23 LENGUAJE DE DESCRIPCIÓN CIRCUITAL: V H D L

A la hora de describir un diseño microelectrónico (es decir, de realizar la descripción de un circuito digital, en formato informático, para ser, posteriormente, «compilado» sobre un dispositivo programable o sobre una librería de celdas de un ASIC), tal descripción puede hacerse en forma gráfica (esquema circuital) o en forma de texto (programa).

El "Versatile Hardware Description Language" VHDL (cuyas siglas proceden de un nombre aún más largo y restrictivo: Very high speed integrated circuit HDL) se desarolló inicialmente como lenguaje de documentación, de simulación y de «validación» (simulación en el contexto en que debe funcionar) de circuitos integrados digitales. Para documentar y simular se requiere una descripción precisa, carente de ambigüedades y estructurada y tal descripción puede ser directamente utilizada para diseñar el circuito descrito (mediante su «compilación» sobre los recursos booleanos disponibles). Las dos páginas que siguen (apartado 1 de este capítulo) amplían la breve presentación que del lenguaje VHDL se hace en este párrafo y deben ser leídas como parte de esta introducción.

Habida cuenta de que un circuito integrado no tiene «finalidad propia», sino que forma parte de un sistema más amplio, cuyo funcionamiento controla o supervisa, uno de los propósitos de VHDL era simular el circuito en el contexto del sistema de que forma parte; de manera que no se limita a describir sistemas digitales sino que abarca, también, cualquier otro tipo de sistema activo (eléctrico, mecánico,...).

Es obvio que un tratado sobre VHDL requeriría todas las páginas de este volumen y muchas más. Por ello, el contenido de este capítulo se restringe a la parte de VHDL que se utiliza habitualmente en el diseño digital. Su objetivo es enseñar las bases de la descripción de circuitos digitales en VHDL a quienes desconozcan por completo este lenguaje.

Quizás la mejor forma de aprender un lenguaje sea utilizarlo y practicarlo. Por eso este capítulo, que es simplemente una presentación parcial y utilitaria de VHDL, está construido, fundamentalmente, con ejemplos de diseño; se apoya en múltiples descripciones y diseños de subcircuitos y de pequeños sistemas digitales y prescinde, en gran medida, de lo que pudiera ser una exposición académica o descriptiva del propio lenguaje.

La misma organización del capítulo está dirigida directamente al diseño digital y, así, los diversos epígrafes de introducción del lenguaje se refieren, sucesivamente, a la descripción de sistemas combinacionales (apartado 2), de sistemas secuenciales y síncronos (apartado 3) y de grafos de estado (apartado 4) y a otros recursos avanzados de diseño digital, aplicados también a ejemplos concretos (apartado 5).

Tomás Pollán Santamaría. Tecnología Electrónica. E.U.I.T.I.Z. Universidad de Zaragoza. tpollan@posta.unizar.es

23.1. VHDL como lenguaje para describir, simular, validar y diseñar

En un principio, la «captura de esquemas» fue la forma habitual de diseño CAD (apoyado y almacenado en un computador). Pero, hoy día, ha sido sustituida (casi por completo) por su descripción funcional en texto (programa que detalla el funcionamiento de las diversas partes del circuito y la conexión entre ellas), utilizando para ello un lenguaje de descripción de *hardware* (HDL).

La forma textual presenta numerosas ventajas: suele requerir menor tiempo y esfuerzo para comprender lo que el circuito hace (en el caso de sistemas complejos); es independiente de la implementación a bajo nivel (en puertas, bloques, biestables y registros); es directamente trasladable a los diversos dispositivos programables y a las diversas librerías de ASICs;... y, sobre todo, resulta mucho más sencillo revisar e introducir modificaciones en el texto descriptor que en el esquema gráfico del mismo.

Actualmente, son dos los lenguajes de descripción circuital que se han impuesto como estándares para el diseño digital: VHDL y Verilog; y, de entre ellos, en el contexto europeo predomina el VHDL (si bien Verilog resulta, en buena medida, más cercano al *hardware* y a los esquemas gráficos que se utilizaban anteriormente).

El lenguaje de descripción de sistemas digitales VHDL nació como herramienta de documentación y de comunicación en relación con los circuitos integrados digitales. Un circuito integrado complejo, a lo largo del tiempo pasa por muchas manos («por muchas mentes»): quien lo define y quien lo utiliza no suele ser el mismo que quien lo diseña; quien repara o quien modifica los sistemas en que participa el circuito no suele ser el propio diseñador; e, incluso, quien lo ha diseñado, al volver a su circuito cuando ha transcurrido un cierto tiempo, difícilmente recordará los detalles de las diversas funciones y recursos con los que lo ha configurado.

De ahí la necesidad de un lenguaje que ofrezca una descripción funcional precisa, carente de ambigüedades, estructurada y de fácil lectura e independiente de la «implementación» concreta a bajo nivel. Que quien lea esa descripción sea capaz de comprender, con poco esfuerzo y absoluta claridad, las funciones que hace el circuito y cómo las hace; sin tener que descender al nivel booleano de puertas y biestables que, por su mayor amplitud en componentes, resultaría más difícil de analizar.

Además, una misma descripción funcional puede configurarse circuitalmente de formas muy variadas y tal configuración dependerá de la librería de celdas estándar disponibles (para el diseño de un ASIC) o, en su caso, del dispositivo programable en que se inserte; en principio no es necesario conocer su configuración a nivel booleano para utilizar eficientemente un circuito digital.

Lo que sí es necesario es disponer de una adecuada y detallada descripción funcional. La escritura es la herramienta de las ideas; el vehículo hacía la «claridad», la precisión, la estructura (disposición, orden y enlace de las partes que configuran el todo) y la comunicación (transmisión de las ideas «en el espacio y en el tiempo»); la escritura es el mejor medio para poder trasladar las ideas a otras personas (comunicación espacial) y poder contrastarlas y debatirlas y, también, para recordar las ideas (comunicación temporal) y poder recuperarlas y revisarlas pasado el tiempo.

La documentación es un requisito inexcusable para considerar que un diseño se ha completado y hecho efectivo y, cuando el diseño se refiere a un sistema complejo, requiere un lenguaje preciso y estructurado (estructurado tanto en su forma de expresar el diseño como en el sentido de que confiera estructura al propio diseño).

Por otra parte, antes de «construir» un diseño complejo, con el coste económico y de tiempo que ello supone, conviene saber si el diseño es correcto; para ello es de gran ayuda efectuar una simulación «virtual» de su funcionamiento. Un lenguaje que describa con precisión el comportamiento de un sistema servirá, sin duda, para simular su comportamiento: bastará con una aplicación informática que «ejecute» la descripción del sistema en relación con el transcurso del tiempo.

Aún más, un circuito integrado no tiene sentido funcional «por sí mismo», sino que formará parte de un sistema más amplio; generalmente será una pieza de control de un sistema con partes eléctricas y mecánicas. El objetivo «final» no es que el circuito integrado funcione (individualmente) bien, sino que el sistema global actúe correctamente: en definitiva, el objetivo que se persigue no es el funcionamiento del propio circuito integrado, sino el del sistema de que forma parte.

De manera que el propósito de un lenguaje eficiente se refiere a la capacidad de describir y simular, además de los circuitos integrados digitales, los mecanismos y sistemas controlados por ellos. Es decir, «validar» al circuito integrado en el entorno funcional para el que ha sido diseñado.

Estamos, pues, planteando un objetivo cada vez más ambicioso: de la documentación se pasa a la simulación y se pretende, también, la validación de los circuitos integrados complejos. Con esa intencionalidad ha sido desarrollado el *Very high speed integrated circuit Hardware Description Language*, que conocemos como VHDL, siglas que podemos referir mejor a *Versatile Hardware Description Language* (pues la referencia a la alta velocidad de los circuitos es superflua; de igual forma puede ser utilizado para documentar, describir, simular y validar circuitos integrados lentos).

Un lenguaje con capacidad para describir con precisión y simular con eficiencia puede ser fácilmente utilizado para diseñar: una vez descrito un sistema digital, basta «compilar» la descripción sobre una «librería de recursos».

Se entiende por compilación el paso de la descripción funcional a la configuración circuital (del algoritmo al circuito), utilizando como componentes de dicho circuito los contenidos en una «librería de recursos»:

- en el caso de CPLDs dicha librería se refiere a las funciones booleanas en forma de suma de productos (configuración PAL) y biestables tipo D;
- para las FPGAs los recursos son las funciones booleanas, con un limitado número de variables (dividiendo, en su caso, las funciones más amplias), expresadas en forma de «tabla de verdad» (configuración LUT) y los biestables D;
- y en el diseño de ASICs, la librería de celdas básicas prediseñadas, propia del diseño con librería (*standard cell*).

El compilador (la herramienta informática de compilación) «traslada» la descripción funcional al circuito que la «materializa», conformado por componentes disponibles en la librería de recursos sobre la que se compila.

VHDL es una excelente herramienta de documentación, simulación, validación y diseño de sistemas digitales, estandarizada y en permanente proceso de actualización bajo los auspicios del IEEE (*Institute of Electrical and Electronics Engineers*). El único lenguaje alternativo que goza, también, de amplia aceptación es Verilog, pero su difusión en el contexto europeo es mucho menor (aunque, en buena medida, Verilog es un lenguaje más directo y más cercano al propio circuito digital).

23.2. VHDL básico para diseñar circuitos combinacionales

El lenguaje VHDL no se refiere solamente a sistemas digitales sino que está abierto a la descripción de sistemas de cualquier tipo; por otra parte, VHDL es un lenguaje tremendamente versátil y potente. Su descripción requiere todo un amplio volumen y su estudio precisa de, al menos, un curso específico dedicado solamente a este lenguaje.

Interesa aquí la utilización de VHDL para diseñar sistemas digitales, es decir, aquella parte del lenguaje que es «compilable» para dar como resultado un circuito digital. A continuación, se expone un breve resumen, a la vez parcial y útil, como primera aproximación a este lenguaje y, a fin de ser lo más directa y práctica posible, esta introducción se realiza a través de ejemplos concretos de descripción circuital.

