INTRODUCCIÓN A LOS VHDL

AUTOR: Cavallero, Rodolfo Antonio. rcavallero@scdt.frc.utn.edu.ar

INTRODUCCION

El lenguaje de programación **VHDL** (**V**ery High Speed Integrated Circuit **H**ardware ***D***escription **L**anguaje) es una nueva metodología de diseño de circuitos digitales. Es un lenguaje que describe el comportamiento del circuito, es decir describe el hardware

VHDL fue desarrollado como un lenguaje para el modelado y simulación lógica el cual posee una sintaxis amplia y flexible y permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido, y el desarrollo de modelos de simulación.

Algunas ventajas del uso de VHDL para la descripción de hardware son:

- VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de puertos
- Circuitos descriptos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados 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
- Esta basado en un standart IEEE std 1076-1987, IEEE std 1076-1993
- Modularidad: VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades mas pequeñas.

ESTILOS DE DESCRIPCIÓN CIRCUITAL

Existen dos formas de **describir un circuito**. Por un lado se lo puede describir indicando los diferentes componentes que lo forman y su interconexión **(estructura)**, de esta manera se tiene especificado un circuito y se sabe como funciona. Esta es la forma habitual, siendo las herramientas utilizadas para ello la ***captura esquematica*** y las de descripción ***netlist.***

La segunda forma consiste en **describir un circuito** indicando lo que hace o como funciona, es decir, describiendo su **comportamiento**. Naturalmente esta forma de describir un circuito es mucho mejor para un diseñador puesto que lo que realmente le interesa es el funcionamiento del circuito mas que sus componentes.

ESTILOS DE DESCRIPCIÓN EN VHDL

VHDL presenta tres estilos de descripción de circuitos dependiendo del nivel de abstracción. El menos abstracto es una descripción puramente estructural. Los otros dos estilos representan una descripción comportamental o funcional, y la diferencia viene de la utilización o no de la ejecución en serie.

En la Fig 1 se observan los tres estilos de descripción

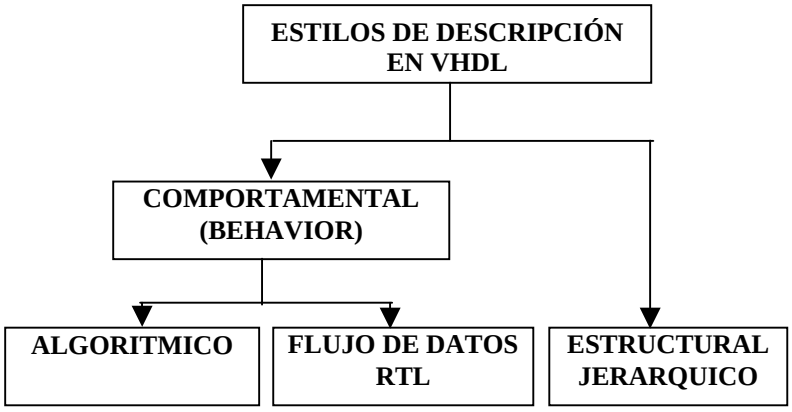


Fig. 1 Estilos de descripción VHDL

El estilo de descripción comportamental, puede ser dividido en dos tipos de estilo: Flujo de Datos y Algoritmico, tal como se observa en la Fig. 1
La representación de Flujo de Datos describe como se mueve el dato a través del sistema (en realidad entre registros). El Flujo de Datos hace uso de sentencias concurrentes, las cuales son ejecutadas en paralelo tan pronto como el dato llega a la entrada. Por otra parte las sentencias secuenciales son ejecutadas en las secuencias que son especificadas.

DESCRIPCIÓN ESTRUCTURAL:

Aunque no es la caracteristica mas interesante del VHDL, tambien permite ser usada como *Netlist* o lenguaje de descripción de estructura. Las caracteristicas mas significativas son:

- Descomponer la entidad en su estructura de elementos mas simples: **COMPONENTES** lo que implica diversos niveles de descripción (jerarquías)
- Declarar los **COMPONENTES** dentro de la arquitectura
- No se define el comportamiento, solo los terminales externos
- Realizar el interconexionado

DESCRIPCIÓN ALGORITMICA:

Aquí se describe el comportamiento del sitema, no se esta indicando ni los componentes ni sus interconexiones, sino simplemente lo que hace. Las caracteristicas mas significativas son:

- Se emplean sentencias secuenciales y no concurrentes: **PROCESOS**
- Se ejecutan en un orden determinado, se finaliza la ejecución de una sentencia antes de pasar a la siguiente

DESCRIPCIÓN FLUJO DE DATOS:

Es una descripción intermedia entre la comportamental y la estructural. En el se describe el sistema mediante diagramas de transferencia entre registros, tablas de verdad o ecuaciones booleanas. Los elementos básicos son: registros, memorias, lógica combinacional y buses.
Las caracteristicas mas significativas son:

- Se emplean sentencias concurrentes, es decir de ejecución paralela
- Se puede dividir su funcionamiento en una serie de pasos y en cada paso el circuito debe realizar cierta funcion que se traduce en la transferencia de datos entre registros y evaluar ciertas condiciones para pasar al siguiente paso

Presentaremos algunos ejemplos de utilización de código VHDL y gradualmente nos iremos introduciendo en el lenguaje. No es pretensión de este trabajo un estudio profundo del lenguaje, sino dar a conocer una herramienta poderosa para el desarrollo de los sistemas digitales. Aquellos que quieran profundizar en este tema pueden consultar la bibliografía que a continuación se detalla.

VHDL – Lenguaje para síntesis y modelado de circuitos – Fernado Pardo y Jose Boluda
Editorial Alfaomega
VHDL - David Maxinez - Editorial C.E.C.S.A

DESCRIPCION DE UN MODELO VHDL

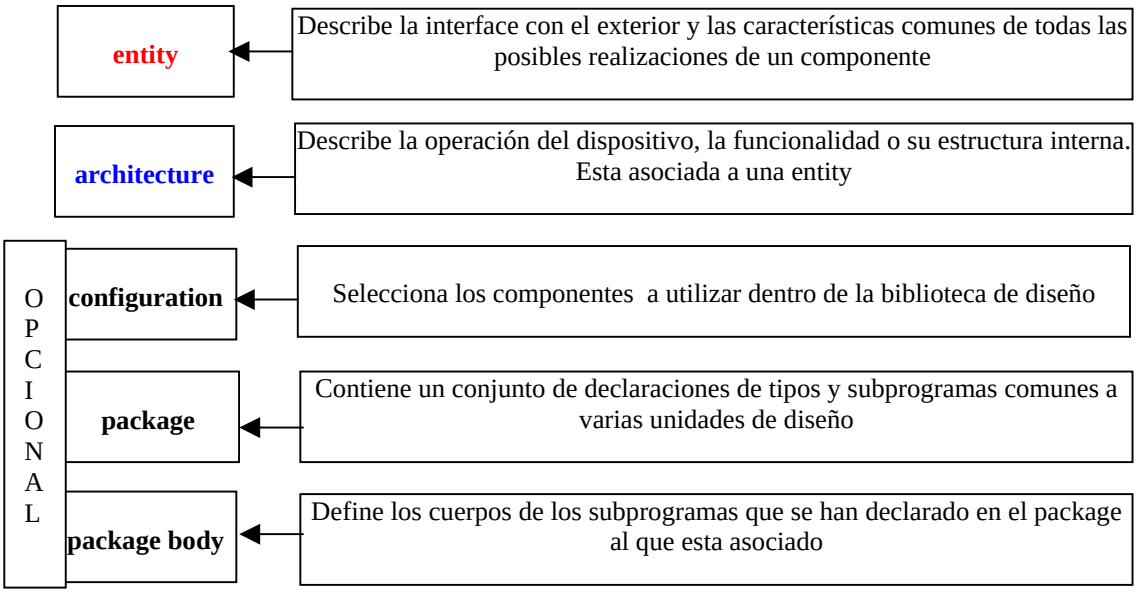


Fig. 2 Descripción de un modelo VHDL

ESTRUCTURA BASICA DE UN ARCHIVO FUENTE VHDL

ENCABEZAMIENTO	**Library** <nombre_libreria> **Use**<nombre_librería>.<nombre_paquete>. **all**
ENTIDAD	**Entity**<nombre_entidad>**is** <listado de puertos> --Declaración de pines **end** <nombre_entidad>;
ARQUITECTURA	**Architecture**<nombre_arquitectura> **of**<nombre_entidad> **is** --Declaracion de señales internas --Declaracion de tipos de datos definidos por el usuario --Declaracion de componentes en caso de instanciación **begin** --Cuerpo de la arquitectura --Se define la funcionalidad del diseño con: --Asignaciones concurrentes --Procesos --Instanciación de componentes **end**<nombre_arquitectura>;

