# Lenguaje descripción de hardware: VHDL

## Generalidades

A partir del desarrollo de circuitos Integrados Digitales programables con una gran cantidad de componentes lógicos y la necesidad de sistema digitales para aplicaciones mas complejas, las herramientas de diseño tradicionales se vuelven cada vez más ineficientes y poco efectivas para lograr desarrollos adecuados, por lo tanto las empresas fabricantes de circuitos integrados desarrollan herramientas más útiles, originándose así los HDL ó Lenguajes de Descripción de hardware. Así cada empresa crea el suyo estableciendo una diversidad de lenguajes muy grande.

Para tratar de unificar estas herramientas, entre los años 1984 y 1987, el IEEE y el Departamento de Defensa de los Estados Unidos (DoD) patrocinan el desarrollo de un Lenguaje llamado VHDL. Su nombre viene de VHSIC HDL, o sea Lenguaje de Descripción de hardware para circuitos integrados de muy alta velocidad.

Considerando que un lenguaje de descripción de hardware es una herramienta formal que permite describir la estructura y comportamiento de un sistema para lograr una adecuada especificación, documentación y simulación del mismo antes de su realización real; para su implementación se establecieron ciertas características fundamentales, que aún hoy siguen siendo válidas, y son:

- Cada elemento de diseño tiene una interfaz única y perfectamente definida, que permite conectarla a otros elementos.
- Cada elemento tiene un comportamiento preciso y unívocamente definido, que permiten su posterior simulación.
- La especificación de comportamiento que permite definir la operatividad puede realizarse a través de un algoritmo ó de una estructura de hardware real.
- Los diseños mantienen una estructura jerárquica, que permite descomponerlo adecuadamente.
- Las características concurrentes, temporizadas y de sincronismo (por ej. reloj) pueden ser modeladas adecuadamente.
- Se puede simular cualquier operación lógica y de temporización.

Se establece así una herramienta que además tiene amplias características de modelado y documentación. De esta forma cualquier circuito digital se puede especificar y simular adecuadamente.

Luego del desarrollo de este lenguaje aparecieron las herramientas adecuadas de síntesis que completan el panorama de diseño de un sistema digital. Así de esta forma se puede decir que si utiliza VHDL se puede diseñar, simular y sintetizar cualquier sistema digital, desde el combinacional más simple hasta la estructura secuencial más compleja. Las nuevas versiones de HDL permiten también el desarrollo de circuitos analógicos.

## Ambiente y Flujo de Diseño

Antes de abocarnos al desarrollo específico del lenguaje, debemos analizar el ambiente y etapas del proceso de un diseño.

En cualquier diseño basado en VHDL podemos subdividir el flujo de diseño en dos partes bien diferenciadas:

1. **Etapa de Desarrollo**: Tiene varios pasos, a saber:

    a. **Planteo General del Sistema**: Consiste en hacer un Diagrama en Blocks funcional y jerárquico del sistema a desarrollar. Definición de módulos e interfaces generales del sistema.
    b. **Codificación**: Consiste en escribir el código VHDL para todos los componentes planteados, módulos específicos e interfaces. Se trata de un simple discurso de texto, por lo que puede realizarse en cualquier editor conocido por el diseñador. Sin embargo, los ambientes de diseño incluyen un Editor VHDL especializado, por lo que el desarrollo de esta etapa es más sencilla. Estos editores incluyen características propias del lenguaje, como sangría automática, resaltado de palabras clave, verificación sintáctica, etc.
    c. **Compilación**: En esta etapa el compilador VHDL transforma el programa fuente en objeto, por lo cual analiza la sintaxis de lo escrito y verifica la compatibilidad con cualquier otro módulo ingresados como fuente del presente programa. Junto a esto produce toda la información necesaria para la posterior simulación del proyecto. Nota: A veces, en proyectos complejos conviene realizar compilaciones parciales, ganando así tiempo posterior de desarrollo.
    d. **Verificación:** Esta etapa es muy importante, pues permite establecer si el circuito obtenido funciona como se pretendió al fijar las pautas de diseño. Existen dos características a verificar, y ellas son las Funcional y la Temporal. En el primer caso se analiza el funcionamiento lógico del circuito, sin considerar el tiempo como variable. Aquí los elementos lógicos son considerados como ideales, es decir sin retardos. En la verificación temporal se analizan los resultados del circuito real, considerando retardos estimados, ya que aún no se han seleccionado los circuitos reales de síntesis. De esta manera se verifica la funcionalidad temporal de circuitos combinacionales y especialmente los secuenciales con características de memorización. La verificación funcional y temporal se realiza a través de un proceso complejo conocido como simulación, el cual permite detectar errores en el diseño obtenido, y de esta forma hacer las correcciones adecuadas antes de pasar a la etapa de síntesis. **Simulación:** Este procedimiento de verificación permite definir entradas y aplicarlas al prototipo de software, analizar el comportamiento de los diversos módulos definidos y observar las salidas. Todo esto sin tener que realizar el prototipo físico con circuitos reales. Para proyectos pequeños se pueden generar entradas y verificar salidas en forma manual; pero en grandes sistemas se crean ¨bancos de prueba" con la capacidad de establecer entradas, verificar salidas y realizar las comparativas con lo valores esperados. La simulación funcional es completa y precisa, sin embargo la temporal sólo es aproximada pues se basa en valores estimados, y sirve para establecer que vamos en el camino correcto. Esto es así pues es muy dependiente de la síntesis, dónde se establecen los circuitos específicos y es allí dónde salen los verdaderos valores de retardos de acuerdo al circuito, a la tecnología, al layout, tipo de componentes discretos, etc.

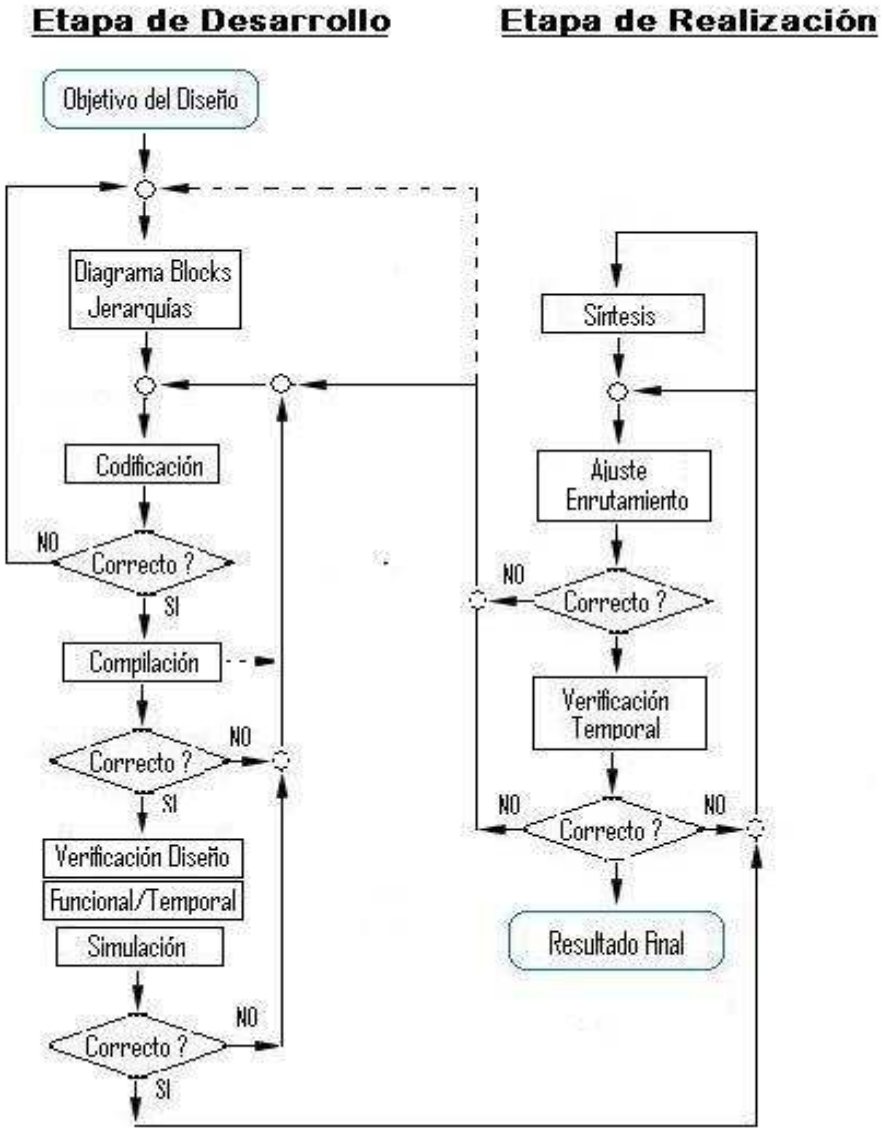
2. **Etapa de Realización**: Las características y herramientas son algo diferentes de la anterior. Tiene varios pasos, a saber:

    a. **Sintesís**: Convierte el modelo descripto por VHDL en un conjunto de primitivas ó componentes que se articularán en un circuito real en una tecnología adecuada. Generalmente estas herramientas

presentan la posibilidad de establecer filtros, premisas ó restricciones específicas del tipo de circuito ó tecnología. Por ejemplo en los circuitos lógicos programables, como PAL, GAL, CPLD, FPGA las **herramientas de síntesis** pueden generar ecuaciones booleanas; y en el caso de ASIC ó ASSP generar una lista de compuertas y la malla que fija el cableado correspondiente de interconexión.

b. **Ajuste y Enrutamiento**: Son herramientas que mapean las ecuaciones ó componentes de acuerdo a los recursos disponibles de cada dispositivo. Como en el paso anterior, el diseñador puede especificar restricciones ó asignaciones específicas para lograr el módulo correcto.

**c.** **Verificación temporal y total del circuito**: Este es la etapa crucial ya que establece la funcionalidad temporal correcta basada en todos los parámetros reales introducidos en el diseño, como circuitos integrados reales, longitud de los buses incorporados, longitud de los alambres, cargas de los elementos físicos, etc. Generalmente se aplican las mismas señales ó bancos de prueba utilizadas en la etapa de desarrollo.



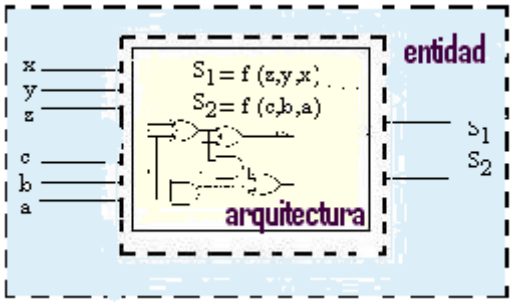
Como vemos en el diagrama de flujo de diseño planteado, aparecerán muchos avances y retrocesos, hasta lograr el diseño adecuado. Lo importante es no dejarse llevar solo por el uso de herramientas automáticas, sino poner en cada paso la experiencia adquirida en los diseños anteriores.

## Programación VHDL: Estructura

Este lenguaje está diseñado con principios de programación estructurada basado en Pascal y ADA. Damos las siguientes definiciones:

- *Entidad VHDL:* Declaración de los vectores del módulo, generalmente entradas y salidas. En otras palabras, podemos decir que se trata de identificador Vectorial del sistema.
- *Arquitectura VHDL:* Descripción detallada de la estructura interna ó comportamiento del módulo.

Observado el siguiente diagrama en blocks,




vemos que la ***entidad*** implica los lazos que tiene un sistema para ser excitado y entregar información a otros, es decir, implica todo lo externo del diagrama en blocks, mientras que la **arquitectura** describe el interior del bloque. En el desarrollo jerárquico se definen como entidades de nivel bajo ó inferior a aquellas que no usan de otras entidades para describir su comportamiento, y entidades de nivel superior a aquellas que hacen uso de las de menor grado ó nivel.

Para realizar un circuito a través de VHDL se deben declarar ambas características, la entidad y la arquitectura. Por lo tanto, un programa VHDL es un archivo de texto dónde se declara en primer lugar la entidad y luego se define la arquitectura, como vemos en la fig Nº 3, para una compuerta AND.

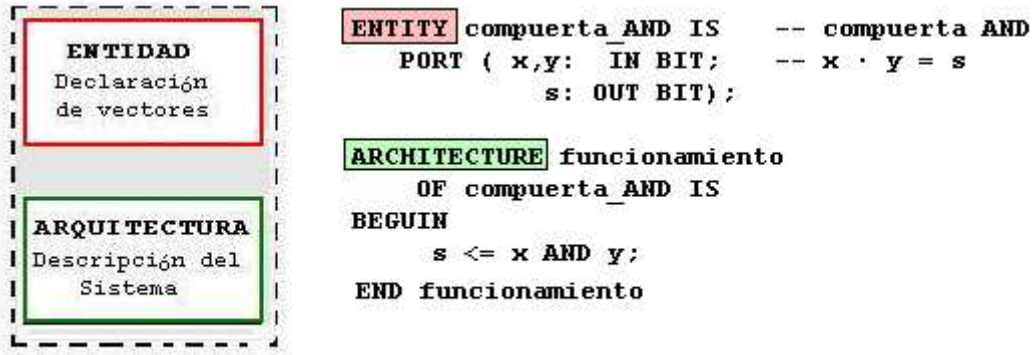



Fig. Nº 3: Estructura del archivo del programa VHDL y Programa VHDL de una compuerta AND

## Descripción General

- VHDL ignora los saltos de línea y espacios, por lo tanto el contexto formal de la escritura del programa depende de la mejor inteligibilidad del mismo.

- Comentarios: Comienzan con dos guiones – y finaliza automáticamente con el final de línea.

- Palabras clave, especiales ó reservadas: Son cadenas de caracteres especiales reservados por VHDL para su sintaxis. En el ejemplo pueden leerse varias ENTITY, PORT, IN, OUT, ACHITECTURE, ETC.

- Identificadores: Son aquellas cadenas de caracteres que el diseñador utiliza para nominar o identificar algo del diseño. En nuestro ejemplo: X, Y , Z, BIT, Compuerta, etc. Hay algunos identificadores especiales que se usan para ciertos tipos de definiciones. Es el caso de "BIT" en este ejemplo.

- Las palabras clave y los identificadores no reconocen diferencia entre mayúsculas y minúsculas.

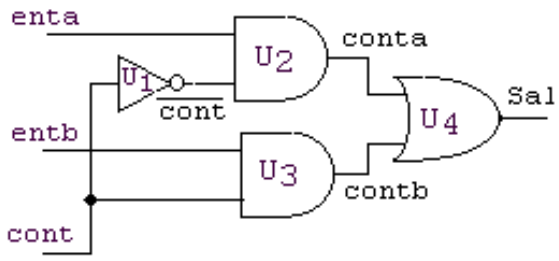
- Como mencionamos, hay tres tipos de descripción. Planteamos el desarrollo general, para el siguiente sistema lógico:

  - Planteo verbal: circuito lógico de dos entradas y una salida, que a condición de una entrada de control, deja pasar a la salida una u otra de las entradas.

  - Desarrollo circuital por el método tradicional:

De acuerdo a lo enunciado, tenemos la siguiente resolución circuital:

| cont | enta | entb | Sal |
|---|---|---|---|
| 0 | X | a | a |
| 1 | b | X | b |




$Sal_{(cont,enta,entb)} = \overline{cont} \cdot enta + cont \cdot entb$

- En VHDL todas las versiones tendrán la misma entidad:

```
ENTITY mux2a1 IS
   PORT (enta, entb, cont : IN  BIT ;
                     sal  : OUT BIT);

   END mux2a1;
```

Y veamos ahora en forma simple los tres tipos de descripciones previstos. cabe indicar que se trata solo de una presentación preliminar, y como aún no sabemos la sintaxis específica es posible que algunos detalles no los entendamos.

## 1 - Descripción Estructural

```
Arquitectura
    Declaración
    caracteristicas
Descripción
    Enunciado estructural
    del circuito a describir
Final
```

```
ARCHITECTURE dmux_est OF mux2a1 IS
  COMPONENT  INV
    PORT (I : in  BIT ;
          O : out BIT ;
    END Component
  COMPONENT  AND2
    PORT (I1,I2: in  BIT ;
          O : out BIT ;
    END Component
  COMPONENT  OR2
    PORT (I3,I4: in  BIT ;
          O : out BIT ;
    END Component
  SIGNAL contneg, conta, contb : BIT;
BEGIN
  U1:inversor PORT MAP(cont, cont);
  U2:AND2 PORT MAP(enta, cont, conta);
  U3:AND2 PORT MAP(entb, cont, contb);
  U4: OR2 PORT MAP(conta, contb, Sal);
END dmux est ;
```

La instrucción de presentación de componentes COMPONENT implica una descripción anterior de la estructura y funcionalidad de los mismos. También pueden introducirse a través de la librería correspondiente donde están descriptos .

Veamos aquí los correspondientes a este programa.

**Descrición de componentes**

```
ENTITY inversor IS
  PORT (I : in  BIT ;
        O : out BIT ;
  END  inversor
ARCHITECTURE finv OF inversor IS
BEGIN
   O <= NOT I
END finv
```

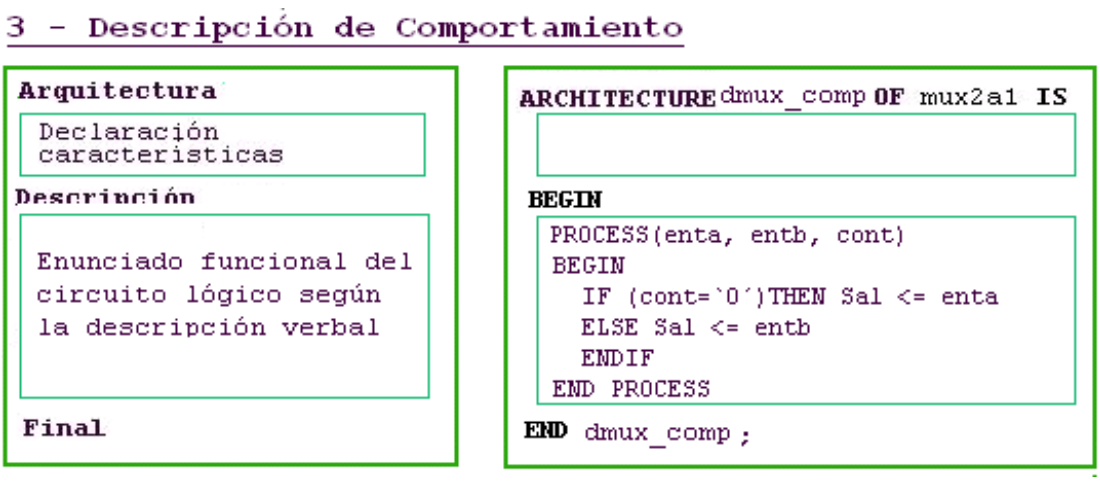
```
ENTITY AND2  IS
  PORT (I1,I2 : in BIT  ;
              O : out BIT ;
  END  AND2
ARCHITECTURE fAND2 OF AND2 IS
BEGIN
   O <= I1 AND I2
END  fAND2
```

```
ENTITY  OR2  IS
  PORT (I3,I4 : in BIT  ;
              O : out BIT ;
  END  OR2
ARCHITECTURE fOR2 OF  OR2 IS
BEGIN
   O <= I3 OR I4
END  fOR2
```

## 2 - Descripción de Flujo de Datos

```
Arquitectura
    Declaración
    caracteristicas
Descripción
    Enunciado de la ecuación
    booleana que describe el
    circuito
Final
```

```
ARCHITECTURE dmux_RTL OF mux2a1 IS
  SIGNAL contneg, conta, contb : BIT;
BEGIN
  contneg <= cont
  conta   <= enta AND cont
  contb   <= entb AND cont
     Sal <= conta OR contb
END dmux_RTL;
```




## Sintaxis General

1. **Entidad:** La declaración básica tiene el siguiente propósito:

   a. Nombrar la entidad
   b. Declaración de puertos, la cual define sus puertos ó señales de interfaz externa, que se manifiesta y representa de la sig. forma sintáctica:

```
ENTITY   nombre-entidad  IS
  PORT  (nombre-señales : mode tipo-señal ;
         nombre-señales : mode tipo-señal;
        ……..
        nombre-señales : mode tipo-señal) ;
   END   nombre-identidad ;
```

Para establecer esta declaración hay dos tipos de textos bien diferenciados:

a. Palabras clave : **ENTITY** , **IS** , **PORT** y **END.**

*b.* Palabras nominativas:

- ***nombre-entidad:*** Identificador que nombra la entidad, no puede ser palabra clave.
- ***nombre-señales:*** Identificadores que nombran las variables y funciones de la interfaz externa.
- ***modo:*** Identificador que especifica la dirección de la señal, usando 4 palabras clave, que son:
  - **IN** : Indica una señal de entrada a la entidad
  - **OUT** : Indica una señal de salida. El valor de la señal no puede ser leído dentro de la arquitectura de la entidad.
  - **BUFFER**: Indica una señal de salida. El valor de la señal si puede ser leído dentro de la arquitectura de la entidad.
  - **INOUT**: Indica una señal de entrada o salida de la entidad. Se utiliza mucho en los terminales entrada/Salida de tercer estado de los dispositivos lógicos programables.
- ***Tipo-señal:*** Identificador de la señal definida o integrada por el usuario.

2. **Arquitectura:** La declaración básica tiene por finalidad especificar la operatividad y funcionamiento de la entidad.
   Las señales de puertos externos de una arquitectura se heredan desde la parte de declaración de puertos de la correspondiente declaración de entidad, que se manifiesta y representa de la sig. forma sintáctica:

```
ARCHITECTURE nombre-arquitectura OF  nombre-entidad  IS
    declaración de señal
    declaración de constantes
    declaración de tipo
    definición de funciones
    definición de procedimiento
    declaracion de componentes
begin
    enunciado concurrente
    enunciado concurrente
    ……..
    enunciado concurrente
end nombre-arquitectura
```

teniendo en cuenta las siguientes características generales:

- ***nombre-arquitectura****:* Identificador elegido a gusto del usuario, generalmente relacionado con la entidad, pudiendo tener el mismo nombre.

- ***nombre-entidad:*** Debe ser el mismo seleccionado para nominar la entidad.

- ***declaraciones:*** son las especificaciones que definen la arquitectura, son de varias características y pueden aparecer en cualquier orden, a saber:

  - ***declaración de señal:*** Especifica la misma información que en una declaración de puerto, excepto que no se establece ningún modo.

    ***Signal*** *nombre señal : tipo señal:*

    Puede definirse desde ninguna a varias señales dentro de una arquitectura y pueden corresponder a los nodos que se identifican en un diagrama lógico.

  - ***declaración de constante:*** *En VHDL e*sta declaración sirve para dar legibilidad, mantenimiento y portabilidad a los programas, como ocurre en cualquier lenguaje, cuya sintaxis es:

    ***constant*** *nombre constante : nombre tipo := valor;* *(ver tipo más adelante)*

    Las constantes pueden utilizarse siempre que el valor se pueda necesitar. El valor de una constante puede ser una expresión ó función simple.

    *y consideremos los siguientes ejemplos:*

    ***constant*** LSB: entero:= Ancho_BUS-1; -- número de bit del LSB
    ***constant*** U: character:= *`U´;* -- Valor lógico No Inicializado
    ***constant*** Tamaño_Bus: *entero:= 16;* -- Ancho del bus

- ***declaración de tipo***: Especifica el conjunto ó intervalo de valores que un objeto puede tomar cuando se trabaja con señales, variables o constantes. También hay un conjunto de operadores asociado a un determinado ***tipos***, cuya sintaxis general es***:***

  **type** *nombre-tipo* **IS** declaración del tipo *;*

  VHDL permite también crear subtipos de un tipo determinado, como veremos en cada caso particular.

  **subtype** *nombre-subtipo* **IS** nombre tipo cominzo **to** final ;

  La declaración de tipo puede ser Predefinida ó definida por el usuario, como se indica a continuación:

  a. **Predefinida:** Son nueve, pero sólo usaremos algunos de ellos, ya que resultan más útiles las definidas por el usuario, y son:

  | | | |
  |---|---|---|
  | ***bit*** | ***bit_vector*** | ***boolean*** |
  | ***character*** | ***integer*** | ***Real*** |
  | ***Severity_level*** | ***string*** | ***time*** |

  ***Tipo Character*** ó carácter, representa el conjunto de caracteres ISO de 8 bits. De esta definición, los primeros 128 caracteres corresponden al conjunto ASCII.

  ***Tipo integer*** ó entero se define como el intervalo de enteros que hay entre $-2^{31}+1$ a $+2^{31}-1$; pudiendo cambiarse el mismo. El tipo integer tiene 2 subtipos predefinidos, a saber

  **subtype** *natural* **IS** rango entero 0 **to** mayor entero *;*

  **subtype** positivo **is** rango entero 0 **to** mayor entero *;*

  ***Tipo boolean* ó** booleano se define con dos valores true: **verdad** y false: **Falso.**

  En las siguientes tablas vemos los operadores enteros y booleanos:

  | **Operadores enteros** | | **Operadores Booleanos** | |
  |---|---|---|---|
  | + | Suma | And | AND |
  | - | Resta | Or | OR |
  | * | Multiplicación | Nand | NAND |
  | / | División | Nor | NOR |
  | Mod | Modulo Div | Xor | OR Exclusiva |
  | Rem | Mod. residuo | Xnor | NOR exclusiva |
  | Abs | Valo. absoluto | not | Complementación |
  | ** | exponenciación | | |

  b. Definidas por el usuario**:** Es la definición de tipo más usada en VHDL. Hay varios tipos definidos por el usuario.

     - Tipos **enumerados:** Son los más comunes; para lo cual corresponde la sig. sintaxis:

       **type** *nombre-tipo* **IS** *( lista valores) ;*

Lista de valores es una lista de todos los valores del tipo especificado, se enumeran en una lista separado por comas. Y también pueden incluir un subtipo, aplicando la siguiente sintaxis:

**subtype** *nombre-subtipo* **IS** nombre tipo start **to** end*;*
**subtype** *nombre-subtipo* **IS** nombre tipo start **downto** end*;*

Los valores en le subtipo, tal como se indica deben ser un intervalo contiguo de valores del tipo base, como se indica en la sintaxis del el *inicio* hasta el *final*.
Para un tipo "enumerado, se refiere a las posiciones en la lista de valores definida originalmente. El orden de un intervalo puede especificarse en forma ascendente o descendente, para lo cual se emplea **to** o **downto**.

Por ejemplo:

**subtype** *dosvalores_lógicos* **IS** rango std_logic `0 ´ **to** `1 ´ *;*

**subtype** *cincovalores_lógicos* **IS** rango std_logic `Z ´ **to** `- ´ *;*

**subtype** enteros_negativos **IS** rango enteros -2147483647 **to** -1 *;*

**subtype** tramatransmisión **IS** rango entero 255 **downto** 0 *;*

Los valores pueden ser:

1) Definidos por el usuario: Se utiliza para definir casos o estados, en el caso de una máquina secuencial.
Por ejemplo:
**type** semáforo **is** (inicializar, parar, esperar, seguir);

2) Caracteres: Cada uno de ellos es un carácter ISO encerrado en comillas simples. Se refiere a un tipo lógico Standard definido por el usuario **std_logic**. Incorpora siete valores útiles para simular una señal lógica en un circuito lógico real, no solo '0´ y `1´.

```
type  STD_ULOGIC IS  ( 'U´,  -- No inicializado
                       'X´,  -- Forzar desconocido
                       '0´,  -- Forzar 0
                       '1´,  -- Forzar 1
                       'Z´ ,  -- Alta Impedancia
                      'W´ ,  -- Débil desconocido
                      'L´ ,  -- Débil 0
                      'H´ ,  -- Débil 1
                      '_´ ,  -- Sin importancia
                     ) ;
```

- **Tipos arreglo:** Un arreglo se define como un conjunto ordenado de elementos del mismo tipo. **Indice del arreglo** permite seleccionar cada elemento del arreglo. En VHDL hay varias versiones para la sintaxis de un arreglo:

**type** *nombre-tipo* **is array** *( inicio* **to** *final)* **of** *tipo-elemento;*

**type** *nombre-tipo* **is array** *(inicio* **downto** *final)* **of** *tipo-elemento;*

Aquí se define el número total de elementos del arreglo y el intervalo posible del índice a través de los enteros *inicio* y *final.*

**type** *nombre-tipo* **is array** *(rangotipo)* **of** *tipo-elemento;*

**type** *nombre-tipo* **is array** *(rangotipo* **rango** *comienzo* **to** *final)*
**of** *tipo-elemento;*

**type** *nombre-tipo* **is array** *(rangotipo* **rango** *comienzo* **downto** *final)*
**of** *tipo-elemento;*

- ***declaración de función***: Una función VHDL acepta un cierto número de argumentos y devuelve un resultado. Estos argumentos y resultados tienen un tipo determinado. La sintaxis general es:

**function** *nombre-función (*
*nombre-señal : typo-señal ;*
*nombre-señal : typo-señal ;*
*…….*
*nombre-señal : typo-señal ;*
) **return** tipo-retorno **is**
*declaración de tipo*
*declaración de constantes*
*declaración de variables*
*definición de funciones*
*definición de procedimiento*
**begin**
*enunciado secuencial*
*enunciado secuencial*
*……..*
*enunciado secuencial*
***end*** *nombre-función*

Después de presentar el nombre, se enumeran una serie de parámetros formales que se usan dentro del cuerpo de la función. Cuando se llama a la función los parámetros reales sustituyen a los formales, y se establece que ambos parámetros deben ser del mismo tipo ó subtipo. Cuando se llama una función dentro de una arquitectura, se devuelve un valor del tipo **tipo-retorno** en lugar de la llamada de función.

Como vemos en la sintaxis indicada, una función puede definir sus propios parámetros de tipos, constantes, variables y procedimientos, y funciones anidadas. Vemos que cuando la función es invocada se ejecuta una serie de instrucciones secuenciales anidadas a través de las palabras claves **begin** y **end. Ejemplo**

**ARCHITECTURE** pulsosx*seg* ***OF*** frecuencimetro **IS**
**function** compuerta *(señal, control: BIT_VECTOR)* **return** *BIT_VECTOR* **is**
**begin**
*IF control = `1´THEN RETURN señal;*
*ELSE RETURN '0´;*
*END IF;*
***end*** *compuerta*

**begin**
*Sal <= compuerta (A,B);*
***end*** pulsosx*seg*

La palabra clave **return** indica cuando la función debería dar su resultado a quien la llamó, le sigue una expresión con el valor que se devolverá a quién la llamó.

El **tipo-retorno** debe coincidir con el tipo resultante de esta expresión.

El paquete lógico Standard IEEE 1164 define muchas funciones que operan que operan sobre los tipos Standard **std_logic** y **std_logic_vector**. El paquete especifica un número de tipos definidos por el usuario y operaciones lógicas básicas sobre estos tipos.

La definición de función, al igual que procedimiento permite declarar variable, y por ello la vemos a continuación:

- ***declaración de variable:*** *Implican una e*specificación similar a las señales, solo que no tienen correspondencia física en un circuito lógico, de hecho no la vemos declarada en la sintaxis mostrada para la definición de una arquitectura. Sin embargo la variables se utilizan especialmente en las funciones, procedimientos y procesos VHDL. Cuando se trabaja con algunos de estos elementos de programa , la sintaxis es:

  ***variable*** *nombre variable : tipo variable;*

- ***declaración de procedimiento:*** Es similar a una función , excepto que no regresa un resultado. Así como una llamada de función puede emplearse en lugar de una expresión, una llamada de procedimiento puede utilizarse en lugar de una instrucción.

## Librería VHDL:

Se trata de un lugar específico dónde el compilador VHDL almacena información acerca de un proyecto de diseño en particular , incluyendo archivos intermedios que se utilizan en el análisis, simulación y síntesis de diseño. La ubicación de la librería depende estructuralmente de la implementación, y se trata de un sistema de archivos.

Considerando un diseño determinado, el compilador VHDL crea y utiliza en forma automática una librería llamada **"work".**

Un diseño completo VHDL tiene muchos archivos, donde cada uno guarda diferentes unidades del diseño; incluyendo entidades y arquitecturas. El compilador analiza cada archivo de diseño y coloca el resultado en la librería "work".

No toda la información necesaria puede estar en la librería "work", pero se puede usar la librería de otro proyecto haciendo mención del mismo. Los proyectos modestos pueden hacer uso de alguna librería estándar, y en cuyo caso el diseñador hace referencia a través de la cláusula ***library*** al comienzo del archivo de diseño. La cláusula "***library*** **work"** está incluida en forma implícita al principio de todo archivo de diseño. Caso contrario se define específicamente , y en cuyo caso, esa especificación nos da acceso a cualquier entidad y arquitectura previamente almacenadas en la librería. Por ejemplo:

***Library IEEE;*** hace referencia a las definiciones estándar IEEE

Esto no nos da acceso a las definiciones de tipos y semejanzas, lo cual se hace a través de la función "paquetes" y "cláusulas use".

## Paquete VHDL:

Se refiere a un archivo que contiene definiciones de objetos que pueden ser usados en otros programas.

Un paquete incluye declaraciones de señal, tipo, constante, función, procedimiento y componentes. Y se definen señales globales disponibles para cualquier entidad que haga referencia al mismo, a saber:

- Los tipos y constantes definidos son conocidos en cualquier archivo que los mencione.
- Las funciones y procedimientos pueden ser llamados en archivos que utilicen el paquete.
- Los componentes pueden ser ejemplificados en arquitecturas que hacen uso del paquete.

Hay paquetes escritos por el usuario y estándares.

Los escritos por el usuario, pueden ser de dos tipos universales y locales,

a. Los universales son visibles en cualquier archivo de diseño que use dicho paquete, y tienen la sintaxis:

**Package** *nombre-paquete* **is**
*declaración de tipo*
*declaración de señales*
*declaración de constantes*
*declaración de componentes*
*declaración de funciones*
*definición de procedimiento*
***end*** *nombre-paquete;*

Notar, que aquí sólo se indican declaración de funciones y no definiciones.

b. Los locales son visibles solo en el archivo que se está tratando, y tienen la sintaxis:

**Package body** *nombre-paquete* **is**
*declaración de tipo*
*declaración de constantes*
*definición de funciones*
*definición de procedimiento*
***end*** *nombre-paquete;*

La definición de función completa está determinada en el cuerpo del paquete y no es visible para los usuarios de la función.

La declaración de función es:

**function** *nombre-función (*
*nombre-señal : typo-señal ;*
*nombre-señal : typo-señal ;*
*…….*
*nombre-señal : typo-señal ;*
) **return** tipo-retorno

El uso de un paquete específico se realiza a través de la cláusula **use** al comienzo del archivo de diseño.

Los paquetes estandares están definidos de antemano por alguna entidad específica. Por ejemplo:

**Library IEEE** **--** Especifica el nombre de la librería, IEEE en este caso, del cual se usaran definiciones específicas.

***use*** IEEE.std_logic_1164.all; -- Permite usar las definiciones del paquete standard IEEE1164. El sufijo .all le sentencia al compilador que use todas la definiciones de este archivo.

Se pueden usar otras alternativas para acotar más el uso de partes de alguna librería, por ej.

IEEE.std_logic_1164.std_ulogic; -- Permite usar sólo las definiciones del paquete std_ulogic, sin todos los tipos y funciones relacionadas.

## Descripción del Sistema

Una vez que hemos estudiado las definiciones estructurales podemos establecer la parte ejecutable de una arquitectura.

Si recordamos la sintaxis de una arquitectura, vemos que el cuerpo principal incluye una serie de instrucciones concurrentes; es decir que cada una de ellas se ejecuta simultáneamente con todas las otras instrucciones concurrentes.

**ARCHITECTURE** *nombre-arquitectura* ***OF*** *nombre-entidad* **IS**
*declaración de señal (variable)*
*declaración de constantes*
*declaración de tipo*
*definición de funciones*
*definición de procedimiento*
*declaracion de componentes*
**begin**
***enunciado concurrente***
***enunciado concurrente***
***.....***
***enunciado concurrente***
***end*** *nombre-arquitectura*

Este comportamiento difiere radicalmente con la ejecución secuencial que introducen todos los lenguajes de software convencionales. Estas son necesarias para simular el comportamiento del hardware, donde los componentes circuitales se interrelacionan entre sí de forma continua, y no con un ordenamiento por etapas temporales específicas.

Por lo mencionado en los párrafos anteriores, en toda estructura VHDL el simulador actualiza continuamente los cambios y resultados hasta que los valores de las señales especificadas se estabilizen en el tiempo.

VHDL tiene varias instrucciones concurrentes y un mecanismo específico para agrupar un conjunto de instrucciones secuenciales para simular funcionalidad concurrente, las cuales de acuerdo a su forma de utilización establecen tres tipos diferentes de descripción arquitectónica., y son:

## 1. Descripción estructural ó diseño estructural:

Se define así a todo diseño que utiliza componentes, pues define la estructura específica de interconexión de señales y entidades que expicita la entidad.
Se utiliza la instrucción concurrente elemental de VHDL que es la *instrucción* ***component,*** cuya sintaxis básica es:

etiqueta: nombre-componente ***port map*** (señal1, señal2, …., señaln);

etiqueta: nombre-componente ***port map*** (port1=>señal1, ..., portn=>señaln);
dónde:

- nombre-componente es el nombre de una entidad previamente definida, y que se utilizará dentro del cuerpo de una arquitectura actual.
- ***Port map*** palabra clave que introduce una lista asociando los puertos de la entidad nombrada con señales en la arquitectura actual. Hay dos pformas de escribir una lista, a saber:
  - ***Formato posicional***: En forma similar a los lenguajes convencionales, y dónde las señales en la lista están asociados con los puertos de la entidad en orden idéntico al que se estableció en la definición de ***entidad.***
  - ***Formato explícito***: Dónde cada uno de los puertos se conecta a una señal en forma directa ó explicita, a través del operador "=>".

Todo componente antes de ser utilizado en una arquitectura debe ser declarado explícitamente en una ***declaración de componente*** en la definición de arquitectura como se vio oportunamente, y cuya sintaxis es:

```
component  nombre-componente
   port (nombre- señal : modo tipo-señal ;
         nombre- señal : modo tipo-señal ;
         ……
         nombre- señal : modo tipo-señal ;
end component ;
```

Los componentes usados en una arquitectura pueden ser los mismos que se definieron como parte de un diseño ó pueden surgir de una librería definida***.***

Ejemplo para un decodificador de 2 a 4:

```
LIBRARY IEEE;
use  IEEE.std_logic_1164.all

ENTITY  dec2a4 IS
PORT  (enta, entb: IN std_logic;
       Sal: OUT std_logic_VECTOR (3 downto 0);
END dec2a4

ARCHITECTURE   estruc OF dec2a4 IS
   SIGNAL entaneg, entbneg : std_logic
  COMPONENT  INV
   PORT (ent: IN std_logic;
         Sal: OUT std_logic);
   END COMPONENT

COMPONENT  AND2ent
   PORT (ent1, ent2: IN std_logic;
         Sal: OUT std_logic);
   END COMPONENT
```

```
BEGIN
    U1 : INV PORT MAP (enta, entaneg) ;
    U2 : INV PORT MAP (entb, entbneg) ;
    U3 : AND2ent  PORT MAP (entaneg, entbneg, Sal0) ;
    U4 : AND2ent  PORT MAP (entaneg, entb, Sal1) ;
    U5 : AND2ent  PORT MAP (enta, entbneg, Sal2) ;
    U6 : AND2ent  PORT MAP (enta, enta, Sal3) ;
END estruc
```

Esta descripción estructural se basa en la descripción de los componentes que conforman el circuito, en este caso un inversor y una AND de 2 entradas. Estos componentes deben ser descriptos con anterioridad, conformando un nivel más bajo en la jerarquía del diseño.

```
ENTITY  INV IS
PORT  (ent: IN std_logic;
        Sal: OUT std_logic);
END INV
ARCHITECTURE   INV OF INV IS
BEGIN
    Sal <= NOT ent,
END INV

ENTITY  AND2ent IS
PORT  (enta, entb: IN std_logic;
        Sal: OUT std_logic);
END AND2ent
ARCHITECTURE  AND OF AND2ent IS
BEGIN
    Sal <= enta AND entb,
END AND2ent
```

Esta declaración de cada componente puede server como libreria dentro de la presente descripción ó diseños futuros.

**2. Descripción de flujo de datos ó diseño de flujo de datos:**

Esta descripción se puede realizar por el uso de otro tipo de instrucciones concurrentes, que le permiten a VHDL describir un circuito lógico en términos del flujo de la información y operaciones activadas en el circuito.
Algunas de ellas son:

a. Instrucción de asignación de señal concurrente: Es una de las más usadas y su sintaxis es

   *nombre-señal <= expresión [* Se lee *"nombre señal* obtiene *expresión"]*

   En este caso el tipo de expresión debe ser compatible con el de nombre señal.

Ejemplo:

```
LIBRARY IEEE;
use  IEEE.std_logic_1164.all

ENTITY  dec2a4 IS
PORT  (enta, entb: IN std_logic;
```

```
        Sal: OUT std_logic_VECTOR (3 downto 0);
END dec2a4

ARCHITECTURE   flujo_datos1 OF dec2a4 IS
BEGIN
    SIGNAL entaneg, entbneg : std_logic
     entaneg <=  NOT enta
     entbneg <=  NOT entb
     Sal0 <= entaneg AND entbneg ;
     Sal1 <= entaneg AND entb ;
     Sal2 <= enta AND entbneg ;
     Sal3 <= enta AND entb ;
END flujo_datos1
```

b. Forma condicional de la instrucción de asignación de señal concurrente: Aquí se aplican las palabras claves ***when*** y ***else***, a través de la siguiente sintaxis es

```
nombre-señal <= expresión when expresión-booleana else    ;
                expresión when expresión-booleana else    ;
                ……..
                expresión when expresión-booleana else    ;
                expresión ;
```

En este caso una expresión booleana combina términos booleanos sencillos y básicos usando operadores booleanos de VHDL, tales como **and, or, nor, not,** etc. Los términos booleanos son generalmente variables booleanas o el resultado de comparaciones, dónde se emplean los operadores relacionales =, =/; >, >0, etc.

```
LIBRARY IEEE;
use  IEEE.std_logic_1164.all

ENTITY  dec2a4 IS
PORT  (enta, entb: IN std_logic;
        Sal: OUT std_logic_VECTOR (3 downto 0);
END dec2a4

ARCHITECTURE   flujo_datos2 OF dec2a4 IS
BEGIN

    Sal <= `0001´ WHEN  enta= `0´ y entb = `0´  ELSE
           `0010´ WHEN  enta= `0´ y entb = `1´  ELSE
           `0100´ WHEN  enta= `1´ y entb = `0´  ELSE
           `1000´ ;
END flujo_datos2
```

c. Asignación de señal seleccionada: En esta instrucción se evalúa la expresión planteada, y cuando el resultado satisface una de las opciones planteadas asigna el correspondiente valor-señal al nombre–señal, de acuerdo a la siguiente sintaxis:

```
with expresión select
     nombre señal  <= valor-señal when opción ,
                      valor-señal when opción ,
                     ……..
                      valor-señal when opción ;
```

- Las opciones en cada cláusula when puede ser un valor simple de expresión o una lista de valores separados por barras verticales (|).
- Las opciones para toda la instrucción deben ser mutuamente exclusivas y todas inclusivas.
- Para indicar todas las opciones de la expresión que no hayan sido cubiertas se utiliza la palabra clave ***others*** en la última cláusula ***when.***

***Ejemplo relacionando con el estilo anterior***

**3. Descripción basado en comportamiento ó diseño de comportamiento:**

Esta forma permiten describir un circuito lógico en términos del comportamiento del mismo, para lo cual necesitamos conocer algunos elementos adicionales de VHDL que veremos a continuación.

Es importando indicar que en VHDL la forma más clara de describir un comportamiento es a través del " proceso". Para ello debemos definir un ***proceso*** como un conjunto de instrucciones secuenciales que se ejecutan en paralelo con otros procesos e instrucciones concurrentes. Utilizando un proceso se puede especificar una compleja interacción de señales y eventos, concretandose un circuito combinacional ó secuencial que realiza la operación modelada.

a. Instrucción ***process*****:** Es una instrucción de proceso que puede emplearse en cualquier parte del programa dónde pueda utilizarse una instrucción concurrente, y tiene la sig. sintaxis:

```
process ( lista de sensibilidad )
    declaración de tipo
    declaración de variable
    declaración de constante
    definición de funciones
    definición de procedimiento
begin
    enunciado secuencial
    enunciado secuencial
    ……..
    enunciado secuencial
end  process;
```

(Lista de sensibilidad)= (nombre-señal, nombre-señal, …., nombre-señal)
La lista de sensibilidad es opcional. Cuando no existe dicha lista, el proceso comienza a ejecutarse en el tiempo cero en la simulación.

Observar algo importante y es que dentro de un proceso no está permitida la definición de señales

- Una instrucción de proceso está escrita dentro del ámbito de una arquitectura cerrada, tiene visibilidad de los tipos, señales, constantes, funciones y procedimientos que están demacrados o son visible dentro de la arquitectura cerrada, pero se pueden definir tipos, señales, constantes, funciones y procedimientos que sean locales al proceso

- Un proceso puede declarar solo variables y no señales. Una variable toma los valores del estado dentro de un proceso y no es visible fuera de ese proceso.
- Una variable puede o no dar origen a una señal dentro del proceso modelado. La definición de una variable dentro de un proceso es similar a la declaración de una señal dentro de la arquitectura , a saber:

***variable*** nombre-variable : tipo-variable

- Un proceso tiene dos posibles estados: ejecución ó suspendido. Todo proceso se encuentra originalmente suspendido, y comienza ó reanuda su ejecución cuando su lista de sensibilidad cambia de valor. La ejecución comienza con la primera instrucción secuencial y así hasta el final. Cuando una señal de la lista de sensibilidad cambia de valor como resultado de la ejecución de un proceso, el mismo se ejecuta de nuevo., y esto ocurre hasta que ninguna de las señales cambie de valor. En la simulación todo este trámite ocurre en un tiempo nulo (Tiempo Simulación cero)
- Todo proceso se suspenderá luego de una o varias ejecuciones. Sin embargo, por error se pueden escribir procesos que nunca se suspenden, por ejemplo " lista de sensibilidad" "(X) " y un instrucción " X <= not X" . Los simuladores tienen la capacidad de detectar una ejecución infinita y abortarla indicando el correspondiente error, luego de algunos miles de pasadas.

b. Instrucciones ***secuenciales*****:** VHDL tiene varias clases de este tipo de instrucciones, y veamos**:**

I. Instrucción de asignación de señal secuencial: Es una instrucción que ocurre dentro de un proceso y tiene la misma sintaxis que su versión concurrente:

*nombre-señal <= expresión ;*

II. Instrucción de asignación de variable secuencial: Es una instrucción que ocurre dentro de un proceso y tiene una sintaxis similar que su versión para señales, dónde solo varia el operador de asignación como vemos:

*nombre-variable := expresión ;*

III. Instrucción ***if-then***: Es una instrucción secuencial muy usada y permite mejorar la descripción de comportamiento del circuito que se pretende diseñar. Su sintaxis es:

***if*** expresión-booleana ***then*** enunciado secuencial
***end if*** *;*

En este caso se prueba una expresión booleana y se ejecuta una instrucción secuencial si se cumple el valor de esa instrucción, es decir si el valor resulta verdadero **(true)**

IV. Instrucción ***if-then-else***: Su sintaxis es:

***if*** expresión-booleana ***then*** enunciado secuencial
***else*** enunciado secuencial
***end if*** *;*

En este caso se prueba una expresión booleana y se ejecuta una instrucción secuencial si se cumple el valor de esa instrucción, es decir si el valor resulta verdadero **(true)** y en caso de ser falso **(false)** se ejecuta otra instrucción secuencial establecida. Es decir, si se cumple la expresión booleana se realiza una acción, y en caso de no cumplirse se realiza otra, aprovechando la misma.

V. Instrucción ***elsif***: Permite crear instrucciones anidadas *if-then-else,* La instrucción secuencial de la cláusula **elsif** se ejecuta *si su expresión booleana es verdadera y todas las expresiones precedentes fueran falsas.* Su sintaxis es:

```
if      expresión-booleana then enunciado secuencial
elsif   expresión-booleana then enunciado secuencial
……..
elsif   expresión-booleana then enunciado secuencial
end if ;
```

Existe una alternativa que incluye una cláusula else que implica ejecutar esa instrucción si fueran falsas todas las expresiones booleana santeriores, cuya sintaxis es:

```
if      expresión-booleana then enunciado secuencial
elsif   expresión-booleana then enunciado secuencial
……..
elsif   expresión-booleana then enunciado secuencial
else    enunciado secuencial
end if ;
```

VI. Instrucción ***case***: Es una instrucción que permite seleccionar entre múltiples alternativas basadas en el valor de una señal o expresión. Se usa como una alternativa más legible y para producir un circuito mejor sintetizado frente a las instrucciones ***if*** vistas anteriormentes.
Su sintaxis es:

```
case      expresión is
    when  alternativas => enunciado secuencial
    ……..
    when  alternativas => enunciado secuencial
end case ;
```

Esta instrucción evalúa la expresión dada, obteniendo un valor y ejecuta la instrucción secuencial correspondiente a la alternativa u opción que coincide con ese valor.

- Las opciones puede ser un valor simple u múltiples valores separados por barras verticales (|).

- Las opciones deben ser mutuamente excluyentes e incluir todos los posibles valores del tipo de expresión.
- Existe una alternativa para indicar todos los valores que no han sido cubierto, así:

```
case      expresión is
    when  alternativas => enunciado secuencial
    ……..
    when  alternativas => enunciado secuencial
    when  others       => enunciado secuencial
end case ;
```

- La instrucción ***case*** resulta ser la instrucción secuencial de su versión concurrente ***select,*** permitiendo ambas apreciar en forma muy sencilla el comportamiento funcional buscado.

```
ARCHITECTURE   flujo_datos3 OF dec2a4 IS
BEGIN
    PROCESS (enta, entb)
       variable  c: std_logic_vector (1 down to 0)
    BEGIN
        C(1) := enta
        C(0) := entb
      CASE c IS
        WHEN "00" => Sal <= "0001";
        WHEN "01" => Sal <= "0010";
        WHEN "10" => Sal <= "0100";
        WHEN  others => Sal <= "1000";
      END CASE;
  END PROCESS;
END flujo_datos3;
```

VII. Instrucción ***loop***: Es otra instrucción secuencial muy importante y de mucho uso. Con la siguiente sintaxis permite crear un ciclo ó lazo infinito:

***loop***
enunciado secuencial
……..
enunciado secuencial
***end loop*** *;*

Este tipo de instrucción, aunque indeseada en los lenguajes convencionales resulta de mucha utilidad en la descripción y modelado de hardware.

VIII. Instrucción ***for***: Es otra instrucción que permite realizar ciclos:

***for*** indentificador ***in*** rango ***loop***
enunciado secuencial
……..
enunciado secuencial
***end loop*** *;*

La variable del ciclo es el identificador establecido y es del mismo tipo que el rango.

IX. Instruccioes ***exit*** y ***next***: Se trata de un par de instrucciones muy útiles en el caso de ciclos ó lazos.
- ***exit:*** cuando se ejecuta, transfiere el control a la instrucción que sigue inmediatamente al final del ciclo.
- ***Next:*** al ejecutarse produce que cualquier instrucción del ciclo sea omitida y comience la siguiente iteración del ciclo.

X. Instrucción ***while***: Es otra instrucción que permite realizar ciclos, Aquí la expresión booleana se prueba antes de cada iteración del ciclo, y el ciclo se ejecuta solo si el valor de dicha exprsión es verdader (true); de la sig. forma:

***while*** expresión booleana ***loop***
enunciado secuencial
……..
enunciado secuencial

***end loop ;***

Existen muchas otras instrucciones secuenciales que se verán luego.

## Temporización en VHDL

Todo lo estudiado hasta ahora en VHDL prescinde de la dimensión de tiempo; es decir, todo el funcionamiento del circuito ocurre en un tiempo nulo. VHDL tiene muy buenas posibilidades describir y modelar circuitos introduciendo esta importante variable. Se introducen algunos conceptos elementales, ya que se trata de un estudio verdaderamente complejo y que no forma parte de este curso.

A continuación veremos algunas instrucciones que permiter utilizar el tiempo:

A. Palabra clave ***after*** : Permite especificar un retardo temporal en cualquier instrucción de asignación de señal, para los casos de asignaciones secuenciales, concurrentes, condicionales y seleccionadas.

Por ejemplo podemos describir una compuerta inhibitoria, con la sig. descripción donde no interviene para nada el tiempo, :

```
entity inhibición is
  port (a,b :    IN BIT;
              f : OUT BIT) ;
end inhibición

architecture funcionamiento of inhibición is
begin
   f <= `1´ when a= `1´ y b = `0´ else `0´ ;

end funcionamiento
```

Y ahora podemos modelar los retardos, considerando:
Transición de 0 a 1 : 5 nseg
Transición de 1 a 0 : 3 nseg
Con el sig. programa.

```
entity inhibición is
   port (a,b :    IN BIT;
              f : OUT BIT) ;
end inhibición

architecture funcionamiento of inhibición is
begin
   f <= `1´ after 5 ns when a= `1´ y b = `0´ else `0´ after 3 ns;

end funcionamiento
```

Las mayorías de las librerías incluyen tales parámetros de retardo en sus modelos VHDL para los componentes de bajo nivel. Así, empleando estas condiciones, un simulador VHDL puede predecir el comportamiento temporizado de un sistema digital completo.

B. Instrucción ***wait*** : Es otra forma de establecer la dimensión temporal a través de una instrucción secuencial. Puede utilizar para suspender un *proceso* durante un período especificado de tiempo.
**Ejemplo**

**Simulación en VHDL**

Como vimos oportunamente, una vez que se ha desarrollado un programa en VHDL en toda dimensión, es decir con sintaxis y semántica adecuada viene la etapa de verificación funcional, lo cual se establece a través de la simulación.

La simulación tiene una serie de etapas básicas que verifican diferentes dimensiones del proyecto, a saber:

a. **Tiempo simulación cero TS0:**

   - El simulador inicializa las señales a un valor predeterminado, establecido en la herramienta adecuada.
   - Inