VHDL

Contenidos

Artículos

Referencias

Licencias de artículos

Elementos básicos del lenguaje

Antes de comenzar es preciso conocer algunos elementos básicos del lenguaje VHDL.

Comentarios

Los comentarios van precedidos de dos guiones. En una línea se ignorará todo aquello que vaya después de dos guiones seguidos. Ejemplo:

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

Identificadores

Son cualquier cadena de caracteres que sirven para identificar variables, señales, procesos, etc. Puede ser cualquier nombre compuesto por letras (*aux*) o números y letras (*aux1, aux2, aux3, ...*), incluyendo el símbolo de subrayado "_". Las mayúsculas y minúsculas son consideradas iguales, por lo tanto los identificadores *TMP* y *tmp* representan el mismo elemento. No es posible crear un identificador que coincida con alguna palabra reservada del lenguaje.

Números

Cualquier número se representa en base 10. Aunque es posible poner los números en otras bases utilizando diferentes símbolos, como se muestra en la siguiente sección.

Bases

Para escribir un número se puede hacer en binario, octal, decimal y hexadecimal.

- Para vectores de bits:
  - "01111" binario
  - O"17" octal
  - X"F" hexadecimal
- Para enteros y reales:
  - 2#1100# binario
  - 12 decimal
  - 16#C# hexadecimal

l

Tipos de datos

Como en cualquier lenguaje de programación existen varios tipos de datos, en VHDL se pueden diferenciar dos: escalares y compuestos.

Tipos escalares

Son tipos simples que contienen algún tipo de magnitud.

- *Enteros*: Son datos con un valor numérico entero. La forma de definirlos es con la palabra **RANGE**. Realmente se dice que un número está en un límite establecido.

```
    TYPE byte IS RANGE 0 TO 255;
```

- *Físicos*: Se trata de datos que corresponden con magnitudes físicas, que tienen un valor y unas unidades.

```
TYPE longitud IS RANGE 0 TO 1.0e9
  UNITS
    um;
    mm=1000 um;
    m=1000 mm;
    in=25.4 mm;
  END UNITS;
```

- *Reales o coma flotante*: Se definen igual que los enteros con la diferencia que los límites son números reales.

```
TYPE nivel IS RANGE 0.0 TO 5.0
```

- *Enumerados*: Son datos que puede tomar siempre que se encuentre en una lista o conjunto finito. Es idéntico a las enumeraciones en C (*enum*).

```
TYPE color IS (ROJO, VERDE, AMARILLO);
```

Tipos compuestos

Son tipos de datos compuestos por los que se han visto anteriormente.

- *Matrices*: Se trata de un conjunto de elementos del mismo tipo, accesibles mediante un índice. Los hay de dos tipos: monodimensionales o multidimensionales.

```
TYPE word IS ARRAY (31 DOWNTO 0) OF bit;
TYPE tabla IS ARRAY (1 TO 4, 1 TO 4) OF real;
```

En este punto es necesario explicar la palabra reservada **OTHERS**, donde es posible asignar un determinado valor a todos los elementos de la matriz.

```
word <= (OTHERS => '0');   -- Asigna '0' en todas las posiciones
```

Las palabras reservadas **TO** y **DOWNTO** sirven para indicar los índices de una matriz. El primero indica un rango ascendente (de x a y), mientras que el segundo es descendente (desde x hasta y).

```
-- word1 y word2 son elementos idénticos
TYPE word1 IS ARRAY (31 DOWNTO 0) OF bit;
TYPE word2 IS ARRAY (0 TO 31) OF bit;
```

Dependiendo de la opción elegida el bit más significativo corresponderá al primer bit (0) o al último (31). También es posible obtener un *trozo* de una matriz para poder realizar operaciones con ella.

```
TYPE word IS ARRAY (31 DOWNTO 0) OF bit;
TYPE subword IS ARRAY (7 DOWNTO 0) OF bit;
             ...
subword <= word(31 DOWNTO 24);
```

Además, es posible asignar a una matriz una lista separada por comas, de forma que el primer elemento de la lista corresponde al primero de la matriz.

```
semaforo <= (apagado, apagado, encendido);
luces <= (apagado, verde, azul, ..., amarillo);
```

- *Registros*: Es equivalente al tipo *record* de otros lenguajes.

```
TYPE trabajador IS
  RECORD
    nombre : string;
    edad : integer;
  END RECORD;
```

Para acceder a algún atributo del registro se utilizará el punto.

```
trabajadorA.nombre="Juan"
```

Subtipos de datos

En VHDL se pueden definir subtipos, los cuales corresponden a tipos ya existentes. Se pueden diferenciar dos tipos dependiendo de las restricciones que se apliquen.

- *Restricciones de un tipo escalar a un rango*:

```
SUBTYPE indice IS integer RANGE 0 TO 7;
SUBTYPE digitos IS character RANGE '0' TO '9';
```

- *Restricciones del rango de una matriz*:

```
SUBTYPE id IS string(0 TO 15);
SUBTYPE dir IS bit_vector(31 DOWNTO 0);
```

La ventaja de utilizar subtipos se basa a la hora de sintetizar un circuito, ya que si se utiliza el propio tipo, como puede ser el *integer*, se interpretará como un bus de 32 líneas, pero realmente sólo harán falta muchas menos.

Conversión de tipos

En ocasiones puede ser necesario convertir unos tipos a otros, esta operación es conocida como *casting*. Algunas de las conversiones son automáticas, como puede ser el paso de entero a real, otras conversiones deben realizarse de forma explícita, indicando el nombre del tipo al que se quiere pasar seguido del valor entre paréntesis.

```
real(15);
integer(3.5);
```

En muchos diseños es necesario realizar conversiones entre bits y enteros. A continuación se muestran varias funciones de conversión entre ambos tipos.

```
conv_integer(std_logic_vector); -- Conversión de vector a entero
conv_std_logic_vector(integer, numero_bits); -- Conversión de entero a vector de numero_bits de tamaño
```

Constantes, señales y variables

En VHDL existen tres tipos de elementos: señales, constantes y variables. Estas dos últimas tienen un significado similar a cualquier otro lenguaje de programación.

Todos estos elementos son diferentes. Las variables sólo tienen sentido dentro de los procesos o subprogramas, mientras que las señales pueden ser declaradas en arquitecturas, paquetes o bloques concurrentes. Las constantes pueden ser declaradas en los mismos sitios que las variables y señales.

Constantes

Es un elemento que se inicializa con un valor determinado, el cual no puede ser modificado, es decir siempre conserva el mismo valor. Esto se realiza con la palabra reservada **CONSTANT**.

```
CONSTANT e : real := 2.71828;
CONSTANT retraso : time := 10 ns;
```

También es posible no asociar un valor a una constante, siempre que el valor sea declarado en otro sitio.

```
CONSTANT max : natural;
```

Variables

Es lo mismo que una constante, pero con la diferencia que puede ser modificada en cualquier instante, aunque también es posible inicializarlas. La palabra reservada **VARIABLE** es la que permite declarar variables.

```
VARIABLE contador : natural := 0;
VARIABLE aux : bit_vector(31 DOWNTO 0);
```

Es posible, dado un elemento cambiarle el nombre o ponerle nombre a una parte mediante la palabra reservada **ALIAS**.

```
VARIABLE instruccion : bit_vector(31 DOWNTO 0);
ALIAS cod_op : bit_vector(7 DOWNTO 0) IS instruccion(31 DOWNTO 24);
```

Señales

Las señales se declaran con la palabra reservada **SIGNAL**, a diferencia con las anteriores este tipo de elementos pueden ser de varios tipos: normal, register o bus. Es posible asignarles un valor inicial.

```
SIGNAL sel : bit := '0';
SIGNAL datos : bit_vector(7 DOWNTO 0);
```

Atributos

Los elementos como señales y variables pueden tener **atributos**, éstos se indican a continuación del nombre, separados con una comilla simple "'" y pueden incluir información adicional de algunos objetos desarrollados en VHDL, que servirán a las herramientas de diseño para obtener información a la hora de realizar una síntesis.

