III

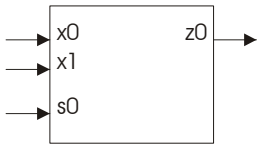
SINTAXIS DEL LENGUAJE

3.1 INTRODUCCIÓN A LA DESCRIPCIÓN EN VHDL DE CIRCUITOS DIGITALES

En este capítulo se discutirán los elementos fundamentales de VHDL que son comúnmente utilizados en síntesis de circuitos. Primero se exponen el diseño de multiplexores y comparadores con VHDL para hacer una analogía con la metodología convencional de diseño. Ya que es muy importante comprender porque VHDL es un lenguaje para describir y no para programar. Posteriormente se expondrán los elementos básicos del lenguaje, tales como: identificadores, objetos de datos, tipos de datos, operadores y tipos de instrucciones.

3.1.1 MULTIPLEXORES

Antes de examinar la descripción en VHDL de multiplexores, analizaremos el funcionamiento interno y la metodología tradicional de diseño utilizada en este tipo de circuitos para después realizar la descripción del circuito en VHDL. En la figura 3.1 se describe externamente a un multiplexor y la tabla resume su funcionalidad.



ENTRADA	SALIDA
s0	z0
0	x0
1	x1

Figura 3.1 Multiplexor 2 a 1

La tabla de verdad completa sería la siguiente.

ENTRADAS			SALIDA
s0	x0	x1	z0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Utilizando cualquier técnica de simplificación de ecuaciones obtenemos: z0 = s0'·x0+s0·x1. Y el circuito quedaría como se muestra en la figura 3.2.

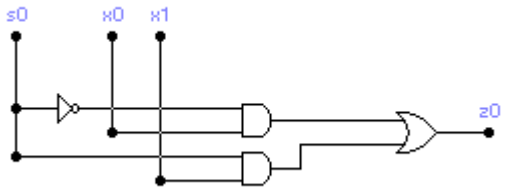


Figura 3.2 Circuito del Multiplexor 2 a 1

En VHDL podemos describir el circuito ya sea mediante la descripción completa de todas las combinaciones de las entradas, utilizando la tabla simplificada de funcionamiento o si lo deseamos es posible realizar la descripción compuerta por compuerta del circuito. Al primer estilo se le conoce como de **flujo de datos** y en este debemos describir como fluyen los datos de entrada hacia la salida. A continuación se muestra la descripción de flujo de datos del multiplexor con VHDL utilizando la tabla completa de funcionamiento del mismo. Las palabras en **negritas** son palabras reservadas en VHDL y los comentarios comienzan con dos guiones (--).

```
ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

ARCHITECTURE data_flow OF multiplexor IS
  SIGNAL temp: bit_vector (2 DOWNTO 0);
BEGIN
  z0 <= '0' WHEN temp = "000" ELSE
        '0' WHEN temp = "001" ELSE
        '1' WHEN temp = "010" ELSE
        '1' WHEN temp = "011" ELSE
        '0' WHEN temp = "100" ELSE
        '1' WHEN temp = "101" ELSE
        '0' WHEN temp = "110" ELSE
        '1';
temp <= s0 & x0 & x1; -- Concatenación
                      -- de las entradas en un
                      -- solo bus.
END data_flow;
```

Las descripciones en VHDL son creadas a partir de dos estructuras que son fundamentales

para el lenguaje: la entidad y la arquitectura. Básicamente la entidad es la estructura en la que se define cuales son las entradas y salidas del circuito que deseamos representar, la cual podemos asociar con una "caja" en la que se que precisan cuales son las interfaces de comunicación con el exterior, siendo la arquitectura donde se detalla el comportamiento interno de esa "caja". Observe que en la descripción anterior se utilizó un objeto de datos llamado SIGNAL para crear el bus "temp" y concatenar "s0", "x0" y "x1" en un solo objeto y así facilitar la descripción. A continuación utilizaremos la tabla simplificada para diseñar el multiplexor, ya que si observamos la salida depende fundamentalmente de la entrada de selección "s0", por lo que atendiendo a esta característica la descripción de flujo de datos quedaría tal y como se muestra a continuación.

```
ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

ARCHITECTURE data_flow OF multiplexor IS
BEGIN
  z0 <= x0 WHEN s0 = '0' ELSE x1;
END data_flow;
```

En ambas descripciones hemos definido el funcionamiento de la salida apoyándonos en las entradas. Note que existe cierto "paralelismo" de las entradas hacia las salidas, es decir, no importa cual combinación de entradas se dé, sólo una opción será asignada a la salida. Por lo que no importa cual orden se haya seguido en la descripción del circuito. Y esta es una de las principales características de VHDL, es decir, no importa tanto el orden de las instrucciones, lo cual no es así en lenguaje de programación de software. Tal vez este "paralelismo" se perciba con mayor detalle realizando la descripción del circuito compuerta por compuerta utilizando la ecuación simplificada del circuito obtenida anteriormente.

```
ENTITY multiplexor IS
  PORT (s0, x0, x1: IN bit;
        z0: OUT bit);
END multiplexor;

ARCHITECTURE data_flow OF multiplexor IS
  SIGNAL  not_s0, and1, and2: bit;
BEGIN
  z0 <= and1 OR and2;
  and1 <= not_s0 AND x0;
  not_s0 <= NOT s0;
  and2 <= s0 AND x1;
END data_flow;
```

La descripción anterior también es de flujo de datos y no sigue ningún orden en particular en el uso de las instrucciones y esto se ha hecho para destacar el comportamiento paralelo o **concurrente** de VHDL. Observe que cada ecuación describe a cada una de las compuertas que se muestran en la figura 3.2, por lo cual no hubiera importado el orden que hayamos seguido siempre y cuando se realicen las **conexiones** correctamente.

Pensemos en el circuito implementado físicamente en el laboratorio. Cada una de estas compuertas posee características eléctricas que les asignan un funcionamiento definido. Estas particularidades eléctricas cumplen con leyes físicas que se están cumpliendo todo momento y, entonces, no importa como se hayan ordenado los circuitos entre sí lo importante es realizar correctamente las conexiones para obtener la función deseada. En VHDL se trata de emular ese comportamiento, por lo tanto el orden que se haya seguido en cada una de las instrucciones anteriores no es importante, ya que hicimos correctamente la **interconexión** entre las compuertas.

3.1.2 COMPARADORES

La figura 3.3 representa a un comparador y la siguiente tabla resume su funcionalidad.

ENTRADAS	SALIDAS		
	x_may_y	equals	x_men_y
x > y	1	0	0
x = y	0	1	0
x < y	0	0	1

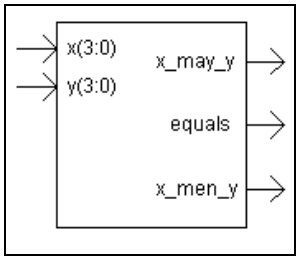


Figura 3.3 Comparador

En el diseño convencional de un comparador tendríamos que realizar una tabla de todos los posibles valores lógicos de las salidas respecto a las entradas. Lo cual en este caso sería de 256 combinaciones, dado que tenemos ocho bits de entrada para obtener las tres ecuaciones de salida. En VHDL basta con describir de una forma general el funcionamiento del hardware y el sintetizador se encargará de generar toda esta tabla de 256 combinaciones y obtener las ecuaciones lógicas de las tres salidas. El código mostrado a continuación corresponde al comparador mostrado en la figura 3.3.

```
ENTITY comparador IS
  PORT ( x: IN bit_vector (3 DOWNTO 0);
         y: IN bit_vector (3 DOWNTO 0);
         equals: OUT bit;
         x_may_y: OUT bit;
         x_men_y: OUT bit);
END comparador;

ARCHITECTURE data_flow OF comparador IS
BEGIN
  equals <= '1' WHEN x = y ELSE '0';
  x_may_y <= '1' WHEN x > y ELSE '0';
  x_men_y <= '1' WHEN x < y ELSE '0';
END data_flow;
```

Nótese que en la primera declaración de puertos, dentro de la entidad del comparador, se definen dos bus de entrada de cuatro bits de magnitud (x, y), en cambio las salidas son de solamente un bit. En la entidad lo único que hacemos es describir como es el circuito, o aquello del circuito que permite a la entidad comunicarse hacia otras entidades, sin mencionar para nada su comportamiento interno. Y es en la arquitectura, después del BEGIN, es donde se realiza la descripción del comportamiento del circuito atendiendo únicamente a la funcionalidad del mismo, es decir, no es necesario analizar cada caso en particular de las 256 posibles combinaciones.

3.1.3 EL ESTILO DE "PROGRAMACIÓN" EN VHDL

Haciendo una comparación con un lenguaje de programación de alto nivel podemos ver que el código es similar en cuanto a las sentencias utilizadas, sin embargo, no es así en el flujo de ejecución de las instrucciones. Un código de programación en VHDL no es precisamente un "programa", ya que un programa es un conjunto de instrucciones que se ejecutan **paso a paso** para llevar a cabo una tarea determinada, y en este caso no podemos decir que las instrucciones se estén ejecutando de esta manera, porque esto no corresponde en la realidad al comportamiento de un circuito. En VHDL las instrucciones se están ejecutando en todo momento lo cual sí se asemeja al comportamiento real de un circuito. Así cuando cambie alguna señal de entrada cambiará inmediatamente la salida y, por consiguiente, estamos **describiendo** cual es el verdadero funcionamiento del circuito. La forma en que se "programa" en VHDL al principio resultará un tanto extraña, pero si asociamos éste código con el circuito que estamos describiendo, podemos darnos cuenta que en él los componentes siempre están activos, y es esto es precisamente lo que describimos mediante VHDL. Tal vez alguna vez ha utilizado **PSPICE** o algún programa de entrada esquemática de diseño para modelar y simular circuitos, estos también son para la descripción de circuitos. En PSPICE la descripción se realiza mediante un listado de conexiones *(netlist)* entre los componentes, en tanto que en los otros lo hacemos de manera gráfica y en ambos se considera que todos los componentes siempre están funcionando para que la simulación o modelado del diseño sea de acuerdo a la realidad. Por esto en VHDL el orden de las instrucciones no es tan importante como en el caso de un lenguaje de programación de software, porque las instrucciones se están ejecutando al mismo tiempo y así sí se modela adecuadamente un circuito. Posteriormente se explicarán los tipos de instrucciones y sus diferencias, ya que VHDL si permite la "descripción secuencial" utilizando instrucciones de esta naturaleza dentro de una estructura llamada PROCESS. En esta estructura las instrucciones se ejecutan "paso a paso" como en los lenguajes de programación de software. Pero de cualquier manera esta estructura siempre esta activa, como si fuera un componente o subcircuito del diseño, por lo que todo lo que se obtenga dentro del proceso se ejecutará paralelamente con el resto de las instrucciones que están fuera de esta estructura.

Otro punto importante es el dispositivo lógico programable que estemos utilizando, ya que si éste no tiene la capacidad para realizar lo que "dice" nuestra descripción nunca podremos

sintetizar el código. Por ejemplo, si en el dispositivo que usemos no es posible que las salidas puedan ser programadas para que trabajen con alta impedancia, aún y cuando la descripción sea correcta nunca podremos sintetizarla en el dispositivo.

Al principio generalmente se comete el error de tratar de "programar" como si fuera C++, Pascal, Visual Basic o cualquier otro lenguaje de programación de software, además de olvidar que el PLD, CPLD, o FPGA que utilicemos tiene características propias que deben ser consideradas cuando se hace la descripción. Pero basta con recordar que estamos diseñando hardware y que por lo tanto no se trata de un lenguaje secuencial de programación para software.

3.2 IDENTIFICADORES

Un identificador se define como un conjunto de caracteres con el cual podemos representar diferentes elementos dentro de una descripción. En VHDL un identificador está compuesto por una secuencia de uno o más caracteres alfabéticos, numéricos, o del carácter de subrayado. Las condiciones que debe de seguir un identificador son las siguientes:

- VHDL permite la utilización de las letras mayúsculas (A.. Z), minúsculas (a.... z), dígitos (0...9), y el carácter de subrayado (_).
- El primer carácter de un identificador debe ser una letra.
- El ultimo carácter de identificador no puede ser el carácter de subrayado. Además, el carácter de subrayado no puede aparecer dos o más veces consecutivas.
- Mayúsculas y minúsculas son consideradas idénticas. Así, Signal_A, signal_a, y SIGNAL_A se refieren al mismo identificador.
- Los comentarios en VHDL comienzan con dos guiones consecutivos (--), y se extienden hasta el final de la línea. Los comentarios pueden aparecer en cualquier lugar dentro de una descripción en VHDL.
- VHDL define un grupo de palabras reservadas, llamadas "palabras clave" (keywords), las cuales no pueden ser usadas como identificadores.

EJEMPLOS

```
-- Este es un comentario.

ENTITY contador IS -- comentario al final
                   -- de una línea
```

Los siguientes ejemplos son de identificadores válidos en VHDL.

```
Mi_entidad

Mux4a2

TTL_7490
```

A continuación se muestran ejemplos de identificadores no válidos en VHDL.

```
3er_Modulo    -- un identificador no
              -- puede iniciar con un
              -- dígito

_salida_x   -- o con el carácter de
              -- subrayado

M__24xmax   -- no se permiten dos
              -- caracteres de
              -- subrayado seguidos

My_design_ -- un identificador no
              -- debe terminar con un
              -- carácter de subrayado

Unidad&     -- el caracter "&", no
              -- es un carácter válido

SIGNAL   -- palabra reservada
```

3.3 OBJETOS DE DATOS

Un objeto de datos en VHDL es un elemento que toma un valor de algún tipo de dato determinado. Según sea este tipo de dato, el objeto poseerá un conjunto de propiedades que se le podrán aplicar, como las operaciones en las que el objeto puede ser usado. En VHDL los objetos de datos son generalmente de una de tres clases: constantes, variables o señales.

3.3.1 CONSTANTES

Una constante es un elemento que puede tomar un único valor de un tipo dado. A las constantes se les debe asignar un valor en el momento de la declaración. Una vez que se le ha asignado algún valor, éste no puede ser cambiado dentro de la descripción del diseño. Las constantes pueden ser

declaradas dentro de las entidades, arquitecturas, procesos o paquetes. Las constantes que se declaren en un paquete pueden ser utilizadas en cualquier descripción en la que se este utilizando dicho paquete. Por otra parte las constantes declaradas dentro de una entidad pueden ser utilizadas por la o las arquitecturas en las que se este haciendo la descripción de dicha entidad, y aquellas constantes que sean declaradas dentro de una arquitectura o proceso, son válidas únicamente dentro de la estructura correspondiente.

DECLARACIÓN DE CONSTANTES

```
CONSTANT identificador: tipo := valor;
```

EJEMPLO

```
CONSTANT byte: integer := 8;
```

3.3.2 VARIABLES

Los objetos de datos de la clase variable son similares a las constantes, con la diferencia que su valor puede ser modificado cuando sea necesario. Las variables en VHDL son similares a cualquier tipo de variable de un lenguaje de programación de alto nivel. A las variables también se les puede asignar un valor inicial al momento de ser declaradas. Se utilizan únicamente en los procesos y subprogramas (funciones y procedimientos). Las variables generalmente se utilizan como índices, principalmente en instrucciones de bucle, o para tomar valores que permitan modelar componentes. Las variables no representan conexiones o estados de memoria.

DECLARACIÓN DE VARIABLES

```
VARIABLE identificador: tipo [:=valor];
```

EJEMPLO

```
VARIABLE aux1, aux2: bit;
```

3.3.3 SEÑALES

Un objeto de la clase señal es similar a un objeto de la clase variable, con una importante diferencia: las señales si pueden almacenar o pasar valores lógicos, mientras que una variable no lo puede hacer. Las señales, por lo tanto, representan elementos de memoria o conexiones y si pueden ser sintetizadas.

