4.3. Lenguaje VHDL

El lenguaje VHDL (Very High Speed Integrates Circuit Hardware Description Language) es un lenguaje estándar utilizado para la descripción de los sistemas digitales. En este apartado se presentan los conceptos básicos para realizar la descripción de un sistema digital.

4.3.1 Elementos de descripción

Un circuito o subcircuito descrito mediante VHDL se denomina *diseño de entidad* (*design entity*). Está compuesto por dos partes: la declaración de la entidad, *entity*, (donde se declaran las señales de entrada y salida, por lo tanto es el modelo de interfaz con el exterior) y la arquitectura, *architecture* (donde se definen los detalles del circuito, es decir, es la especificación del funcionamiento de una entidad).

Por otro lado están las bibliotecas, *libraries*, donde se almacenan los elementos de diseño: tipos de datos, operadores, componentes, funciones, etc...

Hay dos bibliotecas que siempre son visibles por defecto: *std* (la estandard) y *work* (la de trabajo) y que no es necesario declarar.

Los elementos de las librerías se organizan en paquetes (*Packages*) y hay que declararlos para poder utilizarlos.

4.3.2 Unidades de diseño y su sintaxis

Declaración de Packages

La forma general para incluir un *Package* es:

```
LIBRARY library_name;
USE library_name.package_name.all
```

Library_name puede ser una existente en el sistema o creada por el usuario. La sintaxis de declaración es:

```
PACKAGE package_name IS
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package_name;
```

Declaración de *entity*.

En la declaración de entidades, se definen las entradas, salidas y tamaño de un circuito, explicitando cuáles son, de qué tamaño (de 0 a n bits), modo (entrada, salida, ...) y tipo (integer, bit,...) .

`entity circuito_a is`	Cabecera del programa
`port(`	Indica que a continuación vienen los puertos (o grupos señales) de entrada y/o salida
-- puertos de entradas -- puertos de salidas -- puertos de I/O -- puertos de buffers	Se declaran las entradas y/o salidas con la sintaxis que corresponda. Las líneas que comienzan por dos guiones son ignoradas por el compilador. El compilador no distingue las mayúsculas de las minúsculas
`);` `end circuito_a;`	La declaración de puertos de entrada y/o salida ha finalizado y por lo tanto la entidad también.

Cada señal en una declaración de entidad está referida a un puerto (o grupo de señales), el cual es análogo a un(os) pin(es) del símbolo esquemático. Un puerto es un objeto de información, el cual, puede ser usado en expresiones y al cual se le pueden asignar valores.

A cada puerto se le debe asignar un nombre válido. Seguido del nombre del puerto y separado de éste por dos puntos, se debe indicar el *modo.* Este, describe la dirección en la cual la información es transmitida a través del puerto: *in (leer)*, *ou*t (escribir), *buffer* e *inout (leer y escribir).* Si no se especifica nada, se asume que el puerto es del modo *in*. También hay que indicar el *tipo de dato,* que define el conjunto de valores que puede tomar una señal. Los datos pueden ser:

a) Escalares: sólo pueden tener un valor asociado

b) Matriciales (arrays): varios elementos mismo tipo

c) Registros (records): varios elementos y distintos tipos

Modos posibles en los *ports*:

- Modo ***in***: Un puerto es de modo *in* si la información correspondiente al mismo, entra a la entidad.
- Modo ***out***: Un puerto es de modo *out* si la información fluye hacia fuera de la entidad. Este modo no permite realimentación ya que al declarar un puerto como *out* se está indicando al compilador que el estado lógico en el que se encuentra no es legible. Esto le da una cierta desventaja pero a cambio consume menos recursos de los dispositivos lógicos programables.
- Modo ***buffer***: Es usado para una realimentación interna ,es decir, para usar este puerto como un driver dentro de la entidad. Este modo es similar al modo *out*, pero además, permite la realimentación y no es bidireccional, y solo puede ser conectado directamente a una señal interna, o a un puerto de modo *buffer* de otra entidad. Una aplicación muy común de este modo es la de salida de un contador, ya que se debe conocer la salida en el momento actual para determinar la salida en el momento siguiente.
- Modo ***inout***: Es usado para señales bidireccionales, es decir, si se necesita que por el mismo puerto fluya información tanto hacia dentro como hacia afuera de la entidad. Este modo permite la realimentación interna y puede reemplazar a cualquiera de los modos anteriores, pudiéndose usar este modo para todos los puertos.

Tipos Escalares:

- Tipo ***boolean***: puede tomar dos valores: verdadero/true o falso/false.
- Tipo ***bit***: Puede tomar dos valores: 0 ó 1
- Tipo ***integer***: Representa un número binario. Por defecto es un númeor de 32 bits (incluidos negativos). Puede reducirse el rango utilizando *Range*.

  SIGNAL X: INTEGER RANGE –127 TO 127

  Por lo tanto X es de bits.

Se utiliza en operaciones aritméticas. En su lugar puede usarse std_logic_vector.

- Constant data object: su valor no puede cambiar.