Existen muchos atributos, como LEFT, RIGHT, LOW, HIGH, RANGE, LENGTH... Pero el atributo más usado es **EVENT**, que indica si una señal ha cambiado de valor. Por ejemplo la siguiente sentencia captura un flanco de subida de una señal (clk).

```
          ....
if clk'event and clk = '1' then
          ....
```

Definición de atributos

Un atributo definido por el diseñador siempre devolverá un valor constante. En primer lugar se debe declarar el atributo, mediante la palabra reservada **ATTRIBUTE**, indicando el tipo de elemento que se devuelve, seguidos el valor que se retornará. La sintaxís para definir atributos sería la siguiente.

```
ATTRIBUTE nombre : tipo
ATTRIBUTE nombre OF id_elemento : clase IS valor
```

Donde *nombre* es el identificador del atributo, *id_elemento* corresponde al identificador de un elemento del lenguaje definido previamente (señal, variable, etc.), la *clase* es el tipo de elemento al que se le va añadir dicho atributo, es decir si es señal, constante, etc. y el *valor* será lo que devuelva al preguntar por dicho atributo. Un ejemplo de todo esto puede ser el siguiente.

```
SIGNAL control : std_logic;

ATTRIBUTE min : integer;
ATTRIBUTE min OF control : SIGNAL IS 4;
          ....
IF control'min > 20 THEN
```

Operadores

Los operadores que proporciona el lenguaje son:

- *Lógicos*: Actúan sobre los tipos *bit, bit_vector y boolean*. En el caso de utilizar este tipo de operadores en un vector, la operación se realizará bit a bit.

Operadores: **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR** y **NOT**.

- *Aritméticos*:
  - **+** (suma o signo positivo): Sirve para indicar una suma entre dos números. También puede actuar como símbolo si se sitúa delante de una expresión.
  - **-** (resta o signo negativo): Sirve para indicar la resta entre dos números. Si va delante de una expresión modifica el signo de la expresión.
  - ***** (multiplicación): Multiplica dos números de cualquier tipo.
  - **/** (división): Divide dos números de cualquier tipo.
  - ****** (exponencial): Eleva un número a una potencia. El número de la izquierda puede ser entero y real, pero el de la derecha sólo puede ser entero. Ejemplo: 4**2 sería 4^2.
  - **ABS()** (valor absoluto): Devuelve el valor absoluto de su argumento.
  - **MOD** (módulo): Calcula el módulo de dos números.
  - **REM** (resto): Calcula el resto de la división.
- *Relacionales*: Siempre devuelven un valor booleano (true o false).
  - **==, /=** (igualdad): El primero devuelve verdadero si los operando son iguales y falso en caso contrario. El segundo indica desigualdad, funcionando al revés que el anterior.
  - **>, >=, <, <=** (menor mayor): Poseen el significado habitual (mayor que, mayor o igual que, menor que, menor o igual que, respectivamente). La diferencia con los anteriores reside en su uso, en este caso los tipos de datos que pueden manejar son siempre de tipo escalar o matrices.
- *Desplazamientos*: (incluidas en la revisión de 1993)
  - **SLL** (Shift Left Logic) y **SRL** (Shift Right Logic), desplazamiento lógico a la izquierda y desplazamiento lógico a la derecha, respectivamente, rellenando de ceros los huecos.

- **SLA** (Shift Left Arithmetic) y **SRA** (Shift Right Arithmetic), desplazamiento aritmético a la izquierda y derecha respectivamente.
- **ROL** (ROtate Left) y **ROR** (ROtate Right), rotación a la izquierda y a la derecha respectivamente. En este caso los huecos son ocupados por los bits que van quedando fuera.

A continuación se muestran ejemplos sobre los operadores de desplazamiento:

```
-- Inicialmente a vale 1100 1101
a sll 4 -- El resultado es 1101 0000
a sla 4 -- El resultado es 1101 1111 (extensión del último bit)
a srl 4 -- El resultado es 0000 1100
a sra 4 -- El resultado es 1111 1100
a rol 4 -- El resultado es 1101 1100 (los primeros 4 bits pasan a la última posición)
a ror 4 -- El resultado es 1101 1100
```

- *Otros*:
  - **&** (concatenación): Concatena vectores de manera que la dimensión de la matriz resultante es la suma de las dimensiones de las matrices con las que se opera.

Hay que decir que no todos los operadores pueden funcionar sobre todos los tipos de datos. También hay operadores que en determinadas circunstancias no pueden ser utilizados, por ejemplo al hacer código sintetizable no es recomendable usar multiplicadores (excepto si uno de los operadores es potencia de dos, puesto que se trataría de un simple desplazamiento de bits).

El orden de preferencia, de mayor a menor es:

1. **, ABS, NOT
2. *, /, MOD, REM
3. +, - (signo)
4. +, -, & (operaciones)
5. =, /=, <, <=, >, >=
6. AND, OR, NAND, NOR, XOR

Entidad

Durante los capítulos anteriores se ha insistido varias veces en que VHDL sirve para describir hardware. Un circuito electrónico puede ser parte de otro más grande, en este caso el primero sería un subcircuito del segundo. Por lo tanto, un circuito puede estar compuesto por muchos subcircuitos y estos subcircuitos se interconectarían. Así aparece una *jerarquía* en el diseño. En la parte alta de la jerarquía aparecerían los circuitos más complejos, que estarían compuestos por subcircuitos y, a su vez, cada uno de estos subcircuitos podría estar compuesto por subcircuitos más sencillos.

Un ejemplo de jerarquía sería un microprocesador. El circuito más complejo y el más alto en la jerarquía sería el propio microprocesador. Éste estaría compuestos por subcircuitos, por ejemplo el de la unidad de control, el de la unidad aritmético-lógica, memorias, registros, etc. Estos subcircuitos estarían conectados por líneas eléctricas, pueden ser simples como un cable o complejas como un bus. Una unidad de control estaría compuesta por más subcircuitos, más registros, más buses, etc.

Cuando se está diseñando en un determinado nivel, seguramente se empleen elementos de niveles más bajos. Para usar estos elementos de nivel bajo en un nivel más alto sólo se necesita conocer su interfaz, es decir, sus entradas y salidas, sobre ellas se conectarían los cables o buses que correspondieran.

Declaración de entidad

La *entidad* sirve para definir las entradas y salidas que tendrá un determinado circuito. Para definir una entidad se realizará mediante la palabra reservada **ENTITY**.

En principio pudiera parecer que esta definición sea equivalente a la cabecera de una función de un lenguaje cualquiera de programación. En VHDL es más conveniente ver a la entidad como una caja negra con cables para las entradas y salidas. La ventaja de pensar en una entidad como en una caja negra a la que se conectan cables es que es más fácil comprender la ejecución concurrente que ocurrirá en el hardware. La descripción de cómo funciona por dentro esa caja negra es la *arquitectura*, que se verá en el siguiente capítulo.

A continuación se muestra la sintáxis de una entidad.

```
ENTITY nombre IS
  [GENERIC(lista de parámetros);]
  [PORT(lista de puertos);]
END [ENTITY] nombre;
```

La instrucción **GENERIC** define y declara propiedades o constantes del módulo. Las constantes declaradas en esta sección son como los parámetros en las funciones de cualquier otro lenguaje de programación, por lo que es posible introducir valores, en caso contrario tomará los valores por defecto. Para declarar una constante se indicará su nombre seguido de dos puntos y el tipo del que se trata, finalmente se indicará el valor al que es inicializado mediante el operador de asignación **:=**. En el caso que existan más constantes se terminará con un punto y coma, la última constante no lo llevará.

```
nombre_constante : tipo := inicializacion;
```

La instruccion **PORT** definen las entradas y salidas del módulo definido. Basicamente consiste en indicar el nombre de la señal seguido de dos puntos y la dirección del puerto (se verá más adelante), además del tipo de señal del que se trata. Al igual que antes, si existe más de una señal se finalizará con un punto y coma, exceptuando la última señal de la lista.

```
nombre_señal : dirección tipo;
```

A continuación se muestra un ejemplo de una entidad, con una serie de constantes y señales de entrada y salida.

```
ENTITY mux IS
  GENERIC(
    C_AWIDTH : integer := 32;
    C_DWIDTH : integer := 32
  );
  PORT(
    control  : IN bit;
    entrada1 : IN bit;
    entrada2 : IN bit;
    salida   : OUT bit
  );
END mux;
```

En este ejemplo la entidad de llama *mux*. Su interfaz se compone de las señales *control*, *entrada1* y *entrada2* como entradas de tipo bit y de la señal llamada *salida* como salida, también de tipo bit. Además, se incluyen dos constantes que servirán a la parte declarativa para realizar alguna operación.

