguiente o siguientes líneas se indican los paquetes de la biblioteca que se van a utilizar, haciendo que estén disponibles todas las definiciones, funciones y procedimientos que contienen. La sintaxis de la declaración de una biblioteca es la siguiente:

library <*nombre_biblioteca*>;
use <*nombre_biblioteca* . *nombre_paquete* . *package_parts*>;

[10] En la literatura técnica, al conjunto de instrucciones que describen un circuito digital se le denomina *código* con el fin de diferenciarlo de un *programa* de ordenador.

Hay dos bibliotecas estándar que se incluyen por defecto en todos los proyectos y que por lo tanto no necesitan ser declaradas:

- *std*: esta biblioteca contiene los paquetes *standar* y *textio*. El paquete *standar* contiene, entre otras cosas, las definiciones de diversos tipos de datos, de operadores lógicos, de operadores aritméticos, de operadores de desplazamiento, etc. El paquete *textio* contiene diversos recursos relativos al manejo de archivos de texto.

- *work*: es la biblioteca en la que se guarda por defecto todo lo relacionado con el diseño o descripción que se está realizando.

Hay una tercera biblioteca, denominada *ieee*, que contiene diversas definiciones y funciones estándar utilizadas habitualmente en la descripción de circuitos. Para poder utilizarla en un diseño es necesario declararla (está disponible en todos los sintetizadores). Esta biblioteca contiene varios paquetes de los cuales, para la mayoría de los diseños, es suficiente con utilizar los paquetes *ieee.std_logic_*1164*.all* e *ieee.numeric_std.all*.

El paquete *ieee.std_logic_*1164*.all* es un paquete básico del *ieee* que, entre otras cosas, contiene:

- La definición de los tipos de datos *std_ulogic_vector*, *std_logic_vector, std_ulogic* y *std_logic*[11]
- La definición de los operadores lógicos *not*, *and*, *nand*, *or*, *nor*, *xor* y *xnor*.
- Funciones de conversión entre los tipos de datos anteriores.
- Funciones de detección de flancos de subida y de bajada.
- Operadores de relación: =, /=, <, <=, >, >=
- Operadores de desplazamiento de los bits guardados en un vector.

El paquete *ieee.numeric_std.all* contiene las definiciones necesarias para realizar operaciones aritméticas con datos de tipo *signed* y *unsigned*.

Para evitar problemas de exportabilidad del código se desaconseja el uso de los paquetes definidos por la empresa *Synopsys*: *ieee.std_logic_arith.all*, *ieee.std_logic_unsigned.all* e *ieee.std_logic_signed*, debido a que no los tienen todos los compiladores.

[11] Los tipos de datos *std_logic* y *std_logic_vector* son estándares industriales. Es obligatorio declarar los terminales de entrada y de salida de un circuito a implementar en una *FPGA* de uno de estos tipos. En el caso de que se declaren de otro tipo, el sintetizador los redeclarará asignándole uno de estos tipos.

En la práctica, en la mayoría de los diseños basta con declarar:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

1.4.2 Entidad (*Entity*)

Una entidad sirve para establecer los terminales de entrada y de salida del circuito a diseñar así como el tipo de dato existente en dichos terminales. La sintaxis de la declaración de una entidad es:

```
Entity <nombre_entidad> is
  Port(<nombre terminal>  :  <tipo de terminal>  <tipo de dato>;
        . . .
       <nombre terminal>  :  <tipo de terminal>  <tipo de dato>);
end <nombre_entidad>;
```

cumpliéndose que:

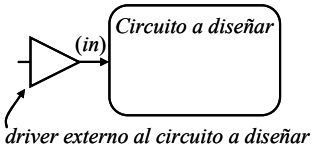
<*nombre_entidad*>: coincide con el nombre del circuito a diseñar. A la hora de escribirlo hay que tener en cuenta lo siguiente:

- Se pueden utilizar las letras del alfabeto tanto en minúsculas como en mayúsculas, los símbolos del sistema decimal y guiones bajos '_'.
- No se diferencia entre minúsculas y mayúsculas, pero dado que algunas herramientas de simulación sí lo hacen, se recomienda no utilizar nombres que sólo difieran en el uso de letras mayúsculas y minúsculas.
- El nombre debe comenzar con una letra, no puede terminar con un guión bajo, ni puede tener dos guiones bajos seguidos. No puede empezar con un número.
- No se puede utilizar como nombre una palabra reservada (ver apéndice A.10)
- No hay un límite máximo en cuanto al número de caracteres de un nombre.

