Capítulo 3

El lenguaje de descripción de hardware VHDL

VHDL es un lenguaje diseñado para describir sistemas electrónicos digitales. Surgió del programa VHSIC (Very High Speed Integrated Circuits) impulsado por Departamento de Defensa del gobierno de los Estados Unidos de América. Durante el transcurso del programa se hizo notoria la falta de un lenguaje para describir circuitos electrónicos. De esta forma se desarrolló el lenguaje VHDL (VHSIC Hardware Design Language) que fue estandarizado inicialmente en el año 1987 por el Instituto de Ingenieros Eléctricos y Electrónicos (Institute of Electrical and Electronics Engeneers, IEEE) en los Estados Unidos de América. Dicho estándar se conoce como IEEE Std 1076-1987. Posteriormente en el año 1993 se revisó este estándar originando el estándar conocido como IEEE Std 1076-1993 con algunos cambios respecto del anterior. El lenguaje VHDL está diseñado para cubrir varias necesidades que surgen durante el proceso de diseño. Permite realizar una descripción funcional o de comportamiento del circuito, utilizando técnicas procedurales y familiares de programación. Permite describir la estructura del diseño y declarar las entidades y subentidades que lo forman especificando una jerarquía entre las mismas y como son sus interconexiones. Por último permite simular el diseño y sintetizarlo con herramientas de síntesis especiales, permitiendo su manufacturado y encapsulado como un microcircuito. Un aspecto que resulta importante destacar es que gracias a este tipo de lenguajes, se elimina la fase de prototipación de componentes lo cual reduce los costos de diseño y producción.

3.1 VHDL describe comportamiento

Cuando comienza la fase de diseño de un sistema electrónico digital, resulta útil realizar una descripción funcional o de comportamiento del mismo. Una descripción VHDL usualmente está compuesta por un conjunto de entidades y subentidades, algunas de las cuales se pueden adquirir manufacturadas. De esta forma se elimina la necesidad de diseñarlas por completo. Sin embargo su comportamiento resulta necesario durante la fase de simulación del sistema. En este punto es conveniente realizar una descripción funcional del componente, que podrá ser utilizada como versión previa para simular el sistema final. La descripción funcional no hace referencia a la estructura interna del componente, que es visto como una caja negra, sino sólo se refiere a su funcionamiento.

Muchas veces la descripción funcional se divide a su vez en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones. Estas dos formas se denominan *algorítmica* y de *flujo de datos*.

3.2 VHDL describe estructura

Un sistema electrónico digital puede ser descripto como un módulo con puertos de entrada y de salida como muestra la figura 3.1. Los valores de los puertos de salida son una función de los valores de los puertos de entrada y del estado del sistema.

Figura 3.1

Una manera de describir un componente o sistema digital es a través de los componentes que lo forman. Cada componente es la instancia de alguna entidad. Sus respectivos puertos están interconectados por señales (figura 3.2). Este tipo de descripción se denomina *estructural* o *de estructura*.

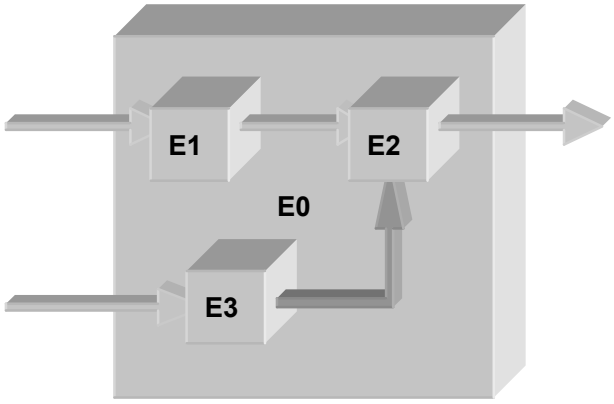

Figura 3.2

3.3 Un ejemplo de descripción en VHDL

El lenguaje VHDL presenta tres estilos de descripción que dependen del nivel de abstracción. El menos abstracto es el nivel estructural mientras que el mas abstracto y lejano a una posible implementación física es el algorítmico. A modo de ejemplo, para ilustrar cada uno de los estilos, se describirá un multiplexor de dos bits, cuya descripción se reutilizará en el trabajo final. Un posible esquema del multiplexor se muestra en la figura 3.3.

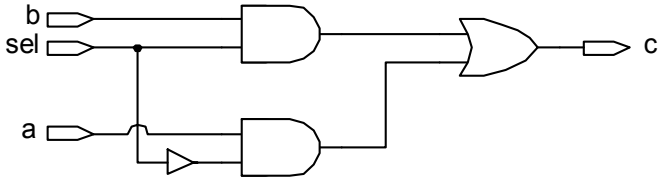

Figura 3.3

La descripción de un circuito en VHDL comienza con la definición de la entidad. Lo primero que se definen son sus entradas y salidas. Se denomina entidad porque en VHDL la palabra clave que se utiliza para su definición es `ENTITY`.

```
ENTITY Mux IS

      PORT( a:    IN bit;
            b:    IN bit;
            sel:  IN bit;

            c:    OUT bit
      );

END Mux;
```

La entidad definida con el nombre `Mux` tiene tres entradas y una salida. Las entradas y la salida son de tipo `bit`. El tipo `bit` es un tipo predefinido del lenguaje que contiene los valores '0' y '1'.

Para este ejemplo la entidad en VHDL es única, ya que se describe el mismo esquema del multiplexor, con la finalidad de ilustrar los distintos estilos, sin embargo esto no es necesario.

La entidad anteriormente declarada tiene asociados tres cuerpos descriptos con diferentes estilos. Estos cuerpos se denominan arquitecturas y se definen en VHDL mediante la palabra clave `ARCHITECTURE`. A continuación se muestran éstas arquitecturas. Cada arquitectura está descripta con un estilo distinto, comenzando por la descripción algorítmica, pasando por la descripción de flujo de datos para culminar con la descripción estructural.

En la descripción algorítmica no se hace referencia a los operadores lógicos presentes en el esquema del multiplexor, sino sólo al funcionamiento del mismo. Con una secuencia sencilla de instrucciones se puede describir el circuito. La forma en que se ejecutan las descripciones algorítmicas es secuencial, como se verá mas adelante en este capítulo, por eso se le da la denominación de algorítmica para diferenciarla de las descripciones concurrentes. La ejecución secuencial se hace posible mediante la inclusión de la palabra clave `PROCESS`. Sin embargo ésta no es la única construcción secuencial que posee VHDL, otras construcciones de naturaleza secuencial son las funciones y los procedimientos, estructuras de alto nivel que también pueden emplearse para realizar descripciones funcionales.

```
ARCHITECTURE Algoritmica OF Mux IS
BEGIN
      PROCESS(a, b, sel)
      BEGIN
            IF (sel='1') THEN
                  c<=b;
            ELSE
                  c<=a;
            END IF;
      END PROCESS;

END Algoritmica;
```

Una descripción algorítmica de mas alto nivel puede realizarse utilizando un procedimiento. Este tipo de descripción es la más lejana a una implementación física. Una descripción procedural del multiplexor puede ser de la siguiente forma

```
PROCEDURE Mux(a,b,sel:IN bit;c:OUT bit) IS
BEGIN
      IF (sel='1') THEN
            c:=b;
      ELSE
            c:=a;
      END IF;

END Mux;
```

A veces resulta interesante describir el circuito de forma que esté mas cercano a una posible implementación física. El lenguaje VHDL posee una forma de describir circuitos que permite la paralelización de instrucciones, y a su vez es más parecida a una descripción estructural pero que se clasifica como de comportamiento. Todas las sentencias que se encuentran entre el `BEGIN ... END` del cuerpo de la arquitectura se ejecutan en forma paralela y mientras dure la simulación. Esta es una característica importante frente a los lenguajes de programación convencionales, en los que las sentencias a iterar deben encerrarse dentro de un bucle. VHDL soporta la ejecución concurrente por omisión en el cuerpo de una arquitectura. El estilo que se presenta a continuación se denomina de flujo de datos.

```
ARCHITECTURE FlujoDeDatos OF Mux IS
BEGIN
      c<=(a and (not sel)) or (b and sel);

END FlujoDeDatos;
```

Una descripción estructural es la más cercana a la implementación física de un circuito y hace referencia a los componentes que intervienen y a sus interconexiones. Los componentes que se utilizan son declarados en la parte declarativa de la arquitectura e instanciados en el cuerpo de la misma las veces que sea necesario.

```
ARCHITECTURE Estructural OF Mux IS

      COMPONENT not IS

            PORT( a:    IN bit;
                  b:    OUT bit
                  );

      END COMPONENT;

      COMPONENT and IS

            PORT( a:    IN bit;
                  b:    IN bit;
                  c:    OUT bit
                  );

      END COMPONENT;
```

```
        COMPONENT or IS

              PORT( a:    IN bit;
                    b:    IN bit;
                    c:    OUT bit
                    );

        END COMPONENT;

  SIGNAL x,y,z:bit;

  BEGIN
        Not1: not PORT MAP(sel,x);
        And1: and PORT MAP(b,sel,y);
        And2: and PORT MAP(a,x,z);
        Or1: or PORT MAP(y,z,c);

  END Estructural;
```

Una descripción en VHDL puede ser mixta, es decir puede comprender varios estilos. Por ejemplo, en una descripción estructural, los elementos mas bajos de la jerarquía de diseño pueden definirse en forma de flujo de datos, debido a que el lenguaje incluye un conjunto de operadores predefinidos, como es el caso de los operadores lógicos, que pueden emplearse en la descripción.