En la introducción se vio como asignar las señales de entradas y salidas mediante la palabra **PORT MAP**, para el caso de los genéricos se realiza con la palabra reservada **GENERIC MAP**, esta parte se estudiará con mayor detalle en los siguientes capítulos. Un ejemplo para utilizar el código anterior como un componente sería el siguiente.

```
mux_1 : ENTITY work.mux
  GENERIC MAP(
    C_AWIDTH => C_AWIDTH,
    C_DWIDTH => C_DWIDTH
  )
  PORT MAP(
    control  => control,
    entrada1 => entrada1,
    entrada2 => entrada2,
    salida   => salida
  );
```

Obsérvese que en todo momento se habla de señales y no de variables, para el caso de los puertos.

Direcciones de los puertos de una entidad

Las señales representarían la función que harían los cables en un diseño hardware tradicional, es decir, sirven para transportar información y establecer conexiones. Dentro de una **entidad** los puertos son considerados como señales, en donde se pueden diferenciar varios tipos.

- **IN**: Son señales de entrada, las cuales sólo se pueden leer, pero no se le pueden asignar ningún valor, es decir, no se puede modificar el valor que poseen. Por lo tanto, su funcionalidad es similar a las constantes.
- **OUT**: Corresponden a las señales de salida, en este caso su valor puede ser modificado, pero en este caso no pueden leerse, es decir no pueden ser utilizadas como argumentos en la asignación de cualquier elemento.
- **INOUT**: Este tipo es una mezcla de los dos anteriores, pueden ser utilizados tanto como de lectura o de escritura.
- **BUFFER**: Es idéntico al anterior, con la diferencia de que sólo una fuente puede modificar su valor.

Arquitectura

Como se ha dicho en el capítulo anterior, la arquitectura es lo que define cómo se comporta un circuito. En el primer capítulo también se mostraron varias arquitecturas en las que se describía un multiplexor. La primera línea de cada una era

```
ARCHITECTURE mux_comportamiento OF mux IS
ARCHITECTURE mux_rtl OF mux IS
ARCHITECTURE mux_estructural OF mux IS
```

Los nombres de cada arquitectura son *mux_comportamiento*, *mux_rtl* y *mux_estructural* respectivamente. Todas están asociadas a la entidad *mux*. El nombre de la arquitectura se usará para indicar qué arquitectura se debe usar en caso que haya varias para una misma entidad.

Después de esta línea pueden aparecer varias instrucciones para indicar la declaración de señales, componentes, funciones, etc.. Estas señales son internas, es decir, a ellas no se puede acceder desde la entidad, por los que los circuitos de nivel superior no podrían acceder a ellas. En un símil con un microprocesador, estas señales podrían ser las líneas que comunican la unidad central con la ALU, a las que no se puede acceder directamente desde el exterior del microprocesador. Obsérvese que en este caso no se indica si son entradas o salidas, puesto que al ser internas pueden ser leídas o escritas sin ningún problema. En esta parte de la arquitectura también pueden aparecer otros elementos, como pueden ser las constantes. Lo siguiente es la palabra clave **BEGIN**, que da paso a la descripción del circuito, mediante una serie de sentencias. Por lo tanto, la sintaxis de una arquitectura sería.

```
ARCHITECTURE nombre OF nombre_entidad IS
  [declaraciones]
BEGIN
  [sentencias concurrentes]
END [ARCHITECTURE] [nombre];
```

Un ejemplo de una arquitectura podría ser la siguiente.

```
ARCHITECTURE mux_rtl OF mux IS
  SIGNAL int1, int2, int3 : BIT;
BEGIN
  int1 <= NOT control;
  int2 <= entrada1 AND int1;
  int3 <= entrada2 AND S;
  salida <= int2 OR int3;
END mul_rtl;
```

La descripción puede ser de tres tipos:

1. Descripción de flujo de datos
2. Descripción de comportamiento
3. Descripción estructural

Descripción de flujo de datos

A la hora de plantearse crear un programa en VHDL no hay que pensar como si fuera un programa típico para ordenador. No hay que olvidar que en VHDL hay que describir un hardware, algo que no se hace en un programa para ordenador. Un circuito electrónico puede tener muchos elementos que estén ejecutando acciones a la vez, por

ejemplo en un circuito puede tener una entrada que se aplique a dos puertas lógicas y de cada una obtener una salida, en este caso tendría dos caminos en los que se ejecutarían acciones (las puertas lógicas) de forma paralela. Esto es lo que se llama **concurrencia**.

VHDL es un lenguaje concurrente, como consecuencia no se seguirá el orden en que están escritas las instrucciones a la hora de ejecutar el código. De hecho, si hay dos instrucciones, no tiene porqué ejecutarse una antes que otra, pueden ejecutarse a la vez.

Sentencias Concurrentes

La instrucción básica de la ejecución concurrente es la asignación entre señales a través del símbolo **<=**. Para facilitar la asignación de las señales VHDL incluye elementos de alto nivel como son instrucciones condicionales, de selección, etc, que se verán a continuación.

WHEN ... ELSE

Sentencia de selección múltiple. En hardware es necesario incluir todas las opciones posibles. En este caso es obligatorio siempre acabar la expresión con un **ELSE**.

```
<señal> <= <asignación1> WHEN <condición1> ELSE
           <asignación2> WHEN <condición2> ELSE
                         ...
           <asignaciónN> WHEN <condiciónN> ELSE
           <asignaciónM>;
```

Un posible ejemplo de este tipo de sentencias podría ser la siguiente.

```
s <= "00" WHEN a = b ELSE
     "01" WHEN a > b ELSE
     "11";
```

Siempre es obligatorio asignar algo, aunque es posible no realizar acción alguna, para ello se utiliza la palabra reservada **UNAFFECTED**. De esta forma se asignará el mismo valor que tenía la señal.

```
s1 <= d1 WHEN control = '1' ELSE UNAFFECTED;
s2 <= d2 WHEN control = '1' ELSE s2;
```

Las dos sentencias anteriores parecen iguales, pero en la segunda se produce una transacción, aspecto que en la primera no sucede.

WITH ... SELECT ... WHEN

Es similar a las sentencias **CASE** o **SWITCH** de C. La asignación se hace según el contenido de un objeto o resultado de cierta expresión.

```
WITH <señal1> SELECT
  <señal2> <= <asignación1> WHEN <estado_señal1>,
              <asignación2> WHEN <estado_señal2>,
                            ...
              <asignaciónN> WHEN OTHERS;
```

Un ejemplo de esta sentencia es la siguiente.

```
WITH estado SELECT
  semaforo <= "rojo"     WHEN "01",
              "verde"    WHEN "10",
              "amarillo" WHEN "11",
              "roto"     WHEN OTHERS;
```

La cláusula **WHEN OTHERS** especifica todos los demás valores que no han sido contemplados. También es posible utilizar la opción que se contempló en el caso anterior (**UNAFFECTED**).

BLOCK

En ocasiones interesa agrupar un conjunto de sentencias en bloques. Estos bloques permiten dividir el sistema en módulos, estos módulos pueden estar compuestos de otros módulos. La estructura general es la siguiente.

```
block_id: BLOCK(expresión de guardia)
  declaraciones
BEGIN
  sentencias concurrentes
END BLOCK block_id;
```

El nombre del bloque es opcional (*block_id*), al igual que la expresión de guardia. Un ejemplo de esto podría ser el siguiente.

```
latch: BLOCK(clk='1')
BEGIN
  q <= GUARDED d;
END BLOCK latch;
```

Descripción de comportamiento

Como la programación concurrente no siempre es la mejor forma de describir ideas, VHDL incorpora la programación serie, la cual se define en bloques indicados con la sentencia **PROCESS**. En un mismo diseño puede haber varios bloques de este tipo, cada uno de estos bloques corresponderá a una instrucción concurrente. Es decir, internamente la ejecución de las instrucciones de los **PROCESS** es serie, pero entre los bloques es concurrente.

A continuación se verán la estructuras más comunes de la ejecución serie y sus características.

PROCESS

Un **PROCESS**, como se ha dicho antes, es una sentencia concurrente en el sentido de que todos los **PROCESS** y todas las demás sentencias concurrentes se ejecutarán sin un orden establecido. No obstante las sentencias que hay dentro del **PROCESS** se ejecutan de forma secuencial.

Por lo tanto se puede decir que una estructura secuencial va en el interior de un **PROCESS**.

La estructura genérica de esta sentencia es:

```
PROCESS [lista de sensibilidad]
  [declaración de variables]
BEGIN
  [sentencias secuenciales]
END PROCESS;
```

La lista de sensibilidad es una serie de señales que, al cambiar de valor, hacen que se ejecute el **PROCESS**.