Los puertos de una entidad son implícitamente declarados como señales en el momento de la declaración, ya que estos representan conexiones. También pueden ser declaradas en las arquitecturas antes del BEGIN, lo cual nos permite realizar conexiones entre diferentes estructuras de programación. La asignación de valores a un objeto de datos del tipo señal no es inmediata como en el caso de las variables, esto se explicará más detalladamente cuando se exponga la estructura PROCESS y los tipos de instrucciones.

DECLARACIÓN DE SEÑALES

```
SIGNAL identificador: tipo [:=valor];
```

EJEMPLOS

```
SIGNAL A, B: bit := '0';   -- el valor
                           -- inicial es
                           -- opcional

SIGNAL dato: bit_vector (7 DOWNTO 0);
```

3.3.4 ALIAS

Un ALIAS no es precisamente un objeto de datos. La instrucción ALIAS permite que utilicemos un identificador diferente para hacer referencia a un objeto de datos, o a parte de él, ya existente. Este no es un objeto de datos nuevo, sino que nos permite manipular fragmentos del objeto de datos original para facilitar la programación. Al modificar el ALIAS se modifica el objeto de datos al que señala.

DECLARACIÓN DE ALIAS

```
ALIAS identif: tipo IS identif2 <rango>;
```

EJEMPLO

```
ALIAS instr: bit_vector (3 DOWNTO 0) IS
                  dato (7 DOWNTO 4);
```

3.4 TIPOS DE DATOS

Un tipo de dato especifica el grupo de valores que un objeto de datos puede tomar así como las operaciones que son permitidas con esos valores. En VHDL es sumamente importante el tipo de

dato, los objetos de datos no pueden tomar o no se les puede asignar un objeto de datos de otro tipo, y no todas las operaciones se pueden utilizar con los diferentes tipos de datos a menos que se utilicen las librerías adecuadas en las que estén definidas funciones para la conversión de tipos. Además, es posible que el usuario defina subtipos y tipos compuestos, modificando los tipos básicos, así como definir tipos particulares con combinaciones de los diferentes tipos. A continuación se discutirán las dos categorías de tipos de datos más utilizadas en síntesis: escalares y compuestos.

3.4.1 TIPOS ESCALARES

Los tipos escalares tienen un orden especifico lo cual permite que sean usados con diferentes operadores. Existen cuatro clases de tipos escalares: enteros, reales o de punto flotante, enumerados, y físicos.

- **ENTERO**

VHDL permite especificar la gama del entero (integer) de manera diferente. Sin embargo, la gama debe extender desde por lo menos -(2^31-1) a +(2^31-1), o - 2147483648 a +2147483647. Una señal o variable declarada como tipo entero y que tenga que ser sintetizada en elementos lógicos, debe ser limitada con un rango.

EJEMPLO

```
VARIABLE n: integer RANGE -15 TO 15;
```

- **REAL**

El rango de valores que puede tomar este tipo de dato se encuentra entre -1.038E38 a +1.038E38. El Real rara vez es usado en síntesis y en la gran mayoría de las herramientas de software de VHDL para síntesis no es posible utilizar este tipo de dato.

- **ENUMERADOS**

Un tipo enumerado es un tipo de dato con un grupo de posibles valores asignados por el usuario. Los tipos enumerados se utilizan principalmente en el diseño de maquinas de estado.

DECLARACIÓN DE UN TIPO ENUMERADO

```
TYPE nombre IS ( valor [,valor...] );
```

El orden en el que los valores son listados en la declaración del tipo enumerado define el orden léxico para ese tipo.

EJEMPLOS

a) En este ejemplo se define un tipo enumerado llamado “arith”, y los posibles valores son add, sub, mul, y div.

```
TYPE arith IS (add, sub, mul, div);
```

b) Ahora se define un tipo enumerado llamado “estados”, con 4 posibles valores: estado0, estado1, estado2 y estado3.

```
TYPE estados IS ( estado0, estado1,
                  estado2, estado3 );
```

Existen varios tipos de datos enumerados, algunos predefinidos en los programas de síntesis, para el lenguaje pero generalmente los siguientes tipos enumerados son los más comúnmente utilizados para síntesis de circuitos.

BOOLEAN

El tipo BOLEAN es un tipo enumerado con dos valores, FALSE y TRUE, donde FALSE < TRUE. Las funciones lógicas y de comparación retornan siempre un valor booleano.

Cuando en una operación lógica o de comparación se utilza un tipo de dato no booleano, el bit por ejemplo, existen funciones de conversión que permiten realizar dichas operaciones con distintos tipos de datos. En el caso del bit se utiliza la siguiente función.

```
boolean_var := (bit_var = '1');
```

BIT

El bit es un tipo enumerado que representa valores binarios: '0' y '1'. Las operaciones lógicas en las participa este tipo de dato regresan valores binarios mediante las siguiente función.

```
IF (boolean_var) THEN
  bit_var := '1';
ELSE
  bit_var := '0';
END IF;
```

CHARACTER

El tipo character es un tipo enumerado que contiene el conjunto de los símbolos contenido en el ASCII.

STD_LOGIC

El tipo std_logic es similar al tipo bit pero con la excepción que éste no esta definido dentro del lenguaje. El paquete std_logic_1164 de la IEEE define al std_logic como un tipo de dato el cual puede tomar los valores 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'. Para poder utilizar este tipo de dato es necesario incluir el paquete dentro de la descripción utilizando las siguientes dos líneas antes de la declaración de la entidad.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

El significado de los valores del std_logic se muestra en la siguiente tabla.

'U' -- Uninitialized	'W' -- Weak Unknow
'X' -- Forcing	'L' -- Weak 0
'0' -- Forcing 0	'H' -- Weak 1
'1' -- Forcing 1	'-' -- Don't care
'Z' -- High Impedance	

Los valores '0', '1', 'L' y 'H', se utilizan en síntesis de circuitos, los valores 'Z' y '-' tienen restricciones sobre como y donde pueden ser usados. Los valores 'U', 'W' y 'X' se utilizan únicamente para simulación y evaluación de diseños mas no para síntesis. Para la mayoría de los diseños se utiliza este tipo de dato ya que es más completo que el tipo bit por proporcionar los valores 'Z' y '-'.

SOBRECARGA DE UN TIPO ENUMERADO

Es posible cargar el valor de algún tipo enumerado incluyendo dicho valor en la definición de dos o más tipos enumerados. Cuando se utilice en una descripción dos o más tipos que usen el mismo valor generalmente el programa de síntesis identifica de cual tipo proviene, pero bajo ciertas condiciones esta determinación es imposible. En estos casos es necesario indicar explícitamente de cual tipo se trata. En el siguiente ejemplo se muestra como realizar esta indicación explicita.

```
TYPE color is (red, green, yellow, blue,
                violet);
TYPE primary_color IS (red, yellow, blue);

. . .

SIGNAL A: color;
SIGNAL B: primary_color;
SIGNAL C: bit;
SINAL D: std_logic;

. . .

A <= color'(red); -- Se indica a que
                  -- tipo estamos
                  -- haciendo
                  -- referencia

B <= blue; -- Posiblemente el
           -- programa de síntesis
           -- marque error en este
           -- caso

C <= '1';  -- En estas dos
           -- asignaciones el
D <= '1';  -- programa de síntesis
           -- identifica de que
           -- tipo se trata
```

CODIFICACIÓN DE LOS TIPOS ENUMERADOS

Los tipos enumerados se ordenan de acuerdo a sus valores. Los programas de síntesis automáticamente codifican binariamente los valores del tipo enumerado para que estos puedan ser sintetizados. Algunos programas los hacen mediante una secuencia binaria ascendente, otros buscan cual es la codificación que mejor conviene para tratar de minimizar el circuito o para incrementar la velocidad del mismo una vez que la descripción es sintetizada. También es posible asignar el tipo de codificación mediante directivas de síntesis.

El siguiente ejemplo muestra la forma en que el programa de síntesis Foundation de Xilinx, Inc. codifica un tipo enumerado de cinco posibles valores.

```
TYPE color IS (red, green, yellow, blue,
                violet);
```

La codificación sería la siguiente: red="000", green="001", yellow="010", blue="011", violet="100".

Para realizar la codificación manualmente se deben utilizar directivas de síntesis que son propias de cada programa. Para ver ejemplo de cómo se utilizan consulte la ayuda del programa y

busque la sección de "directivas de síntesis" o "codificación de tipos enumerados".

- **FÍSICOS**

Los tipos físicos son usados para especificar unidades de medida, ya sea de tiempo o para determinar medidas eléctricas. El único tipo predefinido es el "time", mediante el cual se pueden establecer medidas para simular los retardos de tiempo o para generar diferentes señales que nos permitan simular nuestro diseño. La unidad básica del tipo time es el femtosegundo, y de éste se forman diferente múltiplos.

```
TYPE time IS RANGE -2147483647 TO
                    2147483647

UNITS
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
END UNITS;
```

Los tipos físicos no tienen ningún significado en síntesis, sólo son utilizados para simulación de circuitos.

3.4.2 TIPOS COMPUESTOS

Un tipo compuesto es un tipo de dato formado con elementos de otros tipos. Existen dos formas de tipos compuestos: ARRAYS y RECORDS.

DECLARACIÓN DEL TIPO ARRAY

```
TYPE nombre IS ARRAY ( rango ) OF tipo;
```

DECLARACION DEL TIPO RECORD

```
TYPE nombre IS RECORD
     elemento: tipo_de_dato;
     [;elemento: tipo_de_dato...];
END RECORD;
```

Un ARRAY es un objeto de datos que consiste en una "colección" de elementos del mismo tipo. Los arreglos pueden ser de una o más dimensiones. Los elementos individuales de un arreglo pueden ser utilizados especificando un valor dentro del arreglo. Elementos múltiples de un arreglo pueden ser utilizados especificando más valores.

Un RECORD es un objeto de datos que consiste en una "colección" de elementos de diferentes tipos. La estructura RECORD en VHDL es análoga a los RECORDS utilizados en Pascal o a las estructuras en C. Los campos individuales de un RECORD pueden ser utilizados usando los nombres de los elementos. También se puede utilizar más de un campo.

EJEMPLOS

Las siguientes líneas corresponden a la declaraciones de un tipo ARRAY.

```
TYPE word IS ARRAY (0 TO 15) OF std_logic;

TYPE matriz IS ARRAY (0 TO 13, 0 TO 18) OF
                     std_logic;

TYPE valores IS ARRAY (0 TO 127) OF
                     integer;
```

A continuación se muestra como declarar objetos de datos utilizando estos tipos.

```
SIGNAL mensaje1, mensaje2: word;

SIGNAL arreglo_matriz: matriz;

VARIABLE valor_actual: valores;
```

Algunas posibles maneras de asignar valores a elementos de estos objetos son:

```
mensaje1(0) <= '1'; -- asignación de valor
                    -- al elemento '0' de
                    -- mensaje1


mensaje(5) <= '0'; -- asignación de valor
                    -- al quinto elemento de
                    -- mensaje1


mensaje2 <= mensaje1; -- hace mensaje1
                    -- igual a mensaje2,
                    -- esto es permitido
                    -- ya que se trata de
                    -- dos objetos de datos
                    -- del mismo tipo

mensaje2 (63) <= arreglo_matriz (5, 13);
                    -- transfiere el valor
                    -- de un elemento de
                    -- arreglo_matriz a un
                    -- elemento de mensaje2,
                    -- noté que ambos son
                    -- arreglos del tipo
                    -- std_logic_vector
```

A continuación se muestra un ejemplo de una declaración del tipo RECORD.

```
TYPE operacion IS (add, sub, mul, div);

TYPE instruccion IS RECORD
  operador: operacion;
  op1: integer;
  op2: integer;
END RECORD;
```

Aquí está la declaración de dos objetos usando la declaración del tipo RECORD anterior.

```
VARIABLE inst1, inst2: instruccion;
```

A continuación se muestran algunas posibles maneras de asignar valores a elementos de estos objetos de datos.

```
inst1.operador := add; -- asigna un valor
                    -- a "operador"del
                    -- RECORD inst1

inst2.operador := sub; -- asigna un valor
                    -- a "operador" de
                    -- inst2

inst1.op1 := inst2.op2;
```

```
inst2 := inst1;
```

ARREGLOS RESTRINGIDOS

Un arreglo restringido (Constrained Array) es aquel que está explícitamente definido mediante rango entero especifico de un tipo de datos ya existente. Cuando se declara un objeto de datos con arreglo restringido, el objeto posee el mismo rango.

EJEMPLO:

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF bit;
  -- Este es un arreglo restringido cuyo
  -- rango es: (7, 6, 5, 4, 3, 2, 1, 0)
```

ARREGLOS INDEFINIDOS

Un arreglo indefinido o sin restricciones (Unconstrained Array) es aquel que no está delimitado mediante un rango entero especifico. Un objeto de datos declarado con arreglo indefinido deberá ser delimitado o de lo contrario no podrá ser sintetizado.

EJEMPLO

```
TYPE bit_vector IS ARRAY
             (integer RANGE <>) OF bit;

VARIABLE v: bit_vector(5 DOWNTO -5);
```

A continuación se exponen algunos tipos compuestos comúnmente utilizados en síntesis de circuitos utilizando VHDL.

BIT_VECTOR

Los valores asignados al tipo bit_vector deben ser especificados con comillas dobles (" ") y los valores asignados al tipo bit simple son asignados con comillas simples (' '). El prefijo 'X' o 'x' denota un valor hexadecimal; los prefijos 'O' y 'o' denotan un valor octal; el prefijo 'B' o 'b' denota un valor binario. Si ningún prefijo es especificado, se asume el prefijo binario. Las asignaciones en hexadecimal y octal deben usarse únicamente si el valor puede combinarse directamente con el tamaño del vector. Por ejemplo, sí 'a' es un bit_vector (0 TO 6), entonces la asignación a <= x"B", no podrá hacerse porque el numero hexadecimal 'B' usa cuatro de bits y no equipara el tamaño del vector al que está siendo asignado.

EJEMPLOS

```
x1 <= "0001";
y1 <= B"1011";
x2 <= x"A";
y2 <= x"7";
z_octal <= o"2";
un_bit <= '1';
```

STD_LOGIC_VECTOR

El tipo std_logic_vector al igual que el bit_vector es simplemente un arreglo de elementos del tipo std_logic. La forma de utilizarlo es similar a la del bit_vector.

SIGNED & UNSIGNED

El estándar 1076.3 de la IEEE es un paquete para VHDL en el cual se definen nuevos tipos de datos además de funciones aritméticas y lógicas para ser utilizadas por herramientas de síntesis. Éste define dos paquetes: el numeric_std y el numeric_bit en los que se define dos nuevos tipos de datos: signed y unsigned. Estos tipos son parecidos a los tipos std_logic_vector o bit_vector y son parte de una norma emergente (IEEE 1076.3) para desempeñar operaciones numéricas sobre señales vectorizadas. El paquete numeric_bit define a estos tipos (unsigned y signed) como un vector cuyos elementos son del tipo bit y el paquete numeric_std define los mismos pero con elementos del tipo std_logic.

El propósito de estos dos tipos es el de representar números enteros positivos y negativos en forma binaria. Para ambos tipos, el bit más significativo está a la izquierda. El tipo signed se utiliza para representar un número entero con signo en forma binaria con complemento a dos, y el unsigned es solamente un número entero sin signo en forma binaria. El paquete numeric_std define funciones y operadores aritméticos, relacionales, lógicos y de asignación para ser utilizados con estos tipos de datos. Signed, unsigned y std_logic_vector son tipos diferentes por lo que no se pueden mezclar. Sin embargo, varias funciones de conversión, tales como to_unsigned, son definidas para la conversión entre los tipos.

3.4.3 SUBTIPOS

Un subtipo es un "subgrupo" de un tipo predefinido. Los subtipos son útiles para crear tipos de datos con limitaciones sobre tipos mayores.

DECLARACIÓN DE SUBTIPOS

```
SUBTYPE identif IS tipobase RANGE <rango>;
```

EJEMPLOS

```
SUBTYPE byte IS bit_vector (7 DOWNTO 0);
SUBTYPE digito IS integer RANGE (0 TO 9);
```

Estos ejemplos definen dos subtipos llamados byte y digito. Las señales o variables que son declaradas como byte son del tipo std_logic_vector de 8 bits en orden descendente. Las señales o variables que sean declaradas como tipo digito serán del tipo entero, consistiendo de los posibles valores de los enteros del 0 al 9, inclusive.

En el siguiente ejemplo se muestra como se pueden crear subtipos de datos a partir de aquellos tipos que sean definidos por el usuario.

EJEMPLOS

```
TYPE arith IS (add, sub, mul, div);
SUBTYPE add IS arith RANGE add TO sub;
SUBTYPE mul IS arith RANGE mul TO div;
```

Los únicos subtipos predefinidos son el natural y el positive.

3.4.4 TIPOS PREDEFINIDOS EN VHDL

Dentro del estándar VHDL de la IEEE se describen dos paquetes en los que se especifica el conjunto de tipos de datos y operaciones en las que dichos tipos de datos pueden ser utilizados, estos paquetes son: STANDARD y TEXTIO.

El paquete STANDAR de tipos de datos esta incluido en todos archivos fuente de VHDL, es decir, que no es necesario declararlo dentro de código para poder utilizarlo. El paquete TEXTIO define tipos de datos y operaciones para estos tipos para la comunicación con el software de síntesis que se este utilizando. Este no es necesario para la síntesis de circuitos y algunos programas de síntesis no lo soportan. De hecho, el paquete STANDAR, que es utilizado por la mayoría de las herramientas de síntesis, no es utilizado por completo. A continuación se muestra una parte del paquete STANDAR donde se encuentran los tipos y subtipos predefinidos más utilizados.