3.4 Modelo de tiempo basado en eventos discretos

Una vez que se ha descripto un sistema, se debe realizar una simulación del mismo. Esto se lleva a cabo simulando el pasaje del tiempo en etapas discretas. En algún momento del tiempo de simulación, un módulo es estimulado cambiando los valores de sus señales de entrada. El módulo responde modificando sus señales de salida como función de su entrada a través de su comportamiento, que está caracterizado por su estado interno. Los nuevos valores para las señales de salida, serán visibles a su entorno en un paso discreto de simulación posterior. Si el nuevo valor de la señal, es distinto al que poseía previamente, se dispara un evento que estimula a otro módulo y así sucesivamente.

La simulación comienza con una fase de inicialización en la cual a las señales se les da un valor inicial, se ejecutan las descripciones de los módulos que intervienen en el diseño y el tiempo de simulación se pone a cero. Todas las señales que deben ser activadas en ese instante de tiempo son ejecutadas y simuladas, lo que ocasiona que se disparen eventos en otras señales como consecuencia de la simulación. Luego de la fase inicial, todos los módulos que se han estimulado en la fase inicial ejecutan el cuerpo de su descripción ocasionando cambios en sus señales de salida. Una vez que se ha completado un paso discreto de tiempo, la simulación se repite en el próximo paso.

El objeto de la simulación, es verificar y reunir información acerca del funcionamiento completo del sistema. Usualmente se lleva a cabo en un ambiente controlado con ciertas facilidades para inspeccionar el valor de las señales a lo largo de la simulación y en instantes de tiempo determinados.

3.5 VHDL como lenguaje de programación

El comportamiento de un módulo puede ser descripto haciendo uso de técnicas convencionales de programación que brinda el lenguaje VHDL. La sintaxis del lenguaje se ha basado en el lenguaje de programación ADA que también es un estándar definido por el Departamento de Defensa de los Estados Unidos de América. La elección de utilizar el lenguaje ADA como modelo para VHDL proviene del hecho de que ADA tiene una orientación hacia sistemas de tiempo real y hardware sin limitación de la arquitectura.

3.6 Elementos de sintaxis

3.6.1 Comentarios

Los comentarios en VHDL son cadenas de caracteres alfabéticos y números que comienzan con dos guiones '--' y se extienden hasta el final de la línea actual. Son ignorados por el lenguaje ya que su propósito es precisamente el de introducir aclaraciones.

```
-- Esto es un comentario
```

3.6.2 Identificadores

Los identificadores en VHDL están constituidos por las palabras reservadas del lenguaje y los que define el programador. Los identificadores pueden estar formados por letras, números y el carácter especial '_' (guión bajo). Deben empezar siempre con una letra y el lenguaje no es sensible a mayúsculas y minúsculas.

3.6.3 Números

Los números se pueden representar en base diez o en cualquier base que sea potencia de dos entre dos y dieciséis, sin embargo esto debe indicarse explícitamente por medio del carácter '#' (numeral). Si el número incluye un punto '.' Representa un número real, de lo contrario representa un número entero. Los dígitos de un número entero pueden separarse por medio del carácter '_'. Los números reales pueden contener un exponente al final, precedido por la letra 'E'.

Ejemplos:

```
0     6     12_345_890  47.987_9          -- Números en base diez

2.89E-8                                   -- Número real en donde el
                                          -- punto se desplaza 8
                                          -- lugares hacia la izquierda

1.77E+6                                   -- Número real en donde el
                                          -- punto se desplaza 6
                                          -- lugares hacia la derecha

2#11000010#       2#1.000_111#            -- Números en binario
4#2312#           4#23.122#               -- Números en base cuatro
16#4BAC#          16#4F.8_9#              -- Números en hexadecimal
```

3.6.4 Caracteres

Los caracteres están constituidos por un carácter ASCII encerrado entre comillas simples ’ ’.

Ejemplos:

```
'a'   'A'   '0'   '%'
```

3.6.5 Cadenas de caracteres (Strings)

Los Strings se construyen encerrando una secuencia de caracteres ASCII entre comillas dobles. Para incluir una comilla dentro del String, se deben incluir dos comillas seguidas.

Ejemplos:

```
"Esto es un String"
"Este String contiene una comilla antes del final """
"890*"
```

3.6.6 Cadenas de bits

Las cadenas de bits se utilizan para representar arreglos compuestos por los caracteres ‘0’ y ‘1’. Las cadenas de bits se encierran entre comillas dobles. Además es posible definir cadenas de bits en base dieciséis, ocho o dos precediendo el valor a codificar con las letras `X`, `O` o `B` respectivamente.

Ejemplos:

```
X"4BFF"     O"2320"     B"10011110"
```

3.6.7 Tipos de datos del lenguaje

VHDL proporciona tipos de datos básicos y formas de combinarlos para construir tipos compuestos. Los tipos básicos incluyen números, magnitudes físicas y enumerativos. Existen tipos predefinidos estándar. Los tipos compuestos que provee el lenguaje son arreglos, registros, punteros y archivos. Los tipos se declaran con la palabra clave TYPE.

3.6.7.1 Tipo entero

Una variable de tipo entero toma valores de tipo entero en un rango especificado. Un identificador declarado como entero puede asumir un valor dentro el rango –214783648 a 214783647. Los rangos pueden ser ascendentes o descendentes.

Ejemplos:

```
TYPE Byte IS RANGE 0 TO 255;        -- Declara un tipo BYTE con
                                    -- rango ascendente de 0 a
                                    -- 255
TYPE Index IS RANGE 31 DOWNTO 0;    -- Declara un tipo Index con
                                    -- rango descendente de 31 a 0
```

Existe además un tipo predefinido denominado `Integer`. El rango de este tipo viene definido por la implementación que se utilice puede variar de un compilador a otro. Por lo tanto se podría definir al tipo entero como sigue:

```
TYPE Integer IS RANGE Implementation_defined;
```

en donde el rango `Implementation_defined` va del valor entero mínimo al máximo soportado por la implementación del lenguaje.

3.6.7.2 Tipos físicos

Un tipo de datos físico es un tipo numérico que se puede utilizar para representar alguna magnitud física como por ejemplo tiempo, o tensión eléctrica. La declaración de un tipo físico incluye una magnitud denominada base y un conjunto de subunidades que son múltiplos de la unidad base.

Ejemplo:

```
TYPE Clockunits IS
      UNITS
            seg;
            min = 60 seg;
            hs = 60 min;
      END UNITS;
```

que define el tipo de la unidad de tiempo de un reloj convencional. Para asignar a un identificador declarado de este tipo, se puede especificar una unidad de tiempo completa, como por ejemplo `2 hs 30 min 0 seg`.

3.6.7.3 Tipos de punto flotante

Un tipo de punto flotante es una aproximación discreta al conjunto de los números reales en un rango determinado. Un tipo declarado flotante debe llevar siempre un punto, de lo contrario sería considerado por el lenguaje como entero. El rango máximo de números en punto flotante debe incluir al menos 6 decimales y está determinado por la implementación.

Ejemplos:

```
TYPE Real IS RANGE –1.0E38 TO 1.0E38;          -- Tipo Real predefinido
TYPE Wave IS RANGE -32768.0 TO 32767.0;
TYPE Probability IS RANGE 0.0 TO 1.0;
```

3.6.7.4 Tipo enumerativo

Un tipo enumerativo o enumerado es un conjunto ordenado de identificadores. Cada miembro de ese conjunto ordenado debe ser distinto dentro de una misma declaración.

Ejemplos:

El tipo de datos carácter `Character` es un enumerativo que incluye a todos los caracteres ASCII.

```
TYPE Bit IS ('0','1');
TYPE Boolean IS (True,False);
TYPE Hexdigit
      IS ('0','1','2','3','4','5','6','7','8','9',A,B,C,D,E,F);
```

3.6.7.5 Tipo arreglo

Un arreglo es una colección homogénea de datos indexados. Los arreglos pueden ser de una sola dimensión o varias. Además pueden ser restringidos, lo que significa que los límites se fijan en el momento de la declaración o irrestrictos en cuyo caso los límites se determinan mas adelante en tiempo de ejecución o declaración de identificadores. Un arreglo está formado por un rango discreto y un tipo de datos que indica el tipo de los elementos por los cuales está formado el arreglo.