Un ejemplo sería:

```
     PROCESS(señal1, señal2)
     ...
```

El **PROCESS** anterior sólo se ejecutará cuando *señal1* o *señal2* cambien de valor.

Variables y Señales

Hay que distinguir las señales y las variables, las señales se declaran entre **ARCHITECTURE** y su correspondiente **BEGIN** mientras que las variables se declaran entre **PROCESS** y su **BEGIN**. Dentro de un **PROCESS** pueden usarse ambas, pero hay una diferencia importante entre ellas: las señales sólo se actualizan al terminar el proceso en el que se usan, mientras que las variables se actualizan instantáneamente, es decir, su valor cambia en el momento de la asignación.

Unos ejemplos son:

```
     ENTITY ejemplo
       PORT (c: IN std_logic;
         d: OUT std_logic);
     END ENTITY;

     ARCHITECTURE ejemplo_arch OF ejemplo IS
       SIGNAL a,b: std_logic;
     BEGIN
       PROCESS(c)
         VARIABLE z: std_logic;
       BEGIN
         a<= c and b;    --asignación de señales: después de
ejecutarse esta línea a seguirá valiendo lo mismo, sólo se actualiza al
 acabar el PROCESS
         z:= a or c;     --asignación de variables: en el momento de
ejecutarse esta línea z valdrá a or c (el valor que tenía a cuando
empezó el PROCESS)
       END PROCESS;
     END ARCHITECTURE;
```

Sentencias secuenciales

IF ... THEN ... ELSE

Permite la ejecución de un bloque de código dependiendo de una o varias condiciones.

```
     IF <condición1> THEN
        [sentencias 1]
     ELSIF <condición2> THEN
        [sentencias 2]
     ELSE
        [sentencias N]
     END IF;
```

Un ejemplo es:

```
IF (reloj='1' AND enable='1') THEN
  salida<=entrada;
ELSIF (enable='1') THEN
  salida<=tmp;
ELSE
  salida<='0';
END IF;
```

CASE

Es parecido al anterior porque también ejecuta un bloque de código condicionalmente, pero en esta ocasión se evalúa una expresión en vez de una condición. Se debe recordar que se deben tener en cuenta todos los casos, es decir, incluir como última opción la sentencia **WHEN OTHERS**.

```
CASE <expresión> IS
   WHEN <valor1> => [sentencias1]
   WHEN <valor2> => [sentencias2]
   WHEN <rango de valores> => [sentenciasN]
   WHEN OTHERS => [sentenciasM]
END CASE;
```

Un ejemplo es:

```
CASE a IS
  WHEN 0        =>    B:=0;
  WHEN 1 to 50  =>    B:=1;
  WHEN 99 to 51 =>    B:=2;
  WHEN OTHERS   =>    B:=3;
END CASE;
```

LOOP

LOOP es la forma de hacer bucles en VHDL. Sería el equivalente a un **FOR** o **WHILE** de un lenguaje convencional.

Su estructura es:

```
[etiqueta:] [WHILE <condición> | FOR <condición>] LOOP
  [sentencias]
  [exit;]
  [next;]
END LOOP [etiqueta];
```

Un ejemplo de bucles anidados es:

```
bucle1: LOOP
  a:=A+1
  b:=20;
  bucle2: LOOP
    IF b < (a*b) THEN
```

```
        EXIT bucle2;
      END IF;
      b:=b+a;
    END LOOP bucle2;
    EXIT bucle1 WHEN a>10;
  END LOOP bucle1;
```

Otro ejemplo, este con FOR es:

```
  bucle1: FOR a IN 1 TO 10 LOOP
    b:=20;
    bucle2: LOOP
      IF b<(a*a) THEN
        EXIT bucle2;
      END IF;
      b:=b-a;
    END LOOP bucle2;
  END LOOP bucle1;
```

Otro más con WHILE

```
  cuenta := 10;
  bucle1: WHILE cuenta >= 0 LOOP
    cuenta := cuenta + 1;
    b:=20;
    bucle2: LOOP
      IF b<(a*a) THEN
        EXIT bucle2;
      END IF;
      b := b-a;
    END LOOP bucle2;
  END LOOP bucle1;
```

NEXT y EXIT

NEXT permite detener la ejecución actual y seguir con la siguiente.

```
  [id_next:]
  NEXT [id_bucle] [WHEN condición];
```

Como se puede suponer, la sentencia **EXIT** hace que se salga del bucle superior al que se ejecuta.

```
  [id_exit:]
  EXIT [id_bucle] [WHEN condición];
```

Se puede ver su uso en los ejemplos del apartado anterior.

ASSERT

Se usa para verificar una condición y, en caso de que proceda, dar un aviso.

La sintáxis es:

```
ASSERT <condición>
  [REPORT <expresión>]
  [SEVERITY <expresión>];
```

Este comando se estudiará en el subcapítulo de notificaciones, en la sección de bancos de prueba. Puesto que el uso de este comando se realiza únicamente en la simulación de circuitos.

WAIT

La ejecución de un bloque **PROCESS** se realiza de forma continuada, como si de un bucle infinito se tratara (se ejecutan todas las sentencias y se vuelven a repetir). Esto no tiene mucho sentido, puesto que continuamente se ejecutaría lo mismo una y otra vez, sería interesante poder parar la ejecución. Una forma de hacerlo e mediante las listas de sensibilidad, las cuales se han visto anteriormente, aunque existe otra forma de hacerlo mediante la sentencia **WAIT**, pero es algo más complejo.

```
WAIT ON lista_sensible UNTIL condicion FOR timeout;
```

La *lista_sensible* es un conjunto de señales separadas por comas. La *condición* es una sentencia que activará de nuevo la ejecución. El *timeout* es el tiempo durante el cual la ejecucion esta detenida.

No es necesario utilizar las tres opciones, en caso de hacerlo la primera condición que se cumpla volverá a activar la ejecución.

```
WAIT ON pulso;
WAIT UNTIL counter = 5;
WAIT FOR 10 ns;
WAIT ON pulso, sensor UNTIL counter = 5;
WAIT ON pulso UNTIL counter = 5 FOR 10 ns;
```

Si se utiliza una lista de sensibilidad no es posible utilizar la sentencia **WAIT**, sin embargo si es posible utilizar varias sentencias **WAIT** cuando esta actúa como condición de activación. Este comando se estudiará en el subcapítulo de retrasos, en la sección de bancos de prueba.

Descripción estructural

Las dos descripciones anteriores son las más utilizadas por los diseñadores, ya que son más cercanos al pensamiento humano. Aunque existe otro tipo de descripción, que permite la realización de diseños jerárquicos. VHDL dispone de diferentes mecanismos para la descripción estructural.

Definición de componentes

En VHDL es posible declarar componentes dentro de un diseño mediante la palabra **COMPONENT**. Un componente se corresponde con una entidad que ha sido declarada en otro módulo del diseño, o incluso en alguna biblioteca, la declaración de este elemento se realizará en la parte declarativa de la arquitectura del módulo que se está desarrollando. La sintáxis para declarar un componente es muy parecida a la de una entidad.

```
COMPONENT nombre [IS]
  [GENERIC(lista_parametros);]
  [PORT(lista_de_puertos);]
```

```
END COMPONENT nombre;
```

Si se dispone de un compilador de VHDL'93 no será necesario incluir en los diseño la parte declarativa de los componentes, es decir se pasaría a referenciarlos de forma directa. Un ejemplo de un componente podría ser el siguiente.

```
COMPONENT mux IS
  GENERIC(
    C_AWIDTH : integer;
    C_DWIDTH : integer
  );
  PORT(
    control  : IN  bit;
    entrada1 : IN  bit;
    entrada2 : IN  bit;
    salida   : OUT bit
  );
END COMPONENT mux;
```

Referencia de componentes

La referencia de componentes consiste en copiar en la arquitectura aquel componente que se quiera utilizar, tantas veces como sea necesario para construir el diseño. Para ello, la sintaxis que presenta la instanciación de un componente es la siguiente.

```
ref_id:
  [COMPONENT] id_componente | ENTITY id_entidad [(id_arquitectura)] | CONFIGURATION id_configuración
  [GENERIC MAP (parametros)]
  [PORT MAP (puertos)];
```

Un ejemplo de referenciación del componente anterior sería.

```
mux_1 :
  COMPONENT mux
  GENERIC MAP (
    C_AWIDTH => C_AWIDTH,
    C_DWIDTH => C_DWIDTH
  )
  PORT MAP (
    control  => ctrl,
    entrada1 => e1,
    entrada2 => e2,
    salida   => sal
  );
```

Las señales *ctrl, e1, e2 y sal* deben ser declaradas previamente en la sección de declaraciones de la arquitectura, estas señales sirven para poder conectar unos componentes con otros. También deben declararse las variables que se utilizan en la sección *GENERIC*.

Organización del código

En descripciones de sistemas complejos es necesario una organización que permita al diseñador trabajar con mayor comodidad. En el capítulo anterior se vio como era posible agrupar una serie de sentencias mediante módulos. Pero existen otras formas de organizar el código, a través de *subprogramas*, que harán más legibles dichos sistemas. Por otro lado, estos subprogramas pueden ser agrupados junto con definiciones de tipos, bloques, ... en estructuras, lo que formarían los denominados *paquetes*, que a su vez con elementos de configuración describirían una *librería*.

Subprogramas

Como en otros lenguajes de programación, en VHDL es posible estructurar el código mediante el uso de subprogramas. Realmente, un subprograma es una función o procedimiento que realiza una determinada tarea, aunque existen ciertas diferencias entre ambas.

- Una función devuelve un valor y un procedimiento devuelve los valores a través de los parámetros que le han sido pasados como argumentos. Por ello, las primeras deberán contener la palabra reservada **RETURN**, mientras que las segundas no tienen necesidad de disponer dicha sentencia, en caso de tener una sentencia de ese tipo interrumpirá la ejecución del procedimiento.
- A consecuencia de la anterior, en una función todos sus parámetros son de entrada, por lo que sólo pueden ser leidos dentro de la misma, por el contrario en un procedimiento los parámetros pueden ser de entrada, de salida o de entrada y salida (unidireccionales o bidireccionales).
- Las funciones se usan en expresiones, sin embargo, los procedimientos se llaman como una sentencia secuencial o concurrente.
- Los procedimientos pueden tener efectos colaterales al poder cambiar señales externas que han sido pasadas como parámetros, por otro lado las funciones no poseen estos efectos.
- Las funciones nunca pueden tener una sentencia **WAIT**, pero los procedimientos sí.

Declaración de funciones y procedimientos

Las declaraciones de estos elementos pueden realizarse en la parte declarativas de las arquitecturas, bloques, paquetes, etc. A continuación, se muestra la estructura de un procedimiento.

```
PROCEDURE nombre[(parámetros)] IS
  [declaraciones]
BEGIN
  [sentencias]
END [PROCEDURE] [nombre];
```

La estructura de las funciones corresponden a las siguientes líneas.

```
[PURE | IMPURE]
FUNCTION nombre[(parámetros)] RETURN tipo IS
  [declaraciones]
BEGIN
  [sentencias]          -- Debe incluir un RETURN
```

```
END [FUNCTION] [nombre];
```

Como ya se explicó, la lista de *parámetros* es opcional en ambos casos. Estos parámetros se declaran de forma similar a como se hacen los puertos de una entidad.

```
<nombre del puerto> : <tipo de puerto> <tipo de objeto>
```

Dependiendo de la estructura que se utilice, funciones o procedimientos, los parámetros tendrán un significado u otro. En las funciones sólo es posible utilizar el tipo de puerto **IN**, mientras que en los procedimientos pueden usarse los tipos **IN**, **OUT** e **INOUT**. Además, en las funciones el parámetro puede ser **CONSTANT** o **SIGNAL**, por defecto es **CONSTANT**. Por otro lado, en los procedimientos los parámetros de tipo **IN** son **CONSTANT** por defecto y **VARIABLE** para el resto, aunque también es posible utilizar **SIGNAL** siempre que se declare explícitamente. No se aconseja utilizar señales como parámetros, por los efectos que pueden tener en la ejecución de un programa.

Las funciones **PURE** o puras devuelven el mismo valor para unos parámetros de entrada determinados. Mientras que una función es **IMPURE** o impura si para los mismos valores de entrada se devuelve distinto valor. Estas últimas pueden depender de una variable o señal global. Realmente, estas palabras reservadas hacen de comentario, puesto que una función no se hace impura o pura por indicarlo.

Un ejemplo de una función sería el siguiente.

```
FUNCTION es_uno(din : std_logic_vector(31 downto 0)) RETURN std_logic IS
  VARIABLE val : std_logic;
BEGIN
  IF din = X"00000001" THEN
    val := '1';
  ELSE
    val := '0';
  END IF;
  RETURN val;
END es_uno;
```

Por otro lado, un ejemplo de un procedimiento podría pertenecer a las siguientes líneas.

```
PROCEDURE es_uno(din : std_logic_vector(31 downto 0)
                 dout : std_logic) IS
BEGIN
  IF din = X"00000001" THEN
    dout := '1';
  ELSE
    dout := '0';
  END IF;
END es_uno;
```

En la asignación a la señal de salida se realiza con el operador :=, puesto que no es una señal, es decir que se trata de un tipo **VARIABLE**. Si se tratara de una señal se haría con el operador de asignación <=.

Llamadas a subprogramas

Es muy sencillo realizar una invocación a un subprograma, basta con indicar el nombre de dicho subprograma seguido de los argumentos, los cuales irán entre paréntesis. En VHDL existen varias formas de pasar los parámetros a un subprograma.

- *Asociación implícita*: Poniendo los parámetros en el mismo orden en el que se declaran en el subprograma. Por ejemplo, si se dispone del siguiente procedimiento.

```
PROCEDURE limite(CONSTANT conj : IN std_logic_vector(3 DOWNTO 0);
                 VARIABLE min, max : INOUT integer) IS
                           ...

limite(cjt(31 DOWNTO 28), valmin, valmax); -- Llamada al procedimiento
```

- *Asociación explícita*: Los parámetros se colocan en cualquier orden. Un ejemplo de la llamada al procedimiento anterior podría ser la siguiente.

```
limite(min=>valmin, max=>valmax, conj=>cjt(31 DOWNTO 28));
```

- *Parámetros libres*: En VHDL es posible dejar parámetros por especificar, de forma que tengan unos valores por defecto, y en el caso de pasar un valor a dichos argumentos puedan tomar el valor especificado. Un ejemplo de este tipo de llamadas sería el siguiente.

```
PROCEDURE lee(longuitud : IN integer := 200) IS
BEGIN
            ....
END PROCEDURE;

-- Posibles llamadas
lee;
lee(350);
```

Como ya se indicó al inicio de este capítulo, los subprogramas pueden ser llamados desde un entorno secuencial o concurrente. En caso de ser llamado en un entorno concurrente el procedimiento se ejecutará de forma similar a un bloque **PROCESS**, por lo que hay que tener alguna sentencia que permita suspender la ejecución, porque podría ejecutarse de forma continua. Este tipo de sentencias podría ser una lista sensible o una sentencia **WAIT**, como ya se explicó en los bloques **PROCESS**, como se indicó, no es posible utilizar la lista de sensibilidad junto con una sentencia **WAIT**. Este tipo de sentencias es posible que detengan un procedimiento para siempre si son utilizadas en entornos secuenciales.

Sobrecarga de operadores

Como en otros lenguajes de programación, en VHDL también es posible sobracargar los métodos, es decir, tener funciones con el mismo nombre pero con distintos parámetros. Aunque, en este lenguaje hay que tener un poco de cuidado a la hora de declarar los procedimientos, puesto que al ser posible dejar libres algunos argumentos es muy probable encontrar situaciones en las que dos funciones son llamadas de forma idéntica cuando se hace sin parámetros. Por ejemplo, si se dispone de los siguientes procedimientos la llamada a éstos sin parámetros sería la misma, y no habría forma de diferenciar a qué método se refiere.

```
PROCEDURE lee(longuitud : IN integer := 20) IS
BEGIN
            ....
END PROCEDURE;

PROCEDURE lee(factor : IN real := 100.0) IS
BEGIN
            ....
END PROCEDURE;

lee;
```

Paquetes

Como se comentó al principio, un paquete consta de un conjunto de subprogramas, constantes, declaraciones, etc., con la intención de implementar algún servicio. Así se pueden hacer visibles las interfaces de los subprogramas y ocultar su descripción.