```
PACKAGE standard IS
  TYPE boolean IS (FALSE, TRUE);
  TYPE bit IS ('0', '1');
  TYPE character IS (
NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', DEL);
  TYPE integer IS RANGE -2147483647 TO
                        2147483647;
  SUBTYPE natural IS integer RANGE 0 TO
                        2147483647;
  SUBTYPE positive IS integer RANGE 1 TO
                        2147483647;
  TYPE string IS ARRAY (positive RANGE <>)
                        OF character;
  TYPE bit_vector IS ARRAY (natural RANGE
                              <>) OF bit;
END standard;
```

3.4.5 TIPOS NO SOPORTADOS EN VHDL PARA SÍNTESIS

Algunos tipos no son muy utilizados aunque existen dentro del lenguaje, como el characer o el string, y hay otros que no son soportados por las herramientas de síntesis. Los tipos no soportados son ignorados y por lo tanto no pueden ser sintetizados. Cada herramienta de síntesis define cuales son los tipos, objetos o estructuras de lenguaje que son o no soportados por la misma, a continuación se muestran lo tipos y objetos no soportados por el FOUNDATION.

- Tipos físicos

- Tipos reales o flotantes.

- Objetos de datos ACCESS. Un ACCESS equivale a un apuntador y no es soportado por que no tiene ningún sentido en hardware.

- Objetos de datos FILE. Estos se utilizan para almacenar datos en RAM o ROM de la computadora.

3.5 OPERADORES

Un operador nos permite construir diferentes tipos de expresiones mediante los cuales podemos calcular datos utilizando los diferentes objetos de datos con el tipo de dato que maneja dicho objeto. En VHDL existen distintos operadores de asignación con lo que se transfieren valores de un objeto de datos a otro, y operadores de asociación que relacionan un objeto de datos con otro, lo cual no existe en ningún lenguaje de programación de alto nivel.

El uso de los operadores que aquí son expuestos dependerá del software utilizado, ya que no es regla que los utilicen todos. Para conocer las operaciones que pueden ser utilizadas así como los paquetes incluidos en el software, es recomendable revisar las librerías del programa. De no encontrarse algún operador especial para ser utilizado con algún tipo de dato especifico, es necesario sobrecargar los operadores o en ocasiones crearlo. Como sobrecargar operadores y como crear funciones se expone dentro del tema de subprogramas. Para poder utilizar la mayoría de estos operadores con los tipos signed, unsigned y std_logic_vector, basta con utilizar el paquete donde se encuentran declarados estos tipos, porque dentro de los mismos paquetes ya se encuentran sobrecargada varias funciones aritméticas y lógicas para que sean utilizadas con estos tipos, en temas posteriores se incluyen las funciones que se encuentran en los paquetes std_logic_1164, numeric_std y numeric_bit.

TIPOS DE OPERADORES

LÓGICOS	AND, OR, NAND, NOR, XOR, XNOR, NOT
COMPARACIÓN	=, /=, <, >, <=, >=
ADICIÓN	+, -, &
MULTIPLICACIÓN	*, /, MOD, REM
MISCELÁNEOS	ABS, **
ASIGNACIÓN	<=, :=
ASOCIACIÓN	=>
CORRIMIENTO	SLL, SRL, SLA, SRA, ROL, ROR

3.5.1 OPERADORES LÓGICOS

Los operadores lógicos AND, OR, NAND, NOR XOR, XNOR, y NOT están definidos para ser usados con los tipos **bit** y **boolean**. Para utilizar estos operadores, excepto el operador NOT, los operandos deben ser del mismo tamaño.

Los operadores lógicos no tiene orden de precedencia por lo que para expresiones en las que se utilice más de un operador lógico es necesario indicar mediante paréntesis cual es el orden en que se debe realizar el cálculo.

EJEMPLOS

```
x <= y AND z OR w
     -- esta forma de utilizar los
     -- op. lógicos, es incorrecta y
     -- producirá un error cuando sea
     -- compilado el código


x <= y AND ( z OR w )
     -- forma correcta de utilizar los
     -- op.lógicos
```

3.5.2 OPERADORES DE COMPARACIÓN

Estos tipos de operadores se utilizan para ejecutar pruebas de igualdad, desigualdad, o de magnitud entre dos objetos de datos. Los operandos que participen en la prueba deben ser del mismo tipo y el resultado de la operación es del tipo boolean. Los operadores de igualdad y desigualdad (= y /=) pueden ser utilizados para todos los tipos de datos predefinidos en el lenguaje Los operadores de magnitud ("<" menor que, ">" mayor que, "<=" menor o igual que, y ">=" mayor o igual que) están definidos para ser utilizados con tipos escalares.

EJEMPLO

```
SIGNAL x, y: bit_vector ( 3 DOWNTO 0)

SIGNAL z : bit;

z <= '1' WHEN x >= z ELSE '0';
```

3.5.3 OPERADORES DE ADICIÓN

Los operadores + y - son frecuentemente utilizados para describir sumas y restas además de signos positivo y negativo. Están definidos para ser utilizados con el tipo **entero**. El operador "&" permite concatenar cadenas de bits obteniendo una de mayor tamaño. Los tres operadores tienen la misma precedencia, por lo que para instrucciones en la que se utiliza más de un operador de este tipo es recomendable indicar mediante paréntesis el orden de las operaciones.

Para poder realizar operaciones de suma o resta entre un entero y un objeto de datos que represente una cadena de bits, lo mejor es declarar este objeto de datos como signed o unsigned e incluir el paquete numeric_std o el numeric_bit, ya que en estos se sobrecargaron los operadores "+" y "-" para que pudieran ser utilizados de esta manera. De lo contrario resultaría en un error de compilación al realizar dichas operaciones. Por otra parte no es posible realizar sumas entre bit_vectors de diferente tamaño, y tampoco podremos asignar el resultado de una suma o resta entre dos bit_vectors a un bit_vector de diferente longitud. Si se desea obtener el acarreo de la suma del resultado de una operación aritmética entre dos bit_vectors de la misma longitud, a un bit_vector que sea de mayor longitud en un bit, se permite utilizar el operador de concatenación para incrementar el tamaño **solamente** en el primer bit_vector que participa en la operación, con lo cual se indica que deseamos obtener el acarreo de la suma.

EJEMPLOS

```
SIGNAL conteo: integer RANGE 0 TO 255;
SIGNAL x, y, z: signed( 7 downto 0);
SIGNAL r, m: signed( 8 downto 0);

conteo <= conteo + 1;
x <= y + z + 5;
r <= '0'z + x; -- de esta manera se
               -- obtiene el acarreo de
               -- la suma
m <= r +1;
```

3.5.4 OPERADORES DE MULTIPLICACIÓN

Son los operandos "*" y el "/" que se utilizan para la multiplicación y para la división respectivamente. Los dos operandos tienen el mismo orden de precedencia al igual que los operandos MOD y REM.

Todos los operandos de multiplicación están definidos para ser utilizados con operandos del mismo tipo, siendo estos del tipo entero o bit_vector. El resultado es entonces del mismo tipo que los operandos por lo que también el objeto de datos que recibe el resultado de la operación deberá ser del mismo tipo que los operandos.

La operación REM se define como se muestra a continuación:

```
A REM B = A-(A/B)*B
```

La división es entera, por lo que los operandos deben ser del tipo entero. El resultado toma el signo de A. MOD calcula el módulo de dos números. Se define como:

```
A MOD B = A-B*N
```

Donde N es un entero. El resultado toma el signo de B.

3.5.5 OPERADORES MISCELÁNEOS

En esta categoría se encuentran los operadores "abs" y "**". El operador "abs" devuelve el valor absoluto de un operando del tipo **entero**. El operador ** se utiliza para elevar el primer operador a una potencia definida por el segundo operando, ambos deben ser del tipo entero.

EJEMPLO

```
CONSTANT r: integer := 2;
VARIABLE i: integer;

FOR n IN 0 TO 5 LOOP
    i := i + r**n;
END LOOP;
```

3.5.6 OPERADORES DE ASIGNACIÓN

En VHDL existen dos tipos de operadores de asignación los cuales son: "<=" y ":=". El operador ":=" se utiliza para asignar un valor inicial a constantes, variables y señales en el momento de la declaración, pero para el resto de la descripción únicamente utilizaremos ":=" para ser usado con variables y "<=" para ser usado con señales.

ASIGNACIÓN A VARIABLES

```
nombre_variable := expresión;
```

ASIGNACIÓN A SEÑALES

```
nombre_señal <= expresión;
```

Las asignaciones a variables solamente pueden ocurrir dentro de los procesos o subrutinas, las asignaciones a señales pueden ocurrir en cualquier lugar dentro de la descripción.

Para realizar asignaciones a objetos de datos de tipos compuesto, se pueden realizar utilizando **agregados**. Los agregados son una lista de varios valores encerrados entre paréntesis y separados mediante comas de tal forma que el primer elemento de la lista es asignado al primer elemento del objeto, el segundo elemento de la lista es asignado al segundo elemento del objeto de datos etc. Así mediante una sola instrucción se asignan varios valores al objeto de datos.

EJEMPLOS

```
TYPE op IS (suma, resta, mult, div);
TYPE reg_datos IS RECORD
  operador: op;
  x: integer;
  y: bit;
END RECORD;
. . .
VARIABLE registro: reg_datos;
SIGNAL vec1, vec2: bit_vector(0 TO 3);
. . .
vec1 <= ('0', '1', '1', '0');
-- asignación mediante agregados

vec2 <= vec1;
-- también esta es una asignación
-- mediante agregados

registro := ( resta, 13, '1' );
-- asignación a variable del tipo
-- record mediante agregados

vec2 <= ( '1', OTHERS => '0' );
-- en esta asignación se hace '1'
-- el elemento 0 de vector2 y el resto
-- se hacen cero
```

3.5.7 OPERADORES DE ASOCIACIÓN

En diseños jerárquicos generalmente se hace uso de varios componentes, los cuales son entidades que realizan ciertas funciones especificas. Para poder especificar las conexiones de puertos entre dichos componentes y con los puertos de la entidad principal es necesario utilizar el operador de asociación "=>". El orden con el que se asocian dichas conexiones depende del orden en el que fueron declarados los puertos del componente, además, deben ser del mismo tipo y del mismo modo. Diseños jerárquicos y componentes se explicaran detalladamente en temas posteriores. A continuación se muestra un ejemplo de cómo utilizar este operador de asociación.

EJEMPLO

```
LIBRARY mi_libreria;
USE mi_libreria.sumadores.ALL;

ENTITY sumador IS
  PORT ( ci: IN bit;
         x: IN bit_vector(3 DOWNTO 0);
         y: IN bit_vector(3 DOWNTO 0);
         z: OUT bit_vector(3 DOWNTO 0);
         co: OUT bit);
END sumador;
ARCHITECTURE a_sumador OF sumador IS
  SIGNAL carry1: bit;
  SIGNAL carry2: bit;
  SIGNAL carry3: bit;
BEGIN
  u0: add PORT MAP (  ci => ci,
                      x0 => x(0),
                      y0 => y(0),
                      z0 => z(0),
                      co => carry1 );
  u1: add PORT MAP (  ci => carry1,
                      x0 => x(1),
                      y0 => y(1),
                      z0 => z(1),
                      co => carry2 );
  u2: add PORT MAP ( carry2, x(2), y(2),
                      z(2), carry3 );
  u3: add PORT MAP ( carry3, x(3), y(3),
                      z(3), co );
END a_sumador;
```

El componente “add” esta declarado dentro del paquete “sumadores” de la librería “mi_libreria”, y esta declarado de la siguiente manera.