Ejemplos:

```
TYPE Vector IS ARRAY(1 to 100) OF Integer;
TYPE Matriz_booleana
      IS ARRAY(Integer RANGE <>, Integer RANGE <>) OF Boolean;
```

En el primer caso, se define un arreglo de 100 enteros, en el segundo una matriz booleana de rango irrestricto de enteros. Por ejemplo un objeto puede ser declarado como `Matriz_Booleana` de la siguiente forma:

```
Matriz_adyacencia:Matriz_booleana(1 TO 10, 1 TO 10);
```

lo que declara una matriz de adyacencia de 10 filas por 10 columnas.

El lenguaje presenta dos tipos arreglo predefinidos irrestrictos, el tipo `String` que se declara como

```
TYPE String IS ARRAY(Natural RANGE <>) OF Character;
```

y el tipo vector de bits

```
TYPE Bit_vector IS ARRAY(Natural RANGE <>) OF Bit;
```

Un elemento de un arreglo puede ubicarse indexando el nombre del arreglo. Si se supone que `A` y `B` son arreglos de una y dos dimensiones respectivamente entonces los objetos indexados `A(3)` y `B(1,2)` se refieren a elementos indexados de los arreglos. De hecho se puede acceder a un conjunto de elementos de un arreglo utilizando la técnica que se conoce como slicing. Si A es un arreglo de 10 elementos, se pueden obtener los elementos de las posiciones 5 a 8 mediante `A(5 to 8)`. De la misma manera se pueden asignar valores a las posiciones 5 a 8 utilizando un aggregate. Si se supone que A es un arreglo declarado como

```
TYPE A IS ARRAY(1 TO 10) OF Integer;
```

y se desean asignar valores a los elementos del mismo se puede utilizar un aggregate posicional, de la forma

```
(1,4,0,0,3,3,0,0,0,0)
```

en la que los elementos están listados en el orden en que serán asignados al arreglo de izquierda a derecha. Alternativamente se puede utilizar un aggregate con posicionamiento nombrado

```
(1,4, 5 => 3,6 => 3,OTHERS => 0)
```

que tiene el mismo efecto que el caso anterior. En este caso el índice de cada elemento se da explícitamente.

3.6.7.6 Tipo registro

Un registro es una colección de datos heterogéneos. Los datos son nombrados, lo que significa que a diferencia de los arreglos se deben acceder por nombre de campo. Un campo es un identificador que se asocia con un tipo de datos determinado dentro del registro. Los registros se declaran utilizando la palabra clave `RECORD`. Para acceder a los campos de un registro se utiliza el nombre del objeto de tipo registro separado por un punto del nombre del campo.

Ejemplo:

```
TYPE Persona IS RECORD
      Nombre:String(1 TO 30);
      Edad:Natural;
END RECORD;
```

Si se desea acceder al nombre de la persona `P` se puede hacer `P.Nombre`

Para asignar valores a los registros, se pueden utilizar aggregates como ocurre con arreglos

```
("Juan Perez", 28)
```

da valores a los campos de un objeto registro `Persona`.

3.6.7.7 Tipo puntero

VHDL soporta la declaración de punteros mediante la palabra clave `ACCESS`. Estas variables pueden crearse dinámicamente, reservando memoria en tiempo de ejecución. Los punteros son una estructura de alto nivel de VHDL, por lo que se utilizan solo para realizar la descripción algorítmica del sistema y pueden ser utilizados sólo con variables.

Ejemplos:

```
TYPE PEntero IS ACCESS Integer;          -- Declara un puntero a entero
```

La creación de un puntero se hace mediante la palabra clave `NEW` y el espacio se libera mediante la sentencia `DEALLOCATE`.

Ejemplos:

```
p:PEntero;        -- Declara un puntero a Integer

p:=NEW Integer;   -- Crea el puntero dinámicamente
DEALLOCATE(p);    -- Libera el espacio
```

3.6.7.8 Tipo archivo

El tipo de datos archivo es una estructura de alto nivel de VHDL, la cual es apropiada para almacenar datos durante la simulación o realizar bancos de prueba (Test-Bench). Los archivos se almacenan en disco.

Un archivo puede almacenar cualquier tipo de datos de VHDL. La forma de declarar un archivo difiere en ambos estándares de VHDL (‘87 y ‘93), como se muestra a continuación

```
TYPE Nombre_tipo IS FILE OF Integer;           -- Común a ambos

FILE Nombre_lógico:Nombre_tipo IS [ modo ] "archivo.ext"
FILE Nombre_lógico:Nombre_tipo [ OPEN modo ] IS "archivo.ext"
```

La segunda declaración corresponde al estándar ’87, mientras que la tercera al estándar ’93.

El `Nombre_lógico` de un archivo es el nombre por el cual se lo referencia en el código. En VHDL ’87 el modo puede ser `IN` u `OUT`, indicando que se va a leer o escribir el archivo. En VHDL ’93, el modo puede ser `WRITE_MODE`, `READ_MODE` o `APPEND_MODE` y `READ_MODE` se toma por omisión.

El tipo de datos archivo como tipo de datos abstracto, posee las siguientes operaciones asociadas

```
PROCEDURE Read(VARIABLE Nombre:INOUT Nombre_tipo; Valor:OUT Integer);
PROCEDURE Write(Nombre:INOUT Nombre_tipo;Valor:IN Integer);
FUNCTION Endfile(VARIABLE Nombre:IN Nombre_tipo) RETURN Boolean;
```

3.6.7.9 Subtipos

Los subtipos son restricciones o subconjuntos de tipos existentes. Existen dos clases de subtipos. La primera clase se utiliza para restringir tipos de un tipo escalar a un rango específico. La otra clase de subtipos se utiliza para restringir el rango de un tipo arreglo irrestricto especificando valores para sus índices. Se declaran mediante la palabra clave `SUBTYPE`.

Ejemplos:

```
SUBTYPE Natural IS Integer RANGE 0 TO Highest_integer;
SUBTYPE Digits IS Character RANGE ‘0’ TO ‘9’;
SUBTYPE Mayusculas IS Character RANGE ‘A’ TO ‘Z’;

SUBTYPE Nombre IS String(1 TO 30);
SUBTYPE Bus16 IS Bit_vector(15 DOWNTO 0);
```

3.6.8 Declaración de objetos de datos

Un objeto en VHDL es un elemento nombrado que tiene asociado un valor de un tipo de datos específico. Los objetos en VHDL se clasifican en tres grupos: constantes, variables y señales.

Las constantes se declaran mediante la palabra clave `CONSTANT`, las variables mediante `VARIABLE` y las señales mediante `SIGNAL`.

Las variables son elementos secuenciales de alto nivel, por lo que pueden aparecer únicamente dentro de construcciones secuenciales o procedurales. Las variables se utilizan en descripciones algorítmicas y no se pueden mapear en hardware.

Una señal especifica una conexión entre dos entidades, módulos o componentes y representan las interconexiones entre los dispositivos físicos de un sistema. Pueden aparecer en construcciones secuenciales o concurrentes. Una señal se declara de la siguiente forma:

```
SIGNAL Nombre_señal:Tipo_datos Tipo_señal [:= Valor_omision];
```

En donde `Tipo_datos` es el conjunto de valores que transporta la señal.

Ejemplos:

```
CONSTANT PI:Real := 3.1416;
CONSTANT e:Real := 2.7182;

VARIABLE a:Integer;
VARIABLE b:Real:=0;                -- A las variables se les puede
                                   -- dar un valor inicial

SIGNAL bus:Bit_vector(31 DOWNTO 0);
```

3.6.9 Atributos

Los tipos y objetos en VHDL pueden tener información asociada denominada atributos. Existen un número predefinido de atributos asociados a un tipo determinado.