Fig. 3 Estructura básica de un archivo fuente VHDL

ENTIDAD Y ARQUITECTURA

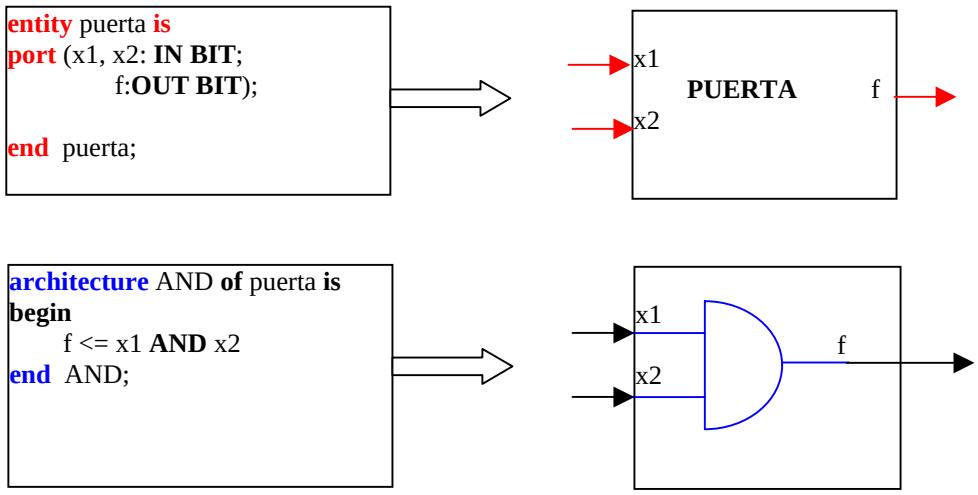


Fig. 4 Entidad y Arquitectura

Ejemplo 1

X3	X2	X1	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

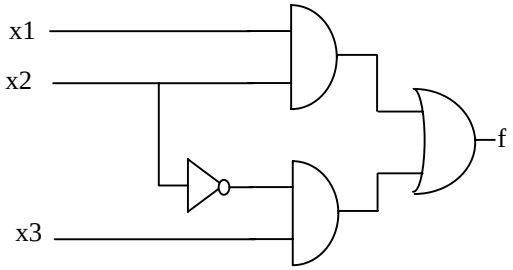


Fig. 5 Diagrama circuital

La función de salida una vez minimizada es: $f = x1.x2 + \bar{x2}.x3$

A continuación escribimos el codigo VHDL correspondiente

```
ENTITY ejemplo1 IS
        PORT (x1, x2, x3 :  IN     BIT;
                    f            :  OUT  BIT);
END ejemplo1;

ARCHITECTURE LogicFunc  OF ejemplo1   IS
BEGIN
        f <= (x1 AND x2) OR (NOT x2 AND x3);
END LogicFunc;
```

Fig. 6 Código VHDL del ejemplo 1

Ejemplo 2

En la Fig. 7 se da otro ejemplo de codigo VHDL . Alli observamos que el circuito tiene 4 entradas (x1, x2, x3, x4) y dos salidas (f, g). El circuito lógico producido por el complilador de VHDL es el mostrado en la Fig. 8

```
ENTITY ejemplo2 IS
        PORT (x1, x2, x3, x4 :  IN     BIT;
                f,g                :  OUT  BIT);
END ejemplo2;

ARCHITECTURE LogicFunc  OF ejemplo2   IS
BEGIN
        f <= (x1 AND x3) OR (NOT x3 AND x2);
        g <= (NOT x3 OR x1) AND (NOT x3 OR x4);
END LogicFunc;
```

Fig. 7 Código VHDL de ejemplo 2

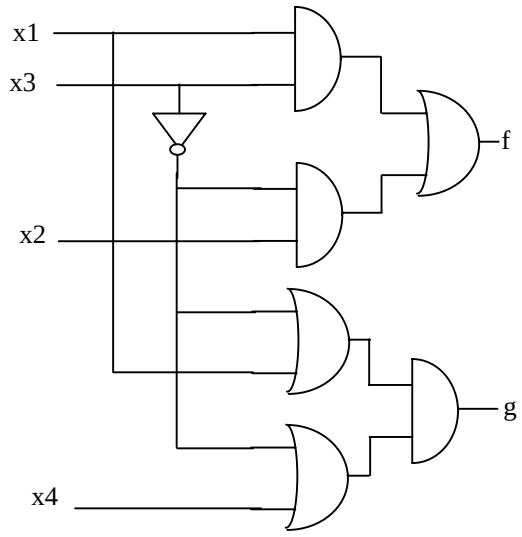


Fig.8 Compilación código fig. 7

Ejemplo 3

Sea la siguiente función canónica y su simplificada

$f = \Sigma\ 1.\ 4.\ 5.\ 6$	$f = x1.\overline{x3} + \overline{x2}.x3$

El código VHDL se muestra en la Fig. 9

```
ENTITY ejemplo3 IS
        PORT (x1, x2, x3 :  IN     BIT;
                f                :  OUT  BIT);
END ejemplo3;
```

```
ARCHITECTURE LogicFunc  OF ejemplo3   IS
BEGIN
        f <= (NOT x1 AND NOT x2 AND x3) OR
             (x1AND NOT x2 AND NOT x3) OR
              (x1 AND NOT x2 AND x3) OR
              (x1 AND X2 AND NOT x3);
END LogicFunc;
```

Fig. 9 Código VHDL funcion lógica ejemplo 3

Ejemplo 4

En los ejemplos anteriores vimos que las señales de I/O eran del tito BIT. Con el objeto de darle mayor flexibilidad, VHDL provee otro tipo de datos llamados STD_LOGIC, los que pueden tener diferentes valores. En la Fig 10 damos un ejemplo utilizando STD_LOGIC. Aquí el código VHDL que se muestra es el mismo del ejemplo 3, de la Fig. 9, excepto que el tipo es STD_LOGIC eb vez de BIT. Además se incluyen dos líneas adicionales que son utilizadas por el complilador como directivas ya que los compiladores primitivos no las incluían.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ejemplo4 IS
        PORT (x1, x2, x3 :  IN     STD_LOGIC;
                    f              :  OUT  STD_LOGIC);
END ejemplo4;

ARCHITECTURE LogicFunc  OF ejemplo4   IS
BEGIN
        f <= (NOT x1 AND NOT x2 AND x3) OR
             (x1AND NOT x2 AND NOT x3) OR
              (x1 AND NOT x2 AND x3) OR
              (x1 AND X2 AND NOT x3);
END LogicFunc;
```

Fig. 10 Código VHDL ejemplo 4

Para STD_LOGIC hay un numero de valores legales, pero los mas importantes son **: 0, 1, z** (alta impedancia) y **-** (condiciones no importa)

Ejemplo 5 (Semisumador)

Desarrollemos el código VHDL para un semisumador de un bit

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY semisumador IS
        PORT (x1, x2,  :       IN     BIT;
                    Suma,carry:  OUT  BIT);
END semisumador;
```

```
ARCHITECTURE semi  OF semisumador   IS
BEGIN
        suma<= x1 XOR x2;
        carry<= x1 AND x2;
END semi;
```

Fig. 11 Código VHDL ejemplo 5

El esquema circuital correspondiente será:

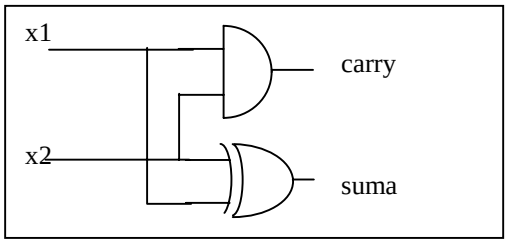


Fig. 12 Esquema circuital semisumador

Ejemplo 6 (Sumador total 1)
En la Fig. 13 se observa un circuito sumador total

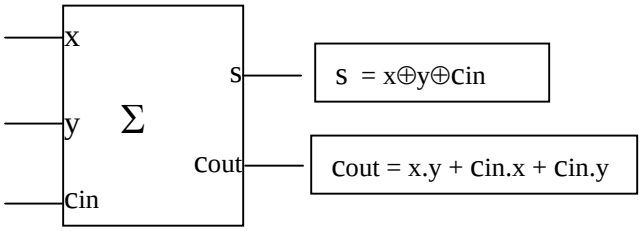


Fig. 13 Diagrama Sumador total

En la Fig 14 se muestra el código VHDL para el sumador total de la Fig. 13

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fulladd IS
        PORT (cin, x, y :  IN     STD_LOGIC;
                s, cout       :  OUT  STD_LOGIC);
END fulladd;