Nota: Todos los ejemplos de código VHDL (incluso las descripciones parciales) que vienen a continuación han sido comprobados en cuanto a su compilación (con MAX+plus II de ALTERA).

23.2.1. Primeras nociones

VHDL no distingue entre MAYÚSCULAS y minúsculas (salvo unas pocas excepciones referidas a valores de las señales). Pueden incluirse comentarios y, para identificarlos, se inician con el símbolo repetido "**--**" que indican al compilador que ignore todo lo que sigue hasta final de línea. Cada «módulo» descriptivo y cada «asignación» se cierran con el símbolo "**;**".

Los elementos básicos de la descripción digital son las señales (*signal*); para ellas suele utilizarse el tipo *std_logic* (*standard logic*) que admite los siguientes nueve valores:

'**0**' -- cero	'**1**' -- uno	*valores booleanos típicos*
'**X**' -- desconocido		*no se conoce el valor*
'**Z**' -- alta impedancia		*propio de tri-estado*
'**U**' -- sin inicializar		*biestables en su situación previa*
'**-**' -- no importa (don't care)		*indiferente (para simplificación)*
'**L**' -- 0 débil	'**H**', -- 1 débil	'**W**', -- desconocido débil

Los valores de una señal se expresan siempre entre comillas simples: *'0', '1'* y los valores X y L no admiten la minúscula (x , l no son válidas).

Los tres primeros valores (0, 1, X) son de tipo fuerte, si se «encuentran» dos de ellos aplicados sobre un nudo el resultado es X (desconocido). Los valores débiles corresponden a determinadas situaciones circuitales que, si confluyen con algún valor fuerte, dan como resultado dicho valor fuerte; en cambio, si se encuentran dos valores débiles sobre un nudo el resultado es W (desconocido débil).

Un conjunto de señales constituye un vector, *std_logic_vector*, que puede ser declarado en forma ascendente *std_logic_vector(0 to 7)* o descendente *std_logic_vector(7 downto 0)*, siendo más frecuente esta segunda declaración porque corresponde a la forma típica en la que el dígito más significativo es el de mayor subíndice; el conjunto de valores que adopta un vector se expresa entre comillas dobles: por ejemplo, *"11010001"*.

Para conjuntos de señales (vectores), se utiliza también el tipo *integer* (entero) que debe ser declarado para un rango determinado: *integer range 0 to 15* (señal de 4 bits).

Los tipos de señales, *std_logic* y *std_logic_vector*, aunque son los más habituales en diseño digital (ya que describen bien las señales electrónicas en todas sus posibilidades), no se encuentran definidos en el propio VHDL básico (están definidos los tipos *bit* y *bit_vector*, que admiten sólo los dos valores booleanos **0** y **1**). Los tipos *standard logic* han sido introducidos en la normalización hecha por IEEE y requieren la declaración de la librería (y de los paquetes) que los definen al principio de la descripción:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

(con el paquete *ieee.std_logic_unsigned* las operaciones se realizan en binario natural; si se desea efectuarlas en complemento a 2 debe utilizarse el paquete *ieee.std_logic_signed*)

En estos paquetes se dispone de dos funciones muy útiles:

CONV_INTEGER(**a**) que convierte el *std_logic_vector* **a** en *integer*

CONV_STD_LOGIC_VECTOR(**b**,**n**) que convierte el *integer* **b** en vector de longitud **n**.

Las **operaciones básicas** entre señales son:

asignación:	<=
operaciones booleanas	and or not xor
comparaciones	= /= > < >= <=
aritméticas	+ - *
concatenación	&

(la concatenación se refiere a poner señales o vectores juntos, formando un vector «más largo», cuyo número de bits es la suma de los números de ambas señales).

La base de la descripción VHDL es la asignación de valores a una señal, la cual puede hacerse directamente o por medio de operaciones entre señales.

Ejemplos de asignaciones directas:

```
signal a, b, c, Y:   std_logic_vector(3 downto 0);
signal m, n: integer range 0 to 15;          -- 4 bits
   y <= "1001";
   Y <= (3 => '1', 0 => '1', others => '0');  -- equivale a la anterior ("1001")
   m <= 9;
   Y <= ((not a) and b) or (a and not b);    -- equivale a y <= a xor b;
   y <= a + b;                               -- suma aritmética
```

El lenguaje VHDL es muy disciplinado: una señal de un tipo no admite asignación de valores o de señales de otro tipo.

Ejemplos de asignaciones incorrectas:

```
y <= 9;          -- y no es de tipo integer
m <= "1001";     -- m no es del tipo std_logic
Y <= m;          -- tipos de señal diferentes
M <= a;          -- tipos de señal diferentes
Y <= "001";      -- faltan componentes
m <= 18;         -- fuera de rango
y <= '1001';     -- faltan comillas dobles
M <= 4           -- falta ;
```

23.2.2. Estructura de una descripción: librerías, entidades y arquitecturas

En VHDL se describe por un lado la «caja» del circuito con sus entradas y salidas, o sea, los terminales de conexión hacia el exterior, y eso se hace en un módulo denominado *entity*, y en otro módulo posterior, denominado *architecture*, se describe «lo que hace» el circuito, es decir, su funcionamiento interno. Además, es preciso declarar previamente las librerías necesarias para compilar el circuito (sobre librerías se trata en el apartado 23.5.6).

En consecuencia, la descripción VHDL tiene la siguiente estructura:

➔ declaración de librerías

➔ módulo de terminales

```
entity nombre_de_la_entidad is
  port(
            declaración de entradas y salidas
                                                  );
end nombre_de_la_entidad ;
```

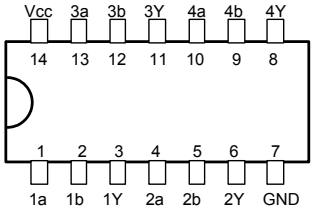
➔ módulo de funciones

architecture nombre_de_la_arquitectura *of* nombre_de_la_entidad *is*
signal declaración de señales internas
begin
descripción del funcionamiento (asignaciones)
end nombre_de_la_arquitectura *;*

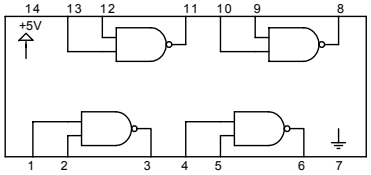
Ejemplo: consideremos un sencillo circuito integrado, como puede ser el 7400 que contiene 4 puertas Nand.

Podríamos representar gráficamente la *entidad* y la *arquitectura* de ese circuito en la siguiente forma:

entity cuatro_puertas is



architecture puertas_nand of cuatro_puertas is



y su descripción en texto VHDL:

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

entity  cuatro_puertas is
        port( a1,b1,a2,b2,a3,b3,a4,b4        :in std_logic;
              Y1,Y2,Y3,Y4                    :out std_logic);
end cuatro_puertas;

architecture  puertas_nand  of cuatro_puertas is
        begin       Y1 <= a1 nand b1;
                    Y2 <= a2 nand b2;
                    Y3 <= a3 nand b3;
                    Y4 <= a4 nand b4;
end puertas_nand;
```


Otra forma, más breve, de describir este mismo circuito es la siguiente:

```
entity  cuatro_puertas is
        port( a,b           :in std_logic_vector(1 to 4);
              Y             :out std_logic_vector(1 to 4));
end cuatro_puertas;

architecture  puertas_nand  of cuatro_puertas is
        begin       Y <= a and b;
end puertas_nand;
```

En la entidad (*entity*) se describen los terminales del circuito dentro del epígrafe de puertos (*ports*); hay cuatro tipos de «puertos»: entrada (*in*), salida (*out*), bidireccionales (*inout*) y «adaptados» (*buffer*). Los «puertos» de salida no se pueden «leer» dentro del circuito, es decir, no pueden figurar como entradas en ninguna de las asignaciones de su arquitectura; en cambio, los «puertos adaptados» son salidas que sí se pueden «leer» dentro del circuito (sin embargo, suele utilizarse poco este tipo de puertos).

23.2.3. Asignaciones concurrentes

Son asignaciones concurrentes aquellas que se ejecutan siempre y directamente sobre una señal; de forma que una señal no puede recibir dos asignaciones concurrentes (daría lugar a error al intentar imponer dos valores a la misma señal).

En lo que sigue, los valores se representan genéricamente con el grafismo *"""""""* y las condiciones (principalmente, comparaciones) con ..... ; denominaremos «expresión» a cualquier conjunto de operaciones entre señales, entre valores y entre ambos y utilizaremos ------- para representarlas; una expresión puede ser un valor, una señal, una operación (o una serie de operaciones) entre señales o entre valores o entre ambos.

Las asignaciones concurrentes pueden ser

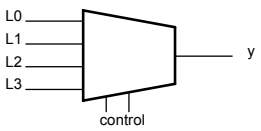
fijas: *señal <= -------;*
es decir, *señal <= valor;* *señal <= señal;*
señal <= operaciones entre señales, entre valores y entre ambos;

condicionales: *señal <= ------- when ..... else*
------- when ..... else
------- when ..... else
-------;

múltiples: *with ------- select*
señal <= ------- when """""",
------- when """""",
------- when """""",
------- when """""",
------- when others;

Ejemplos de asignaciones concurrentes:

Descripción de un multiplexor de cuatro líneas de entrada




```
signal control   : integer range 0 to 3;
```

versión 1

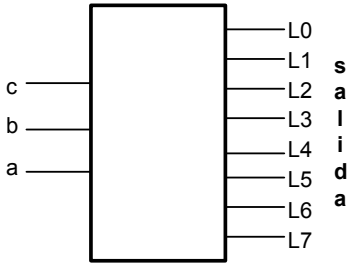
```
y <=    L0  when  control = 0  else
        L1  when  control = 1  else
        L2  when  control = 2  else
        L3;
```

versión 2

```
with  control  select
     y <=   L0  when  0,
            L1  when  1,
            L2  when  2,
            L3  when  others;
```

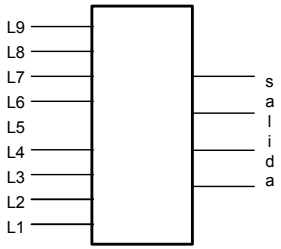
Descripción de un decodificador de ocho líneas