```
-----------------------------------------
-- SUMADORES
-----------------------------------------
-- paquete compilado en la librería
-- "mi_librería"

PACKAGE sumadores IS
. . .
  COMPONENT add
    PORT ( ci: IN bit;
           x0: IN bit;
           y0: IN bit;
           z0: OUT bit;
           co: OUT bit );
  END COMPONENT;
. . .
END PACKAGE;
```

Observe como los puertos de la entidad “sumador”y las conexiones entre los bloques u0, u1, u2 y u3, se hicieron de acuerdo al orden en que los puertos están declarados en el componente. En el bloque u0 primero se hace la conexión del puerto "ci" del componente con el puerto “ci”de la entidad mediante le operador de

asociación "=>", después se hace la conexión del puerto x0 del componente con el elemento 0 (LSB) del vector x de la entidad, y así sucesivamente hasta realizar todas las conexiones del componente "add"utilizado en el bloque u0. Lo mismo se hace con el bloque u1. En los bloques u2 y u3, las conexiones se realizaron con una notación equivalente pero simplificada. Los nombres no tienen que ser necesariamente los mismos e inclusive pueden ser diferentes, ya que cada puerto es un objeto de datos local para la entidad en la que fue declarado. Es importante mencionar que dentro del paquete "sumadores" se encuentra la entidad y la arquitectura correspondiente al componente "add", en los que se describe el mismo.

3.5.8 OPERADORES DE CORRIMIENTO

Incluidos en los paquetes numeric_std y numeric_bit, estos operadores realizan operaciones de desplazamiento o de rotación con los elementos de un vector del tipo **signed** o **unsigned**.

DESPLAZAMIENTOS LÓGICOS SLL Y SRL

Desplazan los bits de un vector *n* veces a la izquierda (SLL) o a la derecha (SRL), introduciendo ceros en los lugares que quedan libres.

EJEMPLO

```
x SRL 3  -- desplaza 3 lugares a la
-- derecha los bits del vector "x"
```

DESPLAZAMIENTOS ARITMÉTICOS SLA Y SRA

También desplazan los bits de un vector *n* veces a la izquierda (SLA) o a la derecha (SRA), introduciendo ceros en los lugares que quedan libres, pero conservan el signo.

ROTACIONES ROL Y ROR

Se desplazan los bits de un vector *n* veces a la izquierda (ROL) o a la derecha (ROR), introduciendo los bits que son desplazados en los lugares que van quedando libres.

ROL ***ROR***

Figura 3.4 Instrucciones ROR y ROL

3.5.9 OPERACIONES CON VECTORES

Todas las herramientas de síntesis proporcionan algún tipo de paquete en el que se encuentre definidas funciones que facilitan la descripción del diseño. Dentro de estos paquetes se encuentran funciones que están hechas específicamente para ser utilizadas con vectores y como por lo general es preferible utilizar vectores estos paquetes son de gran ayuda.

Synopsys desarrolló paquetes basados en el paquete std_logic_1164, que son utilizados por varias herramientas de síntesis existentes en el mercado, como por ejemplo FOUNDATION de Xilinx, Inc. y MAX+PLUS II de Altera Corporation. Estos paquetes son:

- std_logic_arith

- std_logic_signed

- std_logic_unsigned

La compañía Actel también desarrolló su propio paquete de síntesis:

- asyl.arith

Y los paquetes que fueron desarrollados por la IEEE específicamente para síntesis de circuitos digitales.

- numeric_bit

- numeric_std

Además del paquete que es el más utilizado por la mayoría de los paquetes de síntesis.

- std_logic_1164

Todos estos paquetes son los más conocidos y utilizados para síntesis de circuitos, por lo que para poder utilizarlos primero debemos de incluir la librería en que fueron compilados para posteriormente hacer referencia al paquete que deseamos utilizar (una librería puede tener más de un paquete) como se muestra a continuación.

EJEMPLO

```
LYBRARY ieee; -- llamado a la librería

USE ieee.std_logic_1164.ALL; -- referencia
```

```
                    -- o carga del paquete
                    -- std_logic_1164
                    -- "ALL" es para
                    -- indicar que
                    -- deseamos utilizar
                    -- todos los tipos de
                    -- datos y funciones
                    -- incluidas en el
                    -- paquete

USE ieee.numeric_std.ALL;  -- referencia
                    -- al paquete
                    -- numeric_std
```

3.6 ATRIBUTOS

Un atributo es una propiedad que es asociada a señales, entidades o arquitecturas. Estos atributos proporcionan información que nos puede ser útil dentro de una descripción en VHDL. Los atributos se utilizan mediante la comilla simple, por ejemplo el atributo ***'event***, que probablemente sea el más utilizado, nos permite detectar cuando sucede una transición de estado en una señal, por lo que es muy útil en descripciones de circuitos secuenciales.

REFERENCIA A ATRIBUTOS

```
nombre_objeto'nombre_atributo
```

EJEMPLO

```
IF ( clk'event and clk = '1' ) THEN
  A <= '1' ;
END IF;
```

En el ejemplo anterior se utiliza el atributo ***'event***, indicado en color verde, para detectar una transición en la señal clk, y al mismo tiempo comprobamos si esta transición fue positiva. Si ambas condiciones se cumplen entonces se asigna un '1' lógico a "A". El atributo ***'event*** se utiliza solo para señales de *clk* ya que de otra manera no es posible sintetizar una transición en un dispositivo lógico programable, por lo que también debemos indicar que tipo de transición estamos utilizando.

Existen más atributos y a continuación se mencionan algunos que son útiles en descripciones para síntesis.

ATRIBUTOS PARA ARREGLOS

'left	Obtiene el valor que se encuentra a la izquierda de un arreglo.
'right	Regresa el dato que se encuentra a la derecha del arreglo.
'high	Permite obtener el mayor elemento de un objeto de arreglo.
'low	Proporciona el valor más pequeño del arreglo.
'lenght	Con este atributo se obtiene el número de elementos de un arreglo.

Un arreglo es un objeto de datos que esta compuesto por varios elementos de un tipo sencillo, como lo son los bit_vector y std_logic_vector.

ATRIBUTOS DE SEÑALES

'event	El atributo event es del tipo boolean y retorna un valor verdadero cuando ocurre una transición en la señal a la que hace referencia.

EJEMPLOS

```
TYPE secuencia IS integer RANGE 0 TO 10;
SIGNAL conteo: secuencia;
. . .
conteo'left = 0
conteo'rigth = 10
conteo'lenght = 11
conteo'high = 10
conteo'low = 0
```

3.7 ENTIDADES

Una entidad es la abstracción de un circuito, ya sea desde un complejo sistema electrónico hasta una simple compuerta lógica. La entidad únicamente describe la forma externa del circuito, aquí se enumeran las entradas y salidas del diseño. Una entidad es análoga a un símbolo esquemático de los diagramas electrónicos, el cual describe las conexiones del dispositivo hacia el resto del diseño.

DECLARACION DE ENTIDADES

```
ENTITY identificador IS
  GENERIC ( cte_1: tipo := valor;
            cte_2: tipo := valor;
              . . .
            cte_n: tipo := valor
          );
  PORT ( puerto_1: modo tipo;
         puerto_2: modo tipo;
            . . .
       puerto_n: modo tipo
     ) ;
  END identificador ;
```

Note que la ultima línea de declaración de puerto o de genéricos no lleva punto y coma al final de la línea.

EJEMPLO

En este ejemplo se realiza la entidad de un multiplexor 4 a 1 que se muestra en la figura 3.5.

```
ENTITY mux_4_1 IS
  PORT (a, b, c, d: IN bit;
  mux_signal: IN  bit_vector(1 DOWNTO 0);
  x: OUT bit );
END mux_4_1;
```

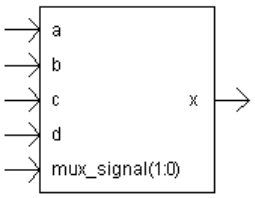


Figura 3.5 Multiplexor de 4 bits a 1

3.7.1 GENÉRICOS

Esta instrucción es opcional y se utiliza para declarar propiedades y constantes del circuito. Estas constantes se utilizan al igual que las que se declaran por el usuario, por lo que nos permiten modelar circuitos en los que se pueden cambiar propiedades, tamaños de los buses de entrada o salida del circuito. Se utilizan generalmente en paquetes.

EJEMPLO

```
ENTITY comparador IS
  GENERIC(msb: integer := 3);
  PORT( x: IN bit_vector(msb DOWNTO 0);
        y: IN bit_vector(msb DOWNTO 0);
        equals: OUT bit;
        x_may_y: OUT bit;
        x_men_y; OUT bit);
END comparador;
```

3.7.2 PUERTOS

Cada entrada y salida de la entidad se declara dentro de la región puertos (PORT), en el momento de la declaración se debe indicar el modo y tipo del puerto. Los puertos los podemos asociar con los pines de un símbolo esquemático y, al igual que estos, algunos son únicamente entradas, otros salidas, o incluso bidirecionales. Un puerto es implícitamente un objeto de datos del tipo **señal** porque representa **conexiones** en el diseño, y puede ser utilizado en expresiones de programación dentro de la arquitectura que describe a dicha entidad. Cada puerto debe tener un nombre, un modo y se debe especificar el tipo de dato mediante el cual manipularemos dicho puerto en la descripción.

3.7.3 MODOS

El modo indica la forma en que los datos fluyen a través del circuito. Estos pueden ser de uno de cuatro tipos:

- IN
- OUT
- INOUT
- BUFFER

Si no se indica ningún modo en la declaración, se asume que es del tipo IN. Un puerto del modo IN describe un pin del circuito que únicamente puede ser utilizado como entrada por lo que solamente podremos leer datos de dicho puerto y nunca escribir sobre él. Por el contrario, un puerto que sea declarado del modo OUT podrá ser utilizado para escribir datos pero no para ser leído, este representa un pin que únicamente es salida del circuito y que en él no existe ningún tipo de retroalimentación hacia dentro del diseño. Un puerto INOUT indica aquellos puertos que son pueden ser utilizados bidireccionalmente mientras que un puerto del modo BUFFER es utilizado para salidas que tienen retroalimentación interna. La diferencia entre el modo BUFFER y el INOUT, es que el INOUT es retroalimentado desde el pin de salida del circuito, en tanto que

como un puerto del modo BUFFER lo hace internamente no puede ser usado como bidireccional.

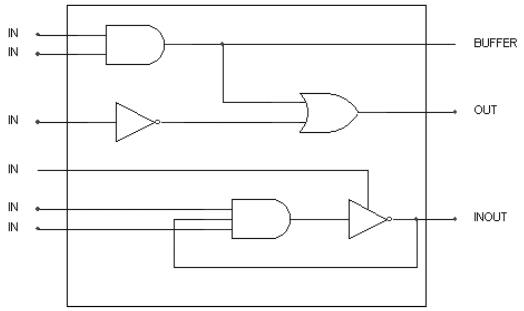


Figura 36 Modos de direccionamiento para puertos

3.8 ARQUITECTURAS

Los pares de entidades y arquitecturas se utilizan para representar la descripción completa de un diseño. Una arquitectura, describe el funcionamiento de la entidad a la que hace referencia. Si una entidad la asociamos con una "caja" en la que se enumeran las interfaces de conexión hacia el exterior, entonces la arquitectura representa la estructura interna de esa caja. Por ejemplo, el símbolo esquemático de un 74LS04 representaría la entidad del diseño, y la forma en que las compuertas son conectadas internamente corresponden a la arquitectura del circuito, y así ambos describen completamente el circuito.

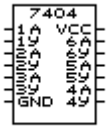


Figura 3.7 Entidad, Símbolo Esquemático

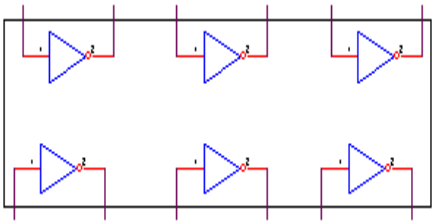


Figura 3.8 Arquitectura, Estructura Interna

DECLARACIÓN DE ARQUITECTURA

```
ARCHITECTURE identificador OF entidad IS
  -- declaraciones de la arquitectura
BEGIN
  -- Código de Descripción
  --   instrucciones concurrentes
  --   ecuaciones booleanas
  --   PROCESS
  --     instrucciones secuenciales
END identificador_arquitectura;
```

Antes del BEGIN se escriben todas las declaraciones que se necesiten dentro de la descripción, tales como: señales, constantes, funciones, alias, componentes, tipos de datos etc. Después del BEGIN es donde se realiza todo el código de descripción del circuito.

EJEMPLO

```
ENTITY mux_4_1 IS
  PORT ( a, b, c, d: IN bit;
  mux_signal: IN  bit_vector(1 DOWNTO 0);
  x: OUT bit );
END mux_4_1;

ARCHITECTURE a_mux_4_1 OF mux_4_1 IS
BEGIN
  x <= a WHEN mux_signal = "00" ELSE
       b WHEN mux_signal = "01" ELSE
       c WHEN mux_signal = "10" ELSE
       d WHEN mux_signal = "11";
END a_mux_4_1;
```

Una arquitectura describe el comportamiento, estructura o flujo de datos de la entidad a la que hace referencia. Una entidad puede tener más de una arquitectura, pero cuando se compile se indica cual es la arquitectura que queremos utilizar. Para describir el funcionamiento de la entidad se puede hacer uso de cualquiera de los tres estilos siguientes:

- Descripción de Flujo de Datos
- Descripción Comportamental
- Descripción Estructural

Los tres estilos son diferentes, pero esto no significa que se tenga que utilizar únicamente un estilo. De hecho lo mejor es tratar de utilizar los tres como mejor nos convenga. En el siguiente tema se explica el estilo de descripción de flujo de datos, así como el tipo de instrucciones que participan en este estilo.

3.9 DESCRIPCIONES DE FLUJO DE DATOS

Una descripción de flujo de datos consiste en especificar como los datos son transferidos de las entradas a las salidas. Cabe mencionar que algunos autores distinguen las descripciones de flujo de datos de las comportamentales, en tanto que para otros ambos estilos son del tipo comportamental. La principal diferencia entre estas es el tipo de instrucciones que utilizan, además que en un estilo comportamental se utiliza el bloque PROCESS en tanto que en el estilo en cuestión no se utiliza.

En este estilo de descripción se utilizan únicamente asignaciones mediante expresiones en las que se indica como cambian los puertos de salida en función de los puertos de entrada, ya sean asignaciones condicionales mediante instrucciones concurrentes o simples ecuaciones. Un ejemplo de descripción de flujo de datos es el comprador utilizado en el primer tema de este capítulo, en éste los datos son los que indican la forma en que cambian las salidas y por esto se le llama de flujo de datos.

3.9.1 INSTRUCCIONES CONCURRENTES

En lenguajes de programación como C o Pascal, cada instrucción de asignación es ejecutada una después de otra en un orden especifico. El orden en el que las instrucciones son ejecutadas es determinado por el orden de las instrucciones en el archivo. Dentro de una arquitectura en VHDL, no existe un orden especifico de ejecución de las asignaciones. El orden en el que las instrucciones son ejecutadas depende de los eventos ocurridos en las señales, similar al funcionamiento de un circuito.

En VHDL todos los bloques son concurrentes, es decir que se están ejecutando en todo momento. Después se explicará el bloque PROCESS, el cual está compuesto por una serie de instrucciones que sí se ejecutan en el orden en el que fueron especificadas. Las instrucciones concurrentes se utilizan fuera de un bloque PROCESS, a diferencia de las instrucciones secuenciales que únicamente se utilizan dentro del bloque concurrente PROCESS y en subprogramas.

3.9.2 ESTRUCTURAS DE EJECUCIÓN CONCURRENTE

- **ASIGNACIÓN CONDICIONAL WHEN... ELSE**

SINTAXIS

```
signal_name <= valor_a WHEN condición ELSE
               valor_b WHEN condición ELSE
               valor_c WHEN condición ELSE
               valor_d WHEN condición ELSE

                  . . .
               valor_n WHEN condición ELSE
               otro_valor;
```

EJEMPLO

```
gray <= "00" WHEN binario = x"0" ELSE
        "01" WHEN binario = x"1" ELSE
        "11" WHEN binario = x"2" ELSE
        "10";
```

- **ASIGNACIÓN WHIT... SELECT... WHEN**

SINTAXIS

```
WITH identificador SELECT
  signal_name1 <= expresión WHEN valor1,
                  valor_a WHEN valor2,
                  valor_b WHEN valor3,
                     . . .
                  valor_n WHEN OTHERS ;
```

EJEMPLO

```
WITH states SELECT
  salida <= "000" WHEN state0,
            "001" WHEN state1,
            "010" WHEN state2,
            "100" WHEN state3,
            "000" WHEN OTHERS;
```

No olvidar la coma al final de cada línea, excepto en la ultima que lleva punto y coma.

- **ECUACIONES BOOLEANAS**

```
signal_name <= ecuación_booleana;
```

EJEMPLOS

```
x <= y AND z;

a <= ( b OR c OR d ) AND e ;
-- cuando se utilice más de un operador
-- lógico es necesario utilizar paréntesis

op1 <= op2 NOR op3 NOR op4;
```

3.9.3 ALU

A continuación se muestra la tabla de funcionamiento de una pequeña unidad aritmético — lógica la cual consta de dos bits de entrada, un bit de salida, acarreo de entrada y acarreo de salida.

TABLA DE FUNCIONAMIENTO

ENTRADAS		SALIDAS	
s1	s0	z	co

0	0	x **AND** y	0
0	1	x **OR** y	0
1	0	x **XOR** y	0
1	1	x + y + ci	acarreo de la suma

CÓDIGO DE DESCRIPCIÓN

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY alu IS
  PORT ( x, y: IN std_logic;
         s0, s1: IN std_logic;
         ci: IN std_logic;
         z: OUT std_logic;
         co: OUT std_logic);
END alu;

ARCHITECTURE a_alu OF alu IS
  SIGNAL seleccion: unsigned(1 DOWNTO 0);
  SIGNAL suma: std_logic;
  SIGNAL and_op: std_logic;
  SIGNAL or_op: std_logic;
  SIGNAL xor_op: std_logic;
  SIGNAL acarreo: std_logic;
BEGIN

  and_op <= x AND y ;

  WITH seleccion SELECT
    z <= or_op WHEN "01",
         and_op WHEN "00",
         suma WHEN "11",
         xor_op WHEN "10",
         '0' WHEN OTHERS;

  seleccion <= s1 & s0;

  suma <= x XOR y XOR ci;

  or_op <= x OR y;

  acarreo <= (x AND y) OR
             (x AND ci) OR
             (y AND ci);

  xor_op <= x XOR y;

  co <= acarreo WHEN seleccion = 3 ELSE
        '0' ;
END a_alu;
```

Este circuito se sintetizo en un GAL22V10 utilizando WARP 5.0 de Cypress Semiconductors. Obsérvese que en el código de descripción de esta *alu* no existe ningún orden en las ecuaciones, asignaciones o instrucciones, de hecho el orden en el que se coloquen las instrucciones no importa dentro de una descripción, recordemos que VHDL describe circuitos y por lo tanto todas las asignaciones (conexiones) siempre están funcionando al igual que todos los dispositivos dentro de un circuito.