ARCHITECTURE LogicFunc  OF fulladd   IS
BEGIN
        s <= (x XOR y XOR cin;)
        cout<= (x AND y) OR (cin AND x) OR ( cin AND Y);
END LogicFunc;
```

Fig. 14 Código VHDL sumador total

En VHDL se dispone de los siguientes mecanismos de *descripción estructural*:

- **Definición de componentes**: en VHDL los componentes se declaran con la palabra clave especial **COMPONENT.** Es algo muy parecido a **ENTITY**
- **Enlace entre componentes y entidades**: Cada componente debe enlazarse con una entidad y una arquitectura definida. De esta manera el componente es como una interfase que se puede referenciar en el diseño tantas veces como se necesite. Este enlace entre componentes, entidades y arquitectura se realiza mediante la sentencia **FOR.**
- **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.
- **Referencia de componentes**: Un mismo componente se puede referenciar o *instanciar* tantas veces como se necesite. La forma en que se hace es mediante las instrucciones de referencia que se veran mas adelante.
- **Configuración**: Algunas de las definiciones anteriores, especialmente la de enlace se pueden especificar en un bloque especial llamado configuración.

Ejemplo 7 (sumador total 2)

Desarrollaremos un codigo mediante la descripción jerarquica/estructural en la cual se descompone la entidad en su estructura de elementos mas simples: **COMPONENTES**, los cuales son declarados dentro de la **arquitectura** antes de la sentencia **begin**
La declaracion es similar a la de una **entidad**, no se define el comportamiento, solo los terminales externos
La sintaxis es:

```
COMPONENT  nombre_componente
   PORT  (port_list):
END  nombre_componente ;
```

El comportamiento debe definirse fuera de la arquitectura donde se declara. Definiendo una **entidad** De igual nombre que el **componente** y con una arquitectura asociada al mismo.
Una vez declarados los componentes, se pueden utilizar dentro de las sentencias concurrentes de la arquitectura asociando señales a entradas y salidas de los mismos según el orden de declaracion
La sintaxis es:

```
Etiqueta: nombre_componente
          PORT MAP  (lista_asociación_ ports);
```

Diseño jerarquico de un sumador de 1 bit (utiliza semisumadores y compuertas)

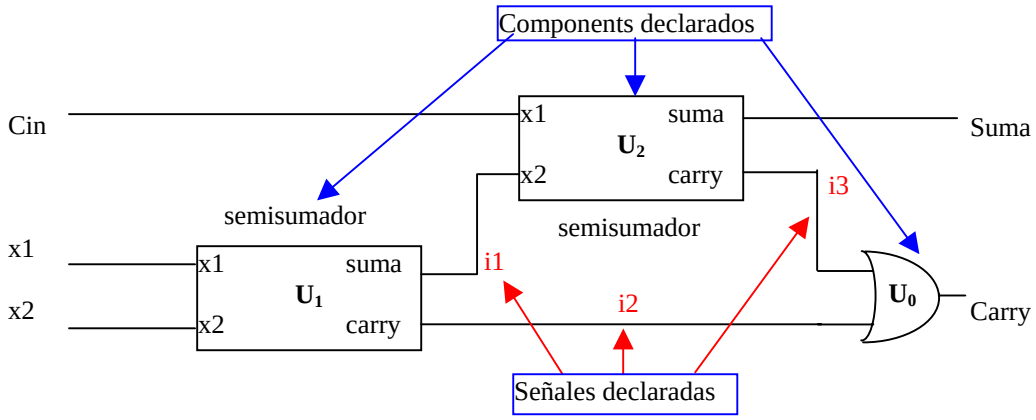


Fig. 15 Diagrama sumador total con semisumadores

. . ENTITY sumador IS PORT (x1,x2,cin: IN BIT; Suma, carry: OUT BIT); END sumador;	
ARCHITECTURE suma OF sumador IS	
SIGNAL i1,i2,i3 : BIT	señales locales
--Declaracion del componente semisumador COMPONENT semisumador PORT (x1, x2: IN BIT; suma,carry: OUT BIT); END COMPONENT; --Puerta como componente COMPONENT puerta_or PORT(x1, x2: IN BIT; z: OUPUT BIT); END COMPONENT;	DECLARACION COMPONENTES
BEGIN u1: semisumador PORT MAP (x1, x2, i1, i2); u2: semisumador PORT MAP (cin,i1, suma, i3); u3: puerta_or PORT MAP (i2, i3, carry); END suma;	ARQUITECTURA CONECTANDO COMPONENTES
--Definicion del componente semisumador ENTITY semisumador IS PORT (x1, x2: IN BIT; Suma, carry: OUT BIT); END semisumador;	DECLARACION DEL COMPONENTE COMO ENTIDAD
ARCHITECTURE semi OF semisumador IS BEGIN Suma <= x1 XOR x2; Carry <= x1 AND x2; END semi;	DEFINICION DE SU COMPORTAMIENTO
--Definicion de components puerta_or ENTITY puerta_or IS PORT(x1, x2 : IN BIT; z: OUT BIT); END puerta_or	PUERTA OR COMO ENTIDAD
ARCHITECTURE puerta-or OF puerta_or Is BEGIN z<= x1 OR x2 END puerta_or	DEFINICION DEL COMPORTAMIENTO

Fig. 16 Código VHDL sumador total

En este ejemplo se podria haber sustituido la linea u3: puerta_or POR MAP (i2, 13, carry);
Por la asignación : carry <= i2 OR i3; sin necesidad de definier el componente puerta_or

Ejemplo 8 (Sumador total de 4 bits)

Siguiendo con el circuito sumador, intentaremos desarrollar un código VHDL para un sumador de 4 bits. Como sumando tenemos : x3, x2, x1, x0, -- y3, y2, y1, y0, y como resultado: s3, s2, s1, s0

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY adder4 IS
        PORT ( Cin          : IN    STD_LOGIC;
               x3,x2,x1,x0  : IN    STD_LOGIC;
               y3,y2,y1,y0  : IN    STD_LOGIC;
               s3,s2,s1,s0  : OUT   STD_LOGIC;
               Cout         : OUT   STD_LOGIC);
END adder4;

ARCHITECTURE Structure OF adder4 IS
        SIGNAL c1,c2,c3: STD_LOGIC;
        COMPONENT fulladd
                PORT ( Cin, x, y : IN       STD_LOGIC;
                       S, Cout  : OUT       STD_LOGIC);
        END COMPONENT;
BEGIN
        Stage0: fulladd PORT MAP (Cin, x0, y0, s0, c1);
        Stage1: fulladd PORT MAP (C1, x1, y1, s1, c2);
        Stage2: fulladd PORT MAP (C2, x2, y2, s2, c3);
        Stage3: fulladd PORT MAP (
                Cin => c3, Cout => Cout,x => x3, y => y3, s => s3) ;
END Structure;
```

Fig. 17 Código VHDL para un sumador de 4 bits

Las tres señales declaradas como : c3, c2, c1, son usadas como señales de carry-out de las tres primera etapas del sumador. La siguiente sentencia *component declaration*, es la que permite que la *entity* fulladd sea usada como componente (subcircuito) en el cuerpo de *architecture*. El sumador de 4 bits esta descripto usando una sentencia de reproducción o réplica (*instantiation*).

Ej: *stage0:* fulladd PORT MAP (Cin, x0, y0, s0, c1);

Donde PORT MAP define la conexion del Puerto

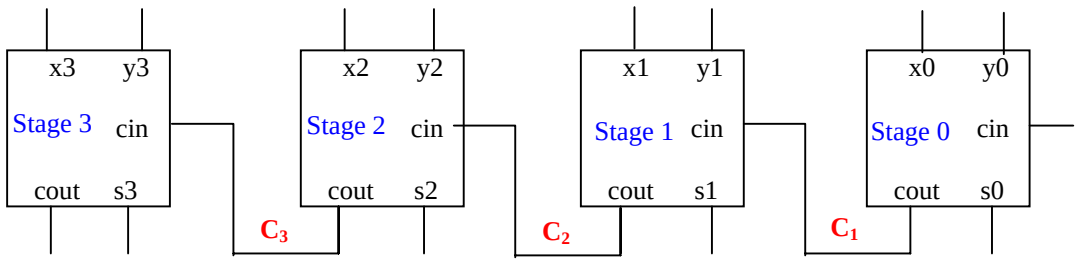


Fig. 18 – Interconexión de bloques

Note que *stage 3*, el nombre de señal Cout es usada tanto en el nombre del puerto del componente, como en el nombre de la señal en la *entity adder4.* Esto no cauda problemas para el compilador VHDL, debido a que el nombre del puerto del componente es siempre el primero del lado izquierdo del carácter =>.
Los nombres de las señales asociadas a cada ejemplo de los componentes del *fulladd*, implícitamente especifican con el full-adder esta conectado. Por ejemplo el *carry-out* de la *stage0* esta conectada al *carry-in* de la *stage1*.