Si t es un tipo enumerativo, entero, flotante o físico se definen los siguientes atributos:

`t'left`	Límite izquierdo del tipo t
`t'right`	Límite derecho del tipo t
`t'low`	Límite inferior del tipo t
`t'high`	Límite superior del tipo t

Si `x` es un miembro de un tipo `t` como el anterior y `n` un entero, se definen los siguientes atributos:

`t'pos(x)`	Posición de `x` dentro del tipo `t`
`t'val(n)`	Elemento `n` dentro del tipo `t`
`t'leftof(x)`	Elemento que está a la izquierda de `x` en `t`
`t'rightof(x)`	Elemento que está a la derecha de `x` en `t`
`t'pred(x)`	Elemento que está delante de `x` en `t`
`t'succ(x)`	Elemento que está detrás de `x` en `t`

Si `a` es un vector y `n` es un entero que está en el rango de dimensiones de `a`, se pueden utilizar los atributos:

`a'left(n)`	Límite izquierdo del rango de dimensión `n` de `a`
`a'right(n)`	Límite derecho del rango de dimensión `n` de `a`
`a'low(n)`	Límite inferior del rango de dimensión `n` de `a`

`a'high(n)`	Límite superior del rango de dimensión `n` de `a`
`a'range(n)`	Rango del índice de dimensión `n` de `a`
`a'length(n)`	Longitud del índice de dimensión `n` de `a`

Suponiendo que s es una señal, los atributos mas usuales que se definen son:

`s'event`	Retorna `True` si se ha producido un evento en la señal `s`
`s'stable(time)`	Retorna `True` si s estuvo estable durante el período de tiempo `time`

3.6.10 Expresiones y operadores

Los operadores se utilizan para combinar expresiones formando expresiones mas complejas.

3.6.10.1 Operadores lógicos

Los operadores lógicos que incluye el lenguaje son `AND`, `OR`, `NAND`, `NOR`, `XOR` y `NOT`, y operan sobre datos de tipo `Bit`, `Bit_Vector` o `Boolean`, o arreglos de dichos tipos elemento a elemento. Para datos de tipo `Bit` o `Boolean`, los operadores binarios se comportan como *short-circuit* lo que significa que únicamente se evalúa la expresión de la derecha si la de la izquierda no alcanza para determinar el resultado.

3.6.10.2 Operadores de desplazamiento

Operan sobre datos de tipo `Bit_Vector`. Se utilizan en forma infija, es decir que el objeto a desplazar se ubica a la izquierda del operador y la cantidad a su derecha.

`SLL, SRL` Desplazamiento lógico a izquierda y a derecha. Desplazan un vector un número de bits a izquierda o derecha, ingresando un cero por el lado opuesto al sentido de desplazamiento.

`SLA, SRA` Desplazamiento aritmético a izquierda y a derecha. Efectúan el mismo desplazamiento que el caso anterior pero conservando el signo.

`ROL, ROR` Rotación a izquierda y a derecha. Realiza la rotación del operando a izquierda o derecha. Los bits que salen por un lado, ingresan nuevamente al operando por el lado opuesto.

Ejemplos:

```
Dato SLL 2;       -- Desplaza Dato dos lugares a la izquierda

Dato SRA 1;       -- Desplaza Dato un lugar hacia la derecha
                  -- conservando el signo

Dato ROR 4;       -- Rota Dato cuatro veces a la derecha
```

3.6.10.3 Operadores relacionales

Esta clase de operadores requieren datos del mismo tipo y retornan siempre un valor booleano.

`=, /=` Operador de igualdad. Compara dos datos del mismo tipo y retorna `True` si son iguales, `False` en caso contrario.

`<,<=,>,>=` Operadores de comparación. Comparan por mayor, menor, mayor o igual y menor o igual, y tienen el significado habitual.

3.6.10.4 Operador de concatenación

El operador de concatenación (`&`) opera sobre vectores. Dados dos vectores `A` y `B`, se define el vector concatenación `C := A & B` como el vector que posee como primeros elementos a los elementos del vector `A` y como últimos a los del vector `B`.

3.6.10.5 Operadores aritméticos

Los operadores aritméticos son los mismos que se presentan usualmente en todos los lenguajes de programación. Operan sobre datos de tipo numérico. Los mas comunes son: (`**`) potencia, `ABS()` valor absoluto, (`*`) multiplicación, (`/`) división, `MOD` resto módulo, `REM` resto, (`+`) suma y (`-`) resta.

3.6.11 Construcciones secuenciales

Como ocurre con los lenguajes procedurales, VHDL ofrece la posibilidad de alterar el estado de los objetos y controlar el flujo de ejecución.

3.6.11.1 Asignación a variables

El valor de una variable se puede modificar con la operación de asignación. La variable adquiere el nuevo valor inmediatamente, en el momento de la asignación. Esta es la diferencia fundamental con las señales, que adquieren su valor en el próximo instante de simulación. Una asignación tiene la siguiente sintaxis:

```
variable := expresión;
```

La variable y la expresión deben tener el mismo tipo base. La variable puede ser un aggregate, en cuyo caso las variables listadas deben ser nombres de objetos. Como primer paso se evalúan los nombres en el aggregate, luego la expresión y por último se procede a la asignación de cada componente a cada objeto.

3.6.11.2 Sentencia IF

La sentencia `IF` permite la selección de las sentencias a ejecutar, basándose en una o mas condiciones. Tiene la siguiente sintaxis:

```
IF Condición THEN
      Sentencias
[ ELSIF Condición THEN
      Sentencias ]
[ ELSE
      Sentencias ]
END IF;
```

Las condiciones se evalúan en el orden en que están listadas y cuando se verifica una, se ejecuta el bloque de sentencias correspondiente.

3.6.11.3 Sentencia CASE

La sentencia `CASE` permite la selección de las sentencias a ejecutar, basándose en una condición de selección. Su sintaxis es:

```
CASE Expresión IS
      WHEN Selección1 =>
            Sentencias
      ...

      WHEN Selección_i =>
            Sentencias

      [ WHEN OTHERS =>
            Sentencias ]
END CASE;
```

La expresión debe ser cualquier tipo discreto o subtipo de un tipo discreto. Las selecciones deben cubrir todas las alternativas posibles de valores que puede adquirir la expresión. No puede haber selecciones duplicadas.

3.6.11.4 Sentencia nula

La sentencia nula `NULL` no tiene efecto alguno. Su función es hacer explícito el hecho de que hay veces en las cuales no se requiere realizar ninguna acción. Se suele usar en sentencias `CASE`, ya que esta última exige considerar todas las alterativas y muchas veces no se debe definir alternativa para las mismas.

3.6.11.5 Aserciones (Assertions)

Una aserción se utiliza para verificar una condición determinada y reportar cuando se la viola. La sintaxis para una aserción es:

```
ASSERT Condición
      [ REPORT Expresión ]
      [ SEVERITY Expresión ];
```

Si la cláusula `REPORT` está presente, el resultado de la expresión debe ser un String. Este es un mensaje que se reporta cuando la condición es falsa. La cláusula `SEVERITY` debe ser de tipo `Severity_level` e indica el nivel de gravedad de la violación. Los posibles niveles son: `NOTE`, `WARNING`, `ERROR` y `FAILURE`. El nivel por omisión es `ERROR`.

3.6.11.6 Sentencias de bucle

VHDL presenta una estructura de bucle básico que puede ser extendido para concebir los bucles `FOR` y `WHILE` presentes en otros lenguajes. La sintaxis es:

```
Etiqueta_loop:
      [ WHILE Condición | FOR Identificador IN Rango_discreto ] LOOP
            Sentencias
      END LOOP;
```

Si se omiten las sentencia `WHILE` o `FOR` el bucle resultante se ejecuta indefinidamente.

La construcción `WHILE` ejecuta el bucle siempre y cuando la condición evalúe a `True`.

La construcción `FOR` itera el bucle la cantidad de veces como elementos hay en el rango. El identificador adquiere los valores en el rango especificado y no puede ser modificado, ya que es una constante que se genera como índice del `FOR`. Sólo puede ser leído su valor.

Existe además dos sentencias adicionales para modificar el comportamiento de un bucle. La primera de ambas sentencias es `NEXT` que termina la iteración actual y pasa a la siguiente y la segunda, `EXIT` termina el bucle actual. La sintaxis de ambas sentencias se define como

```
NEXT [ Etiqueta_loop ] [ WHEN Condición ];
EXIT [ Etiqueta_loop ] [ WHEN Condición ];
```

Si se omite la etiqueta, la sentencia se aplica al bucle mas interno, de lo contrario al nombrado en la misma.

3.6.12 Subprogramas y paquetes

VHDL provee mecanismos de encapsulamiento de código a través de funciones y procedimientos, y de código y datos por medio de paquetes.

3.6.12.1 Funciones y procedimientos

Las funciones y procedimientos en VHDL, se declaran con la siguiente sintaxis:

```
FUNCTION Nombre [ (Lista_parametros_formales) ] RETURN Tipo

PROCEDURE Nombre [ (Lista_parametros_formales) ]
```

`Lista_parametros_formales` representa una secuencia de parámetros formales separados por punto y coma. Esta forma de declaración se denomina diferida, ya que solo se especifica el encabezado y parámetros de los subprogramas y no su cuerpo. En el caso de la función, también se especifica el valor de retorno. Esta forma de declaración se utiliza en declaración de paquetes, ya que en VHDL los identificadores deben ser declarados siempre antes de ser usados.