Las señales suma, acarreo, and_op, or_op y xor_op son las salidas de varios bloques del circuito, cada uno con una función distinta. El bus "seleccion" se compone de los dos bits de entrada s0 y s1, estas se juntan en este bus únicamente para facilitarnos la descripción. Y mediante los bloques de la instrucción WITH... SELECT... WHEN y la instrucción WHEN... ELSE se asignan los datos correspondientes a cada salida de a cuerdo a las combinaciones de los bits s0 y s1.

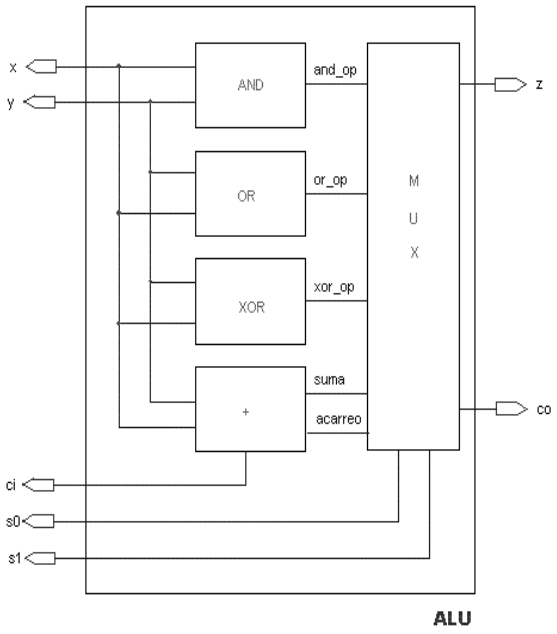


Figura 3.9 Diagrama a bloques de la ALU

El que hayamos realizado la descripción con tantos bloques no significa necesariamente que cuando el compilador sintetice el código respete todos esos bloques y quede exactamente como lo describimos dentro del dispositivo que seleccionamos. El compilador de WARP interpreta nuestra descripción y la sintetiza dentro del dispositivo utilizando el mínimo de compuertas posible al mismo tiempo que trata de respetar la descripción del código, además, que por lo general trata de evitar retroalimentaciones

para que el dispositivo funcione a altas velocidades. De hecho en ocasiones es posible que existan todavía macroceldas libres, pero como el compilador evita retroalimentaciones, entonces no las usa. Como indicar que se usen esas macroceldas libres se explicará posteriormente, ya que este circuito sí puede quedar dentro de un GAL16V8 utilizando directivas de síntesis. A continuación se muestran las ecuaciones, la asignación de pines y el informe de utilización del dispositivo obtenidos por el compilador de WARP.

ECUACIONES

```
PLD Compiler Software: MAX2JED.EXE
02/APR/1999 [v4.02 ] 5.2 IR 17

DESIGN EQUATIONS (05:56:54)

    z =
          /x * /y * s0 * s1 * ci
        + x * /y * s1 * /ci
        + /x * y * s1 * /ci
        + x * /y * /s0 * s1
        + /x * y * /s0 * s1
        + x * y * s0 * ci
        + y * s0 * /s1
        + x * s0 * /s1
        + x * y * /s1

    co =
          y * s0 * s1 * ci
        + x * s0 * s1 * ci
        + x * y * s0 * s1


Completed Successfully
```

PIN – OUT

```
PLD Compiler Software: PLA2JED.EXE
02/APR/1999 [v4.02 ] 5.2 IR 17

PINOUT INFORMATION (06:15:24)

Messages:
Information: Checking for duplicate NODE
logic.
None.

                              C22V10
                 __________________________________
       ci =| 1|                                  |24|* not used
       s1 =| 2|                                  |23|* not used
       s0 =| 3|                                  |22|* not used
        y =| 4|                                  |21|* not used
        x =| 5|                                  |20|* not used
 not used *| 6|                                  |19|* not used
 not used *| 7|                                  |18|* not used
 not used *| 8|                                  |17|* not used
 not used *| 9|                                  |16|* not used
 not used *|10|                                  |15|= z
 not used *|11|                                  |14|= co
 not used *|12|                                  |13|* not used
                 __________________________________

Summary:
Error Count = 0 Warning Count = 0
Completed Successfully
```

UTILIZACIÓN

```
PLD Compiler Software: PLA2JED.EXE
02/APR/1999 [v4.02 ] 5.2 IR 17
RESOURCE UTILIZATION (06:15:25)
Information: Macrocell Utilization.
```

Description	Used	Max
Dedicated Inputs	4	11
Clock/Inputs	1	1
I/O Macrocells	2	10
	7 /	22 = 31 %

```
Information: Output Logic Product Term
Utilization.
```

Node#	Output Signal Name	Used	Max
14	co	3	8
15	z	9	10
16	Unused	0	12
17	Unused	0	14
18	Unused	0	16
19	Unused	0	16
20	Unused	0	14
21	Unused	0	12
22	Unused	0	10
23	Unused	0	8
25	Unused	0	1
		12 /	121 = 9 %

```
Completed Successfully
```

3.10 DESCRIPCIONES COMPORTAMENTALES

Las descripciones comportamentales son similares a un lenguaje de programación de alto nivel, por su alto nivel de abstracción. Mas que especificar la estructura o la forma en que se deben conectar los componentes de un diseño, nos limitamos a describir su comportamiento. Una descripción comportamental consiste de una serie de instrucciones, que ejecutadas secuencialmente, modelan el comportamiento del circuito. La ventaja de una descripción comportamental es que no necesitamos enfocarnos a un nivel de compuerta para implementar un diseño. En VHDL una descripción comportamental necesariamente implica el uso de por lo menos un bloque PROCESS.

3.10.1 INSTRUCCIONES SECUENCIALES

Las instrucciones secuenciales son aquellas que son ejecutadas serialmente, una después de otra. La mayoría de los lenguajes de programación, como C o Pascal, utilizan este tipo de instrucciones. En VHDL las instrucciones secuenciales son implementadas únicamente dentro del bloque PROCESS

3.10.2 PROCESOS

Un proceso es el bloque básico concurrente de codificación secuencial. Contiene una serie de instrucciones secuenciales que permiten modelar el comportamiento del circuito, sin embargo, el bloque PROCESS equivale a una sola instrucción concurrente. Un proceso puede ser utilizado dentro de cualquier arquitectura definiendo para si mismo una región de declaraciones y otra para la codificación secuencial, similar a una arquitectura. La región de codificación puede contener únicamente instrucciones secuenciales (IF, CASE, FOR, etc.) en tanto que la región de declaraciones permite designar constantes, señales, tipos de datos o algún alias.

SINTAXIS

```
PROCESS ( lista sensible )
  -- declaraciones
BEGIN
  -- instrucciones secuenciales
END PROCESS;
```

La lista sensible define cuales señales provocan que las instrucciones dentro del bloque comiencen a ser ejecutadas. Los cambios en alguna de las señales provocan que el proceso sea llamado. Un proceso que no tenga lista sensible debe utilizar una instrucción WAIT para especificar cuando deben ser ejecutadas las instrucciones dentro del bloque. La mayoría de las herramientas de síntesis tienen problemas si las lista sensible no está completamente especificada. Estas consideran que mediante el proceso estamos modelando lógica combinacional o secuencial. La lista sensible es parcialmente declarada cuando alguna de las señales que intervienen en lado derecho de una ecuación o de alguna instrucción secuencial no es mencionada dentro de la lista. El que la lista no este completa generalmente produce que no sea posible modelar totalmente la funcionalidad del diseño y por lo tanto no es posible obtener las ecuaciones durante el proceso de síntesis.

El funcionamiento del proceso es similar a un microprocesador que funciona únicamente con interrupciones. La señales dentro de la lista sensible hacen a su vez de entradas de interrupción y las instrucciones secuenciales se encuentran dentro de una rutina única de servicio de interrupción. Cuando alguna de las señales de la lista sensible cambia, provoca que el proceso comience a funcionar y a ejecutar toda esta rutina de ejecución secuencial con la particularidad de que los que resulte de este procesamiento se asigne únicamente al final de la estructura. Por lo que podemos manipular los valores de las señales y esto no implica que cambien con cada asignación sino solamente hasta que se termina de ejecutar todo el proceso. Y como las asignaciones a los nodos del circuito se hacen al final, entonces todo la estructura del proceso es similar a un dispositivo de ejecución secuencial, como un microprocesador, que forma parte del diseño. Esta comparación con un microprocesador no implica que siempre debamos especificar una señal de reloj para el funcionamiento de la estructura, o que únicamente nos permita modelar circuitos secuenciales. De hecho, si suponemos que la frecuencia de trabajo de este "microprocesador" es muy grande, entonces las instrucciones dentro de la estructura se ejecutan tan rápido que prácticamente lo podríamos considerar combinacional. Si alguna señal de reloj es especificada, entonces estamos limitando a que

las instrucciones dentro del proceso sean ejecutadas únicamente dentro de alguna transición de esta señal, lo cual no permite describir circuitos secuenciales.

EJEMPLO

A continuación se muestra el código de descripción comportamental del comparador de la figura 3.10.

```
ENTITY comparador IS
  PORT( x: IN bit_vector(3 DOWNTO 0);
        y: IN bit_vector(3 DOWNTO 0);
        equals: OUT bit;
        x_may_y: OUT bit;
        x_men_y: OUT bit);
END comparador;

ARCHITECTURE comparador OF comparador IS
BEGIN
  PROCESS(x, y)
  BEGIN
    equals <= '0';
    x_may_y <= '0';
    x_men_y <= '0';
    IF x = y THEN
      equals <= '1';
    END IF;
    IF x > y THEN
      x_may_y <= '1',
    END IF;
    IF x < y THEN
      x_men_y <= '1';
    END IF;
  END PROCESS;
END comparador;
```

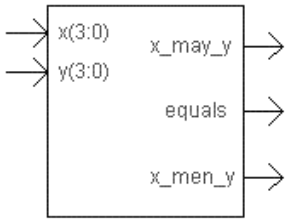


Figura 3.10 Comparador

Este ejemplo corresponde al mismo comprador utilizado en el tema 3.1.2. Se definen 2 vectores de 4 bits y 3 salidas de 1 bit. Esta arquitectura únicamente tiene una instrucción concurrente: el bloque PROCESS, el cual es sensible a los vectores de entrada. Siempre que ocurra un cambio en alguno de estos, el proceso será llamado y generará la lógica de salida. La lista sensible está completa porque, si observamos, el estado de las salidas depende únicamente de las entradas.

Cada instrucción será ejecutada en orden secuencial y cuando todas hallan sido ejecutadas, entonces se asigna el valor procesado a los nodos que se vieron afectados durante el proceso. Una vez que se terminó de ejecutar el proceso, éste se mantendrá inactivo hasta que alguno dos elementos de la lista sensible cambie.

Cuando se utilicen procesos se debe tener cuidado de no olvidar alguna combinación posible de entradas y/o salidas retroalimentadas que tal vez no estemos considerando o que no necesitamos. En estos casos es recomendable utilizar el tipo std_logic o, si son vectores, algún tipo que se base en este. Los valores '-' y 'Z' del std_logic son permitidos en síntesis siempre y cuando se utilicen correctamente.

EJEMPLO

```
ARCHITECTURE simplifica OF entidad_x IS
SIGNAL y_tmp:std_logic_vector(1 DOWNTO 0);
BEGIN
  PROCESS(s)
  BEGIN
    IF (s = 0) OR (s = 3) THEN
      -- s es un vector del tipo unsigned
      y_tmp <= '1';
    ELSIF s = 1 THEN
      y_tmp <= '0';
    ELSE
      y_tmp <= '-';
    END IF;
  END PROCESS;
  y <= y_tmp WHEN enable = '0' ELSE 'Z';
  -- "y" y "y_tmp" son tipo std_logic
END simplifica;
```

En el ejemplo anterior únicamente nos importan las combinaciones s = 3, s = 1, y s = 0. En algunos programas de VHDL para síntesis, y también dependiendo de la arquitectura del dispositivo, la asignación de un "no importa" nos va a permitir que se simplifique la ecuación de y_tmp, quedando como:

```
y_tmp = s(1) +s(0)'
```

De no utilizarla posiblemente quede de la siguiente forma:

```
y_tmp = s(0)'·s(1)' + s(0)·s(1)
```

3.10.3 DIFERENCIAS ENTRE SEÑALES Y VARIABLES

Un objeto de datos del tipo señal es muy diferente a uno del tipo variable. Ya se había mencionado que las señales pueden ser sintetizados en elementos lógicos y/o conexiones, lo cual no es posible con un variable. Una señal representa un nodo de conexión entre elementos lógicos (compuertas, registros, buffers, etc.). Inclusive un mismo nodo puede recibir más de un nombre para facilitar la descripción, sin que esto implique más términos en las ecuaciones de salida. Una señal que se vea involucrada dentro de un proceso no recibe inmediatamente el valor asignado, sólo hasta el final del mismo. Una variable que sea utilizada dentro de un proceso sí recibe el valor de forma inmediata, por lo que son muy útiles para poder obtener el estado de salida deseado para alguna señal de salida. Una variable funciona exactamente igual que cualquier variable de cualquier lenguaje de programación de software.

Podemos decir que una señal está formada por dos partes: un valor actual y un valor futuro (o valor en proceso). El valor futuro es el que se calcula dentro del proceso y una vez que se termina el proceso, los valores futuros de todas las señales se convierten en valores actuales. Al valor futuro se le conoce como ***driver***. En VHDL para síntesis el driver nunca es afectado fuera de un proceso, fuera de éste siempre estamos modificando el valor actual.

EJEMPLO

```
ENTITY proceso IS
  PORT( x, y: IN bit;
        z1, z2, z3: OUT bit );
END proceso;

ARCHITECTURE ejemplo_proceso OF proceso IS
BEGIN
  PROCESS (x,y)
    VARIABLE z_var: bit;
    SIGNAL z_sig: bit;
  BEGIN
    z_var := '1'; -- z_var = '1'
    z_sig <= '1'; -- driver de z_sig = '1'

    z_var := x AND z_var;
                -- z_var = x AND '1'
                -- z_var = x

    z_sig <= x AND y;
         -- driver de z_sig = x AND y
    z1 <= z_var;
      -- driver de z1 = z_var = x,
      -- esto es valido porque son
      -- objetos que manejan el mismo
      -- tipo de datos

    z2 <= z_sig; -- z2 = z_sig = x AND y

END PROCESS; -- finalizado el proceso,
-- valor actual de z1 = x
-- valor actual de z2 = x AND y

z3 <= x OR y;
  -- valor actual de z3 = x OR y,
  -- en todo momento
END ejemplo_proceso;
```

Otro detalle importante en VHDL para síntesis, es que el valor actual de una señal no puede verse modificado más de una vez dentro de la arquitectura, porque las señales representan conexiones y esto equivaldría a unir dos cables. Y esto generalmente resultará en un error de compilación durante el proceso de síntesis.