Ejemplo 9 (Sumador total de 4 bits)

Observando el código VHDL del ejemplo 8 vemos que se ha incluido en la *architecture adder4* la declaración del componente (*component*) fulladd.
Una alternativa a ello es colocar la sentencia de declaración del componente en un VHDL *package* . Esto permite a VHDL construcciones que al ser definidas en un archivo código fuente pueda ser usado en otros archivos de codigos fuente. Dos ejemplos de construcciones que son colocadas en paquetes son las declaraciones de *data type* y *component*

El código de la Fig 19 define el paquete *fulladd_package*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

PACKAGE  fulladd_package IS
        COMPONENT fulladd
                PORT (Cin, x, y : IN        STD_LOGIC;
                        s, Cout    : OUT    STD_LOGIC);
        END COMPONENT;
END fulladd_package;
```

Fig. 19 Declaración de un paquete (package)

Este código puede ser almacenado en un archivo separado del codigo fuente VHDL, o puede ser incluido en el mismo archivo de codigo fuente utilizado para almacenar el codigo para la *entity fulladd*, que se vio en la Fig. 17
Cualquier *entity* VHDL puede luego utilizar el *component fulladd* como un subcircuito haciendo uso del paquete *fulladd_package,* cuyo acceso se realiza de la siguiente forma

```
LIBRARY work;
USE work.fulladd_package.all;
```

La libreria nombrada *work* , representa el directorio de trabajo donde el codigo VHDL define el paquete almacenado. Esta sentencia realmente no es necesaria ya que el compilador VHDL siempre tiene acceso al *working directory* .
En la Fig. 20 mostramos como el codigo dado en la Fig. 17 puede ser reescrito para hacer uso del *fulladd_package* .

En las Figs. 17 y 20, cada una de las 4 entradas y salidas del sumador esta representada usando 4 señales de un solo bit. Un estilo muy conveniente de codigo es usar señales de múltiple bits para representar números.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
        PORT ( Cin          : IN    STD_LOGIC;
               x3,x2,x1,x0  : IN    STD_LOGIC;
               y3,y2,y1,y0  : IN    STD_LOGIC;
               s3,s2,s1,s0  : OUT   STD_LOGIC;
               Cout         : OUT   STD_LOGIC);
END adder4;

ARCHITECTURE Structure OF adder4 IS
        SIGNAL c1,c2,c3 :  STD_LOGIC;
BEGIN
        Stage0: fulladd PORT MAP (Cin, x0, y0, s0, c1);
        Stage1: fulladd PORT MAP (C1, x1, y1, s1, c2);
        Stage2: fulladd PORT MAP (C2, x2, y2, s2, c3);
        Stage3: fulladd PORT MAP (
                Cin => c3, Cout => Cout,x => x3, y => y3, s => s3) ;
END Structure;
```

Fig. 20 Una forma diferente de especificar un sumador de 4 bits

Representación de Números en VHDL

Así como en circuitos lógicos un número es representado por señales en un conexionado de múltiples cables, un número en VHDL es representado como una señal de datos multibit. Un ejemplo de esto es:

```
SIGNAL C : STD_LOGIC_VECTOR (1 TO 3)
```

El tipo de dato STD_LOGIC_VECTOR representa un arreglo lineal del dato STD_LOGIC.
La declaración SIGNAL define a C como una señal STD_LOGIC de 3 bits. Si por ejemplo asignamos

```
C <= "100"
```

Ello significa que C(1) = 1 , C(2) = 0 y C(3) = 0
Otro tipo de declaración SIGNAL es:

```
SIGNAL X: STD_LOGIC_VECTOR (3 DOWNTO 0);
```

Lo que define a X como una señal de STD_LOGIC_VECTOR de 4 bits, especificando que el bit mas significativo de X es designado X(3) y el menos significativo X(0)

Ejemplo: X<= “1100” significa: X(3) = 1, X(2)=1, X(1) = 0 , X(0)= 0

Ejemplo 10 (Sumador total de 4 bits)
En la Fig 21 se ilustra como el código de la Fig. 20 puede ser reescrito para utilizar señales de múltiple bits

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
        PORT ( Cin    : IN     STD_LOGIC;
                X, Y  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                S     : OUT    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                Cout  : OUT    STD_LOGIC);
END adder4;

ARCHITECTURE Structure OF adder4 IS
        SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
        Stage0: fulladd PORT MAP (Cin, X(0), Y(0), S(0), C(1) );
        Stage1: fulladd PORT MAP (C(1), X(1), Y(1), S(1), C(2) ) ;
        Stage2: fulladd PORT MAP (C(2), X(2), Y(2), S(2), C(3) ) ;
        Stage3: fulladd PORT MAP (C(3), X(3), Y(3), S(3), Cout ) ;
END Structure;
```

Fig. 21 Sumador de 4 bits definidos usando señales de multiple bit

Sentencias de Asignación Aritmética

Si definimos:

```
SIGNAL X, Y, S: STD_LOGIC_VECTOR(15 DOWNTO 0);
```

Luego las sentencia de asignacion aritmética

```
S<= X + Y
```

Representa a un sumador de 16 bits

Ejemplo 11 (Sumador de 16 bits)

En la Fig. 22 vemos el uso de estas sentencias y se ha incluido el paquete *std_logic_signed* para permitir el uso del operador adición (+)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
        PORT ( X, Y   : IN     STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                  S       : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
END adder16;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
        S <= X + Y ;
END Behavior;
```

Fig. 22 Código VHDL para una sumador de 16 bits

Ejemplo 12 (Sumador total de 16 bits)
El código de la Fig. 22 no incluye las señales Cin y Cout y tampoco el overflow. Ello se subsana con el código dado en la Fig. 23
El bit 17 designado *sum* , es usado por el Cout desde la posición 15 del sumador.
El término entre paréntesis (0&X), es que el 0 esta concatenado al bit 16 de la señal X para crear el bit 17. En VHDL el operador & se llama operador *concatenate*. Este operador no tiene el significado de la funcion AND. La razón de introducirlo en el código de la Fig. 23 es que VHDL requiere que al menos uno de los operandos X o Y tengan el mismo numero de bits que el resultado
Otro detalle es la sentencia:

```
S <= Sum (15 DOWNTO 0);
```

Esta sentencia asigna los 16 bits menos significativos de *Sum* a la salida S

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
        PORT ( Cin              : IN     STD_LOGIC;
                  X, Y             : IN     STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                  S                : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                  Cout, Overflow  : OUT  STD_LOGIC) ;
END adder16;

ARCHITECTURE Behavior OF adder16 IS
        SIGNAL Sum : STD_LOGIC_VECTOR(16 downto 0) ;
BEGIN
        Sum <= ('0' & X) + Y + Cin ;
        S <= Sum(15 Downto 0) ;
        Cout <= Sum(16) ;
        Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Fig. 23 Sumador de la Fig. 22 con señales de carry y overflow

La sentencia : Cout <= Sum (16); asigna el carry-out de la adición ,(*sum (16)*), a la señal carry-out, (*Cout*)

La expresión de overflow es Cn-1 XOR Cn , en nuestro caso Cn = *Sum (16)* y por otra parte

$$Cn\text{-}1 = X(15) \oplus Y(15) \oplus S(15)$$

El uso del paquete *std_logic_signed* permite que las señales STD_LOGIC sean utilizadas con operadores aritméticos. Este paquete realmente usa otro paquete , el *std_logic-arith*, el que define dos tipos de datos llamados SIGNED Y UNSIGNED, para uso en circuitos aritméticos que tratan con numeros con y sin signo.

Ejemplo 13 (Sumador total de 16 bits)
El codigo de la Fig. 23 puede ser reescrito para usar directamente el paquete *std_logic-arith*, tal como se observa en la Fig. 24

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
        PORT ( Cin              : IN     STD_LOGIC;
                X, Y            : IN     SIGNED(15 DOWNTO 0) ;
                S               : OUT  SIGNED(15 DOWNTO 0) ;
                Cout, Overflow  : OUT  STD_LOGIC) ;
END adder16;

ARCHITECTURE Behavior OF adder16 IS
        SIGNAL Sum : SIGNED(16 downto 0) ;
BEGIN
        Sum <= ('0' & X) + Y + Cin ;
        S <= Sum(15 DOWNTO 0) ;
        Cout <= Sum(16) ;
        Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Fig. 24 Uso del paquete aritmetico

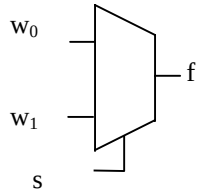
Observe que las señales multibit X, Y, S y *sum* tienen el tipo SIGNED.