La lista de parámetros formales puede estar compuesta por variables, señales o constantes y se les puede asignar un valor por omisión. Los parámetros en un subprograma se clasifican en tres tipos, según el sentido de transferencia de valores:

`IN`	El parámetro puede ser leído únicamente (Entrada)
`OUT`	El parámetro puede ser escrito únicamente (Salida)
`INOUT`	El parámetro puede ser leído y escrito (Entrada/Salida)

Ejemplos:

```
PROCEDURE Operacion(    VARIABLE a:IN Integer;VARIABLE b:IN Integer;
                        VARIABLE res:OUT Integer);
```

Declara el encabezado de un procedimiento que recibe dos variables enteras `a` y `b`, y retorna un resultado en la variable `res`.

```
FUNCTION Convert(       VARIABLE num:IN Real;
                        CONSTANT precision:IN Integer := 16)

                    RETURN Bit_vector;
```

Declara el encabezado de una posible función que recibe un parámetro flotante y la precisión de la conversión a realizar y retorna un vector de bits con el número convertido. Si no se ingresa parámetro actual para la precisión, se asume precisión de 16 bit.

La invocación a los subprogramas se realiza por medio del nombre del mismo con la lista de parámetros actuales adecuados. Un procedimiento se invoca como una sentencia, mientras que una función, se invoca como parte de una expresión o una expresión por si sola. La lista de asociación para los parámetros puede ser posicional, por nombre o una combinación de ambos métodos.

Si `x, y, z` son variables enteras. Para invocar al procedimiento `Operacion` puede utilizarse cualquiera de las tres formas:

```
Operacion(x,y,z);
Operacion(b=>y,res=>z,a=>x);
Operacion(x,y,res=>z);
```

Para especificar el cuerpo de un subprograma se utiliza la sintaxis:

```
FUNCTION Nombre [ (Lista_parametros_formales) ] RETURN Tipo IS
      Declaracion_objetos
BEGIN
      Sentencias
END Nombre;

PROCEDURE Nombre [ (Lista_parametros_formales) ] IS
      Declaracion_de_objetos
BEGIN
      Sentencias
END Nombre;
```

En la parte de declaración de objetos, se pueden declarar variables, constantes tipos, subtipos e incluso otros subprogramas cuya visibilidad se extiende localmente.

Cuando se invoca un subprograma, sus sentencias se ejecutan hasta que se encuentra la sentencia `END` o hasta que se encuentre una sentencia `RETURN`. La sentencia `RETURN` tiene la siguiente sintaxis:

```
RETURN [ Expresion_de_retorno ];
```

en donde `Expresion_de_retorno` es una expresión que se retorna al subprograma invocador una vez que ha sido evaluada. Si una sentencia `RETURN` se utiliza en el cuerpo de un procedimiento, no debe contener expresión de retorno.

Ejemplo:

```
FUNCTION Convert(bits:IN Bit_vector) RETURN Integer IS
      VARIABLE res:Integer:=0
BEGIN

      FOR i IN bits'range LOOP
            res:=res*2+Bit'pos(bits(i));
      END LOOP;

      RETURN res;

END Convert;
```

3.6.12.2 Sobrecarga de operadores (Overloading)

La sobrecarga de subprogramas permite a dos o mas subprogramas tener el mismo nombre, siempre y cuando el número y tipo de parámetros sean distintos en cada uno. Cuando se invoca un subprograma que está sobrecargado, se verifica que los parámetros formales, su orden y tipo concuerden con los reales provistos en la invocación. De esta manera se escoge el subprograma adecuado.

Ejemplo:

```
PROCEDURE Get(c:OUT Character);
PROCEDURE Get(i:OUT Integer);
```

La sobrecarga no solo se limita a nombres de subprogramas, sino también a operadores. Los operadores son tipos especiales de funciones que pueden ser utilizadas en forma infija. Cuando se sobrecarga un operador, el nombre del mismo se encierra entre comillas.

Ejemplo:

```
FUNCTION "+"(a,b:IN String) RETURN String IS
BEGIN
      -- Cuerpo de la función
END "+";
```

La "suma de Strings" puede definirse por ejemplo como concatenación de los mismos.

Un operador puede ser invocado de dos maneras distintas. Para el ejemplo anterior si `a, b, c` son Strings, entonces

```
c := a + b;
```

o bien

```
c := "+"(a,b);
```

3.6.12.3 Paquetes (Packages)

Un paquete en VHDL es una colección de constantes, tipos y operaciones. Normalmente se utilizan para encapsular la implementación y para declarar tipos abstractos. Un paquete se divide en dos partes: una declaración de paquete, que define su interfaz, y un cuerpo que implementa las operaciones que exporta la interfaz. El cuerpo también puede contener declaración de tipos y constantes, y puede ser obviado, en caso de que el paquete conste únicamente de constantes y tipos. El esqueleto genérico de un paquete se define como sigue:

```
PACKAGE Nombre_paquete IS

      Declaración de tipos y subtipos
      Declaración de constantes
      Declaración de encabezados de subprogramas

END Nombre_paquete;

PACKAGE BODY Nombre_paquete IS

      Declaración de tipos y subtipos
      Declaración de constantes
      Implementación de subprogramas

END Nombre_paquete;
```

Las declaraciones de tipos y constantes que aparecen en el encabezado, no necesitan aparecer nuevamente en el cuerpo ya que su visibilidad se extiende en él.

3.6.12.3 Alcance, visibilidad y utilización de los paquetes

Una vez que se ha implementado un paquete, puede ser utilizado mediante la cláusula

```
USE Nombre_paquete.Elemento;
```

El alcance y visibilidad de un paquete se extiende al módulo que contiene la cláusula `USE`. Los objetos exportados por el paquete pueden ser utilizados anteponiendo el nombre del paquete ante los mismos separado por un punto.

La cláusula `Elemento` indica que objeto del paquete nombrado se desea hacer visible en el módulo. Si se desean hacer visibles todos sus componentes, se puede usar la palabra clave `ALL`. De lo contrario se especifica el nombre del objeto.

3.6.13 Declaración de entidad

Un sistema digital se diseña como un conjunto de componentes interconectados. Cada componente tiene un conjunto de puertos que constituyen su interfaz con el entorno. En VHDL la declaración de un componente se denomina entidad y se especifica como sigue:

```
ENTITY Nombre_entidad IS

      GENERIC(Lista_genericos);
      PORT(Lista_señales);

[ BEGIN
      Sentencias ]
END Nombre_entidad;
```

La `Lista_genericos` está formada por parámetros que son pasados a la entidad en el momento de su creación. Deben ser siempre constantes y pueden incluirse varias separados por punto y coma.

La `Lista_señales` la constituyen los puertos que ofician de interfaz con el exterior. Debe incluir únicamente señales las cuales pueden ser de cualquiera de los tres modos `IN`, `OUT` o `INOUT`.

La forma general de declaración de los parámetros es:

```
Nombre_parametro : Modo Tipo;
```

El bloque opcional de sentencias, se ejecuta en el momento que se instancia la entidad y puede utilizarse para verificar por ejemplo si los parámetros genéricos son correctos.

Ejemplo:

```
ENTITY ROM IS

      GENERIC(width,size:Natural:=16);
      PORT( ena:IN Bit;
            addr:IN Bit_Vector(size-1 DOWNTO 0);
            data:OUT Bit_Vector(width-1 DOWNTO 0)
            );

END ROM;
```

Declara la entidad ROM especificando el ancho de palabra y la cantidad de direcciones como parámetros genéricos.

3.6.14 Declaración de arquitectura

Con la declaración de entidad, se especifica únicamente la interfaz del módulo. El comportamiento del módulo puede ser descripto de varias formas en los cuerpos de la arquitectura. Cada arquitectura representa una vista distinta de la entidad. Se puede realizar la descripción de una arquitectura de manera puramente funcional, utilizando técnicas de programación convencionales. De otro modo se puede implementar una arquitectura como una colección de componentes interconectados de manera estructural.

Un cuerpo de arquitectura se especifica como:

```
ARCHITECTURE Nombre_arquitectura OF Nombre_entidad IS

      Declaración de tipos y subtipos
      Declaración de subprogramas
      Declaración de componentes
      Declaración de variables, señales, constantes

BEGIN

      Sentencias concurrentes
      Bloques concurrentes
      Procesos

END Nombre_arquitectura;
```

3.6.14.1 Bloques

Las unidades dentro de una arquitectura pueden ser descriptas como bloques, que están conectados a otros bloques por medio de señales.