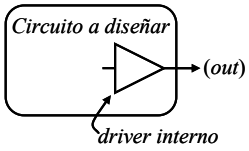
<*nombre terminal*>: es el nombre de un terminal del circuito a diseñar. Se le aplican las normas indicadas en el párrafo anterior.

<*tipo de terminal*>: los tipos de terminales sintetizables son,

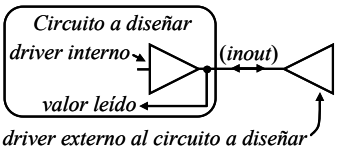
- *in*: indica una entrada (o un conjunto de entradas) del circuito a diseñar. En la arquitectura se pueden poner instrucciones que lean el nivel lógico presente en ella,



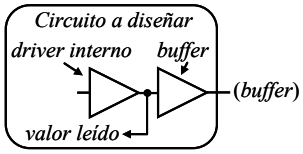
pero no que le asignen un valor. El número de terminales que forman la entrada se especifica mediante el parámetro `<tipo de dato>`.



· *out*[12]: indica una salida (o un conjunto de salidas) del circuito a diseñar. En la arquitectura se pueden poner instrucciones que pongan un valor lógico en ella, pero no que lean el valor presente en ella. El número de terminales que forman la salida se especifica mediante el parámetro `<tipo de dato>`.



· *inout*: indica un tipo de terminal que puede actuar como entrada y como salida. En la arquitectura se pueden poner instrucciones que lean y/o que escriban un valor lógico en este tipo de terminal. Se utiliza habitualmente en terminales que van conectados a un bus y que como consecuencia de ello es necesario poder poner en tercer estado. El número de terminales que forman este tipo de terminal se especifica mediante el parámetro `<tipo de dato>`.



· *buffer*: indica un terminal de salida que se caracteriza porque el valor de dicho terminal puede ser leído por el propio circuito[13]. Este tipo de salida es útil en la descripción de sistemas secuenciales. El número de terminales que forman la salida se especifica mediante el parámetro `<tipo de dato>`. En [28] se desaconseja el uso de este tipo de terminal, debido a que puede ser la causa de que se produzcan errores durante el proceso de síntesis. En el caso de que se necesite leer el valor de una salida, en vez de utilizar un terminal de tipo *buffer* se puede utilizar un terminal de tipo *out* junto con una señal definida en la arquitectura y cuyo valor se le asigna a la salida. Esta solución permite leer el valor de la señal que es el mismo que tiene la salida.

<*tipo de dato*>: el lenguaje vhdl tiene definidos varios tipos de datos. Algunos son sintetizables y otros no. De los que son sintetizables, los que se utilizan en este libro son: *std_logic*, *std_logic_vector*, *signed*, *unsigned* e *integer*. Los tipos de datos asignados a los terminales declarados en una entidad deben ser de tipo *std_logic* o *std_logic_vector*. Si se declaran de otro tipo, al compilar el código el sintetizador le asigna uno de estos tipos[14].

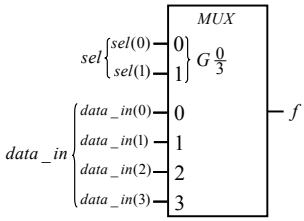
[12] Para describir sistemas combinacionales sólo son necesarios los tipos *in* y *out*.

[13] El valor leído puede no coincidir con el valor presente en el terminal de salida. Ya que dicho valor puede verse modificado, por ejemplo, por una demanda excesiva de corriente por parte de la entrada o entradas conectadas a dicho terminal.

[14] Los tipos de datos *std_logic* y *std_logic_vector* son un estándar industrial.

A continuación se indica a modo de ejemplo la entidad correspondiente al multiplexor indicado en la parte derecha (la declaración de los terminales no es única):

```
Entity Mux_4c is
  Port(sel : in std_logic_vector(1 downto 0);
   data_in : in std_logic_vector(3 downto 0);
         f : out std_logic);
end Mux_4c;
```



A la hora de definir una entidad hay que tener en cuenta lo siguiente:

- Siempre que se defina una entrada y/o una salida de tipo `std_logic_vector` hay que especificar obligatoriamente el número de terminales que tiene. Si no se conoce el número exacto en el momento de definir la entidad, se puede utilizar un tipo de dato genérico (*generic*) para indicarlo.

- No se admite una entrada y/o una salida con más de 1 dimensión.