Definición de paquetes

Los paquetes se separan en dos zonas: *declaraciones* y *cuerpo*, aunque esta última puede ser eliminada si no se definen funciones y/o procedimientos. Las siguientes líneas muestra la estructura de un paquete.

```
-- Declaración de paquete
PACKAGE nombre IS
  declaraciones
END [PACKAGE] [nombre];

-- Declaración del cuerpo
PACKAGE BODY nombre IS
  declaraciones
  subprogramas
      ...
END [PACKAGE BODY] [nombre];
```

El nombre del paquete debe coincidir en la declaración del paquete y del cuerpo. A continuación, se muestra un ejemplo de un paquete.

```
-- Declaración de paquete
PACKAGE mi_paquete IS
  SUBTYPE dir_type IS std_logic_vector(31 DOWNTO 0);
  SUBTYPE dato_type IS std_logic_vector(15 DOWNTO 0);
  CONSTANT inicio : dir_type;    -- Hay que definirlo en el BODY
  FUNCTION inttodato(valor : integer) RETURN dato_type;
  PROCEDURE datotoint(dato : IN dato_type; valor : OUT integer);
END mi_paquete;

-- Declaración del cuerpo
PACKAGE BODY mi_paquete IS
  CONSTANT inicio : dir_type := X"FFFF0000";

  FUNCTION inttodato(valor : integer) RETURN dato_type IS
    -- Cuerpo de la función
  END inttodato;

  PROCEDURE datotoint(dato : IN dato_type; valor : OUT integer) IS
    -- Cuerpo del procedimiento
  END datotoint;

END PACKAGE BODY mi_paquete;
```

Para acceder a los tipos creados en un paquete, se debe indicar el nombre del paquete seguido del elemento que se desea utilizar, separados por un punto. Para el ejemplo anterior sería algo parecido a las siguientes líneas.

```
VARIABLE dir : work.mi_paquete.dir_type;
dir := work.mi_paquete.inicio;
```

Existe otra forma de realizarlo, haciendo *visible* al paquete de esta forma no se necesitará el punto ni tampoco indicar el nombre del paquete. Para ello, se debe utilizar la sentencia **USE** seguido del paquete que se va a utilizar y el método a utilizar, todo ello separado por puntos. El ejemplo anterior se podría realizar de la siguiente forma.

```
USE work.mi_paquete.ALL
VARIABLE dir : dir_type;
dir := inicio;
```

Librerías

Hasta ahora se han visto varios elementos del lenguaje, como pueden ser las entidades, las arquitecturas, los paquetes, etc. Cuando se realiza una descripción en VHDL se utilizan estas unidades, en uno o más ficheros, éstos se denominan *ficheros de diseño*. Posteriormente, estos ficheros serán compilados para obtener una *librería* o *biblioteca de diseño*, de forma que esta biblioteca contiene los elementos que componen el circuito. La biblioteca donde se guardan los resultados de la compilación se denomina *work*.

Una librería se compone de dos partes bien diferenciadas, dependiendo de las unidades que la formen. Por un lado, están las *unidades primarias*, que corresponderán a entidades, paquetes y archivos de configuración. Mientras que las *unidades secundarias* serán arquitecturas y cuerpos de paquetes. Por lo tanto, se puede sacar la conclusión de que

cada unidad secundaria deberá estar asociada con una unidad primaria.

Al realizar una compilación se analizarán las unidades que vayan apareciendo en el texto. Por consiguiente, es importante establecer un orden lógico de las distintas unidades, para que de esta forma se puedan cumplir las dependencias existentes entre las mismas. La forma que toma la *librería* una vez compilada es muy diversa; dependiendo de la herramienta de compilación utilizada así será el resultado obtenido, esto se debe a que en VHDL no existe un estándar para crear bibliotecas.

Para incluir una *librería* a un diseño basta con utilizar la palabra reservada **LIBRARY** seguida del nombre de la biblioteca a utilizar. Además, también es posible hacer visibles elementos internos de estas bibliotecas con el uso de la sentencia **USE**, como se explicó en el apartado anterior. En el caso de querer hacer visible todos los elementos de un paquete se puede utilizar la palabra reservada **ALL**.

```
LIBRARY mis_componentes;
USE mis_componentes.logic.ALL;
```

En VHDL hay dos librerías que no hacen falta importarlas. Por un lado está la librería *work*, que contiene las unidades que se están compilando, y por otro lado, la librería *std* que contiene los paquetes *standard* y *textio*, las cuales contienen definiciones de tipos y funciones para el acceso a ficheros de texto.

Librería *ieee*

Una de las bibliotecas más utilizadas en el mundo de la industria es la denominada *ieee*, la cual contiene algunos tipos y funciones que completan a las que vienen por defecto en el propio lenguaje. Dentro de la librería existe un paquete denominado *std_logic_1164*, con el cual se pueden trabajar con un sistema de nueve niveles lógicos, como puede ser: valor desconocido, alta impedancia, etc. El siguiente código muestra parte de este paquete.

```
PACKAGE std_logic_1164 IS
  TYPE std_ulogic IS( 'U', -- Indefinido
                      'X', -- Desconocido
                      '0', -- 0
                      '1', -- 1
                      'Z', -- Alta impedancia
                      'W', -- Desconocido
                      'L', -- LOW (weak low o 0 débil)
                      'H', -- HIGH (weak high o 1 débil)
                      '-'  -- Desconocido
                     );

TYPE std_ulogic_vector IS ARRAY(NATURAL RANGE <>) OF std_ulogic;

FUNCTION resolved(s : std_ulogic_vector) RETURN std_ulogic;

SUBTYPE std_logic IS resolved std_ulogic;

TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
```

Otros conceptos

Hasta este momento se ha ofrecido una visión general del lenguaje VHDL. EN esta sección se verán algunas cosas que resultan interesantes en algunas ocasiones a la hora de diseñar algún sistema.

Punteros en VHDL

Algunos lenguajes de programación permiten reservar espacio de memoria de manera dinámica mediante *punteros*. Como es sabido, un puntero es una dirección de memoria que apunta a una variable. En VHDL el tipo de datos que permite reservar memoria es a través de la palabra reservada **ACCESS**, la forma de de definir un puntero es la siguiente.

```
TYPE tipo_puntero IS ACCESS tipo_elemento;
VARIABLE p : tipo_puntero;
```

La variable *p* corresponde a un puntero que apunta a variables del tipo *tipo_elemento*. Los punteros sólo pueden ser variables, en ningún caso serán señales, por consiguiente solo podrán estar dentro de un proceso (**PROCESS**). Por otro lado, la creación de un puntero se realiza mediante la palabra **NEW**. Un ejemplo para declarar un puntero es el siguiente.

```
TYPE punteroA IS ACCESS integer RANGE 0 TO 255;
VARIABLE p : punteroA;
          ....
p := NEW integer RANGE 0 TO 255;
p.ALL := 45;
```

Para acceder al contenido de la drección apuntada se utiliza la palabra **ALL**, si no se inicializa el valor que toma, por defecto tomará el valor **NULL**. Si se desea liberar la memoria utilizada por un puntero se hará uso del procedimiento *deallocate*. Para el caso anterior se realizaría de la siguiente forma.

```
Deallocate(p);
```

Ficheros

En algunas ocasiones es interesante utilizar ficheros para leer información o incluso almacenarla. El uso de ficheros únicamente es válido en la simulación, mientras que en la síntesis no es posible su uso. Estos ficheros no pueden almacenar matrices multidimensionales, punteros o ficheros.

Abrir y cerrar ficheros

En primer lugar se debe declarar el tipo de datos que contendrá dicho archivo. Mediante las palabras reservadas **TYPE** y **FILE** se genera la siguiente estructura.

```
TYPE fichero_tipo IS FILE OF tipo;
FILE nombre :
  fichero_tipo IS [modo] "fichero":        -- VHDL'87
  [fichero_tipo [OPEN modo] IS "fichero"]  -- VHDL'93
```

Cuando se tratan con ficheros en VHDL, la estructura anterior depende del compilador que se utilice (VHDL'87 y VHDL'93). La primeras líneas son iguales en ambos casos, pero a la hora de indicar el acceso al fichero la sintáxis

difiere de uno a otro compilador. En VHDL'87 el *modo* puede ser **IN** (por defecto) si se va a leer y **OUT** para escribir. Por otro lado, en VHDL'93 el *modo* es definido mediante un tipo enumerado declarado en la librería correspondiente (*file_open_kind*), donde se encuentra *write_mode*, *read_mode* (por defecto) o *append_mode*, que indica escritura, lectura y concatenación respectivamente. A diferencia que en el anterior, en este caso no es necesario abrirlo obligatoriamente en el momento en que se declara. Para posteriormente abrirlo se podría utilizar el siguiente subprograma.

```
PROCEDURE file_open(status: OUT file_open_status;
                    FILE f: file_type;
                    external_name: IN string;
                    open_kind: IN file_open_kind:=read_mode);
```

Al realizar la apertura después de la declaración conlleva una flexibilidad mayor, como puede ser la petición del nombre del fichero. También es posible obtener el estado en el cual se quedó el fichero: *open_ok*, *status_error*, *name_error* y *mode_error*, aunque este parámetro es opcional.

