GUÍA BÁSICA DEL VHDL

1. INTRODUCCION.

Dentro de la secuencia de diseño electrónico (fig. 1.1), la primera tarea, después de concebir la idea, es realizar una descripción de lo que se pretende hacer. Las computadoras personales ofrecen hoy día herramientas para la creación y verificación de diseños. Con dichas herramientas es posible describir tanto un circuito sencillo que represente una compuerta como un circuito complejo que defina un procesador digital de señales o un microprocesador.

Al principio, las herramientas CAD (Diseño Asistido por Computadora) se limitaban a servir de meros instrumentos de dibujo para poder realizar el diseño. Con la incorporación de herramientas de fabricación de circuitos impresos e integrados, simuladores, etc., la descripción del circuito empieza a desempeñar un papel más importante, ya que sirve como entrada de información a las herramientas posteriores en el flujo de diseño. Esto, junto con la metodología Top-Down de diseño de circuitos electrónicos (proceso de capturar una idea en un alto nivel de abstracción e implementarla partiendo de esta descripción abstracta y después ir hacia abajo incrementando el nivel de detalle según sea necesario), en contraposición a la metodología bottom-up, llevó a la aparición de herramientas de descripción que permiten al diseñador definir el problema de una forma abstracta, utilizando a la PC para materializar la idea.

Las herramientas CAD actuales permiten las siguientes posibilidades de abordar la descripción de una idea o diseño electrónico:

Descripción Estructural. Consiste en enumerar los componentes de un circuito y sus interconexiones. De acuerdo a la herramienta que se utilice se tienen dos formas de hacerlo:

Esquemas. Consiste en la descripción gráfica de los componentes de un circuito (método tradicional).

Lenguaje. Se realiza una enumeración de los componentes de un circuito así como su conexión.

Descripción Comportamental. Es posible describir un circuito electrónico (generalmente digital) simplemente describiendo como se comporta, utilizando un lenguaje de descripción de hardware apropiado.

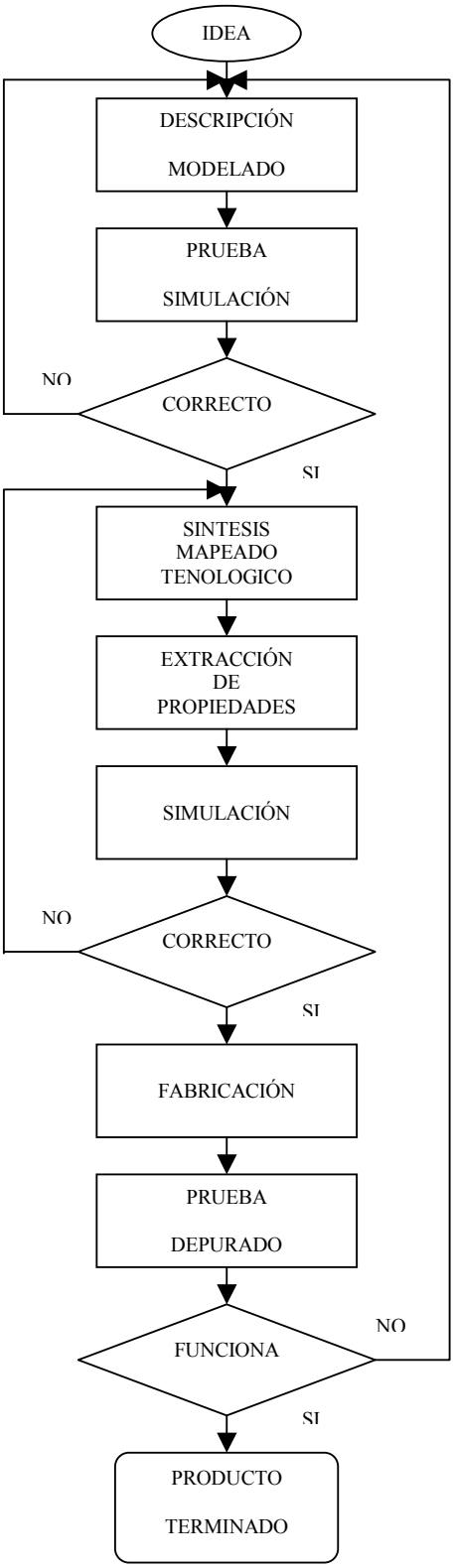

Figura 1.1. Flujo de diseño para sistemas electrónicos y digitales.

1.1 INTRODUCCIÓN AL LENGUAJE VHDL.

La forma más común de describir un circuito consiste en la utilización de esquemas, pero también existe la necesidad de disponer de una descripción del circuito que permita el intercambio de información entre las diferentes herramientas CAD que componen un ciclo de diseño.

Al principio se utilizó un lenguaje de descripción que permitía, mediante sentencias simples, describir completamente un circuito. A estos lenguajes se les llamó Netlist (conjunto de instrucciones que indican la interconexión de los componentes de un diseño, es decir, una lista de conexiones).

A partir de estos lenguajes simples, que ya son auténticos lenguajes de descripción de hardware (HDL), se descubrió el interés que puede tener el representar los circuitos directamente utilizando un lenguaje en vez de usar esquemas. Los esquemas desde el punto de vista humano son más sencillos de entender, pero el lenguaje siempre permite una edición más fácil y rápida.

Con una mayor sofisticación de las herramientas de diseño y con la puesta al alcance de todos de la posibilidad de fabricación de circuitos integrados y de lógica programable, fue apareciendo la necesidad de poder describir los circuitos con un alto grado de abstracción, no desde el punto de vista estructural, sino desde un enfoque funcional. Existía la necesidad de poder describir un circuito pero no desde el punto de vista de sus componentes, sino de acuerdo a su funcionamiento. Este nivel de abstracción se había alcanzado ya con las herramientas de simulación.

1.1.1 EL LENGUAJE VHDL.

Con la aparición de técnicas para la síntesis de circuitos a partir de un lenguaje de alto nivel, se utilizaron precisamente lenguajes de simulación, que si bien alcanzan un altísimo nivel de abstracción, su orientación es básicamente la de simular. De esta manera, los resultados de una síntesis a partir de descripciones con estos lenguajes no es siempre la más optima. Uno de los lenguajes de síntesis que se desarrolló a partir de los lenguajes de modelado y simulación lógica es el VHDL.

El significado de las siglas VHDL es “Very High Speed Integrated Circuit Hardware Description Language” (Lenguaje de Descripción de Hardware para Circuitos Integrados de Muy Alta Velocidad).

VHDL es un lenguaje de descripción y modelado diseñado para describir, en una forma en que los humanos y las máquinas puedan leer y entender, la funcionalidad y la organización de sistemas hardware digitales, placas de circuitos y componentes.

VHDL es un lenguaje con una sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento hardware. Permite el modelado preciso,

en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Los objetivos del lenguaje VHDL son el modelado (desarrollo de un modelo para la simulación de un circuito o sistema) y la síntesis (proceso en donde se parte de una especificación de entrada con un determinado nivel de abstracción y se llega a una realización más detallada, menos abstracta) de circuitos y sistemas electrónicos y digitales.

1.1.2 ALGUNAS VENTAJAS DEL USO DEL VHDL.

- VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de compuertas.

- Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizadas por diversas herramientas de síntesis para crear e implementar circuitos.

- Los módulos creados en VHDL pueden utilizarse en diferentes diseños, lo que permite la reutilización del código. Además, la misma descripción puede utilizarse para diferentes tecnologías sin tener que rediseñar todo el circuito.

- Al estar basado en un estándar (IEEE std 1076-1987, IEEE std 1076-1993) los ingenieros de toda la industria de diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.

- VHDL permite diseño Top-Down, esto es, describir (modelar) el comportamiento de los bloques de alto nivel, analizarlos (simularlos) y refinar la funcionalidad en alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.

- VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas (modularidad).

2. SINTAXIS DEL VHDL.

VHDL es verdaderamente un lenguaje, por lo que tiene sus elementos sintácticos, sus tipos de datos y sus estructuras como cualquier otro tipo de lenguaje.

2.1. ELEMENTOS SINTÁCTICOS.

Comentarios. Cualquier línea que empieza por dos guiones "-- " es un comentario.

Símbolos especiales. De un solo carácter: + - / * ( ) . , : ; & ´ " < > = | #
De dos caracteres: ** => := /= >= <= --

Identificadores. Se utilizan para dar nombre a variables, señales, nombres de rutinas, etc. puede ser cualquier nombre compuesto por letras y números, incluyendo el símbolo del subrayado "_"; nunca puede contener símbolos especiales o palabras claves del VHDL ni empezar por un número o subrayado; tampoco se permite que el identificador acabe con un subrayado ni que haya dos seguidos. Las mayúsculas o minúsculas son consideradas iguales.

Números. Cualquier número se considera que se encuentra en base 10. Se admite la notación científica convencional para números en punto flotante. Es posible manejar números en otras bases utilizando el símbolo del sostenido "#". Ejemplo: 2#11000100# y 16#c4# representan el entero 196.

Caracteres. Es cualquier letra o carácter entre comillas simples: ´1´, ´3´, ´t´ .

Cadenas. Son un conjunto de caracteres englobados por comillas dobles: "esto es una cadena".

Cadenas de bits. Los tipos bit y bit_vector son de tipo carácter y matriz de caracteres respectivamente. En VHDL se tiene una forma elegante de definir números con estos tipos y es mediante la cadena de bits. Dependiendo de la base en que se especifique el número, se puede poner un prefijo B (binario), O (octal), o X (hexadecimal). Ejemplos: B"11101001", O"126", X"FE".

Palabras reservadas. Son aquellas que tienen un significado especial. Son las instrucciones, órdenes y elementos que permiten definir sentencias. La tabla 1 muestra las palabras clave del VHDL`87 y la tabla 2 las que se agregaron en el VHDL`93.

2.2. OPERADORES Y EXPRESIONES.

2.2.1 OPERADOR DE CONCATENACIÓN.

& (concatenación). Concatena matrices de manera que la dimensión de la matriz resultante es la suma de las dimensiones de las matrices sobre las que opera. Por ejemplo, punto<=x&y construye la matriz punto con la matriz x en las primeras posiciones y la matriz y en las últimas.

Abs	configuration	inout	or	Then
Access	constant	is	others	To
After	disconnect	label	out	Transport
Alias	downto	library	package	Type
All	else	linkage	port	Units
And	elsif	loop	procedure	Until
Architecture	end	map	process	Use
Array	entity	mod	range	Variable
Assert	exit	nand	record	Wait
Attribute	file	new	register	When
Begin	for	next	rem	While
Block	function	nor	report	With
Body	generate	not	return	Xor
Buffer	generic	null	select	
Bus	guarded	of	severity	
Case	if	on	signal	
Component	in	open	subtype	

Tabla 1. Palabras clave del VHDL`87.

Group	postponed	ror	sra	
Impure	pure	shared	srl	
Inertial	reject	sla	unaffected	
Literal	rol	sll	xnor	

Tabla 2. Palabras clave agregadas en el VHDL`93.

2.2.2 OPERADORES ARITMÉTICOS.

** (exponencial). Eleva un número a una potencia: 4**2 es 4^2

abs( ) (valor absoluto). Devuelve el valor absoluto de su argumento.

***** (multiplicación). Multiplica dos números de cualquier tipo numérico.

/ (división). Funciona con cualquier dato numérico.

mod (módulo). Calcula el módulo de dos números.

rem (resto). Calcula el resto de una división entera.

+ (suma y signo positivo). Indica suma o signo.

- (resta y signo negativo). Indica resta y signo negativo.

2.2.3 OPERADORES DE DESPLAZAMIENTO.

Estos operadores solo existen en el VHDL`93.

sll, srl (desplazamiento lógico, a la izquierda o a la derecha). Desplaza un vector un número de bits a izquierda o derecha rellenando con ceros los huecos libres. Ejemplo: dato sll 2, desplaza a la izquierda dos posiciones al vector dato, es decir, lo multiplica por 4.

sla, sra (desplazamiento aritmético a la izquierda o a la derecha). La diferencia con el anterior es que este desplazamiento conserva el signo, es decir, conserva el valor que tiene el bit más significativo del vector.

rol, ror (rotación a la izquierda o a la derecha). Es como el desplazamiento, pero los huecos que se forman son ocupados por los bits que van saliendo.

2.2.4 OPERADORES RELACIONALES.

Devuelven siempre un valor de tipo booleano (true o false). Los tipos con los que pueden operar dependen de la operación.

=, /= (igualdad, desigualdad). El primero devuelve true si los operandos son iguales y false en caso contrario. El segundo indica desigualdad, así que funciona al revés. Los operandos deben ser del mismo tipo.

<, <=, >, >= (menor, mayor) Tienen el significado habitual. Los tipos de datos que pueden manejar son escalares o matrices de una sola dimensión de tipos discretos.

2.2.5 OPERADORES LÓGICOS.

Son **not**, **and**, **nand**, **or**, **nor**, **xor** y en el VHDL`93 se añadió **xnor**. Actuan sobre los tipos bit, bit_vector y boolean. En un vector, las operaciones son bit a bit.

2.2.6 PRECEDENCIA DE OPERADORES.

La siguiente tabla muestra la precedencia de los operadores. Los operadores que se encuentran en la misma fila tienen la misma precedencia, por lo tanto, en una expresión, se evaluarán primero siguiendo el orden de izquierda a derecha de la expresión. El uso de paréntesis altera la precedencia a conveniencia del programador.

Mayor |	**	abs	not			
|	*	/	mod	rem		
|	+ (signo)	- (signo)				
|	+	-	&			
|	=	/=	<	<=	>	>=
Menor |	and	or	nand	nor	xor	
xnor						

Tabla 3. Precedencia de los operadores en VHDL.

2.3. TIPOS DE DATOS.

La sintaxis del VHDL es estricta con respecto a los tipos. Cualquier objeto en VHDL debe tener un tipo. Aquí no existen tipos propios del lenguaje, lo que tiene son los mecanismos para poder definir cualquier tipo. Normalmente al compilar un programa, se cargan los tipos de datos que se encuentran en una librería. A continuación se presentan las declaraciones de los dos grupos de tipos de datos del VHDL: los escalares y los compuestos.

2.3.1 TIPOS ESCALARES.

Son tipos simples que contienen algún tipo de magnitud. Se muestra a continuación la forma de declararlos, así como algunos predefinidos.

Enteros. Datos cuyo contenido constituye un valor numérico entero. Se definen con la palabra **range**. No se dice que es un dato entero, sino que está comprendido en cierto intervalo especificando los límites de dicho intervalo con valores enteros. Ejemplos:

type byte **is range** 0 **to** 255;
type index **is range** 7 **downto** 1;
type integer **is range** –2147483648 **to** 2147483647; --Predefinido.

Reales. Números en punto flotante, definidos igualmente con la palabra **range**. Ejemplos:

type nivel **is range** 0.0 **to** 5.0;
type real **is range** –1.0E38 **to** 1.0E38; --Predefinido.

Físicos. Datos que corresponden a magnitudes físicas. Tienen un valor y unas unidades. Ejemplos:

```
type longitud is range 0 to 1.0E9
        units
                um;
                mm = 1000 um;
                m = 1000 mm;
                inch = 25.4 mm;
        end units;
```

Existe un tipo físico predefinido en VHDL que es **time**. Este tipo se utiliza para indicar retrasos y tiene los submúltiplos, desde fs (femtosegundos) hasta hr (horas). Cualquier dato físico se escribe siempre con su valor seguido de la unidad: 10 mm, 1 us, 23 ns.

2.3.2 TIPOS COMPUESTOS.

Son tipos de datos que están compuestos por tipos escalares.

Matrices. Colección de elementos del mismo tipo a los que se accede mediante un índice. Las hay monodimensionales o multidimensionales. Pueden estar enmarcadas en un rango o el índice puede ser libre teniendo la matriz una dimensión teórica infinita. Ejemplos:

type word **is array**(31 **downto** 0) **of** bit;
type transformada **is array**(1 **to** 4, 1 **to** 4) **of** real;
type positivo **is array**(byte **range** 0 to 127) **of** integer;
type string **is array**(positive **range** <>) **of** character; -- Predefinido.
type bit_vector **is array**(natural **range** <>) **of** bit; --Predefinido.
type vector **is array**(integer **range** <>) **of** real;

Los tres últimos ejemplos muestran una matriz cuyo índice no tiene rango sino que sirve cualquier entero. Más tarde, en la declaración del dato, se deberán poner los límites de la matriz: **signal** dato: vector(1 **to** 20);

A los elementos de una matriz se accede mediante el índice: dato(3) es el elemento 3 de la matriz dato. De la misma manera se puede acceder a un rango: datobyte <= datoword(2 **to** 9). También se pueden utilizar en las asignaciones lo que se conoce como agregados o conjuntos (**aggregate**), que no es más que una lista separada por comas de manera que al primer elemento de la matriz se le asigna el primer elemento de la lista y así sucesivamente. Ejemplos.

semaforo <= (apagado, encendido, apagado);
dato <= (datohigh, datolow);
triestado <= (**others** => ´z´);

En este último se ha empleado la palabra **others** para poner todos los bits del triestado a ´z´, de esta forma se puede poner un vector a un valor sin necesidad de saber cuántos bits tiene la señal.

Registros. Es equivalente al tipo registro o record de otros lenguajes. Ejemplo.

```
type trabajador is
        record
                Nombre: string;
                Edad: integer;
        end record;
```

Para referirse a un elemento dentro del registro se utiliza la misma nomenclatura que en Pascal, es decir, se usa un punto entre el nombre del registro y el nombre del campo: persona.nombre = “Carlos”.