EJEMPLOS

```
ARCHITECTURE no_valida1 OF senial IS
BEGIN
  z <= x AND y;
  z <= x OR y;
END no_valida1;

ARCHITECTURE no_valida2 OF senial IS
BEGIN
  PROCESS(x, y)
  BEGIN
    z <= x OR y;  -- driver de z = x OR y
  END PROCESS; -- finalizado el proceso,
  -- valor actual de z = x OR y

  z <= x AND y; -- ERROR, se vuelve a
  -- modificar el valor actual de z
END no_valida2;

ARCHITECTURE valida1 OF senial IS
BEGIN
  PROCESS(x, y)
  BEGIN
    z <= x AND y; -- driver de z = x AND y
    z <= x OR y;  -- se modifica el driver
          -- de z, driver de z = x OR y

  END PROCESS; -- finalizado el proceso,
  -- valor actual de z = x OR y

END valida1;
```

3.10.4 ESTRUCTURAS DE EJECUCIÓN SECUENCIAL

- **IF - THEN - ELSE**

SINTAXIS

```
IF condición THEN
. . .
ELSIF condición THEN
. . .
END IF;
```

EJEMPLO

```
SIGNAL conteo: unsigned(3 DOWNTO 0);
. . .
IF conteo = X"9" THEN
  conteo <= ( OTHERS => '0' ) ;
ELSE
  conteo <= conteo + 1;
END IF;
```

- **CASE - WHEN**

```
CASE expresión IS
  WHEN alternativa1 =>
    . . .
  WHEN alternativa2 =>
    . . .
  WHEN OTHERS =>
    . . .
END CASE;
```

EJEMPLO

```
TYPE estados IS (estado1, estado2,
                 estado3, estado4);
SIGNAL estado_maquina: estados;
SIGNAL motor, alarma: bit;
CONSTANT encendido: bit := '1';
CONSTANT apagado: bit := '0';
. . .
CASE estado_maquina IS
  WHEN estado0 =>
    motor <= apagado;
  WHEN estado1=>
    motor <= encendido;
  WHEN (estado3 OR estado4) =>
    alarma <= encendido;
  WHEN OTHERS =>
  motor <= apagado;
  alarma <= apagado;
END CASE;
```

- **FOR - LOOP**

```
FOR identificador IN rango LOOP
  . . .
END LOOP;
```

EJEMPLO

```
FOR i IN 3 DOWNTO 0 LOOP
  -- i es una variable y no necesita ser
  -- declarada
  IF reset ( i ) = '1' THEN
    data_out ( i ) <= '0';
  END IF;
END LOOP;
```

- **WHILE - LOOP**

```
WHILE condición LOOP
  . . .
END LOOP;
```

EJEMPLO

```
contador := 0;
resultado_tmp := 0;
WHILE contador > 0 LOOP
  contador := contador - 1;
  resultado_tmp:= resultado_tmp+data_in;
END LOOP;
    resultado <= resultado_tmp;
```

- **WAIT**

La instrucción WAIT es utilizada en procesos que no tienen una lista sensible, ya que esta instrucción define implícitamente la lista sensible del proceso. A continuación se muestran las 3 formas de utilizar la instrucción WAIT.

```
WAIT ON  -- espera los cambios de las
         -- señales especificada

WAIT UNTIL -- espera a que se cumpla la
           -- condición especificada

WAIT FOR   -- detiene la simulación
           -- durante el tiempo
           -- especificado
```

La instrucción WAIT ON no es aceptada por la mayoría de los herramientas de síntesis, WAIT FOR solo se utiliza para simulaciones. La única forma en que puede ser utilizada la instrucción WAIT en síntesis sin tener problemas es utilizándola como WAIT UNTIL.

EJEMPLO

```
PROCESS
BEGIN
  WAIT UNTIL rising_edge( clk ) ;
       -- la función rising_edge viene
       -- incluida en el paquete
       -- std_logic_1164 y
       -- equivale a utilizar:
       -- clk'event AND clk = '1'
  IF reset = '1' THEN
```

```
    y <= (OTHERS => '0');
  ELSE
    y <= y + 1;
  END IF;
END PROCESS;
```

3.11 DESCRIPCIONES ESTRUCTURALES

En el siguiente ejemplo se muestra el código para una descripción estructural correspondiente al circuito de la figura 3.11.

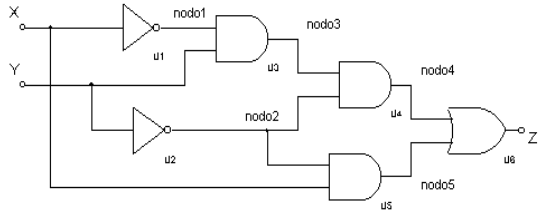


Figura 3.11 Descripciones Estructurales

```
LIBRARY mi_librería;
USE mi_libreria.compuertas.ALL;

ENTITY structural IS
  PORT ( x, y: IN bit;
         z: OUT bit);
END structural;

ARCHITECTURE estructural OF structural IS
  SIGNAL nodo1: bit;
  SIGNAL nodo2: bit;
  SIGNAL nodo3: bit;
  SIGNAL nodo4: bit;
  SIGNAL nodo5: bit;
BEGIN
  U1: not_gate PORT MAP(x, nodo1);
              -- (entrada, salida)

  U2: not_gate PORT MAP(y, nodo2) ;
              -- (entrada, salida)

  U3: and_gate PORT MAP(nodo1, y, nodo3);
            -- (entrada, entrada, salida)

  U4: and_gate PORT MAP(nodo3, nodo2,
                        nodo4 );

  U5: and_gate PORT MAP(nodo2, x, nodo5);

  U6: or_gate PORT MAP(nodo4, nodo5, z);
            -- (entrada, entrada, salida)

END estructural;
```

Esta descripción utiliza entidades descritas y compiladas previamente dentro del paquete "compuertas" de la librería "mi_librería". Una descripción estructural es similar a un netlist de PSPICE. Se declaran los componentes que se utilizan y después, mediante los nombres de los nodos, se realizan las conexiones entre compuertas.

Las descripciones estructurales son útiles cuando se trata de diseños jerárquicos. Este ejemplo pretende mostrar como son este tipo de descripciones, aunque no es una aplicación práctica utilizar este estilo con circuitos sencillos como el anterior.

3.11.1 COMPONENTES

Un componente representa a una entidad declarada en un diseño o librería, la utilización de componentes es útil en diseños jerárquicos como se mostró en el ejemplo anterior. Para poder utilizar una entidad que está dentro de otro diseño, es necesario llamar la librería y el paquete dentro del cual se encuentra esta entidad.

DECLARACIÓN DE COMPONENTES

La declaración de componentes se realiza dentro de paquetes o en la región declarativa de una arquitectura. Es preferible declarar componentes dentro de los paquetes ya que estos son reutilizables, y por esta razón sólo se verán declaración de componentes dentro de paquetes y no en arquitecturas, aunque también sea posible. A continuación se muestra la sintaxis de declaración de componentes.

SINTAXIS

```
COMPONENT identificador
  PORT( senial { , senial}: modo tipo;
        senial { , senial}: modo tipo;
        senial: { , senial}: modo tipo
      );
END COMPONENT;
```

EJEMPLO

```
COMPONENT add
  PORT ( a, b, ci: IN std_logic;
         suma, co: OUT std_logic);
END COMPONENT;
```

DECLARACIÓN DE COMPONENTES CON GENÉRICOS

```
COMPONENT identificador
  GENERICS( generico{ , generico }:
              [ modo ] tipo [ := valor ];
                . . .
              generico{ , generico }:
              [ modo ] tipo [ := valor ]
            );
  PORT( senial { , senial}: modo tipo;
```

```
          . . .
        senial: { , senial}: modo tipo
      );
END COMPONENT;
```

EJEMPLO

```
COMPONENT add_n
  GENERICS(w: integer := 8);
  PORT(a, b: IN bit_vector(w-1 DOWNTO 0);
       ci: IN bit;
       suma: OUT bit_vector(w-1 OWNTO 0 );
       co: OUT std_logic
       ) ;
END COMPONENT;
```

3.11.2 INSTANCIACIÓN DE COMPONENTES

La instanciación de componentes es una instrucción concurrente que especifica la interconexión de las señales del componente dentro del diseño en el que está siendo utilizado. Existen dos formas de hacer la instanciación de componentes: por asociación de identificadores o asociación por posición.

ASOCIACIÓN POR IDENTIFICADORES

En este tipo de instanciación es necesario utilizar el operador de asociación "=>" para indicar como se conectan los puerto del componente con lo puertos o señales de la arquitectura en la que está siendo utilizado dicho componente. Observe que en la asociación " a => b ", "a" pertenece al componente y "b" es una señal, variable o incluso una ecuación booleana en la que intervienen objetos de datos que pertenecen de la arquitectura donde se usa el componente.

```
etiqueta: identificador PORT MAP(
  puerto_componente => señal,
  puerto_componente => variable,
  puerto_componente => expresion,
  puerto_componente => OPEN,

    . . .
  puerto_componente => señal
  );
```

OPEN indica cuando un puerto de salida del componente no se conecta a nada.

EJEMPLO

```
ARCHITECTURE a_reg8 OF reg8 IS
  SIGNAL clock, reset, enable: std_logic;
  SIGNAL data_in: std_logic_vector(7
                    DOWNTO 0);
  SIGNAL data_out: std_logic_vector(7
                    DOWNTO 0);
BEGIN

  reg_1: register8 PORT MAP (
  clk => clock,
rst => reset,
en => enable,
data => data_in,
q => data_out
);

END a_reg8 ;
```

ASOCIACIÓN POR IDENTIFICADORES CON GENÉRICOS

```
etiqueta: identificador GENERIC MAP (
    identificador_generico => señal,
    identificador_generico => variable,
    identificador_generico => expresión,
    identificador_generico => OPEN
      . . .
    identificador_generico => señal
    );
  PORT MAP(
    puerto_componente => señal,
    puerto_componente => variable,
    puerto_componente => expresion,
    puerto_componente => OPEN,

      . . .
    puerto_componente => señal
    );
```

ASOCIACIÓN POR POSICIÓN

En la asociación por posición no es necesario nombrar los puertos del componente. Sólo se colocan las señales, variables, o expresiones en el lugar donde deseamos que sean conectadas. Es importante considerar el orden en el que fueron declarados los puertos del componente porque este orden es el debemos utilizar cuando se haga la instanciación del componente.

```
etiqueta: identificador PORT MAP (
           señal, variable, OPEN, señal,
           variable, OPEN, ..., señal );
```

EJEMPLO

```
ARCHITECTURE a_reg8 OF reg8 IS
  SIGNAL clock, reset, enable: std_logic;
  SIGNAL data_in: std_logic_vector(7
                    DOWNTO 0);
  SIGNAL data_out: std_logic_vector(7
                    DOWNTO 0);
BEGIN

  reg_1: register8 PORT MAP(clock, reset,
```

```
        enable, data_in, data_out);

END a_reg8;
```

ASOCIACIÓN POR POSICIÓN CON GENÉRICOS

```
etiqueta: identificador GENERIC MAP(
     señal, variable, expresión, OPEN,
     ..., señal);
     PORT MAP ( señal, variable, OPEN,
     señal, variable, OPEN, ..., señal);
```

EJEMPLO

```
ARCHITECTURE a_sum4 OF sumador4 IS
  SIGNAL carry_in, carry_out: std_logic;
  SIGNAL x, y, z: std_logic_vector(3
  DOWNTO 0);
BEGIN

-- add_n es el componente de ejemplo en
-- "declaración de componentes con
-- genéricos"

u1: add_n GENERIC MAP(4);
         PORT MAP(x, y, carry_in, z,
         carry_out);

END a_reg8;
```

3.11.3 SENTENCIAS DE GENERACIÓN

Las sentencias de generación de componentes permiten crear una o más copias de un conjunto de interconexiones, lo cual facilita el diseño de circuitos mediante descripciones estructurales.

- **FOR.. GENERATE**

Esta instrucción genera un número finito de conexiones o de instrucciones concurrentes mediante rango discreto.

SINTAXIS

```
etiqueta: FOR indice IN rango GENERATE
     { instucciones_concurrentes }
END GENERATE;
```

La etiqueta es necesaria y el índice de bucle es una variable del tipo entero que no necesita ser declarada anteriormente.

EJEMPLO

El siguiente ejemplo muestra como conectar dos arreglos de cuatro bits un tercer arreglo de ocho bits.

```
SIGNAL a, b : bit_vector(3 DOWNTO 0);
SIGNAL c    : bit_vector(7 DOWNTO 0);
SIGNAL x    : bit;
. . .
genera: FOR i IN 3 DOWNTO 0 GENERATE
  c(2*i + 1) <= a(i) NOR x;
  c(2*i)     <= b(i) NOR x;
END GENERATE;
```

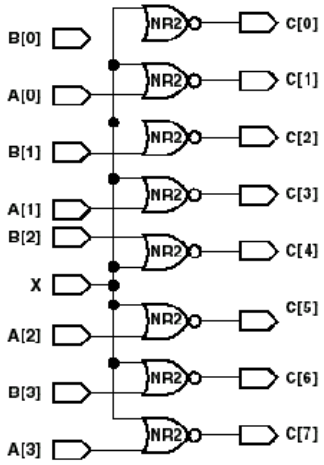
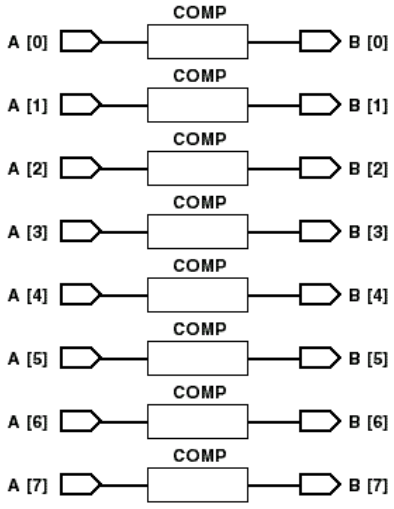


Figura 3.12 For... Generate

El uso más común de las instrucciones de generación es para crear múltiples copias de componentes y procesos. En el siguiente ejemplo se muestra como utilizar estas instrucciones con componentes y en la figura 3.13 se muestra el circuito resultante.

```
COMPONENT comp
  PORT (x :  IN bit;
        y : OUT bit);
END COMPONENT;
 . . .
SIGNAL a, b: bit_vector(0 TO 7);
. . .
gen: FOR i IN a'RANGE GENERATE
  u: comp PORT MAP (x => a(i),
                    y => b(i));
```



```
END GENERATE gen;
```

Figura 3.13 Instaniación de Componentes utilizando la inst. For... Generate

- **IF.. GENERATE**

If... Generate realiza la instrucción de instanciación o la instrucción concurrente sólo si la condición de prueba es válida.