1.4.3 Arquitectura (*Architecture*)

En el cuerpo de una *arquitectura* se describe el comportamiento del circuito cuyos terminales de entrada y de salida se indican en la *entidad* a la que pertenece. Aunque es posible tener una entidad con varias arquitecturas asociadas, representando cada una de ellas una implementación diferente del mismo circuito, sólo una de ellas se sintetiza. La sintaxis de una arquitectura es la siguiente:

```
architecture <nombre_arquitectura> of <nombre_entidad> is
 [declaraciones] -- parte declarativa de la arquitectura (es opcional)
 begin
    [código] -- aquí se describe el comportamiento del circuito.
 end <nombre_arquitectura>;
```

cumpliéndose que:

- `<nombre_arquitectura>`: los nombres de las arquitecturas cumplen las mismas reglas que los nombres de las entidades.

- `<nombre_entidad>`: es el nombre de la entidad a la que está asociada la arquitectura

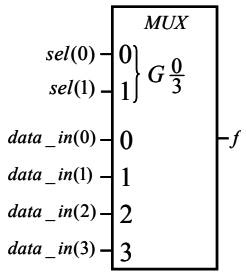
- `[declaraciones]`: esta es la parte declarativa de la arquitectura (está situada entre *is* y *begin*). En ella se pueden declarar señales (*signals*), constantes (*constants*) y componentes (*components*). Todo lo declarado es local a la arquitectura y por lo tanto es visible en toda

la arquitectura. Los terminales que se indican en la entidad a la que está asociada la arquitectura también son visibles en toda la arquitectura.

· `[código]`: este es el cuerpo de una arquitectura (está entre situado entre las palabras reservadas *begin* y *end*). Aquí es en dónde se describe el comportamiento del circuito. En la literatura técnica se consideran 3 estilos a la hora de describir un circuito:

_ Estilo *dataflow*: el hardware se describe utilizando únicamente instrucciones con las que se les asignan valores a señales (código concurrente). A continuación se indica un ejemplo de este estilo que describe el comportamiento del multiplexor de 4 canales indicado en la parte derecha.

```
architecture dataflow of mux_4c is
begin
 f <= (data_in(0) and not sel(1) and not sel(0)) or
      (data_in(1) and not sel(1) and sel(0)) or
      (data_in(2) and sel(1) and not sel(0)) or
      (data_in(3) and sel(1) and sel(0));
end dataflow;
```

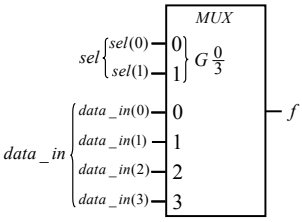


_ Estilo *behavioral*: el hardware se describe utilizando únicamente procesos (código secuencial, ver apartado 4.2). A continuación se describe el multiplexor de 4 canales anterior utilizando este estilo.

```
architecture behavioral of mux_4c is
begin
  process(sel,data_in)
  begin
    if(sel = "00") then
       f <= data_in(0);
    elsif(sel = "01") then
       f <= data_in(1);
    elsif(sel = "10") then
       f <= data_in(2);
    else
       f <= data_in(3);
    end if;
  end process;
end behavioral;
```

_ Estilo *structural*: en este caso la descripción se basa en utilizar una estructura jerárquica formada por la interconexión de diversos sistemas más sencillos. Este estilo resulta particularmente útil cuando hay que describir un sistema complejo, ya que permite describir y comprobar por separado el funcionamiento de cada una de las partes o subsistemas que lo forman. En general, la descripción de sistemas sencillos utilizando un estilo *structural* no

tiene mucho sentido, ya que su descripción suele llevar mucho más tiempo que la descripción utilizando un estilo *dataflow* o *behavioral*. Con el estilo *structural* la descripción de un sistema se realiza utilizando componentes (*component*). A continuación se muestra, a modo de ejemplo, una posible descripción del multiplexor de 4 canales indicado en la parte derecha. En la figura 1.6 se muestran los componentes en los que se ha dividido la descripción del multiplexor.