2.3.3 SUBTIPOS DE DATOS.

VHDL permite la definición de subtipos que son restricciones o subconjuntos de tipos existentes. Se tienen dos alternativas. La primera son los subtipos obtenidos a partir de la restricción de un tipo escalar a un rango. Ejemplos:

subtype raro **is** integer **range** 4 **to** 7;
subtype digitos **is** character **range** ´0´ **to** ´9´;
subtype natural **is** integer **range** 0 **to** entero_mas _alto; -- Predefinido.
subtype positive **is** integer **range** 1 **to** entero_mas_alto; -- Predefinido.

La segunda alternativa de subtipos son los formados por aquellos que restringen el rango de una matriz. Ejemplos:

subtype id **is** string (1 **to** 20);
subtype word **is** bit_vector(31 **downto** 0);

La ventaja de utilizar subtipos es que las mismas operaciones que sirven para el tipo también sirven para el subtipo. Esto tiene especial importancia cuando se describe un

circuito para ser sintetizado, ya que si se utiliza **integer** sin más, esto se interpretará como un bus de 32 líneas (según la herramienta que se utilice) y lo más probable es que se requieran muchas menos.

2.4 ATRIBUTOS.

Los elementos en VHDL, como señales, variables, etc. pueden tener información adicional llamada atributos. Estos atributos están asociados a estos elementos del lenguaje y se manejan en VHDL mediante la comilla simple ´ . Por ejemplo, t´left indica el atributo ´left de t que, en este caso, es un tipo escalar (este atributo indica el límite izquierdo del rango).

Algunos de estos atributos están predefinidos en el lenguaje, mostrándose a continuación los más interesantes. Suponiendo que t es un tipo enumerado, entero, flotante o físico, se tienen los siguiente atributos:

t´left Límite izquierdo del tipo t.

t´right Límite derecho del tipo t.

t´low Límite inferior del tipo t.

t´high Límite superior del tipo t.

Suponiendo un tipo t como el anterior, un miembro x de ese tipo y un entero N, se pueden utilizar los siguientes atributos.

t´pos(x) Posición de x dentro del tipo t.

t´val(N) Elemento N del tipo t.

t´leftof(x) Elemento que está a la izquierda de x en t.

t´rightof(x) Elemento que está a la derecha de x en t.

t´pred(x) Elemento que está delante de x en t.

t´succ(x) Elemento que está detrás de x en t.

Si a es un tipo matriz, u un elemento de éste y N es un entero desde 1 hasta el número de dimensiones de la matríz, se pueden usar los siguientes atributos:

a´left(N) Límite izquierdo del rango de dimensión N de a.

a´right(N) Límite derecho del rango de dimensión N de a.

a´low(N) Límite inferior del rango de dimensión N de a.

a´high(N) Límite superior del rango de dimensión N de a.

a´range(N) Rango del índice de dimensión N de a.

a´length(N) Longitud del índice de dimensión N de a.

Suponiendo que s es una señal, se pueden utilizar los siguientes atributos (se han elegido los más interesantes):

s´event Devuelve true si se ha producido un cambio en la señal s.

s´stable(tiempo) Devuelve true si la señal estuvo estable durante el último periodo tiempo.

El atributo ´event es especialmente importante en la definición de circuitos secuenciales para detectar flancos de subida o bajada del reloj. Por esto es probablemente el atributo más utilizado en VHDL.

2.4.1 ATRIBUTOS DEFINIDOS POR EL USUARIO.

En muchas ocasiones resulta muy interesante agregar información adicional a los objetos que se están definiendo en VHDL. Así, se pueden poner propiedades o atributos a las entidades, arquitecturas, configuraciones, procedimientos, funciones, paquetes, tipos, subtipos, constantes, señales, variables, componentes y etiquetas de instrucción.

Lo que devuelve un atributo definido por el usuario es siempre una constante. Para definir el atributo primero hay que declarar ese atributo, diciendo que tipo de elemento devuelve y luego especificar el valor que devuelve en cada caso. La forma general es:

-- Declaración.
attribute nombre : tipo;

-- Especificación.
attribute nombre **of** id_elemento : clase_elemento **is** valor;

Ejemplos:

signal control: std_logic;
component ne555 ... **end component**;

```
attribute pin: integer;
attribute encapsulado: string;
attribute pin of control: signal is 5;
attribute encapsulado of ne555: component is "dip8";
begin
        if control´pin > 4 then ...
        if ne555´encapsulado = "dip8" then ...
end;
```

2.5 DECLARACIÓN DE CONSTANTES, VARIABLES Y SEÑALES.

Un elemento en VHDL contiene un valor de un tipo especificado. Hay tres tipos de elementos en VHDL: las variables, las señales y las constantes. Las variables y constantes son similares a lo que se encuentra en otros lenguajes. Las señales, en cambio, son elementos con un significado enfocado a la descripción de hardware.

2.5.1 CONSTANTES.

Una constante es un elemento que se inicializa a un valor determinado y que no puede ser cambiado una vez inicializado. Ejemplos:

```
constant e : real := 2.71828;
constant retraso : time := 10 ns;
constant max_size : natural;
```

En la última sentencia la constante max_size no tiene nungún valor asociado. Esto se permite siempre y cuando el valor sea declarado en algún otro sitio y se hace así para las declaraciones en paquetes que se verán más adelante.

2.5.2 VARIABLES.

Una variable en VHDL es similar al concepto de variable en otros lenguajes. Ejemplos:

```
variable contador : natural := 0;
variable aux : bit_vector(31 downto 0);
```

Es posible, dado un elemento previamente definido, cambiarle el nombre o ponerle nombre a una parte. Esto se realiza mediante la instrucción **alias** que suele resultar muy útil. Ejemplos:

variable instrucción : bit_vector(31 **downto** 0);
alias codigo_op : bit_vector(7 **downto** 0) **is** instrucción(31 **downto** 24);

2.5.3 SEÑALES.

Las señales se declaran igual que las constantes y variables, con la diferencia de que las señales pueden ser **normal**, **register** o **bus**. Si en la declaración no se especifica nada, se entiende que la señal es de tipo **normal**; para que sea de tipo **bus** o **register** se debe declarar explícitamente con las palabras clave **bus** o **register**.

Una señal no es lo mismo que una variable, La señal no es exactamente un objeto del lenguaje que guarda un valor, sino que lo hace en realidad es guardar un valor (o varios como se verá más adelante) y hacerlo visible en el momento adecuado. Esto es, se puede decir que la señal tiene dos partes, una donde se escribe y que almacena el valor y otra que se leé y que no tiene por qué coincidir con lo que se acaba de escribir.

Internamente se establece una conexión entre la parte que se escribe y la que se lee. Es posible en VHDL desconectar una parte de la otra, de manera que al leer la señal no se tenga acceso a lo que se escribió. La desconexión de una señal se hace asignándole el valor **null**. La diferencia entre una señal **normal**, **register** o **bus** viene del comportamiento al desconectarse. Las de tipo **normal** no se pueden desconectar. Las de clase **bus** y **register** se pueden desconectar, ya que o bien se trata de señales de tipo resuelto, que admiten varios o ningún valor escrito a un tiempo, o bien de tipo vigiladas, que tienen una condición de desconexión. La diferencia entre ambas es que la señal de tipo **bus** tiene un valor por defecto cuando todas las fuentes de la señal están desconectadas y las de clase **register** no tienen un valor por defecto pero conservan el último valor que se escribió.

Al igual que en variables y constantes, a las señales también se les puede dar un valor inicial si se quiere. Ejemplos:

signal selec: bit := ´0´;
signal datos: bit_vector(7 **downto** 0) **bus** := B”00000000”;

2.5.4 COMPARACIÓN ENTRE CONSTANTES, SEÑALES Y VARIABLES.

Constantes, señales y variables, son cosas diferentes. Las variables sólo tienen sentido dentro de un **process** o subprograma, es decir, solo tienen sentido en entornos de programación donde las sentencias son ejecutadas en serie. Las señales pueden ser declaradas únicamente en las arquitecturas, paquetes (**package**) o en los bloques concurrentes (**block**). Las constantes pueden ser habitualmente declaradas en los mismos sitios que las variables y señales.

Las variables son elementos abstractos del lenguaje, lo que significa que no tienen por qué tener una concordancia física real inmediata. Las señales poseen un significado físico inmediato y es el de representar conexiones reales en el circuito; son visibles por

todos los procesos y bloques dentro de una arquitectura, por lo que en realidad representan interconexiones entre bloques dentro de la arquitectura.

En un diseño las conexiones físicas entre unos elementos y otros son habitualmente declaradas como señales. Las entradas y salidas, definidas en la entidad, son, por lo tanto, consideradas como señales. Aunque estas entradas y salidas son en realidad señales, hay algunas diferencias; por ejemplo, las salidas no se pueden leer y a las entradas no se les puede asignar un valor.

La diferencia principal entre variables y señales es que una asignación a una variable se realiza de forma inmediata. La señal, en cambio, no recibe el valor que se le está asignando hasta el siguiente paso de simulación.

2.6 ELEMENTOS BÁSICOS DEL VHDL.

Una descripción bajo VHDL se compone de las siguientes partes:

- Declaración de librerías.
- Terminales de entrada y salida.
- Arquitectura.
- Banco de pruebas.

El VHDL, al partir de lenguajes de simulación es un lenguaje complejo y poco intuitivo. Esto es cierto particularmente para el banco de pruebas, pero esta sección se puede omitir sin problemas ya que las herramientas CAD cuentan con simuladores gráficos de formas de onda, mucho más simples y con la misma funcionalidad.

2.6.1 DECLARACIÓN DE LIBRERÍAS.

Existe un mecanismo para incorporar elementos de bibliotecas externas y hacerlos visibles al diseño que se está llevando a cabo: la cláusula **library**. Otra cláusula (**use**) permite hacer visibles los elementos internos a los paquetes. Ejemplos.

library componentes; -- Hace visible la biblioteca componentes.
use componentes.logic.and2; -- Hace visible la compuerta “and2” del paquete “logic”.
use componentes.arith.**all**; -- Hace visibles todos los elementos del paquete “arith”.

En VHDL existen dos bibliotecas que son invocadas por default, la biblioteca work, que contiene las unidades de diseño que se están compilando y la biblioteca std, que contiene dos paquetes, el standard y el textio. El paquete standard contiene todas las definiciones de tipos y constantes vistas hasta ahora. Textio contiene tipos y funciones para el acceso a archivos de texto.

Existe una biblioteca que es la más empleada en la industria de los semiconductores y que prácticamente también es estándar, la biblioteca IEEE, con sus paquetes std_logic_1164 y std_logic_1164_ext. El primero contiene la definición de tipos y funciones para trabajar con un sistema de nueve niveles lógicos que incluyen los de tipo bit con sus fuerzas correspondientes, así como los de desconocido, alta impedancia, etc. El segundo paquete, el cual es una extensión del anterior, incorpora alguna función de resolución más, así como operaciones aritméticas y relacionales.

Es por ello que la primera parte de una descripción VHDL contiene la declaración de librerías utilizadas en el proyecto. Con el objeto de respetar la definición IEEE del VHDL se tratara lo más posible de utilizar solo la librería estándar 1164. Existen muchas otras librerías, pero solo se aplican para los dispositivos de algunas empresas, por lo que no son estándar.

Todas las descripciones del texto incluyen las siguientes dos líneas en la sección de librerías:

library IEEE;
use IEEE.std_logic_1164.**all**

Una excepción a lo anterior es cuando se describen memorias, donde se necesita utilizar dos librerías adicionales que también son parte del estándar 1164.

2.6.2 TERMINALES DE ENTRADA Y SALIDA: **ENTITY**.

La segunda sección de la descripción VHDL, la entidad, define la interfaz con el entorno del sistema, ya que contiene las terminales de entrada y salida que lleva un diseño. La entidad es la estructura que permite realizar diseños jerárquicos en VHDL (colección de módulos interconectados entre sí). La forma de declarar una entidad es:

entity nombre **is**
    [**generic** (lista de parámetros);]
    [**port** (lista de puertos);]
    [declaraciones]
[**begin**
    sentencias]
end [**entity**] [nombre];

Las sentencias **generic** y **port** son muy importantes a pesar de que son opcionales. **Generic** sirve para definir y declarar propiedades o constantes del módulo que está siendo declarado en la entidad. Con la palabra clave **port**, se definen las entradas y salidas del módulo que está siendo definido.

Existen cuatro tipos de terminales que pueden ser declaradas:

1 Entradas simples.
2 Salidas simples.
3 Salidas retroalimentadas.
4 Terminales bidireccionales.

Para ejemplificar la forma de dar de alta las terminales de un circuito, se toma como ejemplo la caja negra que aparece en la figura 2.

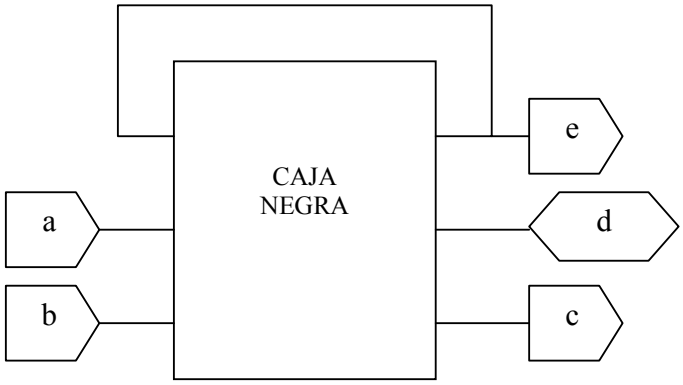

Figura 2. Caja negra con terminales de E/S.

La declaración de esta caja negra en VHDL, con sus correspondientes terminales, utilizando los tipos de la librería IEEE con su paquete std_logic_1164 es:

```
entity Caja_Negra is
        port (
                a ,b : in       std_logic; -- Entradas simples.
                c    : out      std_logic; -- Salida simple.
                d    : inout    std_logic; -- Terminal bidireccional.
                e    : buffer    std_logic  -- Salida retroalimentada.
            );
        end Caja_Negra;
```

La descripción de terminales inicia con la palabra reservada **entity,** continua con el nombre del circuito (en este caso, Caja_Negra) y la palabra reservada **is** y finaliza con la palabra reservada **end** y nuevamente el nombre del circuito con punto y coma. Las terminales son declaradas dentro de una estructura que comienza con la palabra reservada **port** seguida de paréntesis "(" y termina con otro paréntesis y punto y coma ");". Las entradas simples se declaran como **in**, las salidas simples como **out**, las terminales bidireccionales como **inout** y las salidas retroalimentadas como **buffer**.

Los comentarios se inician con dos guiones seguidos y concluyen hasta el final de la línea. Todas las declaraciones de las terminales terminan con punto y coma, excepto la última línea que no termina con punto y coma; ésta es una característica particular del VHDL.

2.6.3 ARQUITECTURA.

La tercera parte de la descripción consiste en la declaración de las asignaciones que definen la función del circuito deseado. El siguiente listado muestra la manera genérica de dar de alta una arquitectura.

architecture nombre_arquitectura **of** nombre_entidad **is**
    [declaraciones]
    begin
        [sentencias concurrentes]
    end [**architecture**] [nombre];

La arquitectura se inicia mediante la palabra reservada architecture seguida por el nombre de la descripción, la palabra reservada **of**, el nombre del circuito y la palabra reservada **is.** En la parte de declaraciones se definen los subprogramas (funciones, procedimientos, etc.), declaraciones de tipo, declaraciones de constantes, declaraciones de señales, declaraciones de alias, declaraciones de componentes, etc. Las asignaciones que describen al circuito se engloban entre un **begin** y un **end** seguido del nombre de la arquitectura.

Una arquitectura siempre está referida a una entidad concreta, por lo que no tiene sentido hacer declaraciones de arquitectura sin especificar la entidad. Una misma entidad puede tener diferentes arquitecturas. Es en el momento de la simulación o de la síntesis cuando se especifica qué arquitectura concreta se quiere simular o sintetizar.

2.6.4 EJEMPLOS DE ELEMENTOS BÁSICOS DEL VHDL.

A continuación se tienen varios ejemplos de descripción de hardware con VHDL para observar su uso básico.

Ejemplo 1. Compuerta NAND de dos entradas.

Realizar la descripción VHDL del circuito de la figura Ej.1.

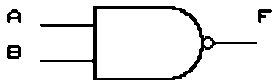

Figura Ej.1. Circuito simple.

Solución:

```
-- Libreria estandar

library IEEE;
 use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Compuerta_NAND is
   port(
      A, B : in  std_logic;  -- Entradas simples
      F:     out std_logic   -- Salida simple
      );
end Compuerta_NAND;

-- Descripcion del circuito

architecture simple of Compuerta_NAND is
   begin
      F <= A NAND B;  -- Compuerta NAND
   end simple;
```

Ejemplo 2. <u>Operadores lógicos.</u>

Realizar la descripción VHDL del circuito combinacional de la figura Ej.2.