Además de este subprograma existe otro para poder cerrar el fichero abierto.

```
PROCEDURE file_close(FILE f: file_type);
```

A continuación se muestra un ejemplo de declaración, apertura y cierre de ficheros.

```
TYPE arch_integer IS FILE OF integer;

-- Declaración y apertura a la vez
FILE datos: arch_integer OPEN read_mode IS "input.txt";

-- Declaración y apertura posterior
FILE datos: arch_integer;
        .....
file_open(datos, "input.txt",read_mode);

-- Cerrar
file_close(datos);
```

Lectura y escritura de ficheros

Una vez que se ha declarado un fichero es posible realizar diferentes operaciones como son la lectura, la escritura o la propia comprobación de fin de archivo. Estas operaciones pueden ser realizadas a través de unos subprogramas que se detallan a continuación.

```
PROCEDURE read(FILE f: tipo_archivo; value: OUT tipo);
PROCEDURE write(FILE f: tipo_archivo; value: IN tipo);
PROCEDURE endfile(FILE f: tipo_archivo) RETURN boolean;
```

El siguiente código muestra un ejemplo de como utilizar estos procedimientos.

```
TYPE type_arch IS FILE OF integer;
FILE arch : type_arch OPEN read_mode IS "arch.txt";
VARIABLE n : integer;
            .....
WHILE NOT endfile(arch) LOOP
```

```
    read(arch, n);
        ....
END LOOP;
```

Ficheros de texto

Trabajar con ficheros es útil, pero hay que recordar que VHDL codifican los datos de forma binaria, lo que implica que es muy poco legible. Por lo tanto, debe haber una manera de *traducir* los datos de forma que el lenguaje pueda entenderlos, y para el usuario sea fácil de escribir dichos archivos. El paquete *textio* permite la conversión de tipos.

Bancos de pruebas

En VHDL es posible describir modelos para la simulación. Estos modelos no tienen demasiadas restricciones, necesitando unicamente un intérprete de las instrucciones VHDL. En cambio, en la síntesis se añaden una cantidad de restricciones como pueden ser aquellas que tienen que ver con las del tiempo, ya que no es posible aplicar retardos a la hora de diseñar un circuito.

Retrasos

El retraso es uno de los elementos más importantes de la simulación, puesto que el comportamiento de un circuito puede cambiar dependiendo del cambio de las diferentes señales. Cuando se realiza una asignación se produce de forma inmediata, puesto que no se ha especificado ningún retraso. Este comportamiento puede ser alterado mediante la opción **AFTER** cuando se asigna un valor a una señal. Su sintaxis corresponde a la siguiente línea.

```
señal <= valor AFTER tiempo;
```

Donde *tiempo* es un valor de tiempo indicado en us, ns, ms, ... Un ejemplo puede ser el siguiente.

```
rst <= '0' AFTER 15 ns;
```

Pero esta sentencia es mucho más compleja, por ejemplo se puede asignar inicialmente un valor y modificarlo posteriormente o incluso que sea modificado cada cierto tiempo.

```
signal clk : std_logic := '0';
      ....
rst <= '1', '0' AFTER 15 ns; -- Inicialmente rst=1, despues de 15 ns rst=0
clk <= not clk AFTER 5 ns; -- Cada 5 ns clk cambia de valor
```

También es posible introducir una espera entre dos sentencias mediante la palabra reservada **WAIT**. La sintáxis de esta operación e más compleja que la anterior.

```
WAIT [ON lista | UNTIL condicion | FOR tiempo];
```

Lista corresponde a una lista de sensibilidad de señales, es decir, se permanecerá en espera hasta que se produzca un cambio en alguna de las señales de la lista. *Condición* se trata de una espera indeterminada hasta que la condición sea verdadera. Por último, el *tiempo* es un valor definido en una unidad de tiempo. A continuación se muestra un ejemplo de las tres esperas posibles.

```
-- WAIT ON
stop <= '1';
WAIT ON semaforo; -- Hasta que semaforo no cambie se permanecerá en el WAIT
stop <= '0';

   ....

-- WAIT UNTIL
ack <= '1';
WAIT UNTIL clk'event and clk = '1'; -- Hasta que no exista un evento de clk y sea uno se permanece en WAIT
ack <= '0';

   ....

-- WAIT FOR
start <= '0';
WAIT FOR 50 ns; -- Se espera 50 ns
start <= '1';
```

Niveles lógicos

Normalmente existen tres niveles lógicos: 0, 1 y X, donde éste último se refiere a un valor desconocido (puede estar a alto o bajo nivel, no se sabe cuál de los dos).

También existen las llamadas fuerzas. *S* equivale a la salida obtenida de una conexión a alimentación o a tierra a través de un transistor. *R* es idéntico al anterior, pero su obtención es a través de una resistencia. *Z* también es igual a las anteriores, pero esta vez es cuando la señal es obtenida mediante una alta impedancia. Por último, se encuentra *I*, que indica que no se sabe qué fuerza existe en el bus.

El tipo *std_logic* amplía estas fuerzas, incluyendo *U* y -. La primera indica que una señal no ha sido inicializada y la segunda que no importa el valor que se ponga.

Notificaciones

En la simulación de circuitos es interesante el uso de las notificaciones, gracias a ellas se pueden advertir que señales han sido activadas o incluso comprobar si una señal ha tomado un valor determinado. El uso de notificaciones se realiza mediante **ASSERT**, seguida de una condición como elemento de activación. La sentencia puede utilizarse tanto en entornos concurrentes como en serie.

```
ASSERT <condición>
  [REPORT <expresión>]
  [SEVERITY <expresión>];
```

Si la condición **no** se cumple aparecerá en la pantalla del simulador el mensaje que se ha especificado y el nivel de gravedad, ambos son opcionales y en el caso de no indicar ningún mensaje aparecerá *"Assertion Violation"*. Los diferentes niveles de gravedad pueden ser (de menor a mayor): *note*, *warning*, *error* (por defecto) y *failure*. Dependiendo del nivel de gravedad la simulación puede detenerse.

A continuación se muestran una serie de ejemplos de esta sentencia.

```
ASSERT adrr = X"00001111";
          ....
ASSERT addr = X"10101010" REPORT "Direccion Erronea";
          ....
ASSERT addr > X"00001000" and addr < X"00002000" REPORT "Direccion correcta" SEVERITY note;
          ....
ASSERT addr < X"00001000" and addr > X"00002000" REPORT "Direccion incorrecta" SEVERITY warning;
```

Descripción de un banco de pruebas

Una de las partes más importantes en el diseño de cualquier sistema son las pruebas para la verificación del funcionamiento de un sistema. Con las metodologías tradicionales la verificación sólo era posible tras su implementación física, lo que se traducía en un alto riesgo y coste adicional. Lo más sencillo es cambiar las entradas para ver cómo son las salidas, en una herramienta de simulación, siempre que su diseño sea sencillo, en caso contrario lo más cómodo sería crear un *banco de pruebas*.