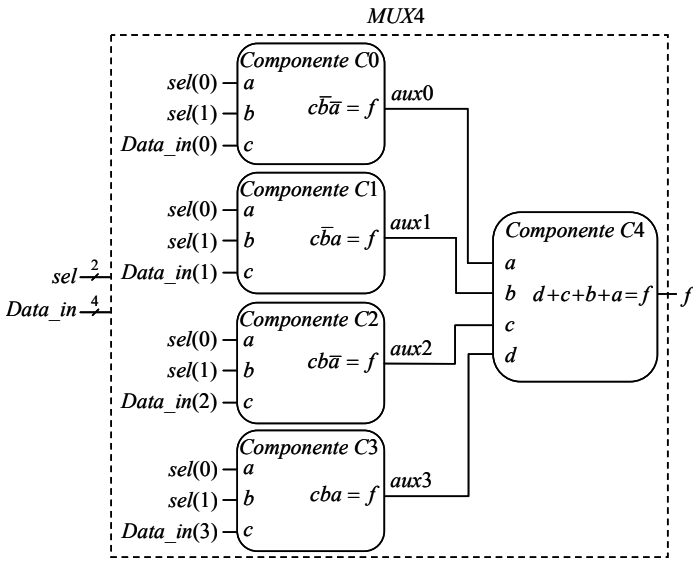


Figura 1.6 Ejemplo de descripción *structural* de un multiplexor de 4 canales

-- Descripción del componente *C*0

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity C0 is
  Port (a,b,c : in std_logic;
        f : out std_logic);
end C0;
architecture Behavioral of C0 is
begin
 f <= c and not b and not a; -- la operación not es prioritaria
end Behavioral;
```

-- Descripción del componente *C*1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity C1 is
   Port(a,b,c : in std_logic;
        f : out std_logic);
end C1;

architecture Behavioral of C1 is
begin
 f <= c and not b and a;
end Behavioral;
```

-- Descripción del componente *C*2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity C2 is
   Port(a,b,c : in std_logic;
        f : out std_logic);
end C2;

architecture Behavioral of C2 is
begin
 f <= c and b and not a;
end Behavioral;
```

-- Descripción del componente *C*3

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity C3 is
   Port (a,b,c : in std_logic;
         f : out std_logic);
end C3;

architecture Behavioral of C3 is
begin
 f <= c and b and a;
end Behavioral;
```

-- Descripción del componente *C*4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity C4 is
   Port (a,b,c,d : in std_logic;
         f : out std_logic);
```

```
end C4;
architecture Behavioral of C4 is
begin
 f <= d or c or b or a;
end Behavioral;
```

-- Descripción del multiplexor de 4 canales utilizando los componentes anteriores

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUX_4_canales is
   Port (sel : in std_logic_vector(1 downto 0);
         data_in : in std_logic_vector(3 downto 0);
         f : out std_logic);
end MUX_4_canales;
architecture Structural of MUX_4_canales is
 Component C0 is  -- Declaración del componente C0
  PORT(a,b,c : in std_logic;
       f : out std_logic);
 end Component;
 Component C1 is  -- Declaración del componente C1
  PORT(a,b,c: in std_logic;
       f : out std_logic);
 end Component;
 Component C2 is  -- Declaración del componente C2
  PORT(a,b,c: in std_logic;
       f : out std_logic);
 end Component;
 Component C3 is  -- Declaración del componente C3
  PORT(a,b,c: in std_logic;
       f : out std_logic);
 end Component;
 Component C4 is  -- Declaración del componente C4
  PORT(a,b,c,d: in std_logic;
       f : out std_logic);
 end Component;
 signal aux0, aux1, aux2, aux3 : std_logic;
begin
 u0: C0 PORT MAP(a => sel(0),b => sel(1),c => Data_in(0),f => aux0); --
```

[15]

[15] Uso del componente *C*0 (*instantiation*) ≡ se indica a qué se conectan los terminales de un componente *C*0 en el circuito que se describe en esta arquitectura (es obligatorio utilizar una etiqueta con el fin de poder distinguirlo de otro componente *C*0 que se pueda utilizar en la descripción).

```
 u1: C1 PORT MAP(a => sel(0),b => sel(1),c => Data_in(1),f => aux1);
 u2: C2 PORT MAP(a => sel(0),b => sel(1),c => Data_in(2),f => aux2);
 u3: C3 PORT MAP(a => sel(0),b => sel(1),c => Data_in(3),f => aux3);
 u4: C4 PORT MAP (a => aux0,b => aux1,c => aux2,d => aux3,f => f);
end Structural;
```

Como se puede apreciar en el ejemplo anterior, la descripción *structural* no resulta adecuada a la hora de describir circuitos sencillos. En la práctica, se suele utilizar una combinación de los tres estilos descritos: *dataflow*, *behavioral* y *structural*.