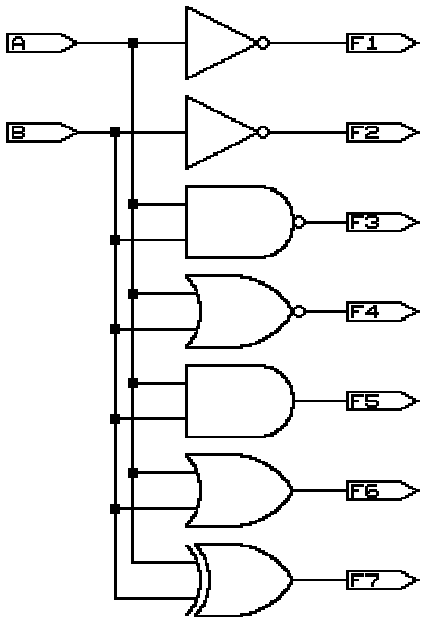

Figura Ej.2. Circuito combinacional con compuertas básicas.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Compuertas_basicas is
   port(
      A, B : in  std_logic;              -- Entradas simples
      F:     out std_logic_vector(1 to 7)-- 7 Salidas simples
      );
end Compuertas_basicas;
```

```
-- Descripcion del circuito

architecture simple of Compuertas_basicas is
   begin
      F(1) <= NOT A;      -- Compuerta NOT
      F(2) <= NOT B;      -- Compuerta NOT
      F(3) <= A NAND B;   -- Compuerta NAND
      F(4) <= A NOR B;    -- Compuerta NOR
      F(5) <= A AND B;    -- Compuerta AND
      F(6) <= A OR B;     -- Compuerta OR
      F(7) <= A XOR B;    -- Compuerta XOR
   end simple;
```

3. ESTILOS DE DESCRIPCIÓN EN VHDL.

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera se tiene especificado un circuito y se sabe como funciona. La segunda forma consiste en describir un circuito indicando lo que hace o cómo funciona, es decir, describiendo su comportamiento. El VHDL permite los dos tipos de descripciones.

La descripción comportamental se divide a su vez en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones: flujo de datos y algorítmica.

Es por ello que el VHDL presenta tres estilos de descripción de circuitos dependiendo del nivel de abstracción. El menos abstracto es una descripción puramente estructural (descripción estructural). Los otros dos estilos (descripciones flujo de datos y algorítmica) representan una descripción comportamental o funcional y la diferencia viene de la utilización o no de la ejecución serie (process).

3.1 DESCRIPCIÓN FLUJO DE DATOS (TRANSFERENCIA DE REGISTROS).

En esta se especifican las ecuaciones de transferencia entre diferentes objetos de VHDL, con una ejecución o interpretación de sentencias concurrente (paralela). Las sentencias, más que mandatos u órdenes, indican conexiones o leyes que se ejecutan no cuando les llega su turno sino continuamente. La mayoría de la descripción de flujo de datos tienen una correspondencia casi directa con su implementación hardware correspondiente. Se le llama flujo de datos puesto que **las instrucciones son todas de asignación**, siendo precisamente los datos los que gobiernan el flujo de ejecución de las instrucciones. Esta descripción también se conoce por las siglas RTL (Register Transfer Level).

Ya que los sistemas digitales tienen múltiples unidades funcionales que trabajan simultáneamente, todos los lenguajes que pretendan describir hardware deben ser como mínimo **concurrentes**.

La descripción de transferencia de registros se encontraría a mitad de camino entre una descripción puramente estructural y la puramente comportamental.

3.1.1 ESTRUCTURAS DE LA EJECUCIÓN FLUJO DE DATOS.

La instrucción básica de la ejecución concurrente es la asignación entre señales que viene gobernada por el operador <=. Para asignaciones complejas, se tienen instrucciones de alto nivel como condicionales, de selección, etc. Ejemplos:

Ejemplo 3. <u>Función lógica simple (descripción flujo de datos).</u>

Definir la descripción VHDL de la siguiente función lógica simple.

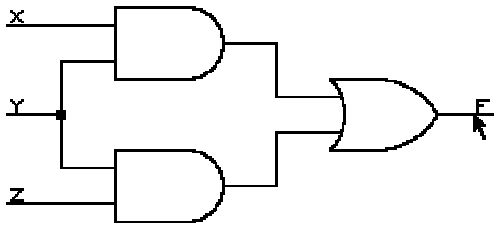

Figura Ej.3. Función lógica simple.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Funcion_simple is
   port(
      X, Y, Z : in  std_logic;  -- 3 Entradas simples
      F:        out std_logic   -- Salida simple
      );
end Funcion_simple;

-- Descripcion del circuito

architecture Basica of Funcion_simple is
   begin
      F <= (X AND Y) OR (Y AND Z);   -- Funcion AND-OR
   end Basica;
```

Ejemplo 4. <u>Multiplexor cuadruple 4-1 (descripción flujo de datos).</u>

Realizar la descripción VHDL del siguiente multiplexor cuadruple 4-1, utilizando asignación directa.

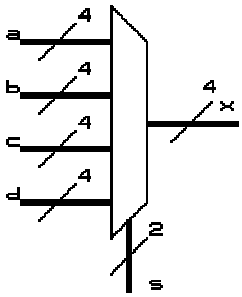

Figura Ej.4. Multiplexor cuadruple 4-1.

<u>Solución:</u>

```
-- Libreria estandar

library ieee;
use ieee.std_logic_1164.all;

-- Descripcion en caja negra

entity mux is
     port(
          a, b, c, d:in std_logic_vector(3 downto 0);
          s:         in std_logic_vector(1 downto 0);
          x:         out std_logic_vector(3 downto 0)
          );
end mux;

-- Descripcion del circuito

architecture archmux of mux is
begin
    x(3) <=     (a(3) and not(s(1)) and not(s(0)))
            OR  (b(3) and not(s(1)) and s(0))
            OR  (c(3) and s(1) and not(s(0)))
            OR  (d(3) and s(1) and s(0));

    x(2) <=     (a(2) and not(s(1)) and not(s(0)))
            OR  (b(2) and not(s(1)) and s(0))
            OR  (c(2) and s(1) and not(s(0)))
            OR  (d(2) and s(1) and s(0));
```

```
    x(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));

    x(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 archmux;
```

Ejemplo 5. Compuerta AND de 5 entradas (descripción flujo de datos).

Este ejemplo de asignación directa muestra como usar atributos para asignar pins. Las señales que no son asignadas a pins pueden ser automáticamente asignadas por el software. Este diseño utiliza el PLCC CYC371.

Solución:

```
-- Libreria estandar

library ieee;
use ieee.std_logic_1164.all;

-- Descripcion en caja negra

entity and5gate is
     port(
           a: in  std_logic_vector(0 to 4);
           f: out std_logic
          );

     attribute part_name of and5gate:entity is "c371";
     attribute pin_numbers of and5gate:entity is
               --espaciós despues de 3 y 5 son necesarios.
                 "a(0):2 a(1):3 "
               & "a(2):4 a(3):5 "--concatenacion (operador &)
               & "f:6";
               --la señal 4 sera asignada a pin por software.
end and5gate;

-- Descripcion del circuito

architecture see of and5gate is
  begin
     f <= a(0) and a(1) and a(2) and a(3) and a(4);
  end see;
```

Ejemplo 6. <u>Comparador de igualdad (descripción flujo de datos).</u>

Describir en VHDL con asignación directa el siguiente comparador, que tiene salida igual a 1 cuando ambos vectores de entrada son iguales y cero en caso contrario.

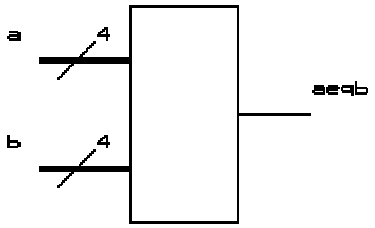

Figura Ej.6. Comparador de dos entradas de 4 bits.

<u>Solución:</u>

```
-- Libreria estandar

library ieee;
use ieee.std_logic_1164.all;

-- Descripcion en caja negra

entity compare is
     port(
          a, b:     in std_logic_vector(0 to 3);
          aeqb:     out std_logic
          );
end compare;

-- Descripcion del circuito

architecture archcompare of compare is
begin
     aeqb <= NOT(
          (a(0) xor b(0)) OR
          (a(1) xor b(1)) OR
          (a(2) xor b(2)) OR
          (a(3) xor b(3)));
end archcompare;
```

3.1.2 ASIGNACIÓN CONDICIONAL: **when..else**

Instrucción condicional que describe hardware de forma concurrente, donde es importante incluir todas las opciones y todos los casos de variación de una señal. Su sintaxis genérica es:

[id_instr:]
señal <= [opciones] {forma_de_onda **when** condicion **else**}
forma_de_onda [**when** condicion];

El "id_instr" es una etiqueta que sirve para identificar las instrucciones concurrentes. Su uso es opcional y suele utilizarse para comentar lo que hace la instrucción, para identificar por dónde va la ejecución durante la simulación o para establecer unos atributos propios para dicha instrucción. En algunas instrucciones concurrentes, esta etiqueta es obligatoria.

Las "opciones " que se ponen justo delante del valor a asignar tienen que ver con una expresión de vigilancia (palabra **guarded**) y con el tipo de retraso que se quiera realizar sobre dicha señal (palabras **transport**, **inercial** y **reject**).

Se pueden anidar varias condiciones en una misma asignación. Ejemplo:

s <= ´1´ **when** a = b **else**
´0´ **when** a > b **else**
´x´

Aunque es obligatorio asignar algo, sea cual sea el resultado de la condición, en el VHDL´93 se introdujo la palabra clave **unaffected** para permitir el caso en que no se realiza ninguna acción. Ejemplo:

Q1 <= d1 **when** nivel = ´1´ **else unaffected** -- VHDL´93

Ejemplo 7. <u>Multiplexor 2-1 (descripción flujo de datos).</u>

Utilizar la asignación condicional when..else para describir en VHDL el siguiente multiplexor 2-1.

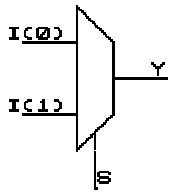

Figura Ej.7. Multiplexor 2-1.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Mux_2_1 is
   port(
      I : in  std_logic_vector(1 downto 0); -- Entradas
      S : in  std_logic;                    -- Seleccion
      Y : out std_logic                     -- Salida
      );
end Mux_2_1;

-- Descripcion del circuito

architecture simple of Mux_2_1 is
   begin
    Y <= I(0) when (S='0') else I(1);--Asignacion condicional
   end simple;
```

Ejemplo 8. <u>Multiplexor cuadruple 4-1 (descripción flujo de datos).</u>

Realizar la descripción VHDL del siguiente multiplexor cuadruple 4-1, utilizando la asignación condicional **when**..**else**.

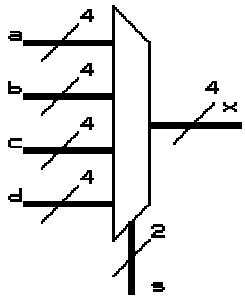

Figura Ej.8. Multiplexor cuadruple 4-1.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity mux is
     port(
     a, b, c, d:    in std_logic_vector(3 downto 0);
     s:             in std_logic_vector(1 downto 0);
     x:             out std_logic_vector(3 downto 0)
     );
end mux;

-- Descripcion del circuito

architecture archmux of mux is
begin
     x <= a when (s = "00") else
          b when (s = "01") else
          c when (s = "10") else
          d;
end archmux;
```

Ejemplo 9. <u>Demultiplexor 1-2 (descripción flujo de datos).</u>

Utilizar la asignación condicional when..else para describir en VHDL el siguiente demultiplexor 1-2.

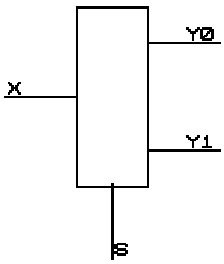

Figura Ej. 9. Demultiplexor 1-2.

<u>Solución:</u>

```
library IEEE;
use IEEE.std_logic_1164.all; -- Libreria estandar

-- Descripcion en caja negra

entity Demux_1_2 is
   port(
      X : in  std_logic;                    -- Entrada
      S : in  std_logic;                    -- Seleccion
      Y : out std_logic_vector(1 downto 0) -- Salida
      );
end Demux_1_2;

-- Descripcion del circuito

architecture simple of Demux_1_2 is
   begin
     Y(0) <= X when (S='0') else '0';--Asignacion condicional
     Y(1) <= X when (S='1') else '0';--Asignacion condicional
   end simple;
```

Ejemplo 10. Comparador de igualdad (descripción flujo de datos).

Describir en VHDL con la asignación condicional when..else el siguiente comparador, que tiene salida igual a 1 cuando ambos vectores de entrada son iguales y cero en caso contrario.

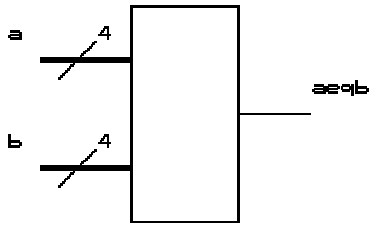

Figura Ej.10. Comparador de dos entradas de 4 bits.

Solución:

```
-- Libreria estandar

library ieee;
use ieee.std_logic_1164.all;

-- Descripcion en caja negra

entity compare is
     port(
          a, b: in std_logic_vector(0 to 3);
          aeqb: out std_logic
     );
end compare;

-- Descripcion del circuito

architecture archcompare of compare is
begin
     aeqb <= '1' when (a = b) else '0';
end archcompare;
```

Ejemplo 11. <u>Lógica de tres estados (descripción flujo de datos).</u>

Describir en VHDL los siguientes buffers de tres estados, uno con activación positiva y otro con activación negativa.

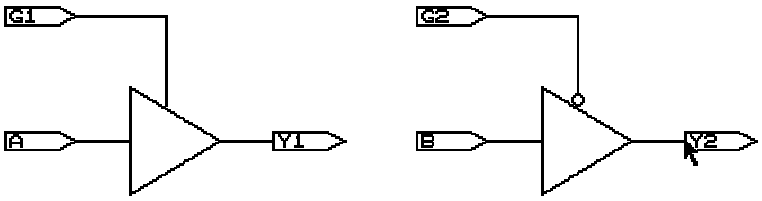

Figura Ej.11. Buffers de tres estados.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Tres_estados is
   port(
      A, B   : in  std_logic; -- Entradas logicas
      G1, G2 : in  std_logic; -- Terminales de control
      Y1, Y2 : out std_logic  -- Salidas
      );
end Tres_estados;

-- Descripcion del circuito

architecture simple of Tres_estados is
   begin
      Y1 <= A when (G1='1') else 'Z';--Asignacion condicional
      Y2 <= B when (G2='0') else 'Z';--Asignacion condicional
   end simple;
```

3.1.3 ASIGNACIÓN CON SELECCIÓN: **with**..**select**..**when**

Es una ampliación del condicional y es similar a las construcciones case o switch del Pascal o C. La asignación se hace según el resultado de cierta expresión. La expresión debe devolver siempre un tipo discreto que sea ennumerado o entero, o bien un tipo vector de una dimensión. La forma general es:

```
with expresion select
    Señal <= [opciones] forma_de_onda   when caso
                        {, forma_de_onda when caso} ;
```

Las opciones son las mismas que en la asignación condicional.

Existen varias posibilidades para el "caso". Por un lado puede ser alguno de los posibles valores que puede tomar la expresión, por otro lado se puede también especificar un rango de valores construido con **to** o **downto**; también se puede dar una lista de valores separados por el simbolo ´|´. Por último, también puede ser la palabra clave **others**. Ejemplo:

```
With estado select
    Semaforo <=  "rojo"          when "01",
                 "verde"         when "10",
                 "amarillo"      when "11",
                 "no funciona"   when others;
```

Es obligatorio, al igual que en la asignación condicional, incluir todos los posibles valores que pueda tomar la expresión. Cuando no se especifican todos los valores en las clausulas **when**, entonces hay que incluir la clausula **when others** que necesariamente debe ir al final. Si se desea no realizar asignación en determinados casos, se puede utilizar la palabra clave **unaffected**.

Ejemplo 12. <u>Demultiplexor 1-4 (descripción flujo de datos).</u>

Describir en VHDL un demultiplexor 1-4 utilizando la asignación con selección with..select..when. Realizar dos versiones de la solución.

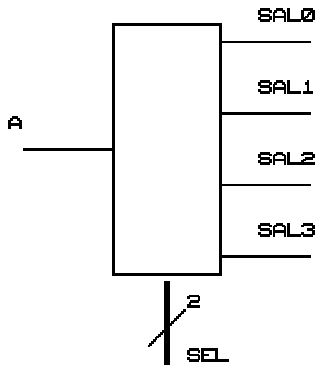

Figura Ej.12. Demultiplexor 1-4.

<u>Solución 1:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity demux is
     port(
          A:   in std_logic;
          sel: in std_logic_vector(1 downto 0);
          sal: out std_logic_vector(3 downto 0)
     );
end demux;

-- Descripcion del circuito

architecture flujo of demux is
     signal aux: std_logic_vector(3 downto 0);
begin
     with sel select
     aux  <=   "0001" when "00",
               "0010" when "01",
               "0100" when "10",
               "1000" when others;
     sal <= aux and A & A & A & A;
end flujo;
```

Solución 2:

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity demux is
     port(
          A:   in std_logic;
          sel: in std_logic_vector(1 downto 0);
          sal: out std_logic_vector(3 downto 0)
     );
end demux;

-- Descripcion del circuito

architecture flujo of demux is
begin
     with sel select
     sal  <=   "000"& A        when "00",
               "00"& A &"0"    when "01",
               "0"& A &"00"    when "10",
               A &"000"        when others;
end flujo;
```

Ejemplo 13. Circuito de desplazamiento (descripción flujo de datos).

Realizar la descripción RTL de un circuito que desplace un bit, a derecha o izquierda, un bus de entrada de 4 bits. El circuito está controlado por una señal de dos bits de manera que cuando esta señal es "00" no desplaza. Si es "01" el desplazamiento es hacia la izquierda. Si es "10" el desplazamiento es hacia la derecha y si es "11" se produce una rotación a la derecha. En el caso de desplazamientos se introduce un cero en el hueco que queda libre. Realizar una descripción con la estructura when..else y otra con la with..select.

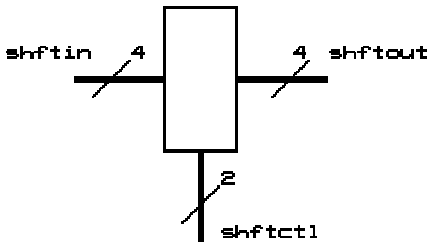

Figura Ej.13. Circuito de desplazamiento de 4 bits.

Solución con when..else:

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity shifter is
     port(
          shftin:   in   std_logic_vector(0 to 3);
          shftout: out std_logic_vector(0 to 3);
          shftctl:in     std_logic_vector(0 to 1)
          );
end shifter;

-- Descripcion del circuito

architecture flujo1 of shifter is
begin
  shftout <= shftin                 when shftctl = "00" else
             shftin(1 to 3)&"0"     when shftctl = "01" else
             "0"&shftin(0 to 2)     when shftctl = "10" else
             shftin(3)&shftin(0 to 2);
end flujo1;
```