VHDL Para Circuitos Combinacionales:

Ejemplo 14 (Mux 2 a 1)

Asignación de las señales de selección

En la Fig. 25 se describe el código de un multiplexor de 2 a 1. La selección de la señal se realiza mediante la entrada *s*.




```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
        PORT ( w0, w1, s        : IN     STD_LOGIC;
                  f             : OUT  STD_LOGIC ) ;

END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
        WITH s SELECT
             f <=   w0 WHEN '0' ,
                    w1 WHEN OTHERS;
END Behavior ;
```

Fig. 25 Código VHDL para un multiplexor 2 a 1

Observamos dentro de la architectura la palabra clave WITH, la que especifica que *s* va ha ser usada para la selección. Hay dos clausulas WHEN, las que establecen ue a la salida *f* le sea asignada el valor de w0 cuando *s* = 0, u otro si *s* = 1 (others: 1, z, -)

Ejemplo 15 (Mux 4 a 1)

En la Fig. 26 se muestra el codigo para un multiplexor de 4 a 1. Al final de la Fig. 26 la entidad *mux4to1* esta definida como un componente en el paquete denominado *mux4to1_package* (esta entidad puede ser luego utilizada como un subcircuito de otro código VHDL

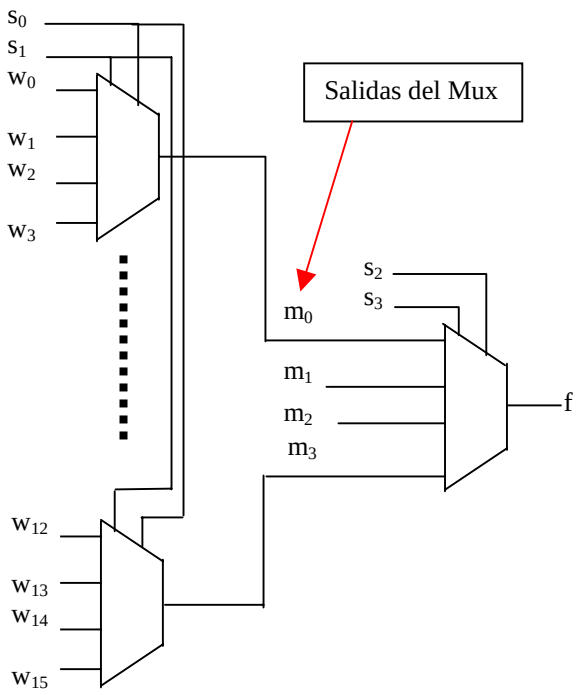
```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
        PORT ( w0, w1, w2, w3   : IN     STD_LOGIC ;
                  s             : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                  f             : OUT STD_LOGIC ) ;
END mux4to1 ;
```

```
ARCHITECTURE Behavior OF mux4to1 IS
BEGIN
        WITH s SELECT
             f <=   w0 WHEN “00”,
                    w1 WHEN “01”,
                    w2 WHEN “10”.
                    w3 WHEN OTHERS ;
END Behavior ;

LIBRARY ieee;
USE ieee.std_logic_1164.all ;
PACKAGE mux4to1 package IS
      COMPONENT mux4to1
           PORT ( w0, w1, w2, w3   : IN     STD_LOGIC ;
                  s                : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                  f                : OUT STD_LOGIC ) ;
END COMPONENT ;
END mux4to1_package ;
```

Fig. 26 Código VHDL para un multiplexor 4 a 1

Ejemplo 16 (Mux 16 a 1)

En la Fig. 27 se presenta un codigo VHDL para un multiplexor de 16 a 1 usando el componente *mux4to1*



Multiplexor 16 a 1

En la linea 11 se definen 4 señales denominadas *m*, que son las salidas de los multiplexores 4 a 1.-

Las lineas 13 a 16 reproducen (replican) los 4 multiplexores. Por ejemplo, la linea 13 corresponde al multiplexor que esta en la parte superior izq del esquema de la Fig. 27 La sintaxis *s*(1 DOWNTO 0) se utiliza para vincular la señal s(1) y s(0) a los dos bits del puerto *s* del componente *mux4to1*. La señal *m(0)* esta conectada al puerto de salida del multiplexor . La linea 17 reproduce el multiplexor que tiene como entradas a m0, m1. m2, m3 y como entradas de selección s2 y s3.-

La compilación del código da como resultado la seguiente función:

$$f = \bar{s}_3.\bar{s}_2.\bar{s}_1.\bar{s}_0.\, w_0 + \bar{s}_3.\bar{s}_2.\bar{s}_1.s_0.\, w_1 + \bar{s}_3.\bar{s}_2.s_1.\bar{s}_0.\, w_2 + \ldots\ldots\ldots + s_3.s_2.s_1.s_0.\, w_{14} + s_3.s_2.s_1.s_0.\, w_{15}$$

```
1     LIBRARY ieee;
2     USE ieee.std_logic_1164.all ;
3     LIBRARY work ;
4     USE work.mux4to1_package.all ;

5     ENTITY mux16to1 ;
6         PORT ( w  : IN     STD_LOGIC_VECTOR(0 TO 15) ;
7                s  : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
8                f  : OUT STD_LOGIC ) ;
9     END mux16to1 ;

10    ARCHITECTURE Structure OF mux16to1  IS
11        SIGNAL m :  STD_LOGIC_VECTOR(0 TO 3) ;
12    BEGIN
13        Mux1: mux4to1 PORT MAP
                ( w(0), w(1), w(2), w(3), s(1 DOWNTO 0), m(0) ) ;
14        Mux2: mux4to1 PORT MAP
                ( w(4), w(5), w(6), w(7), s(1 DOWNTO 0), m(1) ) ;
15        Mux3: mux4to1 PORT MAP
                ( w(8), w(9), w(10), w(11), s(1 DOWNTO 0), m(2) ) ;
16        Mux4: mux4to1 PORT MAP
                ( w(12), w(13), w(14), w(15), s(1 DOWNTO 0), m(3) ) ;
17        Mux5: mux4to1 PORT MAP
                ( m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f ) ;
18    END Structure ;
```

Fig. 27 Código jerárquico para un multiplexor 16 a 1

Ejemplo 17 (Decodificador de 2 a 4)

En	w1	w0	y0	y1	y2	y3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

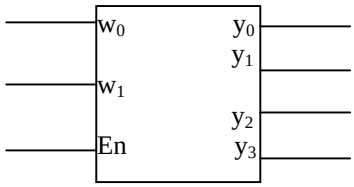


Fig. 28 T.V y diagrama decodificador

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
        PORT ( w     : IN    STD_LOGIC_VECTOR(1DOWNTO 0) ; ;
                  En   : IN    STD_LOGIC ;
                  y    : OUT  STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
     SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
     Enw <= En & w ;
       WITH Enw SELECT
            y <=   "1000"  WHEN "100",
                   "0100"  WHEN "101",
                   "0010"  WHEN "110",
                   "0001"  WHEN "111",
                   "0000"  WHEN OTHERS ;
END Behavior ;
```

Fig. 29 Código VHDL para un decodificador 2 a 4

Se puede utilizar la entrada *select* para describir otro tipo de circuitos tal como un decodificador.

La sentencia Enw <= En&w, indica que Enw es una entrada de 3 bits (En = una entrada y w = 2 entradas) En otras palabras, hemos creado una señal de entrada de 3 bits concatenando (*concatenate*) una de 2 bits (w) con una de un bit (En)

Aquí: Enw(2) = En; Enw(1) = w1; Enw(0) = w0

Asignación de señales condicionales

Similar a la asignación de señales de selección, la asignación de señales condicionales permiteque una señal sea forzada a uno de varios valores.. En la Fig. 30 se muestra una versión modificada del codigo dado en la Fig. 25. Se usa una asignación condicional para especificar que *f* es asignada al valor w0, cuando (WHEN) *s* = 0 o de otro modo (ELSE) *f* es asignada a w1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
        PORT ( w0, w1, s    : IN     STD_LOGIC ;
                    f               : IN     STD_LOGIC ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
      f <= w0 WHEN s = '0' ELSE w1 ;
END Behavior ;
```

Fig.30 Especificaciones para un mux 2 a 1 utilizando señales de asignacion condicionales.