  CONSTANT constant_name : type_name :=constant_value

- Tipo ***bit_vector***: Es un vector de bits. Hay que definir el peso de los bits que lo integran: *downto* (el bit más significativo es el número más alto) o *to*.

`numero : bit_vector (0 to 7);`	El MSB es numero(0) y numero(7) el LSB
`numero : bit_vector (7 downto 0);`	El MSB es numero(7) y numero(0) el LSB

- Tipo std_logic: es más flexible que el tipo BIT ya que admite los siguientes valores: 0, 1, Z (alta impedancia), - (no importa), L, H, U, X y W, aunque solamente son útilies para la síntesis de circuitos digitales los cuatro primeros.
- Tipo std_logic_vector: representa un array de objetos std_logic. Puede utilizarse como número binario en circuitos aritméticos.

Para poder utilizar estos dos últimos tipos, hay que incluir la librería ieee y los packages:

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

- Tipos de datos enumerados: Son tipos de datos escalares definidos por el usuario, pueden definirse con caracteres o con unos nombres literales elegidos a conveniencia

TYPE nombre_tipo IS definición_de_tipo;

```
ARCHITECTURE rtl OF fsm IS
TYPE estado IS (reset,libre,ocupado);
SIGNAL anterior, siguiente: estado;
BEGIN
--Sentencias de la arquitectura
....
END rtl;
```

Para utilizar este tipo de datos hay que declararlos en un paquete y utilizar la biblioteca que corresponda.

Tipos Matrices (*Arrays*):

Un arrray es una colección de datos del mismo tipo.

TYPE bit_vector **IS ARRAY** (rango) **OF** bit;

TYPE std_logic_vector **IS ARRAY** (rango) **OF** std_logic;

Se pueden definir arrays de cualquier tipo por el usuario:

TYPE bit **IS ARRAY** (7 DOWNTO 0) **OF** std_logic;

SIGNAL X: Byte;

Declara la señal *X* con el tipo *byte*.

Declaración de la Arquitectura (*architecture*)

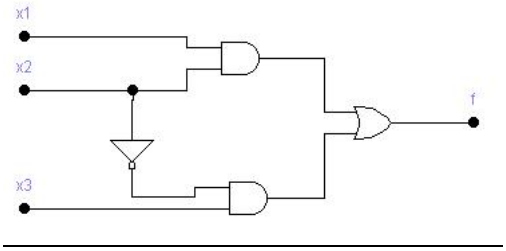
Define la funcionalidad de una entidad y está dividida en dos partes: región de declaración (*declarative region*) y el cuerpo de la arquitectura (*architecture body*).

En la declaración, se pueden declarar señales, tipos definidos por el usuario y constantes. También componentes y atributos. Anticipan términos que van a aparecer.

La funcionalidad se especifica en le cuerpo de la arquitectura después de BEGIN.

```
ARCHITECTURA architecture_name OF entity_name IS
        [SIGNAL declarations]
        [CONSTANT declarations]
        [TYPE declarations]
        [COMPONENT declarations]
        [ATTRIBUTE declarations]
BEGIN
        {COMPONENT instantiation statement;}
        {CONCURRENT ASSIGNMENT statement;}
        {PROCESS statement;}
        {GENERATE statement;}
END [architecture_name];
```

Ejemplo: Circuito AND_OR




```
ENTITY ejemplo1 IS
        PORT (x1, x2, x3  : IN BIT;
                f             : OUT BIT);
END ejemplo1;
ARCHITECTURE LogicFunc OF ejmeplo1 IS
BEGIN
        ft <= (x1 AND x2) OR (NOT x2 AND x3);
END LogicFunc;
```

Ejemplo: Full adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY fulladd IS
        PORT (Cin, x, y : IN STD_LOGIC;
                s, Cout   : OUT STD_LOGIC);
END fulladd;
ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
        s <= x XOR y XOR Cin;
        Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin);
END LogicFunc;
```

4.3.3 Operadores

VHDL utiliza algunos símbolos especiales con funciones diferentes y específicas, tales como el símbolo "+" se utiliza para representar la operación suma y, en este caso, es un operador.

El símbolo "- -" es empleado para los comentarios realizados por el usuario, de tal forma que el programa al encontrar una instrucción precedida por "- -" la saltará ignorando su contenido. De esta forma, el programador puede hacer más comprensible el código del programa.

Los símbolos especiales en VHDL son:

OPERADORES LÓGICOS	NOT, AND, OR, NAND, NOR, XOR	Tipo de operador: boolean Tipo de resultado: boolean
OPERADORES RELACIONALES	= / < <= > >=	Tipo de operador: cualquier tipo Tipo de resultado: boolean
OPERADORES ARITMÉTICOS	+ - * / ** MOD, REM, ABS	Tipo de operador: integer, real, signal Tipo de resultado: integer, real,signal
OPERADOR CONCADENACIÓN	&	Tipo de operador: array tipo de resultado: array

El símbolo " ; " se utiliza para finalizar todas y cada una de las líneas del código dando por terminada dicha sentencia en el programa.

Ejemplo: $f = \left(x_1 \overline{x}_6 + x_2 x_7\right)\left(x_3 + x_4 x_5\right)$

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

ENTITY ejemplo2 IS
        PORT (x1, x2, x3, x4, x5, x6, x7  : IN STD_LOGIC;
                f              : OUT STD_LOGIC);
END ejemplo2;

ARCHITECTURE LogicFunc OF ejemplo2 IS
BEGIN
```

```
    f <= (x1 AND x3 AND NOT x6) OR
          (x1 AND x4 AND x5 AND NOT x6) OR
          (x2 AND x3 AND x7) OR
          (x2 AND x4 AND x5 AND x7);
END LogicFunc;
```