La forma de declarar un bloque es la siguiente:

```
Etiqueta_bloque:
            BLOCK
                  GENERIC(Lista_genericos);
                  GENERIC MAP(Lista_map_genericos);
                  PORT(Lista_señales);
                  PORT MAP(Lista_map_señales);
            BEGIN
                  Sentencias concurrentes
                  Bloques concurrentes
                  Procesos
            END BLOCK [Etiqueta_bloque];
```

`Lista_genericos` y `Lista_señales`, son las constantes genéricas y señales que exporta el bloque.

`Lista_map_genericos` y `Lista_map_señales` son las asociaciones (genéricos y señales) con los que se conecta el bloque al exterior.

3.6.14.2 Declaración de componentes

Una arquitectura puede hacer uso de entidades descriptas en otras librerías. Para ello debe declarar un componente que luego será instanciado en el momento que se emplee. La sintaxis a la que responde la declaración de componente es:

```
COMPONENT Nombre_componente IS

      GENERIC(Lista_genericos_locales);
      PORT(Lista_señales_locales);

END COMPONENT;
```

La declaración de componente debe corresponderse con la declaración de una entidad en la librería.

Por ejemplo para el caso de la memoria ROM ejemplificada anteriormente, la declaración de componente para esa entidad, sería

```
COMPONENT ROM IS

      GENERIC(width,size:Natural:=16);
      PORT( ena:IN Bit;
            addr:IN Bit_Vector(size-1 DOWNTO 0);
            data:OUT Bit_Vector(width-1 DOWNTO 0)
            );

END COMPONENT;
```

3.6.14.3 Instanciación de componentes

Un componente declarado se debe instanciar para su uso. La forma de instanciar un componente se realiza mediante la palabra clave MAP.

```
Etiqueta_componente:Nombre_componente
                  [ GENERIC MAP(Lista_genericos_actuales) ]
                  [ PORT MAP(Lista_señales_actuales) ];
```

en donde `Lista_genericos_actuales` es una lista de constantes (parámetros actuales) pasadas al componente y `Lista_señales_actuales` es la lista de señales por las cuales el componente se conecta en la arquitectura donde está siendo instanciado. La asociación de parámetros actuales puede realizarse de manera posicional o nombrada como ocurría en el caso de los subprogramas.

A modo de ejemplo, si se desean instanciar dos memorias ROM como las declaradas anteriormente se procede de la siguiente manera:

```
Mem1: ROM GENERIC MAP(8,1024) PORT MAP(ena,addr1,data1);
Mem2: ROM GENERIC MAP(16,1024) PORT MAP(ena,addr2,data2);
```

En este caso se instancia a la ROM dos veces. En el primer caso se trata de una ROM de 1204 bytes y en el segundo caso de una ROM de 1024 words.

3.6.15 Asignación a las señales

La asignación a una señal determina una o mas transacciones sobre la señal. Se realiza de la siguiente manera:

```
Señal_destino <= [ TRANSPORT ] Valor [ AFTER Medida_de_tiempo Unidad ];
```

La forma mas simple de asignación, es si se omiten la palabra clave `TRANSPORT` y `AFTER`, con lo que la señal adquiere su valor en el próximo paso de simulación. Si se desea que una señal adquiera el valor asignado luego de un período posterior a la asignación se utiliza la palabra clave `AFTER`. A modo de ejemplo sea `sen` una señal de tipo `Bit_vector` que debe adquirir los valores ‘0’ luego de 5 nanosegundos,

```
sen <= '0' AFTER 5 ns;        -- Adquiere el valor '0' luego de 5 ns
```

Se pueden emplear diversas unidades de tiempo. La unidad mas pequeña es el femtosegundo (fs). Si la asignación anterior es ejecutada a los 15 ns de tiempo de simulación, la señal `sen` tomará su nuevo valor a los 20 ns. Si ahora se realiza otra asignación con la siguiente forma a la señal `sen`,

```
sen <= '1' AFTER 10 ns, '0' AFTER 15 ns;
```

se agregan las transacciones sobre la señal, provocando que la misma cambie cada 5 ns en un período de 15 ns. Si se ejecuta una asignación a una señal y existen transacciones posteriores en el tiempo que aún no han surtido efecto, son eliminadas y las nuevas colocadas en su lugar. Esta última característica en la asignación puede ser modificada incluyendo la palabra clave `TRANSPORT`. Si se omite la palabra clave `TRANSPORT`, se asume que el modo es `INERTIAL`. Por lo tanto,

TRANSPORT: si se incluye esta palabra clave, se dice que la asignación tiene *transport delay*, en cuyo caso todas las transacciones anteriores que deban ser llevadas a cabo luego de la transacción nueva son eliminadas antes de establecer la nueva.

INERTIAL: si se incluye la palabra clave `INERTIAL` o no se incluye palabra clave alguna, se dice que la asignación tiene *inertial delay*. En este caso, todas las transacciones que deban ser llevadas a cabo luego de la transacción nueva son eliminadas antes de establecer la nueva, como ocurre en el caso de transport. Seguidamente, se examinan todas las transacciones anteriores cuyo efecto deba producirse luego del efecto de las nuevas. Si existe alguna con un valor diferente a la nueva transacción, se eliminan todas las anteriores a la que posee el valor distinto. Las restantes, permanecen.

3.6.16 Ejecución secuencial: Procesos y la sentencia WAIT

Como se mencionó anteriormente, para expresar el comportamiento de un sistema digital se puede utilizar la forma de descripción funcional o de comportamiento. La unidad principal para describir a un sistema funcionalmente es el proceso (process). El proceso es un módulo secuencial de código que puede ser activado conforme ocurran eventos sobre señales determinadas. Cuando mas de un proceso se activa al mismo tiempo, éstos ejecutan en forma concurrente. La forma de especificar un proceso es la siguiente:

```
[ Nombre_proceso: ]
      PROCESS [ (Lista_sensibilidad) ]
            Declaración de tipos y subtipos
            Declaración de subprogramas
            Declaración de componentes
            Declaración de variables, señales, constantes
      BEGIN
            Sentencias secuenciales
            [ Sentencia WAIT ]
      END PROCESS;
```

Un proceso es el único lugar junto con las funciones y los procedimientos dentro de un programa en VHDL donde se pueden declarar y utilizar variables. Generalmente las variables se utilizan para almacenar estados en un modelo. Dentro de un proceso puede haber cualquier cantidad de asignación a señales. Las asignaciones a las señales determinar un *driver* para esa señal. Una señal puede poseer a lo sumo un *driver*, de lo contrario provocaría un problema de resolución.

Al iniciarse la simulación, se ejecuta el cuerpo del proceso. Una vez que se ha ejecutado el cuerpo, el proceso se detiene para comenzar nuevamente. Un proceso puede ser suspendido por medio de una sentencia `WAIT`, que tiene la forma:

```
WAIT  [ ON Nombre_de_señal ]
      [ UNTIL Condición_de_espera ]
      [ FOR Tiempo_de_espera ]
```

La lista de sensibilidad que acompaña a la sentencia `WAIT`, especifica una lista de señales `Nombre_de_señal` a las cuales el proceso es sensible mientras está suspendido, en efecto la ejecución continuará si se produce un evento sobre alguna de las mismas y la condición de espera es verdadera. La expresión de `Tiempo_de_espera` indica el tiempo que el proceso permanecerá suspendido, si es obviada el proceso esperará indefinidamente sobre la lista de señales. Se puede agregar una condición que determina que el proceso espere mientras ésta sea verdadera.

Alternativamente se puede expresar la lista de sensibilidad al comienzo del proceso, que tiene el mismo efecto que colocar una sentencia `WAIT` al final del mismo. Si se encuentra presenta la lista de sensibilidad al comienzo del proceso, el mismo no puede incluir sentencias `WAIT` adicionales.

Ejemplo:

```
PROCESS (reset, clock)
      VARIABLE state:Boolean:=False;
BEGIN
      IF reset THEN
            state:=False;
      ELSIF clock THEN
            state:=NOT state;
      END IF;

      q<=state AFTER 1 ns;

END PROCESS;
```

Al comenzar la simulación, el cuerpo del proceso es ejecutado, luego de lo cual se suspende aguardando algún cambio en alguna señal de su lista de sensibilidad. Cuando se produce algún cambio, se vuelve a ejecutar el cuerpo del proceso. Si cambió la señal de `reset` de `False` a `True`, el estado `False` se asigna a la señal de salida `q` cada 1 ns; si `reset` es `False` y `clock` es `True`, cambia el estado entre `True` y `False`, y se asigna a la señal de salida `q` cada 1 ns.

El siguiente ejemplo muestra una señal q que adquiere valores ‘1’ y ‘0’ alternadamente cada 2 ns

```
PROCESS
BEGIN

      WAIT FOR 2 ns;
      q<='1';

      WAIT FOR 2 ns;
      q<='0';

END PROCESS;
```

3.6.17 Asignación concurrente a señales

VHDL provee una notación denominada asignación concurrente a señales, que puede ser utilizada para expresar procesos que describen un driver para una señal. La asignación puede ser condicional o selectiva.