```
entrada <= c & b & a;

with entrada select
salida <=     "10000000" when "000",
              "01000000" when "001",
              "00100000" when "010",
              "00010000" when "011",
              "00001000" when "100",
              "00000100" when "101",
              "00000010" when "110",
                 "00000001" when others;
```



Descripción de un codificador de prioridad de ocho líneas

```
salida  <=    "1001"  when  L9 = '1'  else
              "1000"  when  L8 = '1'  else
              "0111"  when  L7 = '1'  else
              "0110"  when  L6 = '1'  else
              "0101"  when  L5 = '1'  else
              "0100"  when  L4 = '1'  else
              "0011"  when  L3 = '1'  else
              "0010"  when  L2 = '1'  else
              "0001"  when  L1 = '1'  else  "0000";
```



Comparación de números:

```
igual   <=  '1' when  A = B else  '0';
mayor  <=  '1' when (A > B) else  '0';
menor  <=  '1' when (A < B) else  '0';
```

Suma de dos números:

```
R    <=  A + B;
```

También los procesos (*process*), que se describen en el próximo subapartado, son concurrentes (cada uno de ellos considerado globalmente es una asignación concurrente) y no puede asignarse valores a una señal en dos procesos diferentes.

23.2.4. Asignaciones secuenciales

Las asignaciones secuenciales se encuentran dentro de un módulo denominado proceso (*process*) y no se ejecutan hasta que «se ha terminado de leer» todo el módulo. Dentro de un proceso puede haber dos o más asignaciones referidas a la misma señal y es válida la última de ellas; en el caso de asignaciones condicionales (que será el caso general), es válida la ultima de ellas que resulta «efectiva» (es decir, cuyas condiciones se cumplen). Las asignaciones secuenciales no corren peligro de imponer doble valor a una misma señal, pues son consideradas en el orden en que están escrita (igual que un programa de computador) y solamente se aplica la última «efectiva» de ellas.

Los procesos se declaran y se concluyen de la siguiente forma:

nombre_del proceso (*opcional*): *process (*lista de sensibilidad*)*
begin
asignaciones
end process;

La lista de sensibilidad se refiere a las señales que «despiertan» el proceso (que lo hacen operativo) y, en el caso de descripción circuital, debe contener todas las señales que actúan como entradas sobre el proceso. En principio, los compiladores no tienen en cuenta la lista de sensibilidad pero suelen avisar si ésta es incompleta.

Cada proceso, considerado globalmente, es una asignación concurrente (o, si asigna valor a varias señales, un conjunto de asignaciones concurrentes sobre ellas): no se puede efectuar asignación a una misma señal en dos procesos diferentes.

Las asignaciones secuenciales pueden ser fijas, condicionales o múltiples. Las fijas utilizan la misma sintaxis que las asignaciones concurrentes fijas, pero las condicionales y las múltiples utilizan sintaxis diferentes:

condicionales:

```
if .....          then   señal <= -------; señal <= -------; señal <= -------;
    elsif .....   then   señal <= -------; señal <= -------; señal <= -------;
    elsif .....   then   señal <= -------; señal <= -------; señal <= -------;
    else                 señal <= -------; señal <= -------; señal <= -------;
end if;
```

múltiples:

```
case ..... is
        when """""   =>   señal <= -------; señal <= -------; señal <= -------;
        when """""   =>   señal <= -------; señal <= -------; señal <= -------;
        when """""   =>   señal <= -------; señal <= -------; señal <= -------;
        when others  =>   señal <= -------; señal <= -------; señal <= -------;
end case;
```

Dentro de un proceso no pueden utilizarse asignaciones con *when* o con *with* y, de igual forma, las estructuras *if* y *case* no pueden utilizarse fuera de procesos.

Ejemplos de asignaciones secuenciales:

Descripción de un multiplexor de cuatro líneas de entrada

```
process (control,L3,L2,L1,L0)
begin
```

versión 1

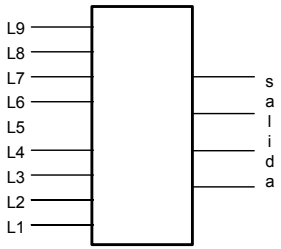
```
if control = 0  then   y  <= L0;   end if;
if control = 1  then   y  <= L1;   end if;
if control = 2  then   y  <= L2;   end if;
if control = 3  then   y  <= L3;   end if;
end process;
```

versión 2

```
case control is
      when 0  =>   y <= L0;
      when 1  =>   y <= L1;
      when 2  =>   y <= L2;
      when others  =>   y <= L3;
end case;
end process;
```

Descripción de un codificador de prioridad de nueve líneas

```
process (L9,L8,L7,L6,L5,L4,L3,L2,L1)
begin
if  L9 = '1'  then              salida  <= "1001";
      elsif  L8 = '1'  then     salida <= "1000";
      elsif  L7 = '1'  then     salida <= "0111";
      elsif  L6 = '1'  then     salida <= "0110";
      elsif  L5 = '1'  then     salida <= "0101";
      elsif  L4 = '1'  then     salida <= "0100";
      elsif  L3 = '1'  then     salida <= "0011";
      elsif  L2 = '1'  then     salida <= "0010";
      elsif  L1 = '1'  then     salida <= "0001";
      else                      salida <= "0000";
end if;
end process;
```

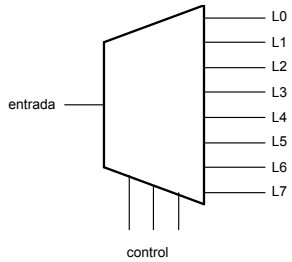


Descripción de un demultiplexor de ocho líneas.

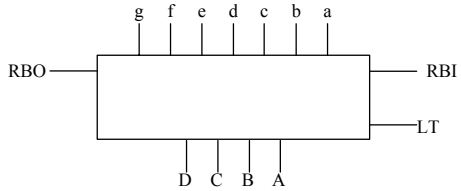
```
control  : integer range 0 to 7

process (control,entrada)
begin
-- asignaciones por defecto
 L0 <= '0'; L1 <= '0'; L2 <= '0'; L3 <= '0';
 L4 <= '0'; L5 <= '0'; L6 <= '0'; L7 <= '0';
```



```
case control is
            when 0  =>   L0 <= entrada;
            when 1  =>   L1 <= entrada;
            when 2  =>   L2 <= entrada;
            when 3  =>   L3 <= entrada;
            when 4  =>   L4 <= entrada;
            when 5  =>   L5 <= entrada;
            when 6  =>   L6 <= entrada;
            when 7  =>   L7 <= entrada;
end case;
end process;
```



23.2.5. Conversor BCD a 7 segmentos de ánodo común



El conversor recibe las 4 entradas BCD y proporciona las 7 salidas (g f e d c b a) correspondientes a la activación de los 7 segmentos (que se activarán con valor **0**, ya que son de ánodo común); además, dispone de una entrada LT para test de lámparas y otra entrada RBI, con su correspondiente salida RBO, para apagado de ceros no significativos.

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

entity BCD_7SEG is
    port (   D,C,B,A,LT,RBI      : in   std_logic;
             SALIDA              : out  std_logic_vector(1 to 7);    -- a b c d e f g
             RBO                 : out  std_logic );
end BCD_7SEG;
```

-- versión 1: con asignaciones concurrentes

```
architecture CODIFICADOR of BCD_7SEG is
signal bcd:       std_logic_vector(3 downto 0);
signal ENTRADA: integer range 0 to 9;
signal AUX:       std_logic_vector(1 to 7);  -- señal auxiliar;
begin
bcd <= D & C & B & A;       ENTRADA <= conv_integer (bcd);
RBO <= '1' when (RBI = '1') and (entrada = 0) else '0';
SALIDA <=       "0000000" when LT = '1' else
                "1111111" when (RBI = '1') and (entrada = 0) else AUX;
```

```
with ENTRADA select                          -- anódo común: activo el 0
        AUX <=          "0000001" when 0,          "1001111" when 1,
                        "0010010" when 2,          "0000110" when 3,
                        "1001100" when 4,          "0100100" when 5,
                        "0100000" when 6,          "0001111" when 7,
                        "0000000" when 8,          "0000100" when 9,
                        "1111111" when others;
end CODIFICADOR;
```

versón 2: con asignaciones secuenciales

```
architecture CODIFICADOR of BCD_7SEG is
signal bcd:       std_logic_vector(3 DOWNTO 0);
signal ENTRADA: integer range 0 to 9;
begin
bcd <= D & C & B & A;       entrada <= conv_integer (bcd);
process(ENTRADA,LT,RBI)
begin
RBO <= '0';
if ( LT = '1' ) then                                SALIDA <= "0000000" ;
   elsif ((RBI = '1') and (ENTRADA = 0)) then       SALIDA <= "1111111";  RBO <= '1';
   else          case ENTRADA is
                          when 0 =>          SALIDA <="0000001";
                          when 1 =>          SALIDA <="1001111";
                          when 2 =>          SALIDA <="0010010";
                          when 3 =>          SALIDA <="0000110";
                          when 4 =>          SALIDA <="1001100";
                          when 5 =>          SALIDA <="0100100";
                          when 6 =>          SALIDA <="0100000";
                          when 7 =>          SALIDA <="0001111";
                          when 8 =>          SALIDA <="0000000";
                          when 9 =>          SALIDA <="0000100";
                          when others =>     SALIDA <="1111111";
                end case;
end if;
end process;
end CODIFICADOR;
```


23.2.6. Decodificador de mapa de memoria

Se trata de situar, en un mapa de memoria cuyo bus de direcciones tiene 16 líneas, un circuito integrado RAM de 8 K al comienzo del mapa, de $0000_{(H}$ a $1FFF_{(H}$, un adaptador de puertos PIA que tiene 4 registros a partir de la posición $A000_{(H}$, y dos circuitos integrados ROM al final de memoria, uno de 2K de $F000_{(H}$ a $F7FF_{(H}$ y el otro de 4K de $F800_{(H}$ a $FFFF_{(H}$; el decodificador de direcciones utiliza, para ello, solamente las 6 líneas superiores del bus de direcciones.