Un banco de pruebas es una entidad sin puertos, cuya estructura contiene un componente que corresponde al circuito que se desea simular y la alteración de las diferentes señales de entrada a dicho componente, para poder abarcar un mayor número de casos de prueba. Es recomendable realizar ls descripción del banco de pruebas de un sistema a la vez que se describe su diseño. Las siguientes líneas muestra la sintaxis de un banco de pruebas.

```
ENTITY nombre_test IS
END nombre_test;
ARCHITECTURE test OF nombre_test IS
  -- Declaraciones
BEGIN
  -- Cuerpo de las pruebas
END test;
```

A continuación se muestran las diferentes metodologías que se pueden llevar a cabo para la realización de un banco de pruebas. Para su explicación se utilizará un ejemplo de un diseño muy sencillo, donde un vector es rotado un bit hacia la izquierda o derecha dependiendo de la entrada *met*, su entidad corresponde al siguiente trozo de código.

```
ENTITY round IS
  clk, rst, met : IN std_logic;
  e : IN std_logic_vector(3 DOWNTO 0);
  s : OUT std_logic_vector(3 DOWNTO 0)
END round;
```

Método tabular

Para verificar la funcionalidad de un diseño se debe elaborar una tabla con las entradas y las respuestas que se esperan a dichas entradas. Todo ello se deberá relacionar mediante código VHDL. Las siguientes líneas muestran un ejemplo con el diseño que se expuso anteriormente.

```
USE work.round;

ENTITY test_round IS
END test_round;
```

```
ARCHITECTURE test OF test_round IS
  SIGNAL clk, rst, met : std_logic;
  SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
  TYPE type_test IS RECORD
    clk, rst, met : std_logic;
    e, s : std_logic_vector(3 DOWNTO 0);
  END RECORD;
  TYPE lista_test IS ARRAY (0 TO 6) OF type_test;
  CONSTANT tabla_test : lista_test :=(
    (clk=>'0', rst =>'1', met=>'0', e=>"0000", s=>"0000"),
    (clk=>'1', rst =>'0', met=>'0', e=>"0000", s=>"0000"),
    (clk=>'1', rst =>'0', met=>'0', e=>"0001", s=>"0010"),
    (clk=>'1', rst =>'0', met=>'0', e=>"1010", s=>"0101"),
    (clk=>'1', rst =>'0', met=>'1', e=>"0000", s=>"0000"),
    (clk=>'1', rst =>'0', met=>'1', e=>"0001", s=>"1000"),
    (clk=>'1', rst =>'0', met=>'1', e=>"1001", s=>"1100"));
BEGIN
  r : ENTITY work.round
  PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

  PROCESS
    VARIABLE vector : type_test;
    VARIABLE errores : boolean := false;
  BEGIN
    FOR i IN 0 TO 6 LOOP
      vector := tabla_test(i);
      clk<=vector.clk; rst<=vector.rst; met<=vector.met; e<=vector.e;
      WAIT FOR 20 ns;
      IF s /= vector.s THEN
        ASSERT false REPORT "Salida incorrecta" SEVERITY error;
        errores:=true;
      END IF;
    END LOOP;
    ASSERT errores REPORT "Test OK" SEVERITY note;
    WAIT;
  END PROCESS;
END test;
```

Uso de ficheros (vectores de test)

En el caso anterior los casos de prueba y el código de simulación permanecían juntos, pero es posible separarlos de forma que, por un lado se encuentren las pruebas y por otro el código. Esto es posible ya que VHDL dispone de paquetes de entradas/salida para la lectura/escritura en ficheros de texto, como ya se comento el paquete *textio* dispone de los subprogramas necesarios para el acceso a dichos ficheros.

Supónganse los casos de prueba desarrollados en el caso anterior, en el siguiente fichero de texto se han escrito los vectores de prueba:

```
clk  rst  met   e     s
 0    1    0   0000  0000
 1    0    0   0001  0010
 1    0    0   1010  0101
 1    0    1   0000  0000
 1    0    1   0001  1000
 1    0    1   1001  1100
```

A continuación se muestra el código relacionado con la simulación, en él se incluye el acceso al fichero anterior que contiene los diferentes vectores de test.

```
    USE std.textio.ALL; -- No es necesario porque se incluye por defecto
    USE work.round;

    ENTITY test_round IS
    END test_round;

    ARCHITECTURE test OF test_round IS
      SIGNAL clk, rst, met : std_logic;
      SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
    BEGIN
      r : ENTITY work.round
      PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

      PROCESS
        FILE vector_test : text OPEN read_mode IS "test.txt";
        VARIABLE errores : boolean := false;
        VARIABLE vector : line;
        VARIABLE clk_tmp, rst_tmp, met_tmp : std_logic;
        VARIABLE e_tmp, s_tmp : std_logic_vector(3 DOWNTO 0);
      BEGIN
        readline(vector_test,vector); -- Lee los nombres (la primera linea)
        WHILE NOT endfile(vector_test) LOOP
          readline(vector_test,vector);
          read(vector,clk_tmp);
          read(vector,rst_tmp);
          read(vector,met_tmp);
          read(vector,e_tmp);
          read(vector,s_tmp);
```

```
        clk <= clk_tmp; rst <= rst_tmp; met <= met_tmp; e <= e_tmp;
        WAIT FOR 20 ns;
        IF s_tmp /= s THEN
          ASSERT false REPORT "Salida incorrecta" SEVERITY error;
          errores:=true;
        END IF;
      END LOOP;
      file_close(vector_test);
      ASSERT errores REPORT "Test OK" SEVERITY note;
      WAIT;
    END PROCESS;
  END test;
```

Metodología algorítmica

Existe otro tipo de test, los cuales se basan en realizar algoritmos para cubrir el mayor número de casos posibles. A continuación se muestra un ejemplo, el cual aplica esta metodología.

```
  USE work.round;

  ENTITY test_round IS
  END test_round;

  ARCHITECTURE test OF test_round IS
    SIGNAL clk : std_logic := '0';
    SIGNAL rst, met : std_logic;
    SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
  BEGIN
    clk <= NOT clk after 10 ns;

    r : ENTITY work.round
    PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

    PROCESS
    BEGIN
      rst <= '1'; met <= '0'; e <= "0000";
      WAIT FOR 20 ns;
      ASSERT (s="0000") REPORT "Error en reset" SEVERITY error;

      rst <= '0';

      e <= "0000";
      WAIT FOR 20 ns;
      ASSERT (s="0000") REPORT "Error desplazamiento izquierda" SEVERITY error;

      e <= "0001";
      WAIT FOR 20 ns;
      ASSERT (s="0010") REPORT "Error desplazamiento izquierda" SEVERITY error;
```

```
        e <= "1010";
        WAIT FOR 20 ns;
        ASSERT (s="0101") REPORT "Error desplazamiento izquierda" SEVERITY error;

        met <= '1';

        e <= "0000";
        WAIT FOR 20 ns;
        ASSERT (s="0000") REPORT "Error desplazamiento derecha" SEVERITY error;

        e <= "0001";
        WAIT FOR 20 ns;
        ASSERT (s="1000") REPORT "Error desplazamiento derecha" SEVERITY error;

        e <= "1001";
        WAIT FOR 20 ns;
        ASSERT (s="1100") REPORT "Error desplazamiento derecha" SEVERITY error;

        ASSERT false REPORT "Test Finalizado" SEVERITY note;
        WAIT;
      END PROCESS;
    END test;
```

Ejemplos

En este capítulo se mostrarán ejemplos de diseños lo más completos posible, van desde la descripción del problema hasta su simulación, se deja al lector si dispone del hardware apropiado su síntesis y verificación real, así como su simulación mediante el software apropiado.

1. Puerta triestado
2. Multiplexor
3. Sumador
4. Contador
5. Biestable-Latch
6. Máquinas de estados
7. ALU

Fuentes y contribuyentes del artículo

Elementos básicos del lenguaje *Fuente*: http://es.wikibooks.org/w/index.php?oldid=192443 *Contribuyentes*: Julian.caba, Rafa, Swazmo, 11 ediciones anónimas

Entidad *Fuente*: http://es.wikibooks.org/w/index.php?oldid=211710 *Contribuyentes*: BalDYxan, Julian.caba, MarcoAurelio, Rafa, 8 ediciones anónimas

Arquitectura *Fuente*: http://es.wikibooks.org/w/index.php?oldid=212921 *Contribuyentes*: Julian.caba, Kranfix, MarcoAurelio, Rafa, Rrmsjp, 22 ediciones anónimas

Organización del código *Fuente*: http://es.wikibooks.org/w/index.php?oldid=195564 *Contribuyentes*: Julian.caba, 5 ediciones anónimas

Otros conceptos *Fuente*: http://es.wikibooks.org/w/index.php?oldid=135216 *Contribuyentes*: Julian.caba, 3 ediciones anónimas

Bancos de pruebas *Fuente*: http://es.wikibooks.org/w/index.php?oldid=172131 *Contribuyentes*: Der Künstler, Julian.caba, MarcoAurelio, 10 ediciones anónimas

Ejemplos *Fuente*: http://es.wikibooks.org/w/index.php?oldid=137121 *Contribuyentes*: Julian.caba, MarcoAurelio, Rafa

Licencia