Ejemplo 18 (Decodificador de prioridad)

Usamos la siguiente tabla de verdad de un codificador de prioridad de 4 a 2

Entradas				Salidas		
w3	w2	w1	w0	y1	y0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

W0 : la mas baja prioridad
W3 : la mas alta prioridad
d : dont care

Fig. 31 T.V Codificador de prioridad

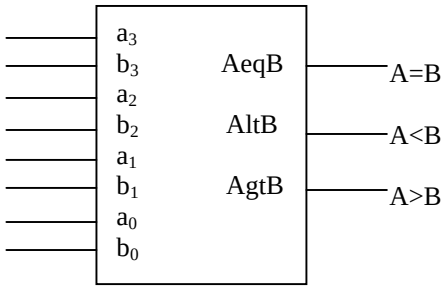
y1 e y0 representan el numero binario de la prioridad. Por ejemplo y1 = 1, y0 = 1, esto representa que w3 es la de mayor prioridad.
Si todas las entradas (wi) son cero entonces z indica esa condicion mediante el valor de z = 0. Si z = 1 indica que hay una entrada activada. El codigo VHDL que describe esta tabla de verdad se observa en la Fig. 31

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;


ENTITY priority IS
        PORT ( w     : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  y     : OUT  STD_LOGIC _VECTOR(1 DOWNTO 0) ;
                  z     : OUT  STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
      y <=   "11"  WHEN w(3) = '1' ELSE,
             "10"  WHEN w(2) = '1' ELSE,
             "01"  WHEN w(1) = '1' ELSE,
             "00" ;
      z <= '0' WHEN w = "0000" ELSE '1' ;
END Behavior ;
```

Fig. 32 Código VHDL para un codificador de prioridad

Ejemplo 19 (Comparador de 4 bits)

En la Fig. 33 se muestra como el circuito comparador del esquema que se muestra a continuación puede ser descripto según el código VHDL.



Cada una de las 3 señales condicionales determinan el valor de una de las salidas del comparador. El paquete denominado *std_logic_unsigned* esta incluido en el código debido a que el especifica que las señales STD_LOGIC_VECTOR, denominadas A, y B pueden ser usadas como números binarios sin signo, con operadores relacionales VHDL. El operador relacional provee una manera conveniente de especificar la funcionalidad deseada. El circuito que se genera, compilador mediante, es similar al del esquema circuital que se da en este ejemplo, pero no idéntico.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
```

```
ENTITY compare IS
        PORT ( A, B                    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  AeqB, AgtB, AltB   : OUT STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
      AeqB <=   '1'  WHEN A = B ELSE `0`;
      AgtB <=   '1'  WHEN A > B ELSE `0`;
      AltB  <=   '1'  WHEN A < B ELSE `0`;
END Behavior ;
```

Fig. 33 Codigo VHDL para un comparador de 4 bits

Otra forma de especificar el circuito es incluir la librería denominada *std_logic_arith* . En ambos casos las señales A y B deberían estar definidas con el tipo UNSIGNED, mas bien que STD_LOGIC_VECTOR. Si nosotros queremos que el circuito trabaje con numeros con signo, las señales A y B deberían ser definidas con el tipo SIGNED. Este código se muestra en la Fig. 34

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY compare IS
        PORT ( A, B                    : IN     SIGNED(3 DOWNTO 0) ;
                  AeqB, AgtB, AltB   : OUT STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
      AeqB <=   '1'  WHEN A = B ELSE `0`;
      AgtB <=   '1'  WHEN A > B ELSE `0`;
      AltB  <=   '1'  WHEN A < B ELSE `0`;
END Behavior ;
```

Fig 34 Codigo de la Fig. 33 para números con signo

Sentencia Generate

El código mostrado en la Fig. 27 representa a un multiplexor de 16 a 1 usando 5 replicas de subcircuitos multiplexores de 4 a 1. Hay una manera de hacer mas compacto este código. VHDL provee un mecanismo llamado sentencia FOR GENERATE para describir códigos jerárquicamente estructurados

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE work.mux4to1_package.all ;
```

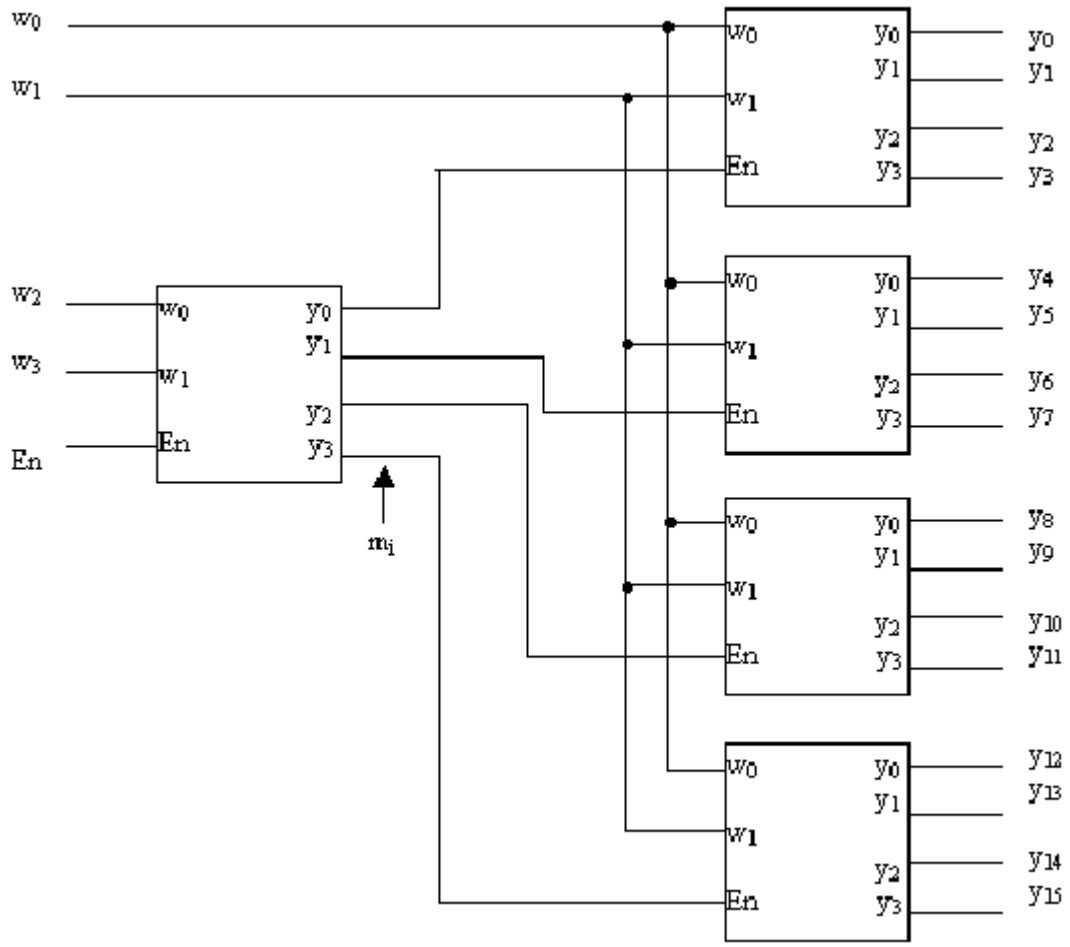
```
ENTITY mux16to1 IS
        PORT ( w     : IN    STD_LOGIC_VECTOR(0 TO 15) ;
                 s     : IN    STD_LOGIC _VECTOR(3 DOWNTO 0) ;
                 f     : OUT  STD_LOGIC ) ;
END mux16to1 ;

ARCHITECTURE Structure OF mux16to1 IS
     SIGNAL m : STD_LOGIC_VECTOR (0 TO 3) ;
BEGIN
     G1:  FOR i IN 0 TO 3 GENERATE
          Muxes : mux4to1 PORT MAP (
                w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 DOWNTO 0), m(i) ) ;
     END GENERATE ;
     Mux5: mux4to1 PORT MAP (m(0), m(1), m(2), m(3), s(3 DOWNTO 2), f) ;
END Structure ;
```

Fig. 35 Código para un Mux 16 a 1 usando sentencia *generate*

En la Fig. 35 se muestra el código rescrito de la Fig. 27 usando la sentencia mencionada
La sentencia GENERATE debe tener una etiqueta que en nuestro caso denominamos G1.
El lazo (loop) reproduce 4 copias del componente *mux4to1*, utilizando el lazo con indice *i* en el rango de 0 a 3. La variable *i* no esta explícitamente declarada en el código, esta variable se define automáticamente como variable local cuyo alcance esta limitado a la sentencia FOR GENERATE. El primer lazo de repetición (iteración) corresponde a la replicación de la sentencia etiquetada *Mux1* de la Fig. 27. El operador * representa multiplicación, aquí para la iteración del primer lazo, el compilador VHDL traduce el nombre de las señales *w(4*i) , w(4*i+1), w(4*i+2) y w(4*i+3)* en señales con nombre w(0), w(1) , w(2) y w(3). El lazo de iteración para i =1 , i = 2 e i = 3, corresponde a la sentencia etiquetada *Mux2, Mux3 y Mux4* en la Fig. 27 . La sentencia etiquetada *Mux5* en la Fig. 27 no encaja dentro del lazo, por ello esta incluida como una sentencia separada en la Fig. 35 . El circuito generado por el código de la Fig. 35 es idéntico al producido usando el código de la Fig. 27 .

Ejemplo 20 (Decodificador 4 a 16)

Veamos ahora la sentencia IF GENERATE, la cual esta incluida en el código de la Fig. 36 junto con FOR GENERATE. El código muestra una descripción jerarquica del decodificador de 4 a 16 que se muestra a continuación



Aquí usamos 5 replicas del componente *dec2to4* definido en la Fig. 29 . La entradas al decodificador es una señal *w* de 4 bits, la habilitacion es *En* y la salida es de 16 bits y se denomina *y*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY dec4to16 IS
        PORT ( w   : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  En : IN     STD_LOGIC  ;
                  y    : OUT  STD_LOGIC_VECTOR (0 TO 5) ) ;
END dec4to16 ;

ARCHITECTURE Structure OF dec4to16 IS
      COMPONENT dec2to4
            PORT ( w    : IN    STD_LOGIC_VECTOR (1 DOWNTO 0) ;
                      En  : IN    STD_LOGIC ;
                      y    : OUT STD_LOGIC_VECTOR (0 TO 3) ) ;
       END COMPONENT ;
      SIGNAL m : STD_LOGIC_VECTOR (0 TO 3) ;
BEGIN
      G1:  FOR i IN 0 TO 3 GENERATE
```

```
        Dec_ri: dec2to4 PORT MAP ( w(1 DOWNTO 0), m(i), y(4*i TO 4*i⊕3) );
        G2: IF i=3 GENERATE
            Dec_left: dec2to4 PORT MAP ( w(i DOWNTO i-1), En, m ) ;
        END GENERATE ;
    END GENERATE ;
END Structure ;
```

Fig. 36 Código jerárquico para un decodificador binario 4 a 16

Podemos observar la definición de la señal *m* , la cual representa las salidas del decodificador de 2 a 4 visto en el esquema de dicho decodificador. Las 5 copias del componente *dec2to4* son reproducidas por la sentencia FOR GENERATE. En cada lazo de iteración, la sentencia etiquetada *Dec_ri* reproduce un componente *dec2to4* , que corresponde a uno de los decodificadores de 2 a 4 ubicados a la derecha del esquema circuital mostrado. El primer lazo de iteración genera el componente *dec2to4* con los datos de entrada *w1 y w0*, habilita la entrada *m0* y las salidas *y0, y1, y2, y3*. Los otros lazos de iteración tambien usan los datos de entrada *w1 w0*, pero diferentes bits de *m* e *y*.
La sentencia IF GENERATE, etiquetada G2, reproduce un componente *dec2to4* en el ultimo lazo de iteración, para la cual la condicion *i = 3* es verdadera. Este componente representa el decodificador 2 a 4 ubicado a la izquierda del esquema circuital mostrado. El posee dos entradas *w3 y w2*, la habilitación *En* y las salidas *m0, m1, m2, m3*
Note que en lugar de usar la sentencia IF GENERATE, podriamos reproducir estos componentes fuera de la sentencia FOR GENERATE. Hemos escrito este codigo simplemente para dar un ejemplo de la sentencia IF GENERATE.
La sentencia GENERATE de los codigos de la Fig. 35 y la Fig. 36 se usan para la reproducción de componentes. Otro ejemplos de uso se veran mas adelante.

Sentencias Concurrente y Secuenciales

Hasta aquí las sentencias vistas son del tipo concurrentes y tienen la particularidad de que el orden en las que ellas aparecen no afectan el significado del código. Por el contrario otra categoría de sentencias, llamadas secuenciales, el orden en el que aparecen si puede afectar el significado. Ellas son *if-then-else* y VHDL requiere que ellas sean colocadas dentro de otra sentencia llamada *process*

Sentencia Process

La Fig. 25 y la Fig. 30 muestran dos manera de describir un multiplexor de 2 a 1 usando las asignación de señales *select y conditional*. El mismo circuito puede describirse usando una sentencia *if-then-else*, pero ellas deben ser ubicadas dentro de la sentencia *process* . En la Fig. 37 se muestra el codigo correspondiente.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
        PORT ( w0, w1, s    : IN    STD_LOGIC ;
                    f       : OUT   STD_LOGIC ;
END mux2to1 ;
```

```
ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS (w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Fig. 37 Mux 2 a 1 usando sentencias *if-then-else*

La sentencia *process*, comienza con la palabra clave PROCESS, seguida por una lista de señales entre paréntesis, llamada *sensitivity list*. El process consiste, en este ejemplo, de una unica sentencia *if-then-else* que describe la función del multiplexor. Asi esta lista sensitiva contiene los datos de entrada *w0 y w1* y la entrada select *s*
En general puede haber varias sentencias dentro de process. Estas sentencias son consideradas como sigue. Cuando hay un cambio en cualquier señal de la lista del process, este se vuelve activo. Una vez activas las sentencias dentro del process son evaluadas en orden secuencial. Cualquier asignación hecha a señales dentro del process no son visibles fuera del process, hasta que todas las sentencias en el process han sido evaluadas. Si hay múltiple asignación a una misma señal, solo la ultima tiene algun efecto visible. Esto esta ilustrado en el diagrama del ejemplo 21.

Ejemplo 21 (Mux 2 a 1)

El código de la Fig. 38 es equivalente al de la Fig. 37. La primera sentencia en process asigna el valor *w0 a f*. Esto provee un valor por defecto de *f* pero la asignación no toma lugar hasta el fin de process. En la jerga VHDL se dice que la asignación es programada para ocurrir después de que todas las sentencias en process han sido evaluadas. Si alguna otra asignación para *f* ocurre, mientras el process esta activo, la asignación por defecto sera forzada. La segunda sentencia en process asigna el valor de *w1 a f* si el valor de *s= 1*. Si esta condicion es cierta, luego la asignación por defecto es forzada. Asi si *s = 0*, luego
f = w0, y si *s = 1*, luego *f = w1*, lo cual define a un multiplexor de 2 a 1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
        PORT ( w0, w1, s    : IN    STD_LOGIC ;
                  f         : IN    STD_LOGIC ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
```

```
BEGIN
    PROCESS (w0, w1, s )
    BEGIN
        f <= w0 ;
        IF s = '1' THEN
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Fig. 38 Código alternativo para un Mux 2 a 1 usando las sentencias *if-then-else*

Consideremos invertir la dos sentencia tal que la sentencia *if-then-else* sea evaluada primero. Si $s = 1$, a f se le asigna el valor de *w1*. La asignación esta programada y no ocurre hasta el fin del process. Sin embargo, la sentencia $f <= w0$ se evalua al ultimo.
Esto fuerza la primera asignación y *f* es asignada al valor *w0* sin importar el valor de *s* . Por lo tanto en vez de describir un multiplexor, este codigo representa un circuito trivial en donde $f = w0$

Ejemplo 22 (Codificador de prioridad)

El codigo de la Fig. 39 describe un codificador de prioridad y es equivalente al dado en la Fig. 32

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
    PORT ( w    : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           y    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           z    : OUT STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    PROCESS ( w )
    BEGIN
        IF w(3) = `1` THEN
            y <= "11" ;
        ELSIF w(2) = `1` THEN
            y <= "10" ;
        ELSIF w(1) = `1` THEN
            y <= "01" ;
        ELSE
            y <= "00" ;
        END IF ;
    END PROCESS ;
    z <= `0` WHEN w = "0000" ELSE `1` ;
END Behavior ;
```

Fig. 39 Codificador de prioridad usando las sentencias *if-then-else*

Ejemplo 23 (Codificador de prioridad)

Mostramos una alternativa de código para un codificador de prioridad usando las sentencias *if-then-else* . La primera sentencia en *process* provee un valor por defecto de 00 para y_1y_0 . La segunda sentencia hace que si $w_1 = 1$, $y_1 y_0 = 01$ De manera similar la tercera y cuarta sentencia hacen que si w_2 o w_3 son 1 , $y_1 y_0$ toma el valor de 10 y 11 respectivamente. Estas cuatro sentencias equivalen a una única sentencia *if-then-else* en el código dado en la Fig. 40 y describen un esquema de prioridad. El valor de *z* se especifica usando una sentencia de default seguida por un *if-then-else* que fuerza el default si $w = 0000$

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY priority IS
      PORT ( w    : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                y    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                z    : OUT STD_LOGIC ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
      PROCESS ( w )
      BEGIN
            y <= “00” ;
            IF w(1) = `1` THEN y <= “01” ; END IF ;
            IF w(2) = `1` THEN y <= “10” ; END IF ;
            IF w(3) = `1` THEN y <= “11” ; END IF ;

            z <= `1` ;
            IF w = “0000” THEN z <= `0 ; END IF ;
       END PROCESS ;
END Behavior ;
```

Fig. 40 Código alternative para un codificador de prioridad

Ejemplo 24 (Comparador de 4 bits)

En el código de la Fig. 33 especificamos un comparador de 4 bits que produce 3 salidas AeqB, A>B, y A<B. En la Fig 41 mostramos como rescribir tales especificaciones usando las sentencias *if-then-else*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY compare1 IS
         PORT ( A, B    : IN   STD_LOGIC ;
                   AeqB  : OUT STD_LOGIC ) ;
END compare1 ;
```

```
ARCHITECTURE Behavior OF compare1 IS
BEGIN
      PROCESS (A, B)
      BEGIN
            AeqB <=   '0' ;
            IF A = B THEN
                  AeqB <=   '1' ;
            END IF ,
      END PROCESS ;
END Behavior ;
```