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

entity MAPA is
    port (   A15, A14, A13, A12, A11, A10     : in    std_logic;
             RAM, IO,ROM1,ROM2                : out   std_logic);     end MAPA;

architecture HABILITACIONES of MAPA is
signal DIR       :std_logic_vector (15 downto 0);
-- se introduce la señal auxiliar DIR para referir las direcciones a las 16 líneas
-- del bus de direcciones (y por tanto 16 bits del mapa de memoria)
begin
DIR      <= A15 & A14 & A13 & A12 & A11 & A10 & "0000000000";
RAM      <= '1'  when DIR <= 16#1FFF# else '0';
-- la notación 16#……# indica que el número es hexadecimal
IO       <= '1'  when DIR = 16#A000#;
ROM1     <= '1'  when (DIR >= 16#F000#) and (DIR <= 16#F7FF#) else '0';
ROM2     <= '1'  when DIR >= 16#F800# else '0';
end HABILITACIONES;
```

23.2.7. Comentarios

Las asignaciones condicionales de tipo concurrente han de ser completas: debe especificarse «qué pasa» en caso de que no se cumplan las condiciones; es decir, deben llevar *else* en el caso de *when* y deben llevar *when others* en el caso de *with*.

La asignación múltiple *case* también debe ser completa; debe llevar *when others =>*. También la asignación con *if* debe ser completa (llevar *else* o, alternativamente, haber dado valores por defecto a las señales) si el circuito es combinacional, pues en caso de no serlo introduce biestables para conservar el valor de las señales (según se verá en el próximo apartado: los procesos producen *memoria implícita*).

En el caso de *if* o de *case*, dentro de una misma condición, se pueden hacer asignaciones a varias señales: *señal <= -------; señal <= -------; señal <= -------;* por eso es necesario acabar cada asignación con ";" y finalizar el conjunto de condiciones con *end if* o *end case*.

En una asignación múltiple (*with* o *case*), para referirse a varios casos se utiliza el símbolo "|" para separarlos (no es correcto utilizar "*or*", ya que, en tal caso, se aplicará la operación booleana "**o**"); por ejemplo: *case entrada is when 1 | 3 | 6 => y <= '1'; ...*
with entrada select y <= 1 when 1 | 3 | 6, ...

23.3. Descripción de circuitos secuenciales y sistemas síncronos

23.3.1. Descripción de biestables

En los biestables, la salida actúa también como entrada (realimentación) y, habida cuenta que las salidas VHDL (*port out*) no pueden «ser leidas» desde dentro del circuito (es decir, no pueden actuar como entradas de ninguna asignación), es necesario utilizar para la realimentación una señal interior, del mismo valor que la salida.

```
.......
port( q :out std_logic;
 .......

architecture nombre_de_la_arquitectura of nombre_de_la_entidad is
signal q_interna: std_logic;
begin
q <= q_interna;
```

Por otra parte, los procesos tienen *memoria implícita*: si una señal recibe una asignación condicional dentro de un proceso y el conjunto de asignaciones no es «completo» (es decir, existe alguna condición en que la asignación a dicha señal no está especificada), el proceso asigna por defecto la conservación del valor de dicha señal. Es como si, al comienzo del proceso existiera la asignación *señal <= señal;*, referida, por defecto, a cada una de las señales que reciben alguna asignación dentro del proceso.

Ejemplo:

```
process(a,b)
begin
if a = '1' then p <= b; end if;
end process;
```

En este caso, cuando **a** = **1**, **p** adopta el valor de **b** y, cuando **a** = **0**, como no se especifica nada dentro del proceso, **p** conserva el valor que tenía anteriormente; es equivalente a cualquiera de las dos descripciones siguientes:

```
process(a,b)
begin
if a = '1' then p <= b;
   else p <= p;
end if;
end process;
```

```
process(a,b)
begin
p <= p;
if a = '1' then p <= b; end if;
end if;
end process;
```


Diversas formas de describir un biestable RS

```
q_interna <= '0' when R = '1' else '1' when S = '1' else q_interna;

process (R,S)
begin
if S = '1' then q_interna <= '1'; end if;
if R = '1' then q_interna <= '0'; end if;
end process; -- borrado prioritario

process (R,S)
begin
if R = '1' then q_interna <= '0'; elsif S = '1' then q_interna <= '1'; end if;
end process; -- también borrado prioritario
```

(téngase en cuenta que un proceso conserva los valores: por ello no es necesario añadir en los dos procesos anteriores *else q_interna <= q_interna ;*)

Diversas formas de describir un biestable D

```
q_interna <= D when E = '1' else q_interna;

with E select q_interna <= D when '1', q_interna when others;

process (D,E)
begin  if E = '1' then q_interna <= D; end if;  end process;
```

23.3.2. Circuitos síncronos: descripción del reloj

La descripción de la señal de reloj **CK** ha de hacerse dentro de un proceso, de las siguientes formas:

- si todo el proceso es síncrono

```
process – sin lista de sensibilidad
begin
wait on CK until CK = '1';          -- flanco ascendente
```

- si hay una parte asíncrona (por ejemplo, un borrado asíncrono con **R**)

```
process(R,CK)
begin
if R = '1' then ……..
elsif CK'event and CK = '1' then    -- flanco ascendente
```

o, también, *elsif rising-edge(CK) then* -- flanco ascendente

Biestable D con habilitación y con borrado asíncrono

```
process (R,D,E,CK)
begin
if R = '1' then q_interna <= '0';
elsif CK'event and CK = '1' then
   if  E = '1'   then q_interna <= D;    end if;
end if;
end process;
```

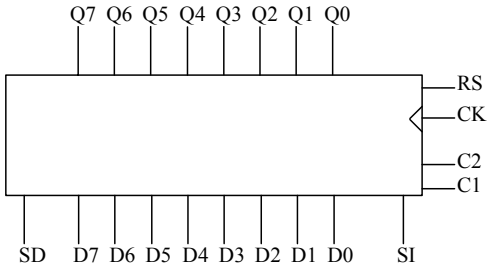
Biestable JK con marcado y borrado asíncronos

```
process (R,S,J,K,CK)
begin
if R = '1' then q_interna <= '0';
elsif S = '1' then q_interna <= '1';
elsif CK'event and CK = '1' then
        if  J = '1'  and K = '1'   then   q_interna <= not q_interna;
        elsif J = '1'              then   q_interna <= '1';
        elsif K = '1'              then   q_interna <= '0';
        end if;
end if;
end process;
```

Registro de desplazamiento bidireccional de 8 bit con carga paralela síncrona

Se trata de diseñar un registro de desplazamiento con las cuatro posibilidades funcionales siguientes, controladas por dos entradas de selección (**C2** y **C1**):

- **00**: retención del valor anterior
- **01**: desplazamiento hacia la izquierda (entrada **SI**)
- **10**: desplazamiento hacia la derecha (entrada **SD**)
- **11**: carga paralelo (entradas **D**)

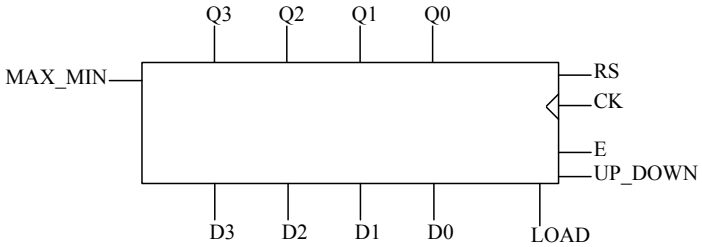



```
library ieee;     use ieee.std_logic_1164.all;

entity REGDESP is
    port (        CK,RS,C1,C2,SI,SD   : in    std_logic;
                  D                   : in    std_logic_vector(7 downto 0);
                  Q                   : out   std_logic_vector(7 downto 0) );
end REGDESP;

architecture SINCRONA of REGDESP is
signal Q_interior                 :std_logic_vector(7 downto 0);
signal control                    :std_logic_vector(2 downto 1);
begin
Q <= Q_interior;        control <= C2 & C1;
REGISTRO: process
-- un proceso puede llevar una etiqueta o «nombre» identificativo delante del mismo
begin
wait until CK = '1';
if ( RS = '1' ) then                              Q_interior <= (others => '0');
      else      case control is
                        when "01" =>      Q_interior <= Q_interior(6 downto 0) & SI;
                        when "10" =>      Q_interior <= SD & Q_interior(7 downto 1);
                        when "11" =>      Q_interior <= D;
                        when others =>
                end case;
end if;
end process;
end SINCRONA;
```

Obsérvese que no es necesario añadir *Q_interior <= Q_interior;* en *when others* ya que un proceso tiene *memoria implícita* (conserva los valores).

23.3.3. Contador década

Contador módulo 10, bidireccional, con habilitación y con borrado y carga paralela síncronos



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

entity DECADA is
    port (      CK,RS,E,UP_DOWN,LOAD     : in   std_logic;
                D                        : in   std_logic_vector(3 downto 0);
                MAX_MIN                  : out  std_logic;
                Q                        : out  std_logic_vector(3 downto 0));
end DECADA;

architecture CONTADOR of DECADA is
signal Q_interior                :std_logic_vector(3 downto 0);
begin
-- COMBINACIONAL:
Q <= Q_interior;
MAX_MIN <= '1' when ((UP_DOWN = '1') and (Q_interior = "1011"))
                or ((UP_DOWN = '0') and (Q_interior = "0000")) else '0';
SINCRONO: process
begin
wait until CK ='1';
if ( RS = '1' ) then                              Q_interior <= "0000";
        elsif ( LOAD = '1' ) then                 Q_interior <= D;
        elsif ( E = '1' ) then
            if ( UP_DOWN = '1') then
                if Q_interior = "1001" then       Q_interior <= "0000";
                   else                           Q_interior <= Q_interior + 1;   end if;
            else  if Q_interior <= "0000" then    Q_interior <= "1001";
                   else                           Q_interior <= Q_interior - 1;   end if;
            end if;
end if;
end process;
end CONTADOR;
```