Solución con with..select:

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity shifter is
     port(
          shftin:   in   std_logic_vector(0 to 3);
          shftout: out std_logic_vector(0 to 3);
          shftctl:in     std_logic_vector(0 to 1)
          );
end shifter;

-- Descripcion del circuito

architecture flujo2 of shifter is
begin
     with shftctl select
          shftout <=     shftin                    when "00",
                         shftin(1 to 3)&"0"        when "01",
                         "0"&shftin(0 to 2)        when "10",
                         shftin(3)&shftin(0 to 2) when others;
end flujo2;
```

3.2 DESCRIPCION ALGORÍTMICA.

Como la programación concurrente no es siempre la más cómoda para la descripción de ideas, sobre todo para sistemas muy complejos, el VHDL también permite una programación algorítmica que combina instrucciones que se ejecutan en serie con otras en operación concurrente. La programación serie se define dentro de bloques indicados con la palabra **process**. Por tanto, siempre que en VHDL se precise de una descripción en serie, se deberá utilizar un bloque de tipo **process**. Es sencillo utilizar este estilo de descripción con un alto nivel de abstracción ya que se acerca a la forma en que los seres humanos abordamos los problemas.

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

3.2.1 EL BLOQUE DE EJECUCIÓN SERIE: **process**.

Instrucción concurrente que define la ejecución serie de un grupo de sentencias. En un mismo programa puede haber múltiples bloques **process**. En el caso de que ocurra, cada uno de ellos equivale a una instrucción concurrente. Es decir, internamente a los **process** la ejecución de las instrucciones es serie, pero entre los propios bloques **process**, que pueden convivir con otras instrucciones concurrentes, la ejecución es concurrente. La forma en que se declara es:

[id_proc:]
[**postponed**] **process**[(lista_sensible)] [**is**]
    declaraciones
begin
    instrucciones_serie
end [**postponed**] **process** [id_proc]

El “id_proc” es una etiqueta opcional que sirve para ponerle nombre a los diferentes procesos de una descripción.

La “lista_sensible” es también opcional y contiene una lista de señales separadas por comas y encerradas entre paréntesis. La ejecución del **process** se activa cuando se produce un evento o cambio, en alguna de las señales de la lista sensible. En el caso de no existir lista sensible, la ejecución se controla mediante el uso de sentencias **wait** dentro del **process**. En cualquier caso debe existir, o bien una lista sensible, o una o más sentencias **wait**; de lo contrario se ejecutaría el proceso una y otra vez sin control.

La parte de “declaraciones” es parecida a la de otras estructuras, de forma que se pueden definir aquí variables, tipos, subprogramas, atributos, etc., **pero en ningun caso**

señales. Es interesante destacar que éste es el único lugar, aparte de los subprogramas, donde se pueden definir las variables.

Las "instrucciones_serie" se ubican entre el **begin** y el **end**. Estas presentan sus propios elementos sintácticos, siendo la asignación simple el único elemento común con la ejecución concurrente.

Por último, existe la posibilidad de "posponer" el proceso. Un proceso **postponed**, significa que debe posponerse su ejecución al final del paso de simulación actual.

3.2.2 SENTENCIA DE ESPERA: **wait**.

Los mecanismos para controlar la ejecución de un **process** y no dejar que este se ejecute en forma continua, sin control, son la inclusión de la "lista sensible" o con el uso de la instrucción **wait**.

Incluir una lista sensible equivale a la adición de una sentencia de espera al final del proceso, para que detenga la ejecución del **process** hasta que cambie alguna de las señales de la lista sensible. Su uso es suficiente para la mayoría de los procesos en un programa.

Otra forma, que es la más genérica, es con la palabra clave **wait**, la cual detiene la ejecución hasta que se cumple una condición o evento especificado en la propia sentencia. Su sintaxis es:

[id_wait:]
wait [**on** lista_sensible] [**until** condición] [**for** tiempo_límite];

En su forma más simple, se puede utilizar **wait** a solas. En este caso, cuando la ejecución llega a esta instrucción, se detiene para siempre.

Los tres elementos opcionales que se observan en la instrucción **wait** indican las condiciones bajo las cuales se reanudará la ejecución del bloque. Se puede especificar una, dos o las tres condiciones en un solo **wait**. Cualquiera que ocurra primero permitirá que se continúe con la ejecución del proceso.

La "lista_sensible" es simplemente una lista de señales separada por comas. Esto es completamente equivalente a la lista sensible del bloque **process**.

La "condición" es una condición que cuando se cumple permite que siga la ejecución.

El "tiempo_límite" indica un tiempo durante el cual la ejecución está detenida; cuando ese tiempo termina, sigue la ejecución. Ejemplos:

wait on pulso;
wait until contador > 7;

wait for 1 ns;
wait on interrupción **for** 25 ns;
wait on clk, sensor **until** contador = 3 **for** 100 ns;

A continuación se muestran dos procesos completamente equivalentes, uno con lista sensible y el otro con la instrucción **wait**:

```
-- Con lista sensible.
p1: process(a,b)
        begin
                a <= a + b + 2;
        end process p1;
```

```
-- Con wait.
p2: process
        begin
                a <= a + b + 2;
                wait on a,b;
        end process p2;
```

3.2.3 SENTENCIA DE SELECCIÓN: **case**.

Es la estructura típica que permite ejecutar unas sentencias u otras dependiendo del resultado de una expresión. Su forma sintactica es la siguiente:

```
[id_case:]
case expresión is
        when caso => sentencias;
        {when caso => sentencias;}
        [when others => sentencias;]
end case [id_case];
```

Esta instrucción es equivalente a la de **with..select** vista en el entorno de ejecución concurrente.

La expresión de selección tiene que ser o bien de tipo discreto o una matriz monodimensional de caracteres. Dependiendo de la expresión se ejecutaran unas sentencias u otras. No pueden haber dos casos duplicados porque marcaría error y todos los posibles casos de valores de la expresión deben estar contemplados en los diferentes **when**. Es por esto conveniente el uso de la palabra **others**, para indicar que se ejecuten ese conjunto de instrucciones, si la expresión toma un valor que no se contempla en ninguno de los casos anteriores.

Ejemplo 14. Tablas de verdad (descripción algorítmica).

La sentencia de selección **case** es útil para declara tablas de verdad. Para este ejemplo se desea describir en VHDL la siguiente tabla de verdad:

x2	x1	x0	y1	y0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

Tabla Ej.14. Tabla de verdad, 3 entradas, 2 salidas.

Solución:

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;


-- Descripcion en caja negra

entity Tabla_de_verdad is
   port(
      x : in  std_logic_vector(2 downto 0); -- Entradas
      f : out std_logic_vector(1 downto 0)  -- Salida
      );
end Tabla_de_verdad;



-- Descripcion de la arquitectura de la tabla
```

```
architecture facil of Tabla_de_verdad is
   begin
    process(x)
      begin
        case x is
          when "000"  => f <= "01"; --Asignacion para x = 000
          when "001"  => f <= "11"; --Asignacion para x = 001
          when "010"  => f <= "00"; --Asignacion para x = 010
          when "011"  => f <= "01"; --Asignacion para x = 011
          when "100"  => f <= "10"; --Asignacion para x = 100
          when "101"  => f <= "10"; --Asignacion para x = 101
          when "110"  => f <= "01"; --Asignacion para x = 110
          when others => f <= "11"; --Asignacion para x = 111
        end case;
      end process;
   end facil;
```

Ejemplo 15. <u>Multiplexor 8-1 (descripción algorítmica).</u>

Describir en VHDL, utilizando la sentencia **case**, el siguiente multiplexor de 8 a 1.

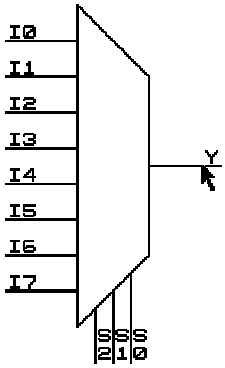

Figura Ej.15. Multiplexor de 8 a 1.

<u>Solución:</u>

```
library IEEE;
use IEEE.std_logic_1164.all; -- Libreria estandar

-- Descripcion en caja negra

entity Mux_8_1 is
   port(
      I : in  std_logic_vector(7 downto 0); -- Entradas
      S : in  std_logic_vector(2 downto 0); -- Seleccion
      Y : out std_logic                     -- Salida
      );
end Mux_8_1;

-- Descripcion del circuito

architecture simple of Mux_8_1 is
  begin
    process(S,I)     -- Declaracion del proceso
      begin
        case S is  -- Declaracion del case
           when "000"  => Y <= I(0); -- Asignacion para S=000
           when "001"  => Y <= I(1); -- Asignacion para S=001
           when "010"  => Y <= I(2); -- Asignacion para S=010
           when "011"  => Y <= I(3); -- Asignacion para S=011
           when "100"  => Y <= I(4); -- Asignacion para S=100
           when "101"  => Y <= I(5); -- Asignacion para S=101
           when "110"  => Y <= I(6); -- Asignacion para S=110
           when others => Y <= I(7); -- Asignacion para S=111
        end case;  -- Fin del case
      end process; -- Fin del proceso
  end simple;
```

Ejemplo 16. <u>Multiplexor cuadruple de 4-1 (descripción algorítmica).</u>

Realizar la descripción VHDL del siguiente multiplexor cuadruple 4-1, utilizando la sentencia de selección **case**.

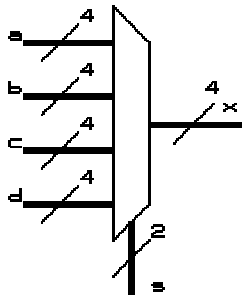

Figura Ej.16. Multiplexor cuadruple 4-1.

<u>Solución:</u>

```
library IEEE;
use IEEE.std_logic_1164.all; -- Libreria estandar

-- Descripcion en caja negra

entity Mux4_4_1 is
 port(
     a,b,c,d : in std_logic_vector(3 downto 0); -- Entradas
     s :       in std_logic_vector(1 downto 0); -- Selección
     x :       out std_logic_vector(3 downto 0) -- Salidas
     );
end Mux4_4_1;

-- Descripcion del circuito

architecture cuadruple of Mux4_4_1 is
   begin
     process(s,a,b,c,d) -- Declaracion del proceso
       begin
         case s is      -- Declaracion del case
           when "00"   => x <= a;  -- Asignacion para s = 00
           when "01"   => x <= b;  -- Asignacion para s = 01
           when "10"   => x <= c;  -- Asignacion para s = 10
           when others => x <= d;  -- Asignacion para s = 11
         end case;      -- Fin del case
       end process;     -- Fin del proceso
   end cuadruple;
```

Ejemplo 17. <u>Demultiplexor 1-4 (descripción algoritmica).</u>

Describir en VHDL un demultiplexor 1-4 utilizando la sentencia de selección **case**.

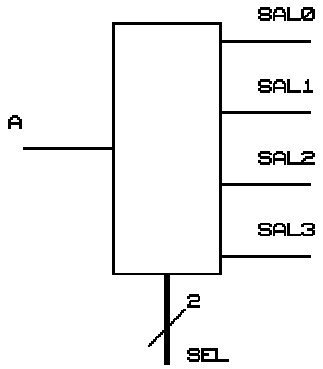

Figura Ej.17. Demultiplexor 1-4.

<u>Solución:</u>

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Demux_1_4 is
   port(
      A   : in  std_logic;                      -- Entrada
      SEL : in  std_logic_vector(1 downto 0); -- Seleccion
      SAL : out std_logic_vector(3 downto 0)  -- Salidas
      );
end Demux_2_4;

-- Descripcion del circuito

architecture simple of Demux_1_4 is
  begin
    process(A,SEL)
      begin
        SAL <= "0000"; -- Valores por omision
        case SEL is
          when "00" => SAL(0) <= A; -- Asignacion concurrente
          when "01" => SAL(1) <= A; -- Asignacion concurrente
          when "10" => SAL(2) <= A; -- Asignacion concurrente
          when others=>SAL(3) <= A; -- Asignacion concurrente
        end case;
      end process;
  end simple;
```

Ejemplo 18. <u>Demultiplexor 1-8 con lógica negativa (descripción algoritmica).</u>

Describir en VHDL un demultiplexor 1-8 con lógica negativa utilizando la sentencia de selección **case**.

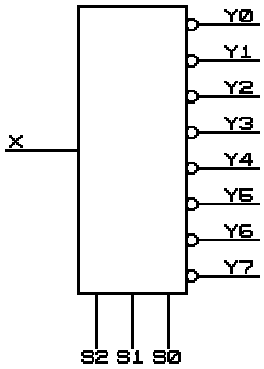

Figura Ej.18. Demultiplexor 1-8 con lógica negativa.

<u>Solución:</u>

```
library IEEE;                    -- Librería estandar
use IEEE.std_logic_1164.all;

entity Demux1-8 is               -- Descripcion en caja negra
     port(
          X : in  std_logic;                      -- Entrada
          S : in  std_logic_vector(2 downto 0); -- Seleccion
          Y : out std_logic_vector(7 downto 0)  -- Salidas
          );
end Demux1-8;

architecture simple of Demux1-8 is --Descripcion del circuito
   begin
      process(X,S)
         begin
            Y <= "11111111";        -- Valores por omision
            case S is
               when "000"  => Y(0) <= not X;
               when "001"  => Y(1) <= not X;
               when "010"  => Y(2) <= not X;
               when "011"  => Y(3) <= not X;
               when "100"  => Y(4) <= not X;
               when "101"  => Y(5) <= not X;
               when "110"  => Y(6) <= not X;
               when others => Y(7) <= not X;
            end case;
         end process;
   end simple;
```

Ejemplo 19. Decodificador de BCD a 7 segmentos (descripción algoritmica).

Realizar en VHDL un decodificador de BCD a 7 segmentos C.C. en forma algoritmica.

Solución:

La tabla siguiente muestra la relación entre los valores BCD y los 7 segmentos para un exhibidor de catodo común. Posteriormente se muestra la descripción VHDL deseada.

BCD	a	B	c	d	e	f	G
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	0	0	1	1

Tabla Ej.19. Tabla de verdad, decodificador BCD a 7 segmentos.

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity BCD_7_segmentos is
 port(
   B: in  std_logic_vector(3 downto 0); -- Entrada BCD
   S: out std_logic_vector(7 downto 1)  -- Salida 7 segmentos
     );
end BCD_7_segmentos;

-- Descripcion del circuito

architecture Decodificador of BCD_7_segmentos is
   begin
      process(B)
         begin
           case B is
               when "0000" => S <= "1111110"; -- Conversion
               when "0001" => S <= "0110000"; -- de codigo
               when "0010" => S <= "1101101"; -- S(7) segm. a
               when "0011" => S <= "1111001"; -- S(1) segm. g
               when "0100" => S <= "0110011";
               when "0101" => S <= "1011011";
               when "0110" => S <= "1011111";
               when "0111" => S <= "1110000";
               when "1000" => S <= "1111111";
               when "1001" => S <= "1110011";
               when others => S <= "-------"; -- No importa
           end case;
         end process;
   end Decodificador;
```

3.2.4 SENTENCIA CONDICIONAL: **if..then..else.**

Es la estructura típica para realizar una acción u otra según el resultado de una expresión booleana, siendo equivalente en significado a estructuras del mismo tipo en otros lenguajes. La forma general es:

```
[id_if:]
if condición then
  sentencias
{elsif condición then
   sentencias}
[else
  sentencias]
end if [id_if]
```

Se observa que aquí también se respeta la posibilidad de anidar **if** consecutivos, mediante la palabra **elsif**.

Tanto el **else** como el **elsif** son opcionales, pero el utilizarlos o no tiene un impacto directo sobre el tipo de hardware que se está describiendo. Se utilizan cuando se describen dispositivos combinacionales y cuando se omiten, la descripción es secuencial, ya que se definen registros. El ejemplo siguiente muestra una descripción serie con esta sentencia:

```
-- Ejecución serie.
process(a,b,c)
 begin
        if  a > b then
                p <= 2;
        elsif a > c then
                p <= 3;
        elsif  (a=c and c=b) then
                p <= 4;
        else p <= 5;
        end if
 end process;
```

Como se ha manejado en los parrafos anteriores, los estilos de descripción en VHDL son equivalentes entre sí, diferenciandolos el grado de abstración que maneja cada uno y con ello la facilidad o no para el programador de describir sistemas complejos. Por ejemplo, un programa equivalente al anterior con instrucciones concurrentes es:

```
-- Ejecución concurrente.
P  <=   2 when a > b else
        3 when a > c else
        4 when (a=c and c=b) else
        5;
```

Para este caso particular, resulta más simple la ejecución concurrente. La regla general para que ambas estructuras sean equivalentes es que el **process** contenga, en su lista sensible, todas las señales que intervienen como argumentos en las asignaciones de la ejecución concurrente.

Ejemplo 20. Candado activado por nivel (descripción algoritmica).

Describir en VHDL un candado activado con un nivel alto en la terminal LD, para que la información de D pase a Q. En el estado inactivo (LD = 0) la salida Q mantiene su estado aunque cambie la información en D.

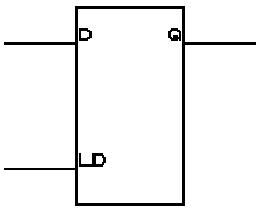

Figura Ej.20. Candado activado por nivel.

Solución:

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Candado is
   port(
      D  : in  std_logic; -- Entrada de datos
      LD : in  std_logic; -- Carga de datos
      Q  : out std_logic  -- Salida
      );
end Candado;

-- Descripcion del circuito

architecture Simple of Candado is
   begin
      process(D, LD)
         begin
            if (LD='1') then
               Q <= D;
            end if;
         end process;
   end Simple;
```

Ejemplo 21. Candado con borrado asincrono (descripción algoritmica).

Describir en VHDL un candado activado con un nivel alto en la terminal LD y que incluya una entrada de borrado asincrono, de acuerdo a la figura siguiente.

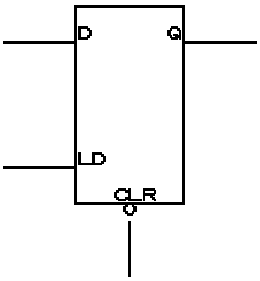

Figura Ej.21. Candado con borrado asincrono.

Solución:

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Candado_ca is
   port(
      D  : in  std_logic; -- Entrada de datos
      LD : in  std_logic; -- Carga de datos
      CLR: in  std_logic; -- Borrado
      Q  : out std_logic  -- Salida
      );
end Candado_ca;

-- Descripcion del circuito

architecture Simple of Candado_ca is
   begin
      process(D, LD, CLR)
         begin
            if (CLR='0') then
               Q <= '0';
            elsif (LD='1') then
               Q <= D;
            end if;
         end process;
   end Simple;
```

Ejemplo 22. Candado con borrado sincrono (descripción algoritmica).

Describir en VHDL un candado activado con un nivel alto en la terminal LD y que incluya una entrada de borrado sincrono, de acuerdo a la figura siguiente.

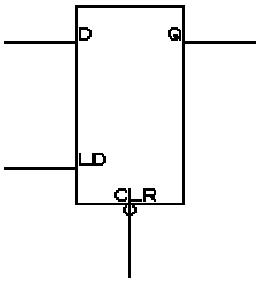

Figura Ej.22. Candado con borrado sincrono.

Solución:

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Candado_cs is
   port(
      D  : in  std_logic; -- Entrada de datos
      LD : in  std_logic; -- Carga de datos
      CLR: in  std_logic; -- Borrado
      Q  : out std_logic  -- Salida
      );
end Candado_cs;

-- Descripcion del circuito

architecture Simple of Candado_cs is
   begin
      process(D, LD, CLR)
         begin
            if (LD='1') then
               if (CLR='0') then
                  Q <= '0';
               else
                  Q <= D;
               end if;
            end if;
         end process;
   end Simple;
```

Ejemplo 23. <u>Flip-flop D con borde de disparo positivo (descripción algoritmica).</u>

Describir en VHDL un flip-flop D con borde de disparo positivo.

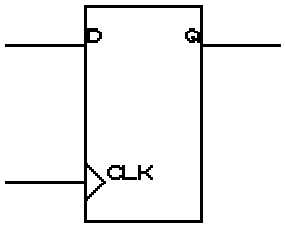

Figura Ej.23. Flip-flop D con borde de disparo positivo.

<u>Solución</u>:

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Flip_flop_d is
   port(
      D   : in  std_logic; -- Entrada de datos
      CLK : in  std_logic; -- Reloj
      Q   : out std_logic  -- Salida
      );
end Flip_flop_d;

-- Descripcion del circuito

architecture Simple of Flip_flop_d is
   begin
      process(CLK)
         begin
            if (CLK'event and CLK='1') then
               Q <= D;
            end if;
         end process;
   end Simple;
```

Ejemplo 24. Flip-flop D con borrado asincrono (descripción algoritmica).

Describir en VHDL un flip-flop D con borde de disparo positivo y borrado asincrono.

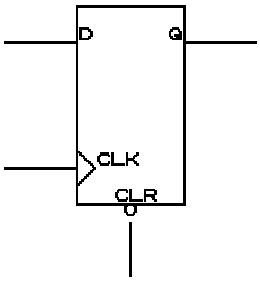

Figura Ej.24. Flip-flop D con borde de disparo positivo y borrado asincrono.

Solución:

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Flip_flop_d_ca is
   port(
      D   : in  std_logic; -- Entrada de datos
      CLK : in  std_logic; -- Reloj
      CLR : in  std_logic; -- Borrado
      Q   : out std_logic  -- Salida
      );
end Flip_flop_d_ca;

-- Descripcion del circuito

architecture Simple of Flip_flop_d_ca is
   begin
      process(CLK, CLR)
         begin
            if (CLR='0') then
               Q <= '0';
            elsif (CLK'event and CLK='1') then
               Q <= D;
            end if;
         end process;
   end Simple;
```

Ejemplo 25. <u>Multiplexor 2-1 (descripción algoritmica).</u>

Utilizar la estructura if para describir en VHDL el siguiente multiplexor 2-1.

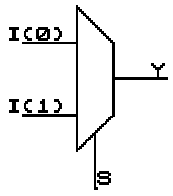

Figura Ej.25. Multiplexor 2-1.

<u>Solución:</u>

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Mux_2_1_if is
   port(
      I : in  std_logic_vector(1 downto 0); -- Entradas
      S : in  std_logic;                    -- Seleccion
      Y : out std_logic                     -- Salida
      );
end Mux_2_1_if;

-- Descripcion del circuito

architecture simple of Mux_2_1_if is
   begin
      process(S,I)        -- Declaracion del proceso
         begin
            if S='0' then -- Inicio del if
               Y <= I(0); -- Asignacion para S=0
            else
               Y <= I(1); -- Asignacion para S=1
            end if;       -- Fin del if
         end process;     -- Fin del proceso
   end simple;
```

3.3 DESCRIPCIÓN ESTRUCTURAL.

Aun cuando la mayoría de descripciones van a ser de flujo de datos o algorítmicas, por ser las más cercanas al pensamiento humano, resulta interesante conocer una descripción que va a permitir la incorporación al diseño de elementos de biblioteca, realización de diseños jerárquicos a partir de componentes, etc.

La descripción estructural es el estilo de VHDL más cercano al Netlist, por lo que soporta la definición y referencia (instanciación) de componentes y la especificación de interconexiones entre unos y otros. Este estilo lo utilizan sobre todo las herramientas CAD para intercambiar información entre ellas.

La descripción estructural en VHDL incluye las dos construcciones básicas: la de componentes y la forma en que se interconectan, es por ello que a continuación se mencionan brevemente la declaración de componentes y la interconexión entre ellos.

3.3.1 DEFINICIÓN DE COMPONENTES.

Los componentes se declaran con la palabra especial **component**. Esto es algo muy parecido a la entidad (entity). La declaración de componentes es:

component nombre [**is**]
    [**generic** (lista_parámetros);]
    [**port** (lista_puertos);]
end component [nombre];

3.3.2 ENLACE ENTRE COMPONENTES Y ENTIDADES.

Cada componente debe enlazarse con una entidad y una arquitectura predefinida, para que se pueda referenciar en el diseño tantas veces como sea necesario. El enlace entre componentes, entidades y arquitecturas se realiza mediante sentencias **for**. La sintaxis para este enlace es:

for lista_refs: nombre_componente
  use entity nombre_entidad(nombre_arquitectura) | **use configuration** nombre_config
  [**generic map** (parámetros)]
  [**port map** (puertos)];

3.3.3 REPETICION DE ESTRUCTURAS: **generate**.

En muchas ocasiones el hardware está compuesto por estructuras que se repiten una y otra vez. Para poder describir estructuras repetitivas, el VHDL tiene la instrucción

generate que repite, tantas veces como se diga, las sentencias especificadas en su interior. Las dos formas de esta sentencia son:

id_generate:
for parámetro_repetitivo **generate**
[declaraciones
begin]
[sentencias]
end generate [id_generate];

id_generate:
if condición **generate**
[declaraciones
begin]
[sentencias]
end generate [id_generate]

La forma más común es la de la izquierda. La etiqueta, al igual que en los **block**, es también obligatoria.

Esta sentencia es concurrente, por lo que no debe estar dentro de los **process** o subprogramas. Pero puede incluir cualquier instrucción concurrente, como sentencias de **block** o **process**.

3.3.4 DEFINICIÓN DE SEÑALES.

Las señales sirven para conectar unos componentes con otros. Se declaran igual que las señales de cualquier otro tipo de descripción y tienen exactamente el mismo significado.

3.3.5 REFERENCIA DE COMPONENTES.

Un mismo componente se puede reproducir (instanciar, referenciar replicar, copiar, etc.) tantas veces como se necesite en la arquitectura. Es una instrucción concurrente y su forma general es la siguiente:

ref_id:
[**component**] nombre_componente | **entity** nombre_entidad [nombre_arquitectura) |
configuration nombre_configuración]
[**generic map** (parámetros)]
[**port map** (puertos)];

3.3.6 CONFIGURACIÓN.

Algunas de las definiciones anteriores, especialmente la de enlace, se pueden especificar en un bloque especial llamado configuración. La forma general de la declaración de la configuración de una entidad es la siguiente:

configuration nombre_conf **of** nombre_entidad **is**
{sentencia_use | def_atributo | def_grupos}
configuración_bloque
end [**configuration**] [nombre_conf]

Ejemplo 26. <u>Lógica combinacional (descripción estructural).</u>

Modelar en VHDL el siguiente circuito combinacional utilizando el estilo de descripción estructural.

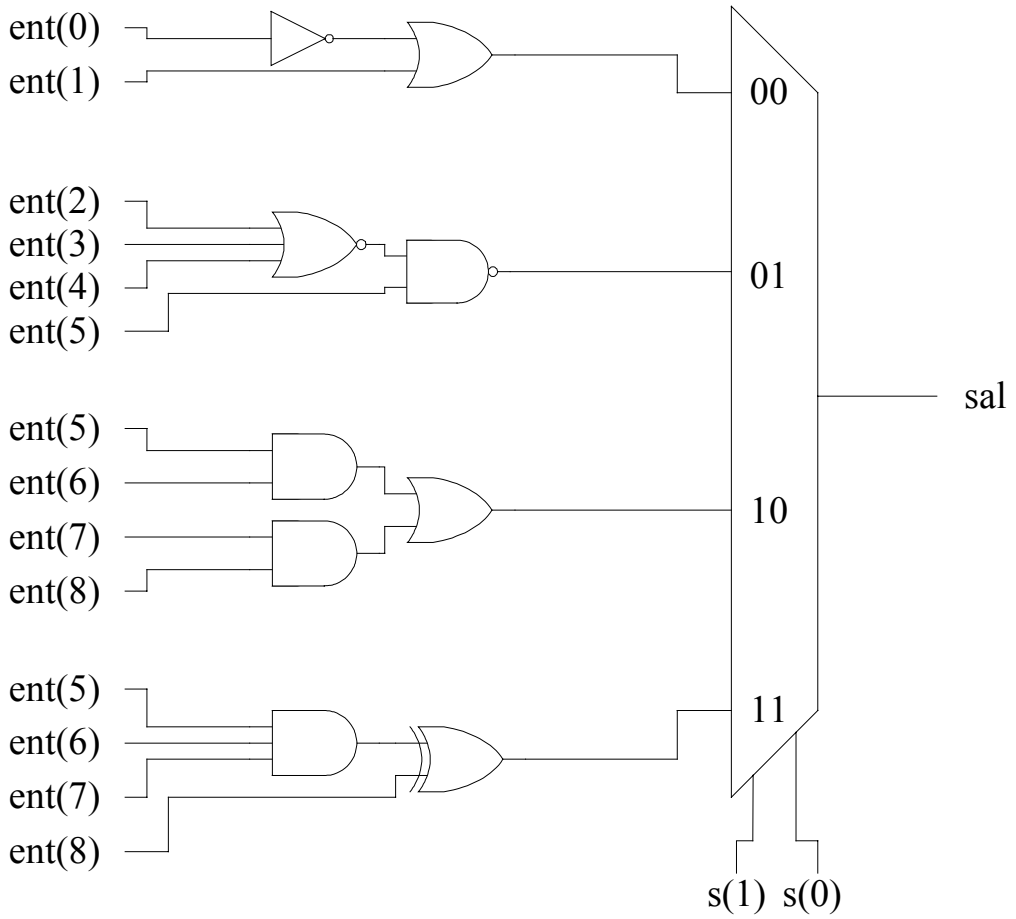

Figura A Ej. 26. Circuito lógico combinacional

<u>Solución:</u>

Como primer paso, se realiza la descomposición del circuito inicial, formando módulos funcionales (componentes) que puedan describirse, simularse y probarse uno a uno, bajo el ambiente del software de diseño en VHDL.

En este ejercicio se definen los componentes U0 a U4, de acuerdo a la siguiente figura:

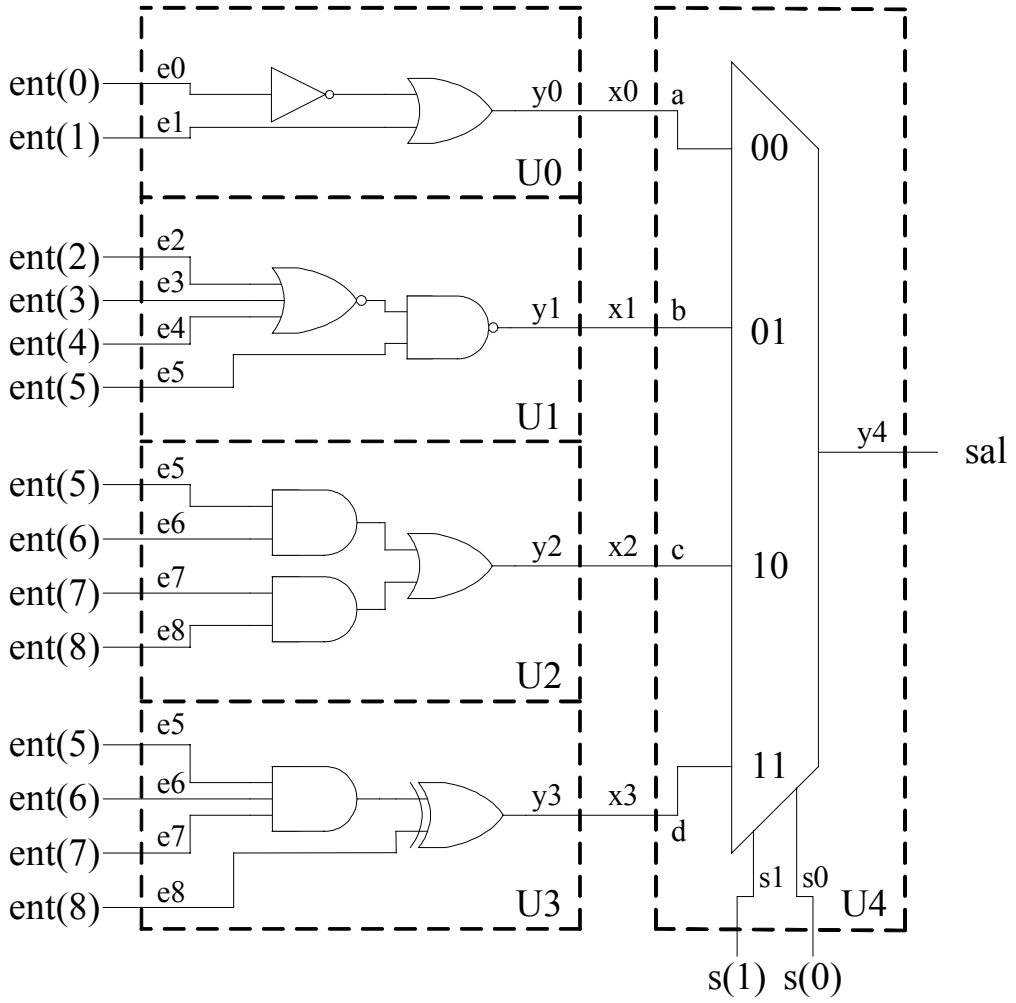

Figura B Ej. 26. Descomposición en componentes del circuito combinacional.

A continuación se realiza la descripción VHDL de cada componente, con la nomenclatura local de las señales de entrada y salida indicada en la figura anterior, tomando como base que cada uno de ellos se puede considerar funcionalmente independiente de los demás.

```
-- Circuito_U0.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Circuito_U0 is
   port(
```

```
      e0, e1: in  std_logic;            -- Entradas simples
      y0    : out std_logic             -- Salida simple
      );
end Circuito_U0;

-- Descripcion del circuito

architecture simple of Circuito_U0 is
   begin
      y0 <= (NOT e0) OR e1;
   end simple;


-- Circuito_U1.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Circuito_U1 is
   port(
      e2, e3, e4, e5: in  std_logic;    -- Entradas simples
      y1            : out std_logic     -- Salida simple
      );
end Circuito_U1;

-- Descripcion del circuito

architecture simple of Circuito_U1 is
   begin
      y1 <= (e2 NOR e3 NOR e4) NAND e5;
   end simple;


-- Circuito_U2.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Circuito_U2 is
```

```
   port(
      e5, e6, e7, e8: in  std_logic;    -- Entradas simples
      y2            : out std_logic     -- Salida simple
      );
end Circuito_U2;

-- Descripcion del circuito

architecture simple of Circuito_U2 is
   begin
      y2 <= (e5 AND e6) OR (e7 AND e8);
   end simple;


-- Circuito_U3.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity Circuito_U3 is
   port(
      e5, e6, e7, e8: in  std_logic;    -- Entradas simples
      y3            : out std_logic     -- Salida simple
      );
end Circuito_U3;

-- Descripcion del circuito

architecture simple of Circuito_U3 is
   begin
      y3 <= (e5 AND e6 AND e7) XOR e8;
   end simple;


-- Circuito_U4.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra
```

```
entity Circuito_U4 is
   port(
      a, b, c, d: in  std_logic;    -- Entradas simples
      s1, s0    : in  std_logic;    -- Entradas simples
      y4        : out std_logic     -- Salida simple
      );
end Circuito_U4;

-- Descripcion del circuito

architecture simple of Circuito_U4 is
   signal selección: std_logic_vector(1 downto 0);
   begin
      seleccion <= s1 & s0;
      y4 <= a when (selección = "00") else
            b when (selección = "01") else
            c when (selección = "10") else
            d;
   end simple;
```

Una vez modelados todos los componentes, se escribe el programa de nivel superior (top_level) donde se hace referencia (instancia) a cada uno de ellos y se interconectan de acuerdo al diagrama del sistema completo. Esto es equivalente a tener un C.I. por cada componente y montarlos y soldarlos en una placa de circuito impreso previamente diseñada.

```
-- Top_level.vhd

-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;


-- Descripcion en caja negra

entity top_level is
   port(
      ent   : in  std_logic_vector(8 downto 0); -- Entradas
      s     : in  std_logic_vector(1 downto 0); -- Seleccion
      sal   : out std_logic                     -- Salida
      );
end top_level;
```

```
-- Descripcion del circuito

architecture superior of top_level is

   -- Señales internas para la interconexión de componentes.

   signal x0, x1, x2, x3: std_logic;

   -- Declaracion de componentes.

   component Circuito_U0 is
      port(
         e0, e1: in  std_logic;         -- Entradas simples
         y0    : out std_logic          -- Salida simple
         );
   end component Circuito_U0;

   component Circuito_U1 is
      port(
         e2, e3, e4, e5: in  std_logic; -- Entradas simples
         y1            : out std_logic  -- Salida simple
         );
   end component Circuito_U1;

   component Circuito_U2 is
      port(
         e5, e6, e7, e8: in  std_logic; -- Entradas simples
         y2            : out std_logic  -- Salida simple
         );
   end component Circuito_U2;

   component Circuito_U3 is
      port(
         e5, e6, e7, e8: in  std_logic; -- Entradas simples
         y3            : out std_logic  -- Salida simple
         );
   end component Circuito_U3;

   component Circuito_U4 is
      port(
         a, b, c, d: in  std_logic;     -- Entradas simples
         s1, s0    : in  std_logic;     -- Entradas simples
         y4        : out std_logic      -- Salida simple
         );
   end component Circuito_U4;
```

```
-- Inicio de la arquitectura.

begin

-- Referencia (instanciacion) de componentes, utilizando
--  asignación nombrada.

U0: Circuito_U0
    port map(
       e0 => ent(0),
       e1 => ent(1),
       y0 => x0
    );

U1: Circuito_U1
    port map(
       e2 => ent(2),
       e3 => ent(3),
       e4 => ent(4),
       e5 => ent(5),
       y1 => x1
    );

U2: Circuito_U2
    port map(
       e5 => ent(5),
       e6 => ent(6),
       e7 => ent(7),
       e8 => ent(8),
       y2 => x2
    );

U3: Circuito_U3
    port map(
       e5 => ent(5),
       e6 => ent(6),
       e7 => ent(7),
       e8 => ent(8),
       y3 => x3
    );

U4: Circuito_U4
    port map(
       a  => x0,
       b  => x1,
       c  => x2,
       d  => x3,
```

```
            s1 => s(1),
            s0 => s(0),
            y4 => sal
        );

-- Fin de la arquitectura.

    end superior;
```

Para evitar problemas de configuración y asociación de los componentes con su entidad y arquitectura, en el ejercicio anterior se aplicaron las siguientes consideraciones, las cuales se recomienda tomar en cuenta en los diseños con descripción estructural:

a) Procurar que el nombre del archivo de cada componente y nivel superior coincida con su nombre de entidad. Per ejemplo, el nombre del archivo del componente U0 (Circuito_U0.vhd) es igual al nombre de su entidad (Circuito_U0) y el nombre del archivo del nivel superior (top_level.vhd) es igual al nombre de su entidad (top_level).

b) Procurar que el nombre del componente sea igual a la entidad a la que viene asociada. Por ejemplo, el componente Circuito_U2, viene asociado a un programa VHDL que tiene una entidad llamada Circuito_U2.

Ejemplo 27. <u>Arreglo de sumadores (descripción estructural).</u>

Diseñar el siguiente circuito combinacional con VHDL utilizando el estilo de descripción estructural:

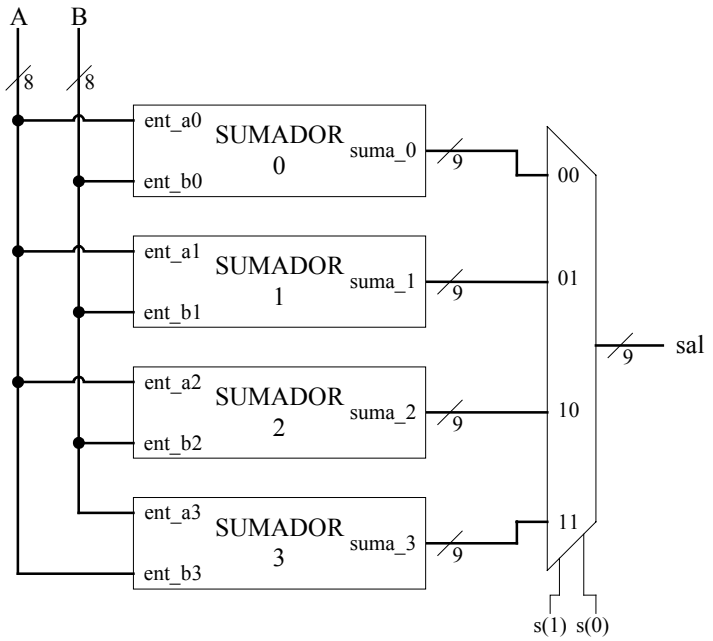

Figura Ej. 27. Arquitectura de 4 sumadores binarios de 8 bits sin signo.

Donde cada bloque de entrada es un sumador binario sin signo de 8 bits, diseñado de la siguiente manera:

a) Sumador 0. Sumador en cascada utilizando medios sumadores y sumadores completos.

b) Sumador 1. Sumador utilizando la técnica de acarreo anticipado (Carry Look-Ahead). Se puede partir con el modelado en VHDL del C.I. sumador binario de 4 bits 74LS83A, conectando 2 de estos C.I. en cascada para completar el sumador de 8 bits.

c) Sumador 2. Sumador utilizando la función '+' de VHDL, contenida en las librerias del software ISE 9.2i use IEEE.STD_LOGIC_ARITH.ALL y use IEEE.STD_LOGIC_UNSIGNED.ALL).

d) Sumador 3. Sumador en base al core IP *Adder Substracter v7.0*, incluido en el grupo *Adders & substracters* de las *Math functions* del *IP Coregen* del software ISE 9.2i.

Realizar un primer diseño puramente combinacional y un segundo diseño segmentado, en el que se registren las entradas y salidas de cada bloque lógico (sumadores y multiplexor), para buscar la operación a la mayor velocidad posible aun cuando se tengan varios ciclos de latencia. Evaluar las estadísticas de área de cada componente y del sistema completo.

4. MAQUNAS DE ESTADOS FINITOS (FSM).

Uno de los ámbitos en el cual ha triunfado totalmente la utilización del VHDL ha sido en el diseño de máquinas de estados finitos, que son parte fundamental del "control-path" de todo sistema digital. En este apartado se dan las nociones básicas del estilo adecuado para su descripción, de tal forma que el sintetizador que lo convertirá a nivel de compuertas genere un circuito lo más sencillo y rápido posible, o lo mas fiable (libre de riesgos) posible.

En el diseño del "control-path" formado básicamente por FSM's que interactúan entre ellas, el VHDL sigue erigiéndose como herramienta fundamental que supera con creces a otros lenguajes y herramientas disponibles en el mercado.

4.1 ESTILOS DE DESCRIPCIÓN.

En general existen dos modos fundamentales de realizar descripciones de máquinas de estados.

Estilo implícito: No aparece en ningún momento una declaración explícita del estado de la máquina. Solo aparecen controles por evento que marcan las transiciones en el circuito de un estado a otro. Este método solo es adecuado en los casos en los cuales el "control path" es un flujo simple donde cada estado puede ser alcanzado únicamente desde otro solo estado. A continuación se muestra un ejemplo en el cual se describe un circuito generador de secuencias de tres bits (...010...) que pasa a través de tres estados de forma cíclica; por tanto el circuito tiene un flujo de control simple, con lo cual el estilo implícito es posible:

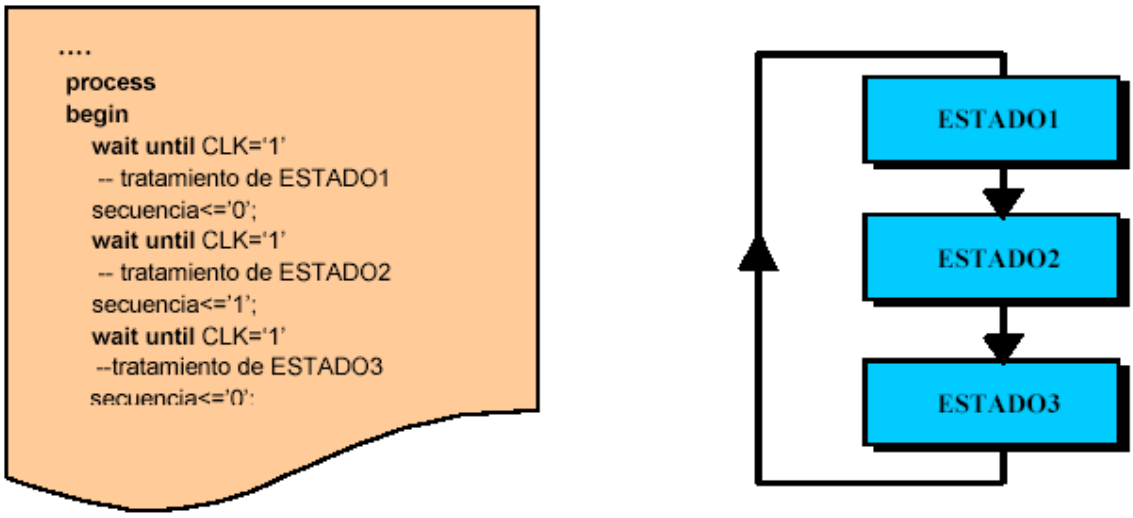

Figura 4.1. Estilo de descripción explícito.

Para este tipo de estilo de descripción, se debe usar la misma fase de reloj para cada expresión de control de evento. Los estado implícitos únicamente pueden ser actualizados si son controlados por una fase de reloj.

La cuestión de la inicialización en estas máquinas no tiene una solución sencilla. Parece claro que existe la inicialización síncrona y la asíncrona; pero lo importante en este caso es que esta posible inicialización rompe el ciclo de control sencillo que imperaba en este tipo de máquinas, lo cual obliga a la búsqueda de una construcción VHDL que permita esta ruptura de un control por defecto cíclico. La opción idónea es utilizar un *loop* dentro del proceso y romper el flujo cíclico a través de los estatutos **next (*when inicialización)*** y **exit (*when inicialización)***. Lamentablemente este tipo de estructuras con estos estatutos no suele formar parte del subconjunto RTL común de las herramientas de síntesis, con lo cual no se va a seguir insistiendo en este tipo de descripción porque en definitiva se depende en exceso de la herramienta que tengamos para poder obtener resultados sintetizables adecuados.

Estilo Explícito: En este estilo aparece de forma explícita una variable que es el estado (normalmente representada mediante un tipo enumerado); dicha variable va siendo actualizada síncronamente dependiendo de los valores de las entradas y del valor actual de esta variable de estado. Es un estilo más fácil de escribir que el implícito. Si se trabaja de forma ordenada, no da problemas y proporciona buenos resultados en la mayoría de las herramientas de síntesis y es fácil de detectar errores en las verificaciones de las máquinas descritas con dicho estilo.

Dentro del mismo existen varios subestilos que dependen del número de procesos utilizados. Se va a adoptar en lo que resta del tema la opción más clara y a la vez normalmente recomendada por los sintetizadores que consiste en separar claramente los procesos combinacionales de los procesos secuenciales. Esto obligará a manejar además de la variable de estado de la máquina, otra del mismo tipo que representará el estado siguiente. Esto no quita que se intente en todo momento reflejar como se puede reducir el número de procesos mediante la combinación de procesos combinacionales o mediante el manejo de una única variable: la de estado.

4.2 MODELO MOORE.

Para cada uno de los modelos se vá a proceder del siguiente modo: En primer lugar se representará el diagrama de bloques que define la funcionalidad del modelo. En segundo lugar se mostrará la metodología en que dicho modelo puede ser representado en procesos VHDL. Se terminará con un ejemplo sencillo en el que se incluirá el diagrama de estados (entendible por cualquier diseñador digital) y su descripción VHDL en la cual se aplicará la metodología de descripción presentada .

DIAGRAMA A BLOQUES.

La funcionalidad del modelo Moore puede representarse tal como se observa en la Figura 4.2, en el que se destaca el hecho de que el circuito combinacional de salida solo depende del estado.

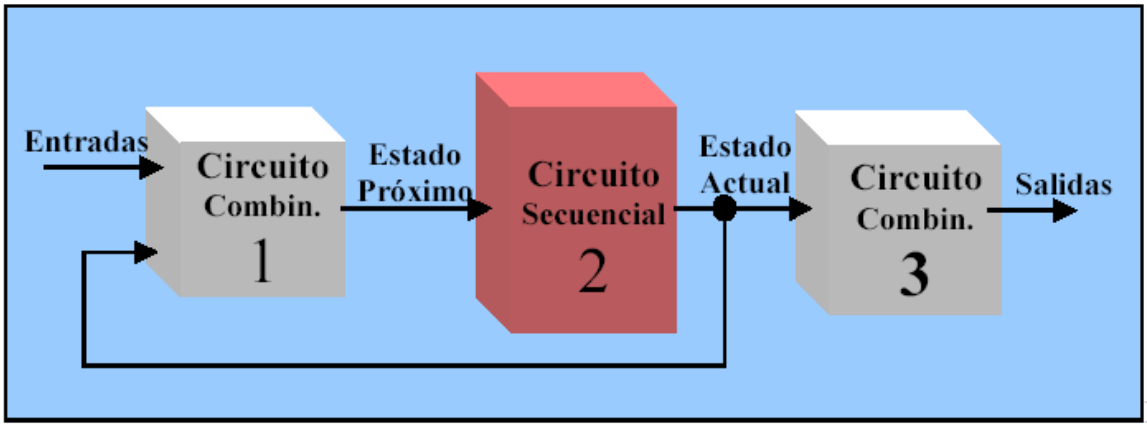

Figura 4.2. Diagrama a bloques del modelo Moore.

MODELO VHDL.

La forma en que este modelo se describe en VHDL es tan sencillo como dedicar un proceso para cada bloque funcional, teniendo en cuenta claro está que dos de ellos son de carácter combinacional y uno de ellos de carácter secuencial. Los procesos combinacionales serán sensibles a sus entradas y el proceso secuencial a la señal de reloj y a una señal asíncrona si la tuviera (por ejemplo el reset asíncrono de inicialización). Así de esta forma se tendría lo siguiente:

combinacional1: **process**(entradas, estado_actual)
proceso sensible a las entradas y al estado de la máquina. En este proceso, de típo combinacional, se modelizará el circuito combinacional que obtiene el próximo estado.

secuencial2: **process**(reloj, reset_a)
proceso que modelizará la lógica secuencial. Se encargará de que en el flanco activo del reloj el estado de la máquina pase a ser el valor que en ese momento tenga el próximo estado. También en este proceso se modelizará la acción del reset de la máquina, sea asíncrono o síncrono.

combinacional3: **process**(estado_actual)
proceso sensible al estado de la máquina. En este proceso, de tipo combinacional, se modelizará el circuito combinacional de las salidas.

Esta es sin duda alguna la forma más natural y sencilla de representar el funcionamiento mediante procesos concurrentes del comportamiento de una máquina de estados. Variantes a este esquema combinan los procesos 1 y 2 en uno solo de tipo secuencial (y por tanto solo sensible a la señal de reloj y señal asíncrona si la hubiera) que elimina la necesidad de manejar la señal estado próximo y que actualiza directamente el estado actual . Una variante también muy habitual es combinar los dos procesos combinacionales en uno solo, aunque esta práctica es mucho más usual en las máquinas Mealy como luego se verá, al compartir ambos procesos la misma lista de sensibilidad.

EJEMPLO MOORE.

Realizar un detector de secuencia, en particular de la secuencia de tres bits (...010...) y que permita el solapamiento. Por tanto el circuito debe de actuar como representa la Figura 4.3.

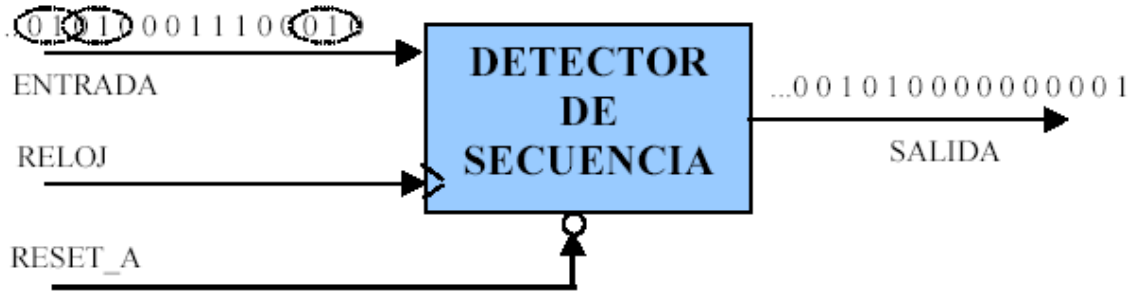

Figura 4.3. Ejemplo: detector de secuencia ...010... con solapamiento.