Fig. 41 Codigo para un comparador de igualdad de 1 bit

Para simplicidad utilizamos numeros de un bit para las entradas A y B, y se muestra unicamente el código para AeqB. El *process* asigna el valor por default de 0 a AeqB y luego la sentencia *if-then-else* cambia AeqB a 1 si A es igual a B
Para el código de la Fig. 42 , una vez que A y B son iguales, resulta que AeqB sera puesto a 1 indefinidamente, aún si A y B no son mas iguales. En VHDL la salida AeqB tiene una memoria implicita debido a que el circuito sintetizado del código recordara o almacenara el valor de AeqB = 1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
        PORT ( A, B    : IN   STD_LOGIC ;
                  AeqB  : OUT STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
      PROCESS (A, B)
      BEGIN
            IF A = B THEN
                  AeqB <=  '1' ;
            END IF ,
      END PROCESS ;
END Behavior ;
```

Fig. 42   Codigo de una memoria implicita

En la Fig. 43 se muestra un circuito sintetizado de ese código. La XOR produce un 1 , cuando A es igual a B y la compuerta OR asegura que AeqB permanezca en 1 indefinidamente.

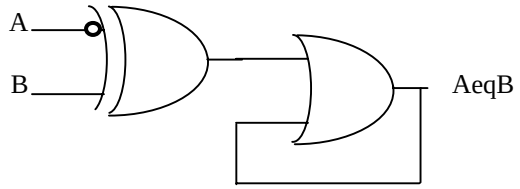


Fig. 43 Esquema memoria

La memoria implicita no es útil, debido a que genera un circuito comparador que no funciona correctamente. Sin embardo demostraremos mas adelante que ella es útil en otro tipo de circuitos.

Sentencia CASE

Una sentencia *case* es similar a la asignación de una señal *select* en que la sentencia case tiene una senal de selección e incluye la cláusula WHEN para varias evaluaciones de esa senal de selección. La Fig. 44 muestra como la sentencia case puede usarse como otra manera de descripción de un circuito multiplexor de 2 a 1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
        PORT ( w0, w1, s    : IN    STD_LOGIC ;
                  f         : IN    STD_LOGIC ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
     PROCESS (w0, w1, s )
     BEGIN
          CASE s IS
               WHEN `0`=>
                    f <= w0 ;
               WHEN OTHERS =>
                    f <= w1 ;
          END CASE ;
     END PROCESS ;
END Behavior ;
```

Fig. 44 Sentencia *case* que representa a un Mux 2 a 1

Esta sentencia comienza con la palabra clave CASE, la cual especifica que *s* va a ser usada como una senal de selección. La primer cláusula WHEN seguida por el símbolo $\Rightarrow$, nos indica que la sentencia deberia ser evaluada cuando $s = 0$. En este ejemplo la única sentencia evaluada cuando $s = 0$ es *f*

<= w_0 . La sentencia case deberá incluir la cláusula WHEN para todas las posibles evaluaciones de la senal de selección . Para la segunda clausula WHEN;, la cual contiene f <= w_1 , se utiliza la palabra clave OTHERS.

Ejemplo 25 (Decodificador 2 a 4)

En la Fig. 29 se vió el código para un decodificador de 2 a 4. Una forma diferente de describir este circuito, es utilizando las sentencias secuenciales , tal como se muestra en la Fig. 45

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY dec2to4 IS
        PORT ( w     : IN    STD_LOGIC_VECTOR(1DOWNTO 0) ; ;
                En   : IN    STD_LOGIC ;
                y    : OUT  STD_LOGIC_VECTOR(0 TO 3) ) ;
END dec2to4 ;

ARCHITECTURE Behavior OF dec2to4 IS
BEGIN
     PROCESS ( w, En )
     BEGIN
          IF En = `1` THEN
               CASE w IS
                    WHEN “00” =>
                         y  <= “1000” ;
                    WHEN “01” =>
                         y  <= “0100” ;
                    WHEN “10” =>
                         y  <= “0010” ;
                    WHEN OTHERS =>
                         y  <= “0001” ;
               END CASE ;
          ELSE
               y <= “0000” ;
          END IF ;
      END PROCESS ;
END Behavior ;
```

Fig. 45 Sentencia *process* que describe un decodificador 2 a 4

El *process* primero usa una sentencia *if-then-else* para chequear el valor de la senal de habilitación *En* del decodificador. Si *En = 1* la sentencia *case* pone la salida *y* con el valor correspondiente a la entrada *w* . La sentencia *case* representa las primeras 4 filas de la Tabla de Verdad del decodificador, ya visto . Si *En = 0* la cláusula ELSE pone a *y = 0000*, como esta especificado en la última fila de la Tabla de Verdad

Ejemplo 26 (Decodificador BCD a 7 segmentos)

Otro ejemplo de la sentencia *case* se da en el código de la Fig. 46. La entidad se denomina *seg7* y representa un decodificador BCD a 7 segmentos.
Las entradas BCD estan representadas por una senal de 4 bits denominadas *bcd*, y las salidas es una senal de 7 bits denominada *leds*
Note que hay un comentario a la derecha de la sentencia *case*, el cual etiqueta las 7 salidas con las letras a, b, c, ...g. Estas letras indican al lector la correlacion entre la senal *leds* de 7 bits y los 7 segmentos del decodificador. Al final la clausula WHEN pone los siete bits de *leds* a - (Recuerde que - significa en VHDL la condicion no importa)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;

ENTITY seg7 IS
      PORT ( bcd    : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                leds   : OUT STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;

ARCHITECTURE Behavior OF seg7 IS
BEGIN
      PROCESS ( bcd )
      BEGIN
            CASE bcd IS                          --     abcdefg
                  WHEN “0000”       => leds <= “1111110” ;
                  WHEN “0001”       => leds <= “0110000” ;
                  WHEN “0010”       => leds <= “1101101” ;
                  WHEN “0011”       => leds <= “1111001” ;
                  WHEN “0100”       => leds <= “0110011” ;
                  WHEN “0101”       => leds <= “1011011” ;
                  WHEN “0110”       => leds <= “1011110” ;
                  WHEN “0111”       => leds <= “1110000” ;
                  WHEN “1000”       => leds <= “1111111” ;
                  WHEN “1001”       => leds <= “1111011” ;
                  WHEN OTHERS     => leds <= “- - - - - - ” ;
            END CASE ;
      END PROCESS ;
END Behavior ;
```

Fig. 46 Codigo que representa un decodificador BCD a 7 segmentos

Ejemplo 27 (ALU)

En la tabla que sigue se especifica el funcionamiento de la ALU 74381. Tiene 2 entrada de datos de 4 bits c/u denominadas A y B, una entrada de selección *s* de 3 bits y una salida *F* de 4 bits. En la tabla el signo + indica adición aritmetica y el signo - significa sustracción aritmética.

OPERACIÓN	ENTRADAS S_2 S_1 S_0	SALIDAS F
Clear	0 0 0	0 0 0 0
B - A	0 0 1	B - A
A - B	0 1 0	A - B
ADD	0 1 1	A + B
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

En la Fig. 47 se describe el código VHDL para la ALU

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
      PORT ( s      : IN     STD_LOGIC_VECTOR(2 DOWNTO 0) ;
                A, B : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                 F     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)) ;
END alu ;

ARCHITECTURE Behavior OF alu IS
BEGIN
      PROCESS ( s, A, B )
      BEGIN
            CASE s IS
                  WHEN “000” =>
                        F <= “0000” ;
                  WHEN “001” =>
                        F <= B – A ;
                  WHEN “010” =>
                        F <= A – B ;
                  WHEN “011” =>
                        F <= A + B ;
                   WHEN “100” =>
                        F <= A XOR B ;
                   WHEN “101” =>
                        F <= A OR B ;
                   WHEN “110” =>
                        F <= A AND B ;
                   WHEN OTHERS =>
                        F <= “1111” ;
            END CASE ;
      END PROCESS ;
END Behavior ;
```

Fig. 47 Codigo que representa la funcionalidad de la ALU 74381

Bibliografía

Titulo: Fundamentals of Digital Logic with VHDL Design
Autores: Stephen Brown, Zvonko Vranesic
Ed. Mc Graw Hill

Titulo: Diseno de Sistema Digitales con VHDL
Autores: Serafin Perez, Enrique Soto, Santiago Fernandez
Ed. THOMSON

Titulo: Introduccion al lenguaje VHDL
Autor: Fernando Nuno Garcia
http://www2.ate.uniovi.es