SINTAXIS

```
etiqueta: IF condición GENERATE
     { instucciones_concurrentes }
END GENERATE;
```

El uso es similar a la instrucción secuencial IF... THEN, la diferencia es que en esta no se pueden utilizar la condiciones extras ELSE o ELSIF. El siguiente ejemplo muestra la descripción de un registro de conversión serie-paralelo de N bits. La información serial DATA es almacenada en los registros CONVERT a través de cnexiones mediante la señal S, en cada transición positiva del reloj. El circuito resultante se muestra en la figura 3.14.

```
ENTITY converter IS
  GENERIC(n: integer := 8);
  PORT(clk, data:   IN bit;
       convert: BUFFER bit_vector(n-1
                  DOWNTO 0));
END converter;

ARCHITECTURE behavior OF converter IS
  SIGNAL s: bit_vector(convert'RANGE);
BEGIN

  g: FOR i IN convert'RANGE GENERATE

    -- Desplaza el bit del registro (N-2)
    -- en el registro superior (N-1).
    -- Ya que el bit N-1 se pierde en
    -- cada transición del reloj.

    g1: IF (i = convert'LEFT) GENERATE
        PROCESS(clk, s)
        BEGIN
          IF clk'EVENT AND clk='1' THEN
            convert(i) <= s(i-1);
          END IF;
        END PROCESS;
    END GENERATE;

    -- Desplaza los bits intermedios
    -- hacia arriba

    g2: IF (i > convert'RIGHT AND
          i < convert'LEFT) GENERATE
          s(i) <= s(i-1) AND convert(i);
          PROCESS(clk, s)
          BEGIN
            IF clk'EVENT AND clk='1' THEN
              convert(i) <= s(i-1);
            END IF;
          END PROCESS;
        END GENERATE;

-- Almacena el bit de entrada DATA en el
-- primer registro

    g3: IF (i = convert'RIGHT) GENERATE
        PROCESS(clk,s)
        BEGIN
            IF clk'EVENT AND clk='1' THEN
              convert(i) <= data;
            END IF;
        END PROCESS;
        s(i) <= convert(i);
    END GENERATE;
  END GENERATE;
END behavior;
```

Como podrá observar en la figura 3.14 las retroalimentaciones en algunas ocasiones se realizaron desde la salida Q de cada registro y en otros usando la salida negada. Esto dependerá del programa de síntesis utilizado y del dispositivo empleado.

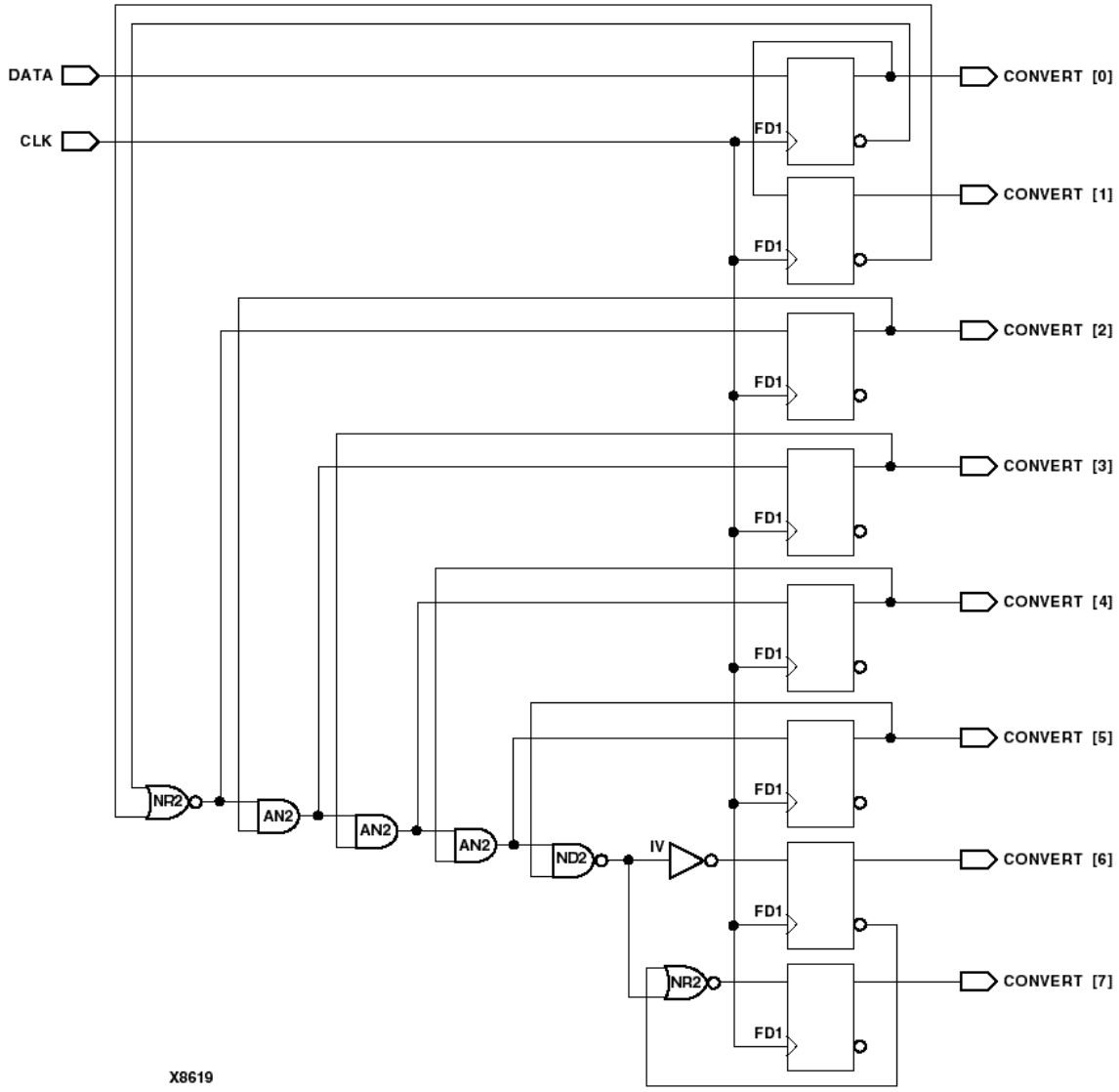


Figura 3.14 Diseño de circuitos utilizando la inst. If... Generate

3.12 SUBPROGRAMAS

Los subprogramas son secuencias independientes de instrucciones y declaraciones que pueden ser llamadas en repetidas ocasiones dentro de una arquitectura, proceso, o cuerpo de un paquete en VHDL. Existen dos tipos de subprogramas: procedimientos y funciones.

Desde el punto de vista del hardware, un llamado a un subprograma es similar a la instanciación de un componente, con la diferencia que el subprograma forma parte del circuito en el cual esta siendo utilizado. La instanciación de un componente o módulo, implica la síntesis de dos o más niveles de jerarquía en el diseño. Un subprograma sintetizado generalmente es un único circuito combinacional (utilícese un proceso si se desea crear un circuito secuencial).

Los subprogramas se declaran habitualmente en paquetes y los cuerpos de dichos subprogramas son implementados en el cuerpo del paquete en el que fueron declarados. Aunque es posible definir los subprogramas dentro de otras estructuras (arquitecturas y procesos), no es común que se haga, además, que algunos sintetizadores restringen la utilización de ellos sólo dentro de paquetes. Acerca de dichas restricciones en el uso de subprogramas, consúltese los manuales de referencia de VHDL o manuales de usuario del sintetizador que se este utilizando.

3.12.1 PROCEDIMIENTOS

Un procedimiento es un algoritmo que puede regresar uno o varios valores y que, además, puede o no tener parámetros. Estos se utilizan generalmente para descomponer grandes descripciones comportamentales en pequeñas secciones, las cuales a su vez pueden ser utilizadas por distintos procesos dentro de la descripción.

Los parámetros que se utilizan en el llamado de un procedimiento deben ser constantes, variables, o señales. Además, también debe especificarse el modo ya sea IN, OUT, o INOUT. A menos que se especifique, un parámetro se considera como una constante si se utiliza en el modo IN, y por omisión una variable si se utiliza el modo INOUT o OUT.

Los procedimientos pueden ser utilizados de manera concurrente o secuencial, es decir, ya sea fuera o dentro de un proceso. Si alguno de los parámetros es un variable, entonces el procedimiento puede ser utilizado sólo secuencialmente. Recordemos que las variables solamente pueden ser declaradas dentro procesos, procedimientos y funciones y por esto un procedimiento que utilice una variable como parámetro puede ser invocado únicamente dentro del proceso en el que se encuentra declarada dicha variable.

Una variable declarada dentro de un procedimiento existe solamente en el momento de ejecución del mismo, similar a la declaración de variables dentro de procesos.

DECLARACIÓN DE PROCEDIMIENTOS

```
PROCEDURE procedimiento ( lista de
                          parámetros );
```

CUERPO DEL PROCEDIMIENTO

```
PROCEDURE procedimiento (lista de
                  parámetros ) IS
  -- declaraciones
BEGIN
  -- instrucciones secuenciales
END procedimiento;
```

EJEMPLO

```
PACKAGE ejemplo IS
  -- declaración de procedimiento
PROCEDURE procedimiento(a: IN bit ;
                        b: INOUT bit);
END ejemplo;

PACKAGE BODY ejemplo IS
    -- cuerpo del procedimiento
  PROCEDURE procedimiento ( a: IN bit ;
                        b: INOUT bit) IS
  BEGIN
    b := a AND b ;
  END;
END ejemplo;
```

3.12.2 FUNCIONES

Una función es un algoritmo que retorna un único valor y puede o no tener parámetros de entrada. Las funciones se utilizan generalmente para:

(1) Convertir objetos de datos de un tipo a otro.

(2) Como simples funciones que realizan operaciones para las más frecuentes situaciones de diseño. Los parámetros de una función siempre son del modo IN y deben

ser señales o constantes. Además, cualquier variable declarada dentro de la función existe solamente dentro de la función.

DECLARACIÓN DE FUNCIONES

```
FUNCTION identificador (lista de
         parámetros) RETURN tipo;
```

CUERPO DE LA FUNCIÓN

```
FUNCTION identificador (lista de
         parámetros) RETURN tipo IS
  -- declaraciones
BEGIN
  -- instrucciones secuenciales
END identificador ;
```

EJEMPLO

```
FUNCTION cuenta_unos(
           vec1: std_logic_vector)
         RETURN integer IS
  VARIABLE temp: integer := 0;
BEGIN
  FOR i IN vec1'low TO vec1'high LOOP
    IF vec1 (i) = '1' THEN
      temp := temp + 1;
    END IF;
  END LOOP;
  RETURN temp;
END cuenta_unos;
```

3.12.3 LLAMADO A SUBPROGRAMAS

Como ya mencionamos un subprograma puede tener o no tener parámetro. Además, en la declaración de un subprograma se define el nombre, modo, y tipo de dato para cada parámetro. Cuando el subprograma es llamado, cada parámetro recibe un valor. El valor que recibe el parámetro (con su tipo correspondiente) puede ser el resultado de una expresión, el valor de un variable, o de una señal. El modo en el que es declarado el parámetro especifica la forma en que puede ser utilizado, similar a los puertos en una entidad.

- IN: lectura.

- OUT: escritura.

- INOUT: lectura y escritura.

Un parámetro que es declarado en el modo OUT o INOUT debe ser una variable o una señal, ya sea para tipos simples como el bit, o arreglos como el bit_vector. Cuando el subprograma es un procedimiento, puede tener múltiples parámetros que pueden utilizar los modos: IN, INOUT, o OUT. Los procedimientos son usados cuando se desea actualizar o modificar algún dato. Un ejemplo puede ser un procedimiento con un parámetro INOUT tipo bit_vector el cual invierte los bits del vector.

Si por el contrario el subprograma es una función, esta puede tener múltiples parámetros, todos del modo IN. Una vez que se ejecuta la función, esta retorna un único valor. Este valor debe ser especificado con un tipo determinado. Un ejemplo es la función ABS que regresa el valor absoluto del parámetro.

LLAMADO A PROCEDIMIENTOS

El llamado a un procedimiento se invoca por su nombre, y este utiliza los parámetros que le son listados.

SINTAXIS

```
identificador_procedimiento(
       [ identificador => ] expresión
       { , [ identificador => ] expresión})
       };
```

Cada expresión puede ser el identificador de una señal, variable, o alguna operación. Al igual que en la instanciación de componentes, la asociación de los parámetros puede ser por el nombre o por posiciones.

EJEMPLO

```
ENTITY proc_ejemplo IS
  PORT(entA: INOUT bit_vector(1 DOWNTO 0);
       entB: INOUT bit_vector(1 DOWNTO 0);
       entC: INOUT bit_vector(1 DOWNTO 0);
       sal0: INOUT bit_vector(1 DOWNTO 0);
       sal1: INOUT bit_vector(1 DOWNTO 0));
END proc_ejemplo;

ARCHITECTURE ejemplo OF proc_ejemplo IS
  PROCEDURE procedimiento(
       a: IN bit_vector(1 DOWNTO 0);
       b: IN bit_vector(1 DOWNTO 0);
       c: INOUT bit_vector(1 DOWNTO 0)) IS
  BEGIN
    c := a AND b ; -- al no especificarse
    -- como señales los parámetros INOUT
    -- son variables por omisión
  END;
BEGIN
  procedimiento(a =>( entA AND entC ),
                b => entB,
                c => sal0);
  procedimiento(entA, entC, sal1) ;
END ejemplo;
```

El siguiente ejemplo muestra un procedimiento local (declarado dentro de un proceso) llamado SWAP el cual compara y ordena dos elementos de un arreglo. El procedimiento es llamado varias veces para acomodar todos los elementos del arreglo.

```
PACKAGE data_types IS
 TYPE dat_element IS integer RANGE 0 TO 3;
 TYPE data_array IS ARRAY (1 TO 3) OF
       dat_element;
END data_types;

USE work.data_types.ALL;
ENTITY sort IS
  PORT(in_array:   IN data_array;
       out_array: OUT data_array);
END sort;
ARCHITECTURE example OF sort IS
BEGIN
  PROCESS(in_array)
    PROCEDURE swap(data: INOUT data_array;
                   low, high: IN integer)
     IS
      VARIABLE temp: data_element;
    BEGIN
      IF (data(low) > data(high)) THEN
        temp := data(low);
        data(low) := data(high);
        data(high) := temp;
      END IF;
    END swap;
    VARIABLE my_array: data_array;
  BEGIN
    my_array := in_array;
    swap(my_array, 1, 2);
    swap(my_array, 2, 3);
    swap(my_array, 1, 2);
    out_array <= my_array;
  END PROCESS;
END example;
```

LLAMADO A FUNCIONES

Una función es llamada por su nombre y utiliza los parámetros que le son dados. Las funciones regresan un único valor.

SINTAXIS

```
identificador_función (
       [ identificador => ] expresión
       { , [ identificador => ] expresión})
       };
```

Al igual que en los procedimientos es posible especificar los parámetros mediante asociación de nombres, o asociación por posiciones

EJEMPLO

```
FUNCTION invert ( a : bit ) RETURN bit IS
BEGIN
  RETURN ( not a ) ;
END;
. . .
PROCESS
  VARIABLE v1, v2, v3: bit ;
BEGIN
  v1 := '1';
  v2 := INVERT (v1) XOR 1 ;
  v3 := INVERT ('0') ;
END PROCESS;
```

INSTRUCCIÓN RETURN

La instrucción RETURN termina un subprograma. Si el subprograma es una función es necesario utilizar la instrucción RETURN, en el caso de los procedimientos es opcional. La sintaxis es la siguiente.

```
RETURN expresión; -- Funciones
RETURN; -- Procedimientos
```

En una función la *expresión* proporciona el valor de retorno de la función. Cada función debe de tener al menos una instrucción de retorno. El tipo de datos que maneja la expresión de retorno debe coincidir con el tipo de dato de retorno declarado en la función.

EJEMPLO

```
PACKAGE ejemplo_return IS
  FUNCTION ejemplo_func (a, b, c: bit)
                         RETURN bit;
END ejemplo_return;

PACKAGE BODY ejemplo_return IS
  FUNCTION ejemplo_func (a, b, c: bit)
                         RETURN bit IS
  BEGIN
    IF (c = '1') THEN
      RETURN(a XOR b);
    ELSE
      RETURN NOT(a XOR b);
    END IF;
  END ejemplo_func;
END ejemplo_return;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.ejemplo_return.ALL;
-- la librería "work" es la librería del
-- presente proyecto

ENTITY uso_funcion IS
  PORT ( SIGNAL in1, in2, in3: IN bit;
  SIGNAL valor_de_retorno: OUT bit);
END uso_funcion;
```

```
ARCHITECTURE funcion OF uso_funcion IS
BEGIN
  valor_de_retorno <= ejemplo_func(in1,
                        in2, in3);
END funcion;
```

En el siguiente ejemplo, la función OPERATE realiza las funciones AND y OR con los operandos de entrada A y B. LA operación realizada depende del valor del parámetro OPERATION. El circuito resultante se muestra en la figura 3.15.

```
FUNCTION operate(a, b, operation: bit)
                  RETURN bit IS
BEGIN
  IF (operation = '1') THEN
    RETURN (a AND b);
  ELSE
    RETURN (a OR b);
  END IF;
END operate;
```

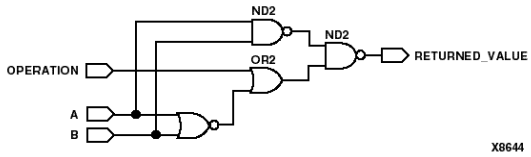


Figura 3.15 Circuito generado mediante el uso de funciones

3.12.4 SOBRECARGA DE OPERADORES

La sobrecarga de operadores consiste en definir nuevas funciones para utilizar tipos de datos con los que no estaba definido anteriormente el operador. Así, por ejemplo el operador AND no está definido de manera predeterminada para ser utilizado con los tipos std_logic_vector, unsigned y signed. Pero dentro de los paquetes std_logic_1164, numeric_std, y numeric_bit, que es donde se definen estos tipos de datos, se sobrecarga el operador AND para poder utilizarlo con estos tipos de datos. También es posible crear operadores para ser utilizados con los tipos de datos definidos por el usuario.

EJEMPLO

```
TYPE mi_bit IS ('0', '1', 'x') ;
-- tipo de datos definido por el usuario

--   Sobrecarga de los operadores AND y OR
-- para ser utilizados con el nuevo tipo
-- de datos

FUNCTION "AND" (input1, input2: IN mi_bit)
                  RETURN mi_bit;
FUNCTION "OR" (input1, input2: IN mi_bit)
                  RETURN mi_bit;
. . .
SIGNAL a, b, c: mi_bit;
. . .
c <= ( a OR b ) AND c;
```

Cuando se sobrecarga un operador en VHDL, es necesario que el nombre del operador se encuentre entre comillas dobles " " para que el programa de síntesis lo interprete como operador. Si no se hace así entonces se considera a la función como tal y no como un operador sobrecargado.

3.13 LIBRERÍAS

Una librería consiste en una colección de unidades de diseño analizadas previamente con lo cual se facilita la utilización de estas en nuevos diseño. Para incluir una librería se utiliza la siguiente sintaxis.

```
LIBRARY identificador_librería;
```

La cláusula **LIBRARY** permite utilizar la librería especificada únicamente para la unidad de diseño en la cual se declara. Una unidad de diseño es una entidad, paquete, arquitectura, o cuerpo de paquete.

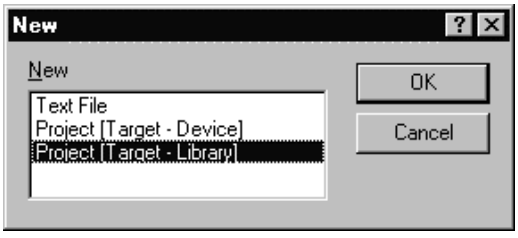
EJEMPLO

```
LIBRARY mi_libreria;
```

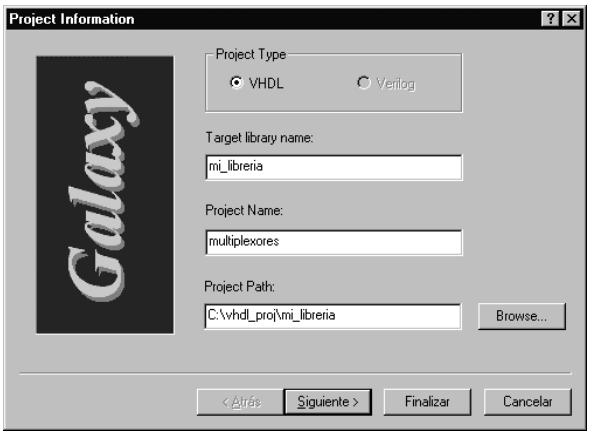
3.13.1 SÍNTESIS DE LIBRERÍAS EN WARP

Para sintetizar librerías en WARP de Cypress Semiconductors necesitas hacer lo siguiente:

1. – Dentro de Galaxy selecciona: *File > New > Project [Target - Library]*



2. – A continuación proporcionas la información del nombre de la librería, nombre del proyecto, y localización del proyecto en disco duro. Cabe mencionar que es posible crear cualquier número de proyectos dentro del mismo directorio, y todos compilando la misma librería. Por ejemplo, podemos crear un proyecto con el nombre *"multiplexores"* para compilar dentro de la librería *"mi_libreria"* en el directorio *"c:\vhdl_proj\mi_libreria"*. Diseñar todas las unidades de diseño deseadas dentro de esta librería y compilarlas en la misma. Después podemos otro proyecto con el nombre *"comparadores"* para compilar en la librería *"mi_libreria"* en el directorio *"c:\vhdl_proj\mi_libreria"*. Diseñar otras unidades de diseño y compilarlas. Cuando se incluya la librería *"mi_librería"* en otros proyectos podemos utilizar cualquier unidad de diseño que se encuentre ya sea en el proyecto de librería *"multiplexores"* o en el de *"comparadores"*. Esto es posible porque ambos proyectos se compilaron en una librería ***con el mismo nombre*** y en ***el mismo directorio***.



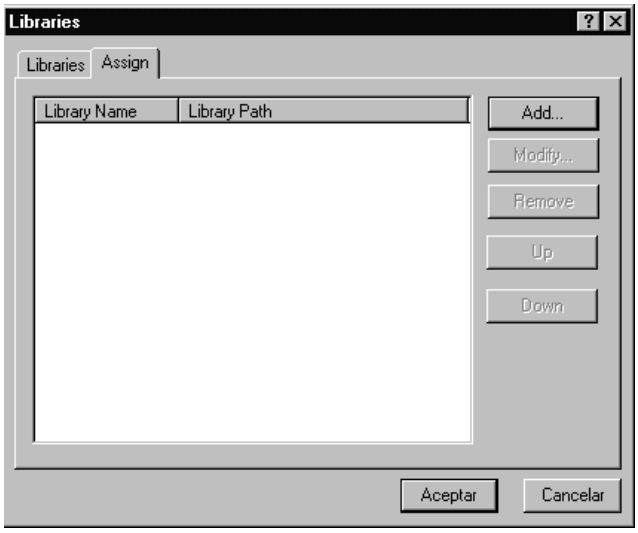
3. – Después aparece un cuadro de dialogo en el cual puedes agregar archivos *.vhd* al proyecto de librería. Si ya los tienes, puedes copiarlos al directorio o buscarlos mediante el botón *Browse...* agregarlos al proyecto. Si no los tienes sólo haz click en *Finalizar* y posteriormente podrás crear los archivos del proyecto de librería.

Como ya se menciono anteriormente, en una librería puedes incluir todas las unidades de diseño que desees, siendo unidades de diseños las estructuras: ENTITY, ARCHITECTURE, PACKAGE, o PACKAGE BODY. Por lo general en los archivos de librería se utilizan paquetes.

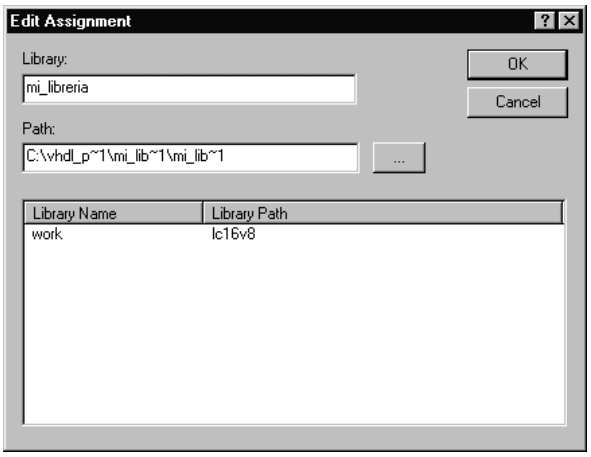
Para incluir una librería creada por el usuario en algún proyecto en particular necesitar hacer lo siguiente:

1. – Seleccionas Project > Library Manager...

2. – Dentro del cuadro de dialogo del administrador de librerías, seleccionas Assign y después haces click en el botón Add...



3. – Ahora se te pide el nombre de la librería que vas a incluir y la ruta en donde se encuentran los archivos de la librería. Por ejemplo, en *Library* podrías poner la librería del ejemplo anterior *"mi_libreria"*. Y en el *Path* escribes la ruta del directorio donde se encuentra compilada la librería. ***Debes*** escribir toda la ruta tal y como aparece en MS-DOS. Esta librería se creo en el directorio *"c:\vhdl_proj\mi_libreria"*, pero la librería se compilo en el directorio *"c:\vhdl_proj\mi_libreria\mi_libreria"* y el nombre MS-DOS del directorio es *" C:\vhdl_p~1\mi_lib~1\mi_lib~1"*.



4. – Después que agregaste la librería, para incluirla basta con que escribas lo siguiente:

```
LIBRARY mi_libreria;
```

Una vez declarada en la descripción podrás utilizar todas las unidades de diseño que se hayan compilado en la librería.

El nombre de la librería de todo proyecto que estés realizando es "*work*". Por lo que si deseas crear un paquete en particular en el proyecto puedes incluirlo en cualquier unidad de diseño de la siguiente manera.

```
USE work.identificador_paquete.ALL;
```

Donde *identificador_paquete* es el nombre del paquete que creaste en el mismo proyecto.

EJEMPLO

Crea un nuevo proyecto en WARP y en un archivo de texto copia toda la siguiente descripción.

```
PACKAGE swap IS
FUNCTION swap4(data: IN bit_vector
                     ( 3 DOWNTO 0 ) )
         RETURN bit_vector;
END swap;

PACKAGE BODY swap IS
  FUNCTION swap4 (data: IN bit_vector
                     ( 3 DOWNTO 0 ) )
    RETURN bit_vector IS
    VARIABLE tempo: bit_vector
                     ( 3 DOWNTO 0 );
  BEGIN
    tempo := data(1 DOWNTO 0)&
             data(3 DOWNTO 2);
    RETURN tempo ;
 END;
END swap ;

-- instanciación de un paquete que se
-- encuentra en el mismo proyecto
USE work.swap.ALL;

ENTITY swap_ent IS
  PORT (x:IN bit_vector(3 DOWNTO 0);
        y: OUT bit_vector(3 DOWNTO 0));
END swap_ent;

ARCHITECTURE swap_ent OF swap_ent IS
BEGIN
  y <= swap4(x) ;
END swap_ent;
```

3.13.2 PAQUETES

Un paquete en VHDL es una colección de declaraciones que pueden ser utilizadas por otras descripciones en VHDL. Un paquete en VHDL consiste de dos secciones: la declaración del paquete y el cuerpo del paquete.

Para incluir un paquete en otra descripción se sigue la siguiente sintaxis:

```
USE libreria.identificador_paquete.ALL ;
```

De esta manera el paquete indicado es visible para la unidad de diseño en la cual está siendo utilizado. Mediante "ALL" indicamos que deseamos incluir todas las declaraciones de funciones, componentes, tipos de datos, subtipos de datos, procedimientos, etc. que encuentren en dicho paquete.

DECLARACIÓN DEL PAQUETE

```
PACKAGE identificador IS
  -- declaración de subprograma
  -- declaración de tipo de datos
  -- declaración de subtipos
  -- declaración de constantes
  -- declaración de señales
  -- declaración de componentes
  -- declaración de atributos
  -- especificación de atributos
  -- instrucción USE
END identificador;
```

CUERPO DEL PAQUETE

```
PACKAGE BODY identificador IS
  -- declaración de subprograma
  -- cuerpo del subprograma
  -- declaración de tipo de datos
```

```
  -- declaración de subtipos
  -- declaración de constantes
  -- instrucción USE
END identificador;
```

En la declaración del paquete se hace mención de todo aquello que puede ser utilizado por otras descripciones cuando se incluye el paquete. El cuerpo del paquete proporciona definiciones y declaraciones adicionales, así como la descripción completa de funciones y procedimientos que fueron declarados previamente en el paquete.

EJEMPLO

```
PACKAGE v3_tbl IS
  SUBTYPE v3 IS std_logic_vector(0 TO 2);
  TYPE v3_array IS ARRAY(0 TO 7) OF v3;
  CONSTANT v3_table : v3_array := (
           "000",
           "001",
           "010",
           "011",
           "100",
           "101",
           "110",
           "111") ;
  FUNCTION int2v3 (ia: integer) RETURN v3;
END v3_tbl;
-- convierte un entero entre 0 y 7 en un
-- vector de 3 bits
PACKAGE BODY v3_tbl IS
  FUNCTION int2v3 (ia: integer) RETURN v3
  IS
  BEGIN
    RETURN v3_table(ia);
END int2v3;
END v3_tbl;
```

EJEMPLO

Para este ejemplo crea un proyecto para compilar la librería *"mi_libreria"*. Crea un nuevo archivo de texto y copia la siguiente descripción en él.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE multiplexores IS
  COMPONENT mux_2_a_1
  GENERIC(msb: integer);
    PORT( selec: IN std_logic;
          x: IN std_logic_vector
                (msb DOWNTO 0);
          y: IN std_logic_vector
                (msb DOWNTO 0);
          z: OUT std_logic_vector
                (msb DOWNTO 0));
  END COMPONENT;
END multiplexores;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux_2_a_1 IS
  GENERIC ( msb: integer := 3 ) ;
-- debe declararse siempre un valor
-- inicial para que en caso de no ser
-- especificado en el momento de la
-- instanciación, el componente tome un
-- valor por omisión
  PORT(selec: IN std_logic;
         x: IN std_logic_vector
                (msb DOWNTO 0);
         y: IN std_logic_vector
                (msb DOWNTO 0);
         z: OUT std_logic_vector
                (msb DOWNTO 0));
END mux_2_a_1;

ARCHITECTURE a_mux_2_a_1 OF mux_2_a_1 IS
BEGIN
  z <= x WHEN selec = '1' ELSE
       y WHEN selec = '0';
END a_mux_2_a_1 ;
```

Sintetiza el proyecto y después crea otro para utilizar el paquete anterior. Para agregar la librería a este nuevo proyecto hazlo desde el administrador de librerías. Crea un nuevo archivo de texto y copia la siguiente descripción en él.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE mi_libreria.multiplexores.ALL;
-- referencia al paquete multiplexores
-- que se encuentra dentro
-- de la librería "mi_libreria"

ENTITY multiplexor IS
  PORT(a,b,c,d: IN std_logic_vector
                  (3 DOWNTO 0);
       selec: IN std_logic_vector
                  (1 DOWNTO 0);
       salida: OUT std_logic_vector
                  (3 DOWNTO 0));
END multiplexor;

ARCHITECTURE estructural OF multiplexor IS
  SIGNAL salida1: std_logic_vector
                  (3 DOWNTO 0);
  SIGNAL salida2: std_logic_vector
                  (3 DOWNTO 0);
BEGIN
-- instanciación del componente mux_2_a_1
u1: mux_2_a_1 PORT MAP
              (a, b, selec(0), salida1);
u2: mux_2_a_1 PORT MAP
              (c, d, selec(0), salida2);
u3: mux_2_a_1 PORT MAP
              (salida1, salida2, selec(1),
              salida) ;
END estructural ;
```

Como se mencionó al principio una librería es una colección de unidades de diseño que pueden ser incluidas es otras descripciones mediante el llamada a la respectiva librería.