La asignación condicional es una abreviación para procesos que contienen asignaciones a señales dentro de sentencias `IF`. Por lo tanto el proceso:

```
PROCESS (clk)
BEGIN
      IF clk='0' THEN
            q<=False;
      ELSE
            q<=True;
      END IF;
END PROCESS;
```

puede expresarse con una asignación condicional de la siguiente manera:

```
q<=False WHEN clk='0' ELSE True;
```

Formalmente, la sentencia de asignación condicional tiene la sintaxis:

```
Nombre_señal<=valor1 WHEN Condición ELSE valor2;
```

La asignación selectiva se utiliza como abreviación de un proceso que tiene asignación a una señal dentro de un bloque `CASE`. Por lo tanto un proceso descripto como:

```
PROCESS
BEGIN
      CASE Expresión IS
            WHEN Caso_1 =>
                  q<=Valor_1;
            WHEN Caso_2 =>
                  q<=Valor_2;

            ...
            WHEN Caso_n =>
                  q<=Valor_n;
      END CASE;

      WAIT [Cláusula_de_sensiblidad]

END PROCESS;
```

posee una notación abreviada equivalente:

```
WITH Expresión SELECT
      q<=valor_1 WHEN caso_1,
            Valor_2 WHEN Caso_2,
            ...
            Valor_n WHEN Caso_n;
```

3.6.18 Unidades y bibliotecas

Las descripciones en VHDL se almacenan en archivos de diseño, que son archivos de texto que contienen las sentencias que conforman los distintos módulos que integran el sistema. Usualmente se almacenan con la extensión .VHD

El compilador es quien se encarga de analizarlas, compilarlas y incluirlas en un archivo de biblioteca, también llamado librería. Cada módulo se incluye por separado y se denomina unidad de biblioteca. Las unidades de biblioteca primarias son las declaraciones de entidad, declaraciones de paquete y declaraciones de configuración. Las unidades de biblioteca secundarias están integradas por las arquitecturas y cuerpos de paquete. Se denominan secundarias, ya que su especificación depende de la declaración de las unidades primarias.

Las unidades de biblioteca se agrupan en bibliotecas que se almacenan como archivos separados con un nombre. Para utilizar las unidades de una biblioteca determinada, se debe especificar su inclusión de la siguiente forma:

```
LIBRARY Nombre_de_biblioteca;
USE Nombre_de_biblioteca.[ ALL ];
```

en donde la cláusula `LIBRARY` indica al compilador VHDL que incluya el archivo de biblioteca con `Nombre_de_biblioteca` en el diseño. La cláusula `USE` especifica que componentes de `Nombre_de_biblioteca` desean utilizarse. Si se indica la palabra clave `ALL`, se asume la inclusión de todos los componentes, en caso contrario se debe utilizar `Nombre_de_biblioteca` seguido del nombre del componente a usar. Por ejemplo,

```
LIBRARY Memory;
USE Memory.ROM;
```

incluye la biblioteca `Memory` en el diseño y hace visible para su utilización únicamente a la unidad `ROM`. La visibilidad de un componente se extiende a la unidad de diseño en donde se lo nombra.

Existen dos bibliotecas que son visibles a todos los diseños y no deben ser incluidas explícitamente. La primera es la biblioteca `WORK`, en donde el compilador incluye el diseño actual. La segunda biblioteca es `STD`, que contiene la funciones y tipos predefinidos por el lenguaje.

3.6.19 La sentencia GENERATE

La sentencia `GENERATE` provee la capacidad de describir estructuras regulares como vectores de bloques, procesos o instancias de componentes dentro del cuerpo de una arquitectura. Su sintaxis es:

```
Etiqueta_generate:
      FOR Rango_generate GENERATE
            [ IF Condición GENERATE
                  Instancia_de_componente
                  Bloque
                  Proceso
            END GENERATE ]

            Instancia_de_componente
            Bloque
            Proceso
      END GENERATE;
```

El `Rango_generate` describe la cantidad de veces que el esquema se va a generar y puede depender de una variable que puede ser utilizada en los esquemas. La sentencia `IF` se utiliza para casos excepcionales en los patrones. El siguiente ejemplo muestra el cuerpo de un sumador binario, utilizando la sentencia `GENERATE`:

```
FOR i IN sum'left DOWNTO 0 GENERATE
LowBit:
      IF i=0 GENERATE
            FA: FullAdder PORT MAP(a2(0),b2(0),cIn,sum(0),c(0));
      END GENERATE;

OtherBits:
      IF i/=0 GENERATE
            FA: FullAdder PORT MAP(a2(i),b2(i),c(i-1),sum(i),c(i));
      END GENERATE;
END GENERATE;
```

Ambos esquemas son similares salvo por el carry de entrada que es una señal proveniente del exterior y debe sumarse únicamente al bit menos significativo. En los casos restantes, cada `FullAdder` se conecta con el siguiente.

3.6.20 La Unidad de configuración

3.6.20.1 Especificación de configuración

Un componente no es mas que una interfaz sin funcionalidad asociada. Por ello se lo debe enlazar con una entidad y arquitectura. Esto se debe a que un mismo componente puede tener mas de una arquitectura relacionada.

La forma de declarar este enlace entre un componente y una entidad se consigue mediante la sentencia FOR. La sintaxis para el caso de un componente es la siguiente

```
FOR Lista_de_referencias : Nombre_de_componente
      [ USE ENTITY Nombre_de_entidad (Nombre_de_arquitectura) ]
      [ USE CONFIGURATION Nombre_de_configuración ]
      [ GENERIC MAP (Parámetros_genericos) ]
      [ PORT MAP (Puertos) ];
```

La `Lista_de_referencias` es una lista separada por comas de las referencias realizadas del componente `Nombre_de_componente`, es decir de las entidades y arquitecturas que se deseen utilizar como descripción del componente. En esta lista se pueden utilizar también las palabras clave `ALL` y `OTHERS` para seleccionar todas las referencias hechas del componente, o todas las que

queden por enlazar con una entidad. Normalmente todas las referencias de un componente están enlazadas con la misma entidad y arquitectura, pero de todos modos hay ocasiones en las que resulta útil utilizar otra entidad o arquitectura diferente para algún componente crítico.

Luego de la sentencia `FOR` se utiliza la cláusula `USE`, que permite enlazar con una entidad seguida con la arquitectura correspondiente entre paréntesis. Opcionalmente se puede enlazar con un bloque de configuración el cual posee información sobre la entidad a enlazar.

La sintaxis de enlazado entre componente y entidad también depende de dónde se coloque la sentencia. La sentencia anteriormente descripta se ubica en la parte declarativa de la arquitectura y se usa para enlazar componentes con sus respectivas referencias.

3.6.20.2 Declaración de configuración

La declaración de configuración se realiza en un bloque especial de VHDL denominado *Unidad de Configuración*. En este bloque se pueden especificar los mismos enlaces de componentes con entidades como ocurre en la especificación de configuración. La forma general de una declaración de configuración es la siguiente

```
CONFIGURATION Nombre_de_configuración OF Nombre_de_entidad IS
      [ Sentencia_use ]
      [ Definición_de_atributo ]
      [ Definición_de_grupos ]
      Bloque_de_configuración
END Nombre_de_configuración;
```

La unidad de configuración está siempre asociada a una entidad. Todo el contenido de una configuración se refiere a la entidad indicada en `Nombre_de_entidad`.

La definición de grupos se utiliza para definir una colección de elementos mediante la palabra clave `GROUP`. Se utiliza normalmente para pasar información acerca de una entidad a la herramienta de síntesis del circuito si ésta lo requiere.