El mismo contador módulo 10, bidireccional, con habilitación pero con borrado y carga paralela asíncronos

Basta cambiar, en la descripción anterior, el proceso SINCRONO por el siguiente, denominado ASINCRONO

```
ASINCRONO: process(CK,RS,Q_interior,LOAD,E,UP_DOWN)
begin
if ( RS = '1' ) then                              Q_interior <= "0000";
        elsif ( LOAD = '1' ) then                 Q_interior <= D;
        elsif (CK'event and CK='1') then
            if ( E = '1' ) then
```



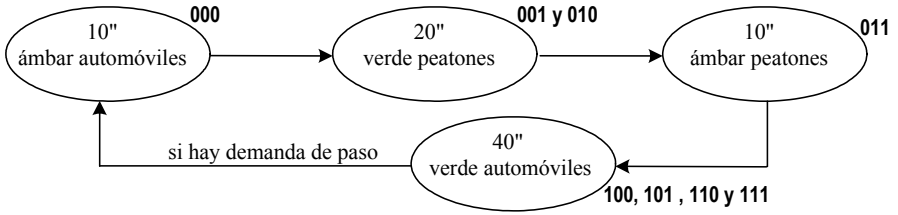
```
            if ( UP_DOWN = '1') then
                if Q_interior = "1001" then       Q_interior <= "0000";
                   else                           Q_interior <= Q_interior + 1;   end if;
            else  if Q_interior <= "0000" then    Q_interior <= "1001";
                   else                           Q_interior <= Q_interior - 1;   end if;
            end if;
        end if;          end if;          end process;
```

23.3.4. Temporizaciones sucesivas: semáforo con demanda de paso para los peatones

Sea un cruce de peatones que cuenta con un semáforo para detener a los automóviles, con un pulsador **P** que debe ser activado por los peatones cuando desean cruzar; la activación de **P** da lugar al siguiente ciclo: 10" en amarillo para detener a los automóviles, 20" en rojo (verde para peatones), 10" en amarillo para peatones, pasando finalmente al estado de circulación de automóviles (rojo para peatones); cuando en dicho estado de circulación se recibe una nueva demanda de paso, es atendida pero asegurando siempre que el intervalo mínimo de paso de automóviles sea de 40".

Se utiliza un biestable **RS** para recoger la demanda de paso por parte de los peatones; dicho biestable se borra en el intervalo de ámbar para peatones (que es cuando se completa el paso de peatones, en respuesta a una demanda anterior). El reloj del sistema es de 1 Hz (1 segundo de período).

El ciclo comienza por el servicio a la demanda de paso (ámbar para automóviles) y dura un total de 80"; mientras hay solicitudes de paso se ejecuta normalmente el ciclo completo, pero, si no hay demanda, el ciclo se detiene al final del mismo (cuarto intervalo de 10" de paso de automóviles) y permanece en dicha situación (paso de automóviles) hasta que se produce una petición de paso por parte de peatones.




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

entity PEATONES is
    port (      CK, RS,DEMANDA                                  : in   std_logic;
                AMBAR,ROJA,VERDE,PAMBAR,PROJA ,PVERDE           : out  std_logic);
end PEATONES;
```

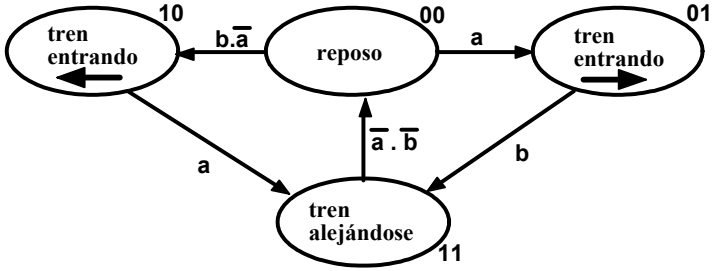
```
architecture TEMPORIZADOR of PEATONES is
signal contador_1          : std_logic_vector(3 downto 0);
signal contador_2          : std_logic_vector(2 downto 0);
signal activo              : std_logic;
begin
-- biestable para guardar la solicitud de paso
SOLICITUD: process(activo, RS, contador_2, DEMANDA)
begin
if  RS = '1'  or contador_2 = "011" then        activo <= '0';
    elsif   DEMANDA = '1'  then                 activo <= '1';
end if;
end process;

TEMPORIZACION: process
-- el contador_1 divide por 10: pasa del reloj de 1" a 10"
begin
wait until CK = '1';
if RS = '1'  then          contador_2 <= "000";     contador_1 <= "0000";
    elsif contador_1 = "1001"  then                 -- cada 10"
            if  contador_2 = "111" then             -- fin de ciclo
                    if activo ='1' then             contador_2 <= "000";
                                                    contador_1 <= "0000";       end if;
                else                                contador_2 <= contador_2 + 1;
                                                    contador_1 <= "0000";
            end if;
    else                                            contador_1  <= contador_1 + 1;
end if;
end process;
SALIDAS: process(contador_2)
begin
VERDE   <= '0';          AMBAR    <= '0';          ROJA   <= '0';
PVERDE <= '0';           PAMBAR <= '0';            PROJA <= '0';
case contador_2 is       when "000" =>     AMBAR <= '1';     PROJA    <= '1';
                         when "001" =>     ROJA    <= '1';   PVERDE  <= '1';
                         when "010" =>     ROJA    <= '1';   PVERDE  <= '1';
                         when "011" =>     ROJA    <= '1';   PAMBAR <= '1';
                         when others  =>   VERDE  <= '1';    PROJA    <= '1';
-- paso de automóviles :others ≡ contador_2 de 100 a 111: 40"
end case;
end process;
end TEMPORIZADOR;
```


23.4. Descripción de grafos de estados

La evolución del estado de un sistema secuencial se describe muy bien con la asignación múltiple *case* para referirse a cada uno de los estados y, dentro de ella, utilizando adecuadamente la asignación condicional *if* para las transiciones. Existen diversas posibilidades para asignar nombres y códigos binarios a los estados; si se prefiere puede dejarse al compilador la tarea de codificar los estados. A continuación se detallan las descripciones VHDL de varios sistemas secuenciales, a partir de sus grafos de estados.

23.4.1. Autómata de Moore: semáforo para cruce de una vía de tren bidireccional con un camino rural

Se trata de un semáforo de aviso de paso de tren en un cruce de vía única bidireccional con un camino; la vía posee, a una distancia adecuadamente grande, sendos detectores de paso de tren **a** y **b**; los trenes circulan por ella en ambas direcciones y se desea que el semáforo señale presencia de tren desde que éste alcanza el primer sensor en su dirección de marcha hasta que pasa por el segundo sensor tras abandonar el cruce.




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

entity SEMAFORO is
    port (      CK,RS,A,B       : in    std_logic;
                SEMF            : out  std_logic);
end SEMAFORO;

architecture GRAFO of SEMAFORO is
-- definición de los estados
subtype mis_estados is std_logic_vector(1 downto 0);
constant reposo                 : mis_estados :="00";
constant entra_por_a            : mis_estados :="01";
constant entra_por_b            : mis_estados :="10";
constant saliendo               : mis_estados :="11";
signal estado    : mis_estados;
begin
```

```
-- evolución del estado:
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                                        estado <= reposo;
else   case estado is
          when reposo  =>        if (A = '1') then          estado <= entra_por_a;
                                    elsif (B = '1') then    estado <= entra_por_b; end if;
          when entra_por_a =>    if (B = '1') then          estado <= saliendo;    end if;
          when entra_por_b =>    if (A = '1') then          estado <= saliendo;    end if;
          when saliendo =>       if (A = '0') and (B = '0') then estado <= reposo; end if;
          when others =>
        end case;
end if;
end process;
-- funciones de activación de las salidas:
SEMF <= '0' when estado = reposo else '1';
end GRAFO;
```

23.3.2. Otras formas de describir los estados

En el ejemplo anterior se ha dado nombre y número binario a los estados mediante la definición de un tipo *mis_estados* y la enumeración de los estados y asignación de valores a los mismos, a través de su declaración como constantes.

```
subtype mis_estados is std_logic_vector(1 downto 0);
constant reposo          : mis_estados :="00";
constant entra_por_a     : mis_estados :="01";
constant entra_por_b     : mis_estados :="10";
constant saliendo        : mis_estados :="11";
signal estado            : mis_estados;
```

Otra forma, más breve, que conduce exactamente a la misma declaración de estados y asignación de valores, es la siguiente:

```
type mis_estados is (reposo, entra_por_a, entra_por_b, saliendo);
attribute enum_encoding: string;
attribute enum_encoding of mis_estados: type is “00 01 10 11”;
signal estado: mis_estados;
```

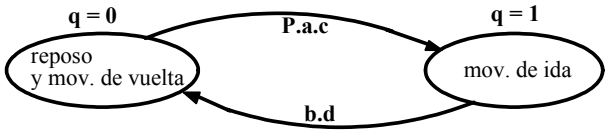
También puede hacerse una declaración de estados sin asignar valores a los mismos, permitiendo que el compilador efectúe esta asignación:

```
type mis_estados is (reposo, entra_por_a, entra_por_b, saliendo);
signal estado: mis_estados;
```


23.4.3. Automáta de Mealy: dos carritos con movimiento de ida y vuelta sincronizados

Sean dos carritos motorizados que se mueven linealmente, entre sendos detectores **a** y **b** el primero y entre **c** y **d** el segundo, de forma que, al activas un pulsador **P**, ambos carritos inician el movimiento desde **a** y **c** y el primero en alcanzar el otro extremo **b** o **d**, espera a que el otro alcance el suyo, para iniciar juntos el movimiento de vuelta.

Un grafo detallado de este sistema de dos carritos puede incluir siete estados (como autómata de Moore) pero puede ser simplificado dando como resultado el grafo siguiente (autómata de Mealy: la necesidad de memoria se limita a distinguir entre dos situaciones: el movimiento de ida hacia **b** y **d** y el de vuelta hacia **a** y **c**):

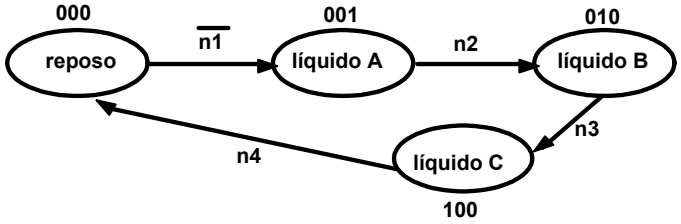



```
library ieee;          use ieee.std_logic_1164.all;

entity CARRITOS is
    port (     CK,RS,A,B,C,D,P              : in    std_logic;
               iz_1,der_1,iz_2,der_2        : out  std_logic);
end CARRITOS;

architecture GRAFO of CARRITOS is
type mis_estados is (vuelta_y_reposo, ida);
signal estado: mis_estados;
begin
-- evolución del estado:
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                        estado <= vuelta_y_reposo;
else   case estado is
          when vuelta_y_reposo =>   if (P and A and C) = '1'  then estado <= ida; end if;
          when ida =>               if (B and D) = '1' then estado <= vuelta_y_reposo;
                                    end if;
       end case;
end if;
end process;
-- funciones de activación de las salidas:
der_1       <= '1' when (estado = ida) and (B='0') else '0';
iz_1        <= '1' when (estado = vuelta_y_reposo) and (A='0') else '0';
der_2       <= '1' when (estado = ida) and (D='0') else '0';
iz_2        <= '1' when (estado = vuelta_y_reposo) and (C='0') else '0';
end GRAFO;
```

23.4.4. Ejemplo realizado con ambas posibilidades, Moore y Mealy: depósito con mezcla de 3 líquidos

Un depósito se llena con una mezcla de tres líquidos diferentes, para lo cual dispone de tres electroválvulas **A**, **B**, **C** que controlan la salida de dichos líquidos y de cuatro detectores de nivel $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_4}$, siendo $\mathbf{n_1}$ el inferior y $\mathbf{n_5}$ el de llenado máximo. Solamente cuando el nivel del depósito desciende por debajo del mínimo $\mathbf{n_1}$ se produce un ciclo de llenado: primero con el líquido A hasta el nivel $\mathbf{n_2}$, luego el líquido B hasta el nivel $\mathbf{n_3}$ y, finalmente, el líquido C hasta completar el depósito $\mathbf{n_4}$.

El grafo de estado correspondiente al autómata de Moore será el siguiente:

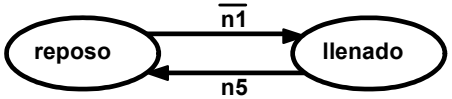



```
library ieee;          use ieee.std_logic_1164.all;

entity DEPOSITO is
    port (      CK,RS,n1,n2,n3,n4          : in    std_logic;
                A,B,C                      : out   std_logic);
end DEPOSITO;

architecture MOORE of DEPOSITO is
subtype mis_estados is std_logic_vector(3 downto 1);
-- código de un solo uno
constant reposo          : mis_estados :="000";
constant liquido_A       : mis_estados :="001";
constant liquido_B       : mis_estados :="010";
constant liquido_C       : mis_estados :="100";
signal estado: mis_estados;
begin
-- funciones de activación de las salidas:
A <= estado(1);    B <= estado(2);    C <= estado(3);
-- evolución del estado:
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                                   estado <= reposo;
```



```
else        case estado is
                when reposo =>      if (n1 = '0') then    estado <= liquido_A;   end if;
                when liquido_A =>   if (n2 = '1') then    estado <= liquido_B;   end if;
                when liquido_B =>   if (n3 = '1') then    estado <= liquido_C;   end if;
                when liquido_C =>   if (n4 = '1') then    estado <= reposo;      end if;
                when others =>
            end case;
end if;
end process;
end MOORE;
```

El mismo sistema secuencial puede configurarse con un número más reducido de estados, según el siguiente grafo que corresponde a un autómata de Mealy (en este caso se necesita solamente una variable de estado **q**, pero las funciones de activación de las salidas resultan más complejas, pues dependen de las entradas, de la información que aportan los detectores de nivel):

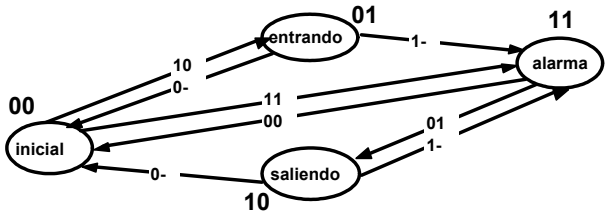



```
architecture MEALY of DEPOSITO is
type mis_estados is (reposo, llenado);
signal estado: mis_estados;
begin
-- evolución del estado
SECUENCIAL: process
begin
wait until CK = '1';
if ( RS = '1' ) then                                   estado <= reposo;
  else      case estado is
                when reposo =>      if (n1 = '0') then    estado <= llenado;   end if;
                when llenado =>     if (n4 = '1') then    estado <= reposo;    end if;
            end case;
end if;
end process;
-- funciones de activación de las salidas:
A           <= '1' when (estado = llenado) and (n2='0') else '0';
B           <= '1' when (estado = llenado) and (n2='1') and (n3 ='0') else '0';
C           <= '1' when (estado = llenado) and (n3='1') else '0';
end MEALY;
```

23.4.5. Ejemplo de grafo con varias transiciones desde cada estado: activación gradual de alarma

Los ejemplos anteriores presentan solamente una transición desde cada estado y en muchos casos actúa solamente una variable de entrada en cada transición; el siguiente grafo, incluye mayor número de transiciones entre estados y dos variables de entrada involucradas en ellas.

*Un sistema de detección de temperatura con cuatro niveles (**00**, **01**, **10**, **11**); la alarma debe activarse cuando se detecta **11** (temperatura muy alta), o si se detecta el nivel **10** (alta) en dos ciclos seguidos de reloj y debe desaparecer cuando se detecta **00** (muy baja), o si se detecta el nivel **01** (baja) en dos ciclos de reloj.*




```
architecture GRAFO of ALARMA is
type mis_estados is (inicial,entrando,alarma,saliendo);
attribute enum_encoding: string;
attribute enum_encoding of mis_estados: type is “00 01 11 10”;
signal estado: mis_estados;
begin
SECUENCIAL: process
begin
wait until CK = '1';
if ( Reset = '1' ) then                                          estado <= inicial;
  else case estado is
        when inicial =>     if (entrada = "10")          then   estado <= entrando;
                               elsif (entrada = "11")    then   estado <= alarma; end if;
        when entrando =>    if (entrada(2) = '1')        then   estado <= alarma;
                               else                              estado <= inicial;  end if;
        when saliendo =>    if (entrada(2) = '1')        then   estado <= alarma;
                               else                              estado <= inicial;  end if;
        when alarma =>      if (entrada = "00")          then   estado <= inicial;
                               elsif (entrada = "01")    then   estado <= saliendo; end if;
      end case;          end if;          end process;

--función de activación de la salida:
y <= '1' when (estado = alarma) or (estado = saliendo) else '0';
end GRAFO;
```


23.5. Otros recursos de VHDL

23.5.1. Tipos de datos

Como ya se hizo en el caso de la declaración de estados en los sistemas secuenciales, se pueden definir y dar valores a constantes, después de la declaración de arquitectura y antes del *begin* de la misma y dichas constantes pueden ser utilizadas para asignación de valores a señales o como parámetros:

```
constant nueve      : std_logic_vector(3 downto 0);="1001";
y <= nueve;
constant k          : integer;=8;
signal ww : std_logic_vector(k-1 downto 0);
```

También pueden definirse parámetros o valores constantes en la declaración de entidad, antes de los puertos, en la forma siguiente:

```
generic( m : integer := 8 );
port( ….
```

Los vectores pueden utilizarse en forma parcial o por componentes; por ejemplo:

```
signal a            : std_logic_vector(15 downto 0);
signal b,c          : std_logic_vector(7 downto 0);
signal d,e          :std_logic;
begin
b <= a(15 downto 8);          -- mitad más significativa del vector a
c <= a(11 downto 4);          -- parte «central» del vector a
d <= a(6);                    -- dígito número 6 del vector a
e <= a(0);                    -- bit menos significativo del vector a
```

Al igual que un vector es un conjunto ordenado y numerado de valores (bits), una matriz (*array*) es un conjunto ordenado y numerado de vectores:

```
signal tt  : array(1 to 64) of std_logic_vector(7 downto 0);
begin
tt(21) <= "10101100";
tt(52,7) <= '1';
```

La declaración anterior de señal introduce una matriz formada por 64 vectores de tipo *standar logic* de 8 bits, numerados de 1 a 64; *tt(21)* se refiere al vector número 21 y *tt(52,7)* señala al bit más significativo del vector número 52.

Variables

En los procesos (y en las funciones, que se explican a continuación), pueden definirse variables internas a los mismos. Mientras que las señales no cambian de valor dentro del propio proceso, sino al final de la aplicación del mismo, las variables cambian de valor dentro del proceso, en cuanto reciben una asignación aplicada sobre ellas.

Las asignaciones a variables se hacen con los símbolos ":=" en lugar de "<=".

Ejemplo de la diferencia funcional entre señales y variables:

***p** señal*	***p** variable*
architecture ejemp1 of entidad is	architecture ejemp1 of entidad is
signal a,b,y,w: std_logic_vector(3 downto 0);	signal a,b,y,w: std_logic_vector(3 downto 0);
signal p : std_logic_vector(3 downto 0);	
begin	begin
....	....
p1: process(a,b,p)	p2: process(a,b)
	variable p : std_logic_vector(3 downto 0);
begin	begin
p <= a or b;	p := a or b;
y <= p;	y <= p;
p <= a and b;	p := a and b;
w <= p;	w <= p;
end process;	end process;
…	…

En el ejemplo de la izquierda (proceso p1) la asignación final que reciben ambas señales **y** y **w** será *a and b*, ya que, como señal, **p** recibe asignación al final del proceso y la anterior asignación (*a or b*) resulta ignorada; en cambio, en el ejemplo de la derecha (proceso p2) la señal **y** recibe la asignación *a or b*, y la señal **w**, *a and b*, (como variable **p** ejecuta sus asignaciones inmediatamente).

Por ejemplo, si el vector **a** tiene valor "1010" y el vector **b** vale "0101", los valores resultantes para **y** y **w** serán:

***p** señal*	***p** variable*
y = "0000" "1010" *and* "0101" = "0000"	y = "1111" "1010" *or* "0101" = "1111"
w = "0000"	w = "0000"

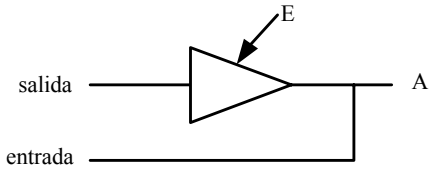
En relación con el diseño digital, las señales se corresponden más directamente con los nudos del circuito; en tal sentido, es preferible la utilización de señales para la propia descripción circuital, mientras que las variables pueden usarse para dar valores a índices, parámetros, etc.

23.5.2. Alta impedancia y bidireccionalidad

Forma de configurar el estado de alta impadancia en señales tri-estado

```
salidas:  out std_logic_vector(7 downto 0)
salidas  <=  "ZZZZZZZZ" when  E = '0' else y;

-- para no especificar el número de valores,
salidas  <=  (others => 'Z')  when  E = '0' else y;

-- dentro de un proceso
if E = '0'  then    salidas  <=  (others => 'Z')
    else            salidas <=  y;              end if;
```

Forma de configurar un terminal bidireccional




```
A                   :inout std_logic;
….
signal entrada,E    : std_logic;
signal salida       :std_logic;
begin
A  <=  salida  when E = '1'  else 'Z';
entrada <= A;
```

Cuando el terminal bidireccional **A** actúa como entrada (E = '0'), recibe el valor de fuera y su asignación como salida debe hacerse a **Z** (alta impedancia).

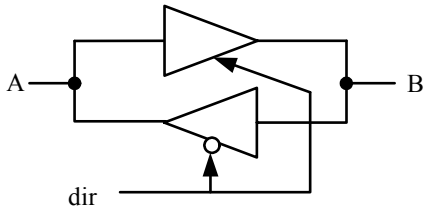
Ejemplo: contador cuyas salidas actúan, también, como entradas paralelo

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

entity  contador is
port(       salidas          :inout std_logic_vector(3 downto 0);
            CK,oe, load      :in    std_logic);
end contador ;

architecture  bidir  of contador is
signal qq: std_logic_vector(3 downto 0);
begin
-- actuación como salidas
salidas  <=  qq when oe = '1' and load = '0' else (others => 'Z');
process
```

```
begin
wait until CK ='1';
if load = '1' and oe = '1'      then      qq <= "0000";
   elsif load = '1'             then      qq <= salidas;
   elsif oe = '1'               then      qq <= qq + 1;
end if;
end process;
end bidir;
```

El contador es completamente síncrono; se borra cuando *load* y *oe* se ponen a **1**, toma el valor de las *salidas* cuando solamente *load* está a **1** y cuenta y presenta el valor del contaje en las *salidas* cuando *oe* se encuentra **1** (con *load* a **0**)

Forma de configurar un adaptador bidireccional (buffer)




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

entity  adaptador is
port(   A,B        :inout std_logic_vector(7 downto 0);
        dir        :in     std_logic);
end adaptador ;

architecture  bidir  of adaptador is
begin
process(A,B,dir)
begin
if dir = '0' then A <= B; else A <= (others => 'Z'); end if;
if dir = '1' then B <= A; else B <= (others => 'Z'); end if;
end process;
end bidir;
```

La descripción anterior corresponde a un adaptador bidireccional de 8 líneas; cuando los terminales actúan como entradas su asignación de salida debe hacerse a Z.

23.5.3. Bucles

Los bucles sirven para aplicar repetitivamente una «instrucción» según un índice que recorre el intervalo de valores que se le fija:

for i in ... to ... loop
asignaciones
end loop;

Para recorrer todo el rango del parámetro i puede utilizarse "*for i in q'range*".

Ejemplo: largo registro de desplazamiento (50 biestables)

```
signal q: std_logic_vector(50 downto 1);
begin
process(reset,clk)
begin
if reset = '1'  then
    for i in 50 downto 1          --- también    for i in q'range
                                  loop      q(i) <= '0';        end loop;
elsif clk'event and clk = '1' then      q(1) <= entrada;
    for i in 50 downto 2   loop         q(i) <= q(i-1);     end loop;
 end if;
end process;
```

Otro ejemplo: pila FIFO (64 registros de 8 bits)

```
type pila is array(1 to 64) of std_logic_vector(7 downto 0);
signal qqqq          :pila;
signal entrada       : std_logic_vector(7 downto 0);
begin
salida <= qqqq(64);
process(reset,clk)
begin
if reset = '1'  then
   for i in qqqq'range       loop      qqqq(i) <= "00000000";   end loop;
elsif clk'event and clk = '1'   then   qqqq(1) <= entrada;
   for i in 2 to 64          loop      qqqq(i) <= qqqq(i-1);      end loop;
end if;
end process;
```

23.5.4. Funciones

Una función consiste en un conjunto de asignaciones, cuya aplicación a las entradas de la función sirve para devolver un valor; se describe al inicio de la arquitectura, antes del *begin* de la misma y puede ser llamada, dentro de la arquitectura, cuantas veces sea necesaria. Dentro de una función no puede ir una instrucción de espera (*wait*), ni una actuación por flancos (*reloj*).

Función: *function* nombre_de_la_función *(entradas: tipo)*
return tipo_de_la_salida is
begin
asignaciones
return ……;
end;

llamada a la función: *señal <= nombre_de_la_función (entradas);*

Ejemplo: descripción de una simple celda sumadora utilizando funciones

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

entity CELDA is
port(   a,b,arrastre_in      : in  std_logic;
        suma,arrastre_out    : out std_logic);        end CELDA ;

architecture SUMADOR of CELDA is

function paridad (a,b,c: std_logic) return std_logic is
begin   return ((a xor b) xor c);       end;

function mayoria (a,b,c: std_logic) return std_logic is
begin   return ((a and b) or (b and c) or (c and a));       end;

begin
suma <= paridad(a,b,arrastre_in);
arrastre_out <= mayoria(a,b,arrastre_in);
end SUMADOR ;
```

Otra función: multiplexor que deja pasar el mayor de 2 números

```
function mayor(a,b: std_logic_vector) return std_logic_vector  is
variable y: std_logic_vector(7 downto 0);
begin   if a > b then y := a;    else y := b;    end if;
return y;       end;
```

a) dejar pasar el mayor de 3 números A, B, C

```
t <= mayor(A,B);          Y <= mayor(t,C);
```

o también,

```
                          Y <= mayor(mayor(A,B),C);
```


b) dejar pasar el número intermedio de entre 3 números A, B, C

```
process(A,B,C)
variable t,u,v: std_logic_vector(7 downto 0);
begin
t := mayor(A,B);          u := mayor(B,C);          v := mayor(C,A);
if (t = u) then Y <= v;   elsif (u = v) then Y <= t;   else Y <= u;   end if;
end process;
```

ahora bien, esta solución, aunque «ingeniosa», requiere mucha circuitería (del orden de cinco comparadores y dos multiplexores de 2 números) y puede resolverse en forma mucho más reducida; por ejemplo,

```
Y <= A;
if (A > B) = (B > C) then Y <= B; end if;
if (A > C) /= (B > C) then Y <= C; end if;
```

que ocupa poco más de la mitad (unos dos multiplexores y tres comparadores).

Otro ejemplo de función: contador de minutos, segundos y décimas (cronómetro)

Se trata de un cronómetro, con resolución de una décima de segundo y capacidad de contaje hasta una hora; se utilizan dos funciones para configurar los diversos contadores módulo 10 y módulo 6, respectivamente.

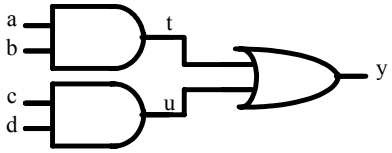
```
library ieee;   use ieee.std_logic_1164.all;   use ieee.std_logic_unsigned.all;

entity CRONO is
port (      clk,rs,hab              : in std_logic;
            max                     : out std_logic;
            decimas                 : out std_logic_vector(3 downto 0);
            minutos,segundos        : out std_logic_vector(6 downto 0));
end CRONO;

architecture CONTADORES of CRONO is

  -- contaje módulo 10
  function mod10 (e :std_logic; q:std_logic_vector) return std_logic_vector is
  variable cont :std_logic_vector(3 downto 0);
  begin
          if e = '1' then
             if q = "1001" then cont := "0000"; else cont := q + 1; end if;
          else cont := q;     end if;
  return cont;    end;
```

```
-- contaje módulo 6
  function mod6 (e :std_logic; q:std_logic_vector) return std_logic_vector is
  variable cont:std_logic_vector(6 downto 4);
  begin
          if e = '1' then
          if q = "101" then cont := "000"; else cont := q + 1; end if;
          else cont := q;     end if;
  return cont;    end;

signal seg,min                          : std_logic_vector(6 downto 0);
signal dec                              : std_logic_vector(3 downto 0);
signal max1,max2,max3, max4             : std_logic;
-- señales para indicar el máximo ( 9 ó 5 ) de los contadores

begin
decimas <= dec;
segundos <= seg;
minutos <= min;
max1 <= '1' when dec(3 downto 0) = "1001" and hab   = '1' else '0';
max2 <= '1' when seg(3 downto 0) = "1001" and max1 = '1' else '0';
max3 <= '1' when seg(6 downto 4) = "101"   and max2 = '1' else '0';
max4 <= '1' when min(3 downto 0) = "1001" and max3 = '1' else '0';
max   <= '1' when min(6 downto 4) = "101"   and max4 = '1' else '0';

process(rs,clk)
begin
if rs = '1' then dec <= (others => '0');    seg <= (others => '0');     min <= (others => '0');
elsif clk'event and clk = '1' then
          dec(3 downto 0) <=   mod10(hab,dec(3 downto 0));
          seg(3 downto 0) <= mod10(max1,seg(3 downto 0));
          seg(6 downto 4) <=  mod6(max2,seg(6 downto 4));
          min(3 downto 0) <= mod10(max3,min(3 downto 0));
          min(6 downto 4) <=  mod6(max4,min(6 downto 4));
end if;
end process;
end CONTADORES;
```


23.5.5. Descripción estructural: conexión de módulos

Hasta aquí, la descripción VHDL de sistemas digitales se ha desarrollado en forma «funcional», pero este lenguaje admite también una descripción «estructural», detallando las conexiones entre celdas o módulos circuitales.

Ejemplo: descripción estructural de una sencilla combinación de puertas en forma de suma de productos




```
library  ieee;   use ieee.std_logic_1164.all;
entity puerta_and is port(a,b  : in std_logic;     y  : out std_logic);   end puerta_and;
architecture uno of puerta_and is   begin  y<='1' when a='1' and b='1' else '0';  end uno;

library  ieee;   use ieee.std_logic_1164.all;
entity puerta_or is port(a,b  : in std_logic;     y  : out std_logic);   end puerta_or;
architecture dos of puerta_or is   begin  y<='1' when a='1' or b='1' else '0';  end dos;

library  ieee;   use ieee.std_logic_1164.all;
entity suma_de_productos is
port(a,b,c,d  : in std_logic;     y  : out std_logic);
end suma_de_productos;

architecture andor of suma_de_productos is

component puerta_and port(a,b : in std_logic;  y  : out std_logic);  end component;

component puerta_or   port(a,b : in std_logic;  y  : out std_logic);  end component;

signal t,u: std_logic;

begin
 u1: puerta_and port map(a=>a, b=>b, y=>t);
 u2: puerta_and port map(a=>c, b=>d, y=>u);
 u3: puerta_or    port map(a=>t, b=>u, y=>y);
end andor;
```

Obsérvese que es necesario declarar la librería y el paquete que se utilizan en cada entidad: la declaración de librerías y paquetes se «satura» (se gasta) cada vez que dicha librería se utiliza.

Los componentes del ejemplo anterior son muy pequeños (simples puertas lógicas básicas), pero de igual forma se procedería para conectar entre sí módulos más amplios (descritos cada uno de ellos con su entidad y su arquitectura), a fin de formar un circuito con ellos.

Otro ejemplo: descripción estructural del cronómetro anterior

Cronómetro con resolución de una décima de segundo y capacidad hasta una hora; se utilizan en esta descripción como módulos los contadores módulo 10 y módulo 6.

```
library ieee;  use ieee.std_logic_1164.all;   use ieee.std_logic_unsigned.all;
entity mod10 is
port(e,ck,rs  : in std_logic;  q  : out std_logic_vector(3 downto 0); max  : out std_logic);
end mod10;
architecture contador1 of mod10 is  signal qqq  : std_logic_vector(3 downto 0);   begin
q <= qqq;   max <= '1' when qqq = "1001" and e = '1' else '0';
process(rs,ck)   begin
      if rs = '1' then qqq <= (others => '0');
      elsif ck'event and ck = '1' then
         if e = '1' then
            if qqq = "1001" then qqq <= (others => '0'); else qqq <= qqq + 1; end if;
      end if;   end if;   end process;   end contador1;

library ieee;  use ieee.std_logic_1164.all;   use ieee.std_logic_unsigned.all;
entity mod6 is
port(e,ck,rs  : in std_logic;  q  : out std_logic_vector(2 downto 0); max  : out std_logic);
end mod6;
architecture contador2 of mod6 is  signal qqq  : std_logic_vector(2 downto 0);   begin
q <= qqq;   max <= '1' when qqq = "101" and e = '1' else '0';
process(rs,ck)   begin
      if rs = '1' then qqq <= (others => '0');
      elsif ck'event and ck = '1' then
         if e = '1' then
            if qqq = "101" then qqq <= (others => '0'); else qqq <= qqq + 1; end if;
      end if;   end if;   end process;   end contador2;

library ieee;   use ieee.std_logic_1164.all;   use ieee.std_logic_unsigned.all;

entity CRONO is
port (      clk,rs,hab              : in std_logic;
            max                     : out std_logic;
            decimas                 : out std_logic_vector(3 downto 0);
            minutos,segundos        : out std_logic_vector(6 downto 0));
end CRONO;
```



```
architecture CONTADORES of CRONO is
component mod10 port(e,ck,rs  : in std_logic;  q  : out std_logic_vector(3 downto 0);
  max  : out std_logic); end component;
component mod6 port(e,ck,rs  : in std_logic;  q  : out std_logic_vector(2 downto 0);
  max  : out std_logic); end component;

signal max1,max2,max3, max4          : std_logic;

begin
u1: mod10 port map(e=>hab, ck=>clk, rs=>rs, q=>decimas, max=>max1);
u2: mod10 port map(e=>max1, ck=>clk, rs=>rs, q=>segundos(3 downto 0),
max=>max2);
u3: mod6 port map(e=>max2, ck=>clk, rs=>rs, q=>segundos(6 downto 4), max=>max3);
u4: mod10 port map(e=>max3, ck=>clk, rs=>rs, q=>minutos(3 downto 0), max=>max4);
u5: mod6 port map(e=>max4, ck=>clk, rs=>rs, q=>minutos(6 downto 4), max=>max);
end CONTADORES;
```

Bucle para generar módulos: generate

etiqueta: for i in ... to ... generate
inserción de módulos
end generate;

Ejemplo: descripción estructural de un registro de desplazamiento de 25 biestables

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

entity biestable is
port(ck,rs,d  : in std_logic;    q  : buffer std_logic);   end biestable;

architecture ff of biestable is   begin
process(ck,rs,q)    begin
if rs='1' then  q<='0';   elsif ck'event and ck='1' then  q<=d;  end if;
end process;   end ff;

library ieee;  use ieee.std_logic_1164.all;

entity registro is
port(ck,rs,entrada  : in std_logic;    y  : buffer std_logic_vector(25 downto 1));
end registro;

architecture desplazamiento of registro is
component biestable port(ck,rs,d  : in std_logic;    q  : out std_logic);  end component;
begin
u1: biestable port map(ck=>ck, rs=>rs, d=>entrada, q=>y(1));
```

```
gen1 : for i in 2 to 25 generate
        u_resto: biestable port map(ck=>ck, rs=>rs, d=>y(i-1), q=>y(i));
end generate;    end desplazamiento;
```

En este ejemplo, la instrucción *generate* da lugar a la inserción de 24 componentes de tipo biestable (numerados del 2 al 25).

Otro ejemplo: divisor de frecuencia por 10^6

Un modulo divisor de frecuencia por un millón sirve, por ejemplo, para pasar de una frecuencia de 1 MHz (período de 1 microsegundo) a la de 1 Hz (período 1 segundo).

```
library  ieee;   use ieee.std_logic_1164.all;    use ieee.std_logic_unsigned.all;
entity contador is port(ck,rs,E  : in std_logic;     max  : out std_logic);   end contador;
architecture modulodiez of contador is   begin
process(ck,rs,E)
variable q: std_logic_vector(3 downto 0);    begin
if E = '1' and q = "1001" then max <= '1'; else max <= '0'; end if;
if rs='1' then  q := "0000";
elsif ck'event and ck='1' then
      if E = '1' then if q = "1001" then  q := "0000"; else q := q +1;  end if;    end if;
end if;  end process;    end modulodiez ;
library  ieee;   use ieee.std_logic_1164.all;    use ieee.std_logic_unsigned.all;
entity divisor is  port(ck,rs,hab  : in std_logic;     y  : out std_logic);   end divisor;
architecture unmillon of divisor is
component contador  port(ck,rs,E  : in std_logic;   max  : out std_logic);  end component;
signal aa :std_logic_vector(6 downto 0) ;
begin
y <= aa(6);       aa(0) <= hab;
gen1 : for i in 1 to 6 generate
        u_resto: contador port map(ck=>ck, rs=>rs, E=>aa(i-1), max=>aa(i));
end generate;     end unmillon;
```

23.5.6. Librerías y paquetes

Una *librería* es una «carpeta» (un directorio) en la que se almacenan diseños (con su entidad y su arquitectura, cada uno de ellos) y paquetes, que pueden ser utilizados en otros diseños. Un *paquete* es un fichero, dentro de una librería, en el que se declaran componentes, constantes, tipos o funciones.

Librerías y paquetes sirven para organizar y ordenar los diseños y para aprovechar el trabajo generando módulos reutilizables. La librería directa de trabajo, en la que se desarrolla el diseño actual, se denomina *work* y no es necesario declararla.

Ejemplo de librerías y paquetes son los declarados habitualmente al inicio del texto:

```
library ieee;
use ieee.std_logic_1164.all; use ieee.std_logic_arith.all; use ieee.std_logic_unsigned.all;
```

Esta declaración indica el uso de la librería "ieee"; dentro de ella se usan los paquetes "std_logic_1164", "std_logic_arith" y "std_logic_unsigned"; y dentro de dichos paquetes se usan todos los elementos contenidos en los mismos. Si se necesitase sólo un elemento, bastaría poner el nombre del elemento (en lugar de *all*).

Un paquete tiene una parte declarativa y otra descriptiva:

declaración de un paquete:

package nombre_del_paquete *is*

-- declaración de tipos, constantes, componentes, funciones y/o procedimientos

subtype nombre *is ... ...;*

constant nombre *:tipo;*

component nombre *port(... ... ...) end component;*

function nombre *(entradas) return tipo;*

end;

descripción de un paquete:

package body nombre_del_paquete *is*

-- valores de las constantes y descripción de componentes, funciones y procedimientos

entity y *architecture* de cada componente

constant nombre*: tipo := valor;*

function nombre *(entradas) return tipo is*

begin

asignaciones

return expresión;

end nombre*;*

end;

para utilizar el paquete

library nombre_de_la_librería *;*

use nombre_de_la_librería.nombre_del_paquete.*all;*