4.3.4 Gramática del lenguaje

En la mayoría de los lenguajes de descripción, la ejecución del programa se lleva a cabo de arriba a abajo, es decir siguiendo el orden en el que se hayan dispuesto las sentencias en el programa, por ello es de vital importancia la disposición de las mismas dentro del código fuente.

VHDL lleva a cabo las asignaciones a señales dentro del cuerpo de un proceso (*process*) de forma secuencial, con lo que el orden en el que aparezcan las distintas asignaciones será el tenido en cuenta a la hora de la compilación.

Asignación a una señal: la señal no cambia su valor hasta que se ha evaluado el proceso en el cual se incluye. Para hacer una asignación a una señal deberemos usar el operador <=, estando la señal a asignar a la izquierda y el valor que debe tomar a la derecha.

```
signal <= signal1 + signal2;
```

```
process
begin
a <= b;
b <= a;
wait on a,b;
end process;
```

```
La señal a tendrá el valor b
La señal b tendrá el valor a

Se actualizan los cambios
AQUÍ
```

Asignación a una variable: A diferencia de las señales, la asignación de un valor a un variable, no tiene un retardo asociado, de manera que la variable toma el nuevo valor justo en el momento de la asignación, de forma que las sentencias que vengan a continuación, la variable recién asignada, tendrá el nuevo valor. De esta forma, el ejemplo expuesto para señales, al ser usado para variables, no se consigue el mismo resultado:

```
a := b;
b := a;
```

a toma el valor **b**
b toma el NUEVO valor de **a** (el de **b**)

Sentencia IF

```
if (condición) then
  Realiza una acción;
else
realiza otra acción diferente;
end if;
```

También:

```
if (condición) then
 Realiza una acción;
elsif (otra condición) then
 realiza otra acción diferente;
else
 realiza otra acción diferente;
end if;
```

Sentencia CASE

```
case (señal a evaluar) is
  when (valor 1) => Realiza una acción;
  when (valor 2) => Realiza otra acción;
  ...
  when (último valor) => Realiza otra acción;
end case;
```

```
case control is          se evalúa la señal control
  when "00" => d <= a;   si control vale "00" entonces d<=a
  when "01" => d <= b;   si control vale "01" entonces d<=b
  when "10" => d <= c;   si control vale "10" entonces d<=c
  when others => d <=    si control no toma ningún valor de
"1111";                  los especificados, toma el valor
                         "1111"
end case;                finaliza la arquitectura
```

Sentencia LOOP

La sentencia *loop* (ciclo en castellano) se usa para ejecutar un grupo de sentencias un número determinado de veces, y consiste tanto en un ciclo *for* como en un ciclo *while*.

```
process (a)
begin
ciclo1: for i in 7 downto 0
loop                              Cabecera del ciclo
    entrada(i) <= ( others => Instrucciones a ejecutar 8
'0' )                         veces
end loop;                     Finalización del ciclo
end process;
```

```
process (a)
variable i: integer := 0;
begin
ciclo2: while i < 7 loop
   entrada(i) <= (others =>   Mientras i sea menor que 7 =>
'0');                         ciclo
   i := i + 1;
end loop;
                              Finaliza el ciclo
end process;
```

Sentencia EXIT

La sentencia **exit** permite salir de un *loop* si se alcanza una condición fijada por el programador.

```
process (a)
begin
ciclo1: for i in 7 downto 0 loop
  if a'length < i then exit ciclo1;
  entrada(i) <= ( others => '0' );
end loop;
end process;
```

Sentencia NEXT

La sentencia **next** también debe estar dentro de un ciclo *loop* y sirve para no ejecutar una o más de las instrucciones programadas.

```
process (a)
begin
ciclo1: for i in 7 downto 0 loop                Cabecera del ciclo
    if i=4 then next;                           Si i vale 4, se salta el ciclo
    else                                        Si no vale 4,...
    entrada(i) <= ( others => '0' );            ... se inicializa entrada
   end if;
end loop;                                       Finaliza el ciclo
end process;
```

Sentencia NULL

La sentencia *null* se utiliza, al igual que en otros lenguajes de programación, para que dada una condición especial no ocurra ninguna acción, es decir, que ninguna señal o variable modifique su valor, y que el programa siga su curso habitual. Su comportamiento dentro de un *loop* es similar al de la sentencia ***next***.