El `Bloque_de_configuración` es la única parte obligatoria dentro del cuerpo de una unidad de configuración. Se refiere a las posibles estructuras presentes en una arquitectura. Dichas estructuras se organizan en tres niveles; los elementos que se encuentran en la propia arquitectura, los elementos que se encuentran en bloques y los elementos dentro de sentencias `GENERATE`. Como éstos dos últimos se estructuran en niveles, la configuración, también. La forma general del bloque de configuración es como sigue

```
FOR Especificación_de_bloque
      [ Sentencia_use ]
      [ Bloque_de_configuración ]
      [ Configuración_de_componente ]
END FOR;
```

Hay tres elementos que pueden constituir la `Especificación_de_bloque`. Usualmente se agrega el nombre de una arquitectura, pero también puede ser el identificador de un bloque `BLOCK` o el de un bloque `GENERATE`. Como ocurre en el caso de la configuración, también se pueden incluir cláusulas USE. Por último se puede volver a definir un `Bloque_de_configuración` debido a la naturaleza jerárquica de esta unidad. Un posible ejemplo para una unidad de configuración podría ser el siguiente

```
CONFIGURATION OperConf OF Oper IS
      FOR FuncionalOper
        FOR Bloque1
          FOR c1: y2 USE ENTITY Gates.And2(Functional);
          END FOR;
          FOR  genear
            FOR c2: Suma USE CONFIGURATION Operadores.SumaConf;
            END FOR;
          END FOR;
        END FOR;
      END FOR;
END OperConf;
```

Para escoger la arquitectura funcional para la entidad `Suma`

```
CONFIGURATION SumaConf OF Suma IS
      FOR Funcional;
      END FOR;
END SumaFuncional;
```

Si el diseño es complejo, conviene realizar todos los enlaces entre componentes y entidades en un bloque de configuración. Las ventajas que presente este enfoque frente a la especificación de la configuración, es que se puede expresar jerarquía y además brinda la posibilidad de incluir la configuración en un archivo para tal fin, evitando editar las distintas arquitecturas cada vez que se desee hacer algún cambio.

3.6.21 Buses y resolución de señales

Como se dijo anteriormente cada señal debe tener una sola fuente, sin embargo en muchos casos es necesario conectar diversas fuentes a la misma señal. Esto puede traer aparejado una serie de problemas ya que si mas de una fuente modifica la señal al mismo tiempo debe existir un modo de arbitrar cual de todos los valores debe adquirir la señal. Esta clase de problemas se denomina resolución de señales. Para manejar este tipo de situaciones VHDL introduce un nuevo tipo de datos para las señales denominado tipo resuelto.

Un tipo resuelto incluye en su definición una función de resolución que toma todas las fuentes para la misma señal para determinar el valor final de la misma. Un tipo resuelto se declara como un subtipo, respetando la sintaxis:

```
SUBTYPE Nombre_tipo_resuelto IS Nombre_función Nombre_tipo_fuentes;
```

El `Nombre_de_funcion` es el nombre de una función de resolución declarada previamente que debe tomar como argumento un vector irrestricto del tipo de las señales a resolver y debe retornar una señal de ese tipo. A modo de ejemplo se introduce lo siguiente:

```
TYPE Logic_tristate IS ('0','1',Z);
TYPE Logic_tristate_array IS ARRAY (Integer RANGE <>)
      OF Logic_tristate;

FUNCTION Resolved(sen:IN Logic_tristate_array)
      RETURN Logic_tristate;

SUBTYPE Res_logic_tristate IS Resolved logic_tristate;
```

```
FUNCTION Resolved(sen:IN Logic_tristate_array)
      RETURN Logic_tristate IS
BEGIN
      FOR i IN sen'range LOOP
            IF sen(i)='0' THEN
                  RETURN '0';
            END IF;
      END LOOP;

      RETURN '1';
END Resolved;
```

El tipo triestado representa tres posibles valores, '0', '1' y Z (alta impedancia). El tipo `Res_logic_tristate` corresponde al tipo resuelto. La función utilizada para resolver retorna '0' si alguna señal es '0', en caso contrario retorna '1', ya que todas las señales son '1' o Z. Este tipo de resolución se denomina AND-cableado que será '0' si al menos uno lo es.

3.6.22 Aserciones concurrentes

Una aserción concurrente es análogo a un proceso que contiene una aserción seguida de una sentencia `WAIT`. Por ejemplo:

```
ASRT: ASSERT Condición REPORT Mensaje SEVERITY Nivel_severidad;
```

es equivalente a:

```
ASRT: PROCESS
      BEGIN
            ASSERT Condición REPORT Mensaje SEVERITY Nivel_severidad;
            WAIT ON Lista_sensibilidad;
      END PROCESS;
```

En este caso la lista de sensibilidad debe contener a todas las señales que intervienen en la condición. Si no se incluyen señales en la lista, el proceso se activa durante la fase de inicialización y se suspende para siempre.

3.6.23 Invocación a procedimiento concurrente

Una invocación a procedimiento concurrente es equivalente a un proceso que contiene únicamente un llamado a procedimiento seguido de una sentencia `WAIT`. Un procedimiento llamado de esta manera no puede tener parámetros formales de tipo variable, ya que las variables tienen sentido únicamente en entornos secuenciales. La lista de sensibilidad del proceso debe incluir las señales que son parámetros de modo `IN` o `INOUT` del procedimiento.

Por ejemplo el llamado:

```
PRO: Nombre_paquete.operacion(p1,p2,p3);
```

es equivalente a:

```
PRO:  PROCESS
      BEGIN
            Nombre_paquete.operacion(p1,p2,p3);
            WAIT ON p1,p2,p3;
      END PROCESS;
```

3.6.24 Transacciones nulas

Una transacción nula especifica la desconexión de una fuente de una señal. Cuando hay varias fuentes conectadas a una misma señal (en efecto, se trata de una señal resuelta) hay circunstancias en las cuales un dispositivo no debe emitir un valor. Su sintaxis es:

```
señal <= NULL AFTER Tiempo;
```

3.7 Descripción de bancos de prueba

Una de las partes mas costosas en tiempo de diseño de sistemas digitales es la verificación de su funcionamiento. Con las metodologías de diseño tradicionales, esto no se podía comprobar hasta que el dispositivo era implementado. Esto traía como resultado que los fallos se tornen insostenibles en algunos casos. El lenguaje VHDL, permite acelerar el proceso de verificación, brindando la posibilidad de llevar a cabo una simulación exhaustiva del modelo, que si es correcto, puede ser implementado directamente. La forma mas directa de llevar a cabo la simulación de un modelo sencillo es modificar las entradas y observar las salidas. Para modelos complejos esta forma de simulación podría resultar poco flexible, por eso VHDL incluye la posibilidad de definir un banco de pruebas.

Un banco de pruebas consiste en un conjunto de entradas denominadas patrones de prueba. Con el lenguaje VHDL se puede modelar el banco de pruebas independientemente de la herramienta de simulación. El banco de pruebas puede ser utilizado en cualquier fase de simulación y para cualquier nivel de abstracción de una definición, posibilitando un ahorro en tiempo y esfuerzo.

Un banco de pruebas es una entidad simple sin puertos de entrada y salida, y su arquitectura tiene como señales internas las entradas y salidas del circuito. El único componente es el correspondiente a la entidad que se va a simular.

La construcción de un banco de pruebas puede realizarse mediante tres métodos, clasificados según como se generen los patrones de prueba.

3.7.1 Método tabular

El método tabular consiste en elaborar una tabla con entradas y la respuesta prevista del circuito. Esta tabla usualmente se almacena en un vector y forma parte del código VHDL en donde está declarado el componente ligado a la entidad que se pretende simular. Se verifica que la respuesta del circuito coincida con la respuesta prevista almacenada previamente en el vector.

3.7.2 Utilización de archivos

El método tabular es independiente del modelo a ser simulado. El modelo y el banco de pruebas pueden estar en archivos distintos, aportando modularidad y portabilidad. El mismo banco de pruebas puede usarse para simular descripciones a distintos niveles de abstracción del componente.

Para simulaciones más costosas o para tener una mejor organización en el banco de pruebas, puede ser apropiado separar los vectores de prueba del mecanismo de simulación. La ventaja de la separación es que el banco de pruebas se independiza aún mas del componente a verificar, ya que si se realizaran modificaciones sobre los patrones no habrá peligro de modificar accidentalmente el componente simulado. El lenguaje VHDL soporta esta separación mediante el uso de archivos y sus operaciones de entrada/salida asociadas.

El acceso a archivos no solo se limita a la lectura de patrones de prueba, sino también al almacenamiento de los resultados de la simulación en un archivo. Esto último puede resultar útil en caso de que se deseen comparar dos archivos con resultados para validar el funcionamiento de un circuito.

La estructura del código que describe el banco de pruebas es similar al caso tabular. La entidad no tiene entradas ni salidas y referencia únicamente al componente que se desea simular.

3.7.3 Utilización de un algoritmo

Los métodos descriptos anteriormente son apropiados para patrones poco extensos. La desventaja que presenta el método que utiliza archivos, es que los accesos al disco incrementan el tiempo, retardando la simulación.

Para salvar estos problemas se puede elaborar un banco de pruebas en donde los patrones se generen en forma algorítmica, es decir, los vectores se generan por medio de un algoritmo o una fórmula matemática. Un ejemplo interesante podría ser la verificación de un circuito de suma. Las sumas pueden realizarse mediante un algoritmo descripto en VHDL en forma funcional y ser comparadas en tiempo de ejecución con el resultado calculado por el circuito.

Los vectores de prueba no están predefinidos, se generan conforme avanza la simulación dentro del código en VHDL.