El diagrama de estado correspondiente al detector de secuencias bajo un modelo Moore es el representado en la Figura 4.4. Su descripción VHDL se tiene a continuación.

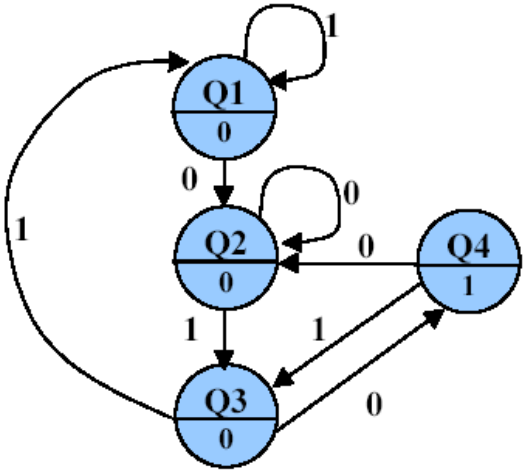

Figura 4.4. Diagrama de estado del ejemplo, modelo Moore.

```
-- Libreria estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
                      SALIDA : out std_logic
  );
end FSM;

-- Descripcion del circuito

architecture MOORE of FSM is

  -- Declaraciones
  type STATE_TYPE is ( Q1, Q2, Q3, Q4);
  signal ESTADO, ESTADO_PROX: STATE_TYPE;
  -- Cuerpo de la arquitectura
  begin

    -- Proceso 1 combinacional
    COMBINACIONAL1: process( ESTADO,ENTRADA)
      begin
        ESTADO_PROX <= ESTADO;
        case ESTADO is
          when Q1 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q2;
            end if;
          when Q2 =>
            if (ENTRADA='1') then ESTADO_PROX<=Q3;
            end if;
          when Q3 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q4;
            else ESTADO_PROX<=Q1;
            end if;
          when Q4 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q2;
            else ESTADO_PROX<=Q3;
            end if;
        end case;
      end process COMBINACIONAL1;
```

```
    -- Proceso 2 secuencial
    SECUENCIAL2: process( RELOJ,RESET_A )
      begin
        if ( RESET_A = '0' ) then
          ESTADO <= Q1;
        elsif ( RELOJ'event and (RELOJ = '1')) then
          ESTADO <= ESTADO_PROX;
        end if;
      end process SECUENCIAL2;

    -- Proceso 3 combinacional
    COMBINACIONAL3: process(ESTADO)
      begin
        case ESTADO is
          when Q4      => SALIDA<='1';
          when others => SALIDA<='0';
        end case;
      end process COMBINACIONAL3;
  end MOORE;
```

ALTERNATIVAS Y COMENTARIOS.

Sin duda alguna la más popular es la sustitución del tercer proceso (COMBINACIONAL3) por una asignación de señal concurrente de tipo "when ..else" o de tipo "with select". Por ejemplo en nuestro caso hubiera bastado con escribir para describir el funcionamiento del circuito combinacional de salida la siguiente asignación en la arquitectura:

SALIDA<= '1' **when** ESTADO=Q4 **else** '0';

Hay que destacar en el modelo descrito la utilización al principio del proceso COMBINACIONAL1 de una asignación de señal por defecto del tipo ESTADO_PROX <= ESTADO; que garantiza de forma fácil que no existan ramas del flujo condicional en los cuales no esté definida la señal ESTADO_PROX , lo cual daría lugar a la inferencia de latches asociados a esta variable, error muy típico cuando se modelizan circuitos combinacionales.

4.3 MODELO MEALY.

DIAGRAMA A BLOQUES.

La funcionalidad del modelo Mealy se representa en la Figura 4.5, destacando, a diferencia del modelo Moore, que el circuito combinacional de salida depende tanto del estado como de las entradas.

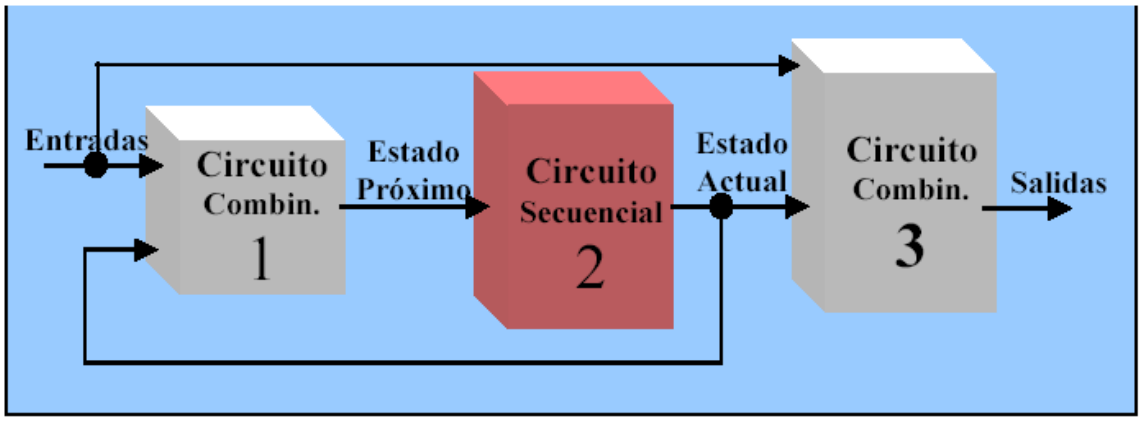

Figura 4.5. Diagrama a bloques del modelo Mealy.

MODELO VHDL.

Como puede adivinarse, los procesos combinacional1 y secuencial2 son totalmente equivalentes al modelo Moore y tan solo el proceso combinacional3 debe reflejar su dependencia en la lista de sensibilidad a los cambios que puedan producirse en las entradas que lógicamente son leídas en el interior de la descripción. Así de esta forma se tiene lo siguiente:

combinacional1: **process**(entradas, estado_actual)
proceso sensible a las entradas y al estado de la máquina. En este proceso, de tipo combinacional, modelizaremos el circuito combinacional que nos obtiene el próximo estado de la máquina Mealy.

secuencial2: **process**(reloj, reset_a)
proceso que modelizará la lógica secuencial. Se encargará de que en el flanco activo del reloj el estado de la máquina pase a ser el valor que en ese momento tenga el próximo estado. También en este proceso se modelizará la acción del reset de la máquina, sea asíncrono o síncrono.

combinacional3: **process**(estado_actual, entradas)
proceso sensible al estado de la máquina y a las entradas. En este proceso, de tipo combinacional, se modeliza el circuito combinacional de las salidas. Como se ha dicho anteriormente, una variante muy habitual es combinar los dos procesos combinacionales en uno solo, puesto que ambos comparten la misma lista de sensibilidad y en ambos es necesario recorrer de igual forma las diferentes transiciones que ocurren desde cada estado y ante cualquier combinación de entradas, para determinar cual es el estado siguiente y la salida asociada a dicha transición. En el ejemplo se aplica esta variante de dos procesos.

EJEMPLO MEALY.

El diagrama de estado correspondiente al detector de secuencias bajo un modelo Mealy es el representado en la Figura 4.6 y su correspondiente descripción VHDL es la que sigue.

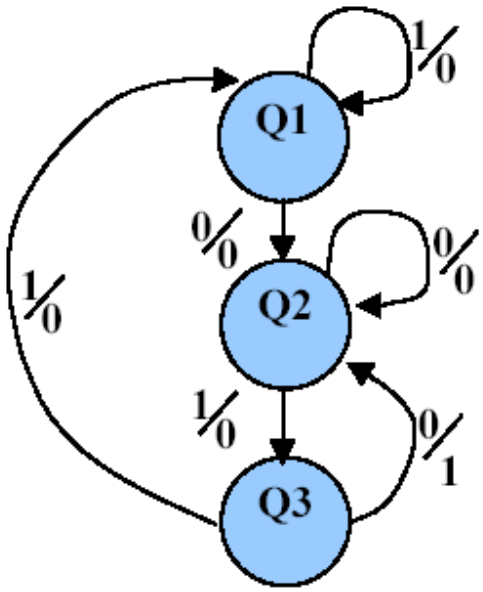

Figura 4.6. Diagrama de estado del ejemplo, modelo Mealy.

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
                      SALIDA : out std_logic
  );
end FSM;

-- Descripcion del circuito

architecture MEALY of FSM is

  -- Declaraciones
  type STATE_TYPE is ( Q1, Q2, Q3);
  signal ESTADO, ESTADO_PROX: STATE_TYPE;
  -- Cuerpo de la arquitectura
  begin
```

```
    -- Proceso 1 combinacional
    COMBINACIONAL1_3: process( ESTADO,ENTRADA)
      begin
        ESTADO_PROX <= ESTADO;
        SALIDA<='0';
        case ESTADO is
          when Q1 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q2;
            end if;
          when Q2 =>
            if (ENTRADA='1') then ESTADO_PROX<=Q3;
            end if;
          when Q3 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q2;
              SALIDA<='1';
            else ESTADO_PROX<=Q1;
            end if;
        end case;
      end process COMBINACIONAL1_3;

    -- Proceso 2 secuencial
    SECUENCIAL2: process( RELOJ,RESET_A )
      begin
        if ( RESET_A = '0' ) then
          ESTADO <= Q1;
        elsif ( RELOJ'event and (RELOJ = '1')) then
          ESTADO <= ESTADO_PROX;
        end if;
      end process SECUENCIAL2;
  end MEALY;
```

ESTADOS ESPÚREOS

En cuanto a los estados espúreos, se puede observar que en este tipo de modelización en la que directamente se utiliza un tipo enumerado específico para las señales *ESTADO* y *ESTADO_PROX* no hay cabida para la descripción de lo que ocurre en los estados espúreos. Por tanto a nivel de descripción no es posible con un tipo enumerado el manifestar un criterio de coste mínimo o de riesgo mínimo.

En el ejemplo que se acaba de ver el sintetizador suele implementar dos flip-flops para almacenar el estado de la máquina, sin embargo la máquina solo dispone de tres estados normales, luego existirá una combinación de la variables de estados que constituirá un estado espúreo. ¿Qué hace el sintetizador con los estados espúreos?. Lo que se haga con ellos dependerá de la herramienta de síntesis que se utilice; afortunadamente la mayoría de ellas tienen como opción de síntesis el utilizar un criterio de coste mínimo: el circuito será lo más rápido y pequeño posible a costa de no importarle lo que le ocurra a la máquina

cuando entra en un estado espúreo; o un criterio de riesgo mínimo: el circuito no será tan óptimo en área y velocidad pero estará protegido ante la aparición de estados espúreos, es decir, lo hará transitar rápidamente a un estado normal de funcionamiento.

4.4. CODIFICACIÓN DE LOS ESTADOS.

De todos los diseñadores de máquinas de estados es sabido que una de las cuestiones más importantes desde el punto de vista de la obtención de óptimos resultados en la síntesis de las FSM es la codificación del estado de la máquina . Así, entre otras, se pueden realizar las siguientes codificaciones:

* Secuencial: binario natural. Utiliza el mínimo número de bits.
* Gray, Johnson: disminuyen la posibilidad de transiciones incorrectas ante entradas que cambian dentro de los márgenes de set-up de los flip-flops.
* One-hot: tantos flip-flops como estados. Cada estado se representa por un flip-flop a uno y el resto a cero. Adecuado en aquellas arquitecturas tecnológicas que disponen de un alto número de flip-flops frente a lógica combinacional para cada registro. Por ejemplo las FPGAs son arquitecturas de este tipo.

Ante estas diferentes posibilidades, que pueden dar mejores o peores resultados, el diseñador puede utilizar inteligentemente los tipos de datos disponibles en VHDL para conseguir alternativas de codificación que le lleven a resultados óptimos sin tener que escribir código de manera exhaustiva y continua. Se resumen a continuación los diferentes tipos de datos que se pueden utilizar para la variable de ESTADO y ESTADO_PROX y las codificaciones que ellos implican.

4.4.1 CODIFICACIONES ASOCIADAS.

Se iniciará con el tipo de dato que se ha utilizado en los ejemplos anteriores y que pasa por ser el más habitual en el diseño VHDL de máquinas de estados: el tipo enumerado.

a) **Tipo enumerado**: La codificación es implícita, es decir corre a cargo del sintetizador. Dicho sintetizador normalmente admite en sus opciones de síntesis diferentes alternativas de codificación como las vistas anteriormente, que pueden ser elegidas y aplicadas sin problemas por el sintetizador siempre y cuando la descripción sea detectada como una FSM por la herramienta. Esto obliga a ser un tanto cuidadoso en el estilo de lo que se describe para permitir que el sintetizador la interprete como una máquina de estados. Si el sintetizador no interpreta la descripción como una máquina de estados, la sintetizará normalmente bien pero aplicando una codificación secuencial a los estados de la enumeración.

b) **Tipo enumerado con atributo de codificación**: La codificación es explicita mediante el uso del siguiente atributo:

type STATE_TYPE **is** (Q0, Q1,Q2,Q3);
signal ESTADO, ESTADO_PROX: STATE_TYPE;
attribute ENUM_TYPE_ENCODING: STRING;
attribute ENUM_TYPE_ENCODING **of** STATE_TYPE: **type is**
"000 001 011 111";

El problema de esta construcción es que no es fácilmente admitida por la mayoría de las herramientas de síntesis.

c) **Tipo integer**: La codificación es implícita pero única: secuencial. Evidentemente debemos utilizar un entero de rango restringido al número de estados de la máquina.

type STATE_TYPE **is integer range** 0 **to** 3;
signal ESTADO, ESTADO_PROX: STATE_TYPE;

d) **Usando tipo vectores (std_logic_vector):** La codificación es explícita (normalmente mediante constantes del mismo tipo) y por tanto especificada y fijada por la descripción. Es a nuestro modo de ver la solución más sencilla cuando se quiere una codificación a la carta de nuestra máquina de estados.

4.5 MÁQUINAS CON SALIDAS DECODIFICADAS EN LOS ESTADOS.

Uno de los casos más usuales en los cuales es necesario hacer una codificación especifica es aquel en el que las salidas están codificados en los estados, con lo cual eliminamos el circuito combinacional3, y por tanto tenemos las salidas libres de "glitches" y con el menor retardo posible. Estas máquinas son conocidas como máquinas Medvedev (Figura 4.7).

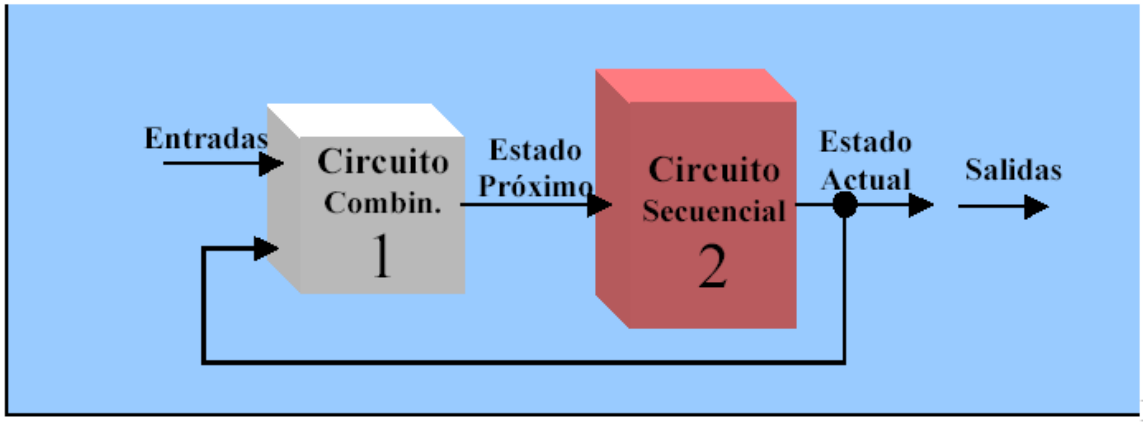

Figura 4.7. Diagrama a bloques de la máquina Medvedev.

Evidentemente, es una variante del modelo MOORE con la particularidad de tener una codificación en la cual algunas de las variables (bits) de estado asumen el valor de las salidas en cada uno de los estados. El precio de esta codificación especial es la necesidad,

en la mayoría de los casos, de un mayor número de bits (flip-flops) que en una codificación de los estados secuencial.

Aplicando esta configuración al ejemplo del detector de secuencia, se va a utilizar el tipo de datos *std_logic_vector* por cuanto es el único que explícitamente permite expresar esta codificación de las salidas en los estados.

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
                      SALIDA : out std_logic
  );
end FSM;

-- Descripcion del circuito

architecture MEDVEDEV of FSM is

  -- Declaraciones
  constant Q1: std_logic_vector(2 downto 0):=”000”;
  constant Q2: std_logic_vector(2 downto 0):=”010”;
  constant Q3: std_logic_vector(2 downto 0):=”100”;
  constant Q4: std_logic_vector(2 downto 0):=”001”;
  signal ESTADO, ESTADO_PROX: std_logic_vector( 2 downto 0);

  -- Cuerpo de la arquitectura
  begin
    -- Proceso 1 combinacional
    COMBINACIONAL1: process( ESTADO,ENTRADA)
      begin
        ESTADO_PROX <= ESTADO;
        case ESTADO is
          when Q1 =>
            if (ENTRADA=’0’) then ESTADO_PROX<=Q2;
            end if;
          when Q2 =>
            if (ENTRADA=’1’) then ESTADO_PROX<=Q3;
            end if;
          when Q3 =>
```

```
            if (ENTRADA='0') then ESTADO_PROX<=Q4;
            else ESTADO_PROX<=Q1;
            end if;
          when Q4 =>
            if (ENTRADA='0') then ESTADO_PROX<=Q2;
            else ESTADO_PROX<=Q3;
            end if;
          when others=>
            ESTADO_PROX<=Q1;
        end case;
      end process COMBINACIONAL1;

    -- Proceso 2 secuencial
    SECUENCIAL2: process( RELOJ,RESET_A )
      begin
        if ( RESET_A = '0' ) then
          ESTADO <= Q1;
        elsif ( RELOJ'event and (RELOJ = '1')) then
          ESTADO <= ESTADO_PROX;
        end if;
      end process SECUENCIAL2;

    SALIDA<=ESTADO(0);

  end MEDVEDEV;
```

Como puede observarse en el ejemplo anterior, la salida esta codificada en el bit menos significativo de la señal de estado, con lo cual no se necesita circuitería combinacional adicional.

4.6 MÁQUINAS CON SALIDAS REGISTRADAS.

Con el fin de tener salidas libres de "glitches" y lo más rápidas posibles con respecto al flanco activo de reloj, suelen necesitarse máquinas de estados finitos con las salidas registradas. Se analiza a continuación lo que ocurre con los modelos vistos anteriormente cuando se quiere registrar las salidas:

1. En realidad el modelo Medvedev visto anteriormente puede contemplarse como una máquina con salidas registradas. No requiere transformación.

2. Las máquinas Mealy son fácilmente transformables, aunque esta transformación va a tener el coste de retrasar un ciclo las salidas. El modelo funcional quedaría como se observa en la figura 4.8.

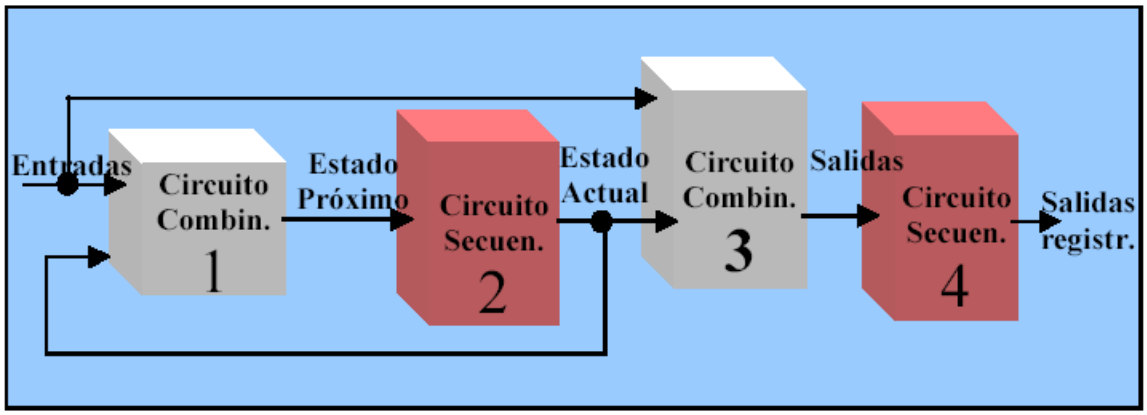

Figura 4.8. Modelo Mealy con las salidas registradas.

Evidentemente se puede adoptar un estilo de descripción con cuatro procesos que modelizan cada uno de los bloques del modelo funcional. Sin embargo, se puede como antes reunir los dos circuitos combinacionales en uno solo y los dos circuitos secuenciales en uno solo, con lo cual se tendrían únicamente dos procesos, como se observa en el ejemplo siguiente.

Muy popular con este tipo de máquinas es la realización de todo el modelo mediante un solo proceso. Es sin duda alguna el tipo de máquinas que mejor se ajusta al estilo de descripción con un solo proceso gobernado por el reloj. Después del ejemplo del modelo Mealy registrado se incluye un ejemplo de dicho tipo de descripción. Es indudable la brevedad del código empleado y la necesidad de únicamente una señal de estado.

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
                  SALIDA_REG : out std_logic
  );
end FSM;

-- Descripción del circuito

architecture MEALY_registrada1 of FSM is
```

```
  -- Declaraciones
  type STATE_TYPE is ( Q1, Q2, Q3);
  signal ESTADO, ESTADO_PROX: STATE_TYPE;
  signal SALIDA: std_logic;

  -- Cuerpo de la arquitectura
  begin

  -- Proceso 1 combinacional
  COMBINACIONAL1_3: process( ESTADO,ENTRADA)
    begin
      ESTADO_PROX <= ESTADO;
      SALIDA<='0';
      case ESTADO is
        when Q1 =>
          if (ENTRADA='0') then ESTADO_PROX<=Q2;
          end if;
        when Q2 =>
          if (ENTRADA='1') then ESTADO_PROX<=Q3;
          end if;
        when Q3 =>
          if (ENTRADA='0') then ESTADO_PROX<=Q2;
            SALIDA<='1';
          else ESTADO_PROX<=Q1;
          end if;
      end case;
    end process COMBINACIONAL1_3;

  -- Proceso 2 secuencial
  SECUENCIAL2_4: process( RELOJ,RESET_A )
    begin
      if ( RESET_A = '0' ) then
        ESTADO <= Q1;
        SALIDA_REG<='0';
      elsif ( RELOJ'event and (RELOJ = '1')) then
        ESTADO <= ESTADO_PROX;
        SALIDA_REG<=SALIDA;
      end if;
    end process SECUENCIAL2_4;

  end MEALY_registrada1;


-- Librería estandar

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

```
-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
                 SALIDA_REG : out std_logic
  );
end FSM;

-- Descripcion del circuito

architecture MEALY_registrada2 of FSM is

  -- Declaraciones
  type STATE_TYPE is ( Q1, Q2, Q3);
  signal ESTADO: STATE_TYPE;

  -- Cuerpo de la arquitectura
  begin
    -- Proceso unico
    UNICO: process( RELOJ,RESET_A )
      begin
        if ( RESET_A = '0' ) then
          ESTADO <= Q1;
          SALIDA_REG<='0';
        elsif ( RELOJ'event and (RELOJ = '1')) then
          case ESTADO is
            when Q1 =>
              if (ENTRADA='0') then ESTADO<=Q2;
              end if;
            when Q2 =>
              if (ENTRADA='1') then ESTADO<=Q3;
              end if;
            when Q3 =>
              if (ENTRADA='0') then ESTADO<=Q2;
                SALIDA_REG<='1';
              else ESTADO<=Q1;
              end if;
          end case;
        end if;
      end process UNICO;
  end MEALY_registrada2;
```

3. Las máquinas Moore con salidas registradas se caracterizan por registrar, en lugar de las salidas, las próximas salidas, lo cual hace necesario prever esas próximas salidas. Esto es sencillo en los modelos Moore puesto que las próximas salidas solo dependen del

próximo estado. Una vez que se sabe cuales son las próximas salidas y se registran, se observa que el efecto resultante es la aparición de las salidas definitivas en el mismo ciclo que las que describieron en ejemplo Moore anterior. Por tanto el modelo que se utilizará es el representado en la figura 4.9.

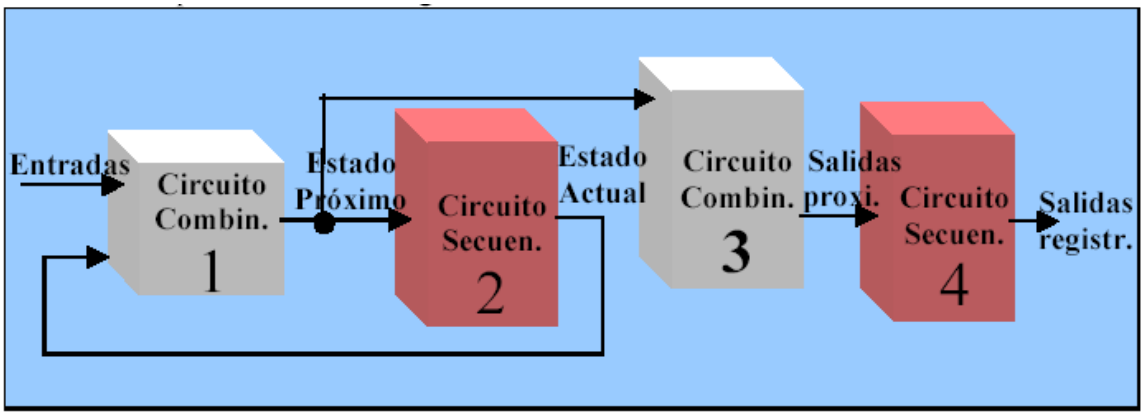

Figura 4.9. Modelo Moore con salidas registradas.

Si se toma el ejemplo de detector de secuencia y lo se describe con un modelo Moore con salidas registradas se tendrá la descripción siguiente.

```
-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity FSM is
  port(
    RELOJ, RESET_A, ENTRADA : in std_logic;
    SALIDA : out std_logic
  );
end FSM;

-- Descripcion del circuito

architecture MOORE_registrada of FSM is

  -- Declaraciones
  type STATE_TYPE is ( Q1, Q2, Q3, Q4);
  signal ESTADO, ESTADO_PROX: STATE_TYPE;
  signal SALIDA_PROX: std_logic;
```

```
-- Cuerpo de la arquitectuta
begin
  -- proceso 1 combinacional
  COMBINACIONAL1: process( ESTADO,ENTRADA)
    begin
      ESTADO_PROX <= ESTADO;
      case ESTADO is
        when Q1 =>
          if (ENTRADA='0') then ESTADO_PROX<=Q2;
          end if;
        when Q2 =>
          if (ENTRADA='1') then ESTADO_PROX<=Q3;
          end if;
        when Q3 =>
          if (ENTRADA='0') then ESTADO_PROX<=Q4;
          else ESTADO_PROX<=Q1;
          end if;
        when Q4 =>
          if (ENTRADA='0') then ESTADO_PROX<=Q2;
          else ESTADO_PROX<=Q3;
          end if;
      end case;
    end process COMBINACIONAL1;

  -- Proceso 2 secuencial
  SECUENCIAL2_4: process( RELOJ,RESET_A )
    begin
      if ( RESET_A = '0' ) then
        ESTADO <= Q1;
        SALIDA <= '0';
      elsif ( RELOJ'event and (RELOJ = '1')) then
        ESTADO <= ESTADO_PROX;
        SALIDA <= SALIDA_PROX;
      end if;
    end process SECUENCIAL2_4;

  -- Proceso 3 combinacional
  COMBINACIONAL3: process(ESTADO_PROX)
    begin
      case ESTADO_PROX is
        when Q4 =>
          SALIDA_PROX <='1';
        when others =>
          SALIDA_PROX <='0';
      end case;
    end process COMBINACIONAL3;
end MOORE_registrada;
```

4.8 RESUMEN Y CONCLUSIONES.

Con esto se concluye todo lo referente a la modelización de máquinas de estados finitos mediante VHDL. Es importante resaltar que cualquier circuito secuencial (contadores, registros de desplazamiento, registros, etc.) puede ser modelizado perfectamente siguiendo las pautas aquí descritas, dado que cualquiera de estos circuitos secuenciales no son más que un modelo con un estado, que puede ser calculado en función de su estado actual y las entradas, y con un conjunto de salidas que pueden ser de tipo Moore y por tanto dependientes únicamente del estado del circuito; o salidas Mealy, y por tanto dependientes del estado en el que se encuentra el circuito y las entradas del mismo.

Lo único que quedaría pendiente por profundizar para alcanzar un perfecto dominio de la descripción VHDL de FSM sería detallar como escribir de manera robusta y eficaz los circuitos combinacionales y secuenciales que pueblan estos modelos.

4.9 CONTADORES.

Los contadores 'cuentan' el número de ocurrencias de un evento que suceden en forma aleatoria o en intervalos aleatorios. En este apartado se describirán dos tipos, los contadores modulo N y los contadores binarios.

4.9.1 CONTADORES MÓDULO N.

Un tipo especial de máquinas de estado finitas (FSM) lo constituyen los contadores que repiten de manera cíclica una secuencia determinada. Su descripción se basa en el modelo Medvedev, tal como se observa en los siguientes ejemplos.

```
-- Contador modulo 10 con reset asincrono

-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra

entity contador_10 is
  port (
    clk : in std_logic;
    rst : in std_logic;
    y   : out std_logic_vector(3 downto 0)
  );
end contador_10;
```

```
-- Descripcion del circuito

architecture machine of contador_10 is

  -- Declaraciones
  signal edo_presente: std_logic_vector(3 downto 0);
  signal edo_proximo : std_logic_vector(3 downto 0);

  -- Cuerpo de la arquitectura
  begin
    -- Proceso combinacional
    combinacional : process(estado_presente)
      begin
        case edo_presente is
          when "0000" => edo_proximo <= "0001";
          when "0001" => edo_proximo <= "0010";
          when "0010" => edo_proximo <= "0011";
          when "0011" => edo_proximo <= "0100";
          when "0100" => edo_proximo <= "0101";
          when "0101" => edo_proximo <= "0110";
          when "0110" => edo_proximo <= "0111";
          when "0111" => edo_proximo <= "1000";
          when "1000" => edo_proximo <= "1001";
          when others => edo_proximo <= "0000";
        end case;
        y <= edo_presente;
      end process combinacional;
    -- Proceso secuencial
    secuencial: process(clk, rst)
      begin
        if (rst = '0') then
          edo_presente <= "0000";
        elsif (clk'event and clk = '1') then
          edo_presente <= edo_proximo;
        end if;
      end process secuencial;
    end machine;


-- Contador modulo 4 ascendente-descendente (reset asíncrono)

-- Librería estandar

library IEEE;
use IEEE.std_logic_1164.all;

-- Descripcion en caja negra
```

```
entity contador_4_ad is
  port (
    A   : in std_logic;
    clk : in std_logic;
    rst : in std_logic;
    y   : out std_logic_vector(1 downto 0)
  );
end contador_4_ad;

-- Descripcion del circuito

architecture machine of contador_4_ad is

  -- Declaraciones
  signal edo_presente : std_logic_vector(1 downto 0);
  signal edo_proximo  : std_logic_vector(1 downto 0);

  -- Cuerpo de la arquitectura
  begin
    -- Proceso combinacional
    combinacional: process(edo_presente, A)
      begin
        case edo_presente is
          when "00"   => if (A = '0') then
                           edo_proximo <= "01";
                         else
                           edo_proximo <= "11";
                         end if;
          when "01"   => if (A = '0') then
                           edo_proximo <= "10";
                         else
                           edo_proximo <= "00";
                         end if;
          when "10"   => if (A = '0') then
                           edo_proximo <= "11";
                         else
                           edo_proximo <= "01";
                         end if;
          when others => if (A = '0') then
                           edo_proximo <= "00";
                         else
                           edo_proximo <= "10";
                         end if;
        end case;
        y <= edo_presente;
      end process combinacional;
```

```
    -- Proceso secuencial
    secuencial: process(clk, rst)
      begin
        if (rst = '0') then
          edo_presente <= "00";
        elsif (clk'event and clk = '1') then
          edo_presente <= edo_proximo;
        end if;
      end process secuencial
  end machine;
```

4.9.2 CONTADORES BINARIOS.

Los siguientes ejemplos infieren diferentes tipos de contadores binarios, pero se debe tomar en cuenta que la mayoría de las herramientas de síntesis no pueden inferir implementaciones optimas de contadores con mas de 8 bits.

```
-- Contador ascendente de 8 bits con habilitacion de cuenta y
-- reset asincrono.

-- Librerias estandar

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-- Descripcion en caja negra

entity counter8 is
  port (
    clk, en, rst : in std_logic;
    count        : out std_logic_vector (7 downto 0)
  );
end counter8;

-- Descripcion del circuito

architecture behav of counter8 is

  -- Declaraciones
  signal cnt: std_logic_vector (7 downto 0);

  -- Cuerpo de la arquitectura
  begin
```

```
    process (clk, en, cnt, rst)
      begin
        if (rst = '0') then
          cnt <= (others => '0');
        elsif (clk'event and clk = '1') then
          if (en = '1') then
            cnt <= cnt + '1';
        end if;
     end process;

     count <= cnt;

  end behav;



-- Contador ascendente de 8 bits con carga y reset asincrono.

-- Librerias estandar

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-- Descripcion en caja negra

entity counter is
  port (
    clk, reset, load : in std_logic;
    data             : in std_logic_vector (7 downto 0);
    count            : out std_logic_vector (7 downto 0)
  );
end counter;

-- Descripcion del circuito

architecture behave of counter is

  -- Declaraciones
  signal count_i: std_logic_vector (7 downto 0);

  -- Cuerpo de la arquitectura
  begin
    process (clk, reset)
      begin
        if (reset = '0') then
```

```
          count_i <= (others => '0');
        elsif (clk'event and clk = '1') then
          if (load = '1') then
            count_i <= data;
          else
            count_i <= count_i + '1';
        end if;
      end process;

      count <= count_i;

  end behave;



-- Contador ascendente de n bits con carga, habilitacion de
-- cuenta, terminación de cuenta y reset asincrono.

-- Librerias estandar

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-- Descripcion en caja negra

entity counter is
  generic (width : integer := n);
  port (
    clk, rst, load, en : in std_logic;
    data               : in std_logic_vector (width-1 downto 0);
    q                  : out std_logic_vector (width-1 downto 0)
  );
end counter;

-- Descripcion del circuito

architecture behave of counter is

  -- Declaraciones
  signal count: std_logic_vector (width-1 downto 0);

  -- Cuerpo de la arquitectura
  begin
    process (clk, rst)
      begin
```

```
        if (rst = '1') then
          count <= (others => '0');
        elsif (clk'event and clk = '1') then
          if (load = '1') then
            count <= data;
          elsif (en = '1') then
            count <= count + '1';
        end if;
    end process;

    q <= count;

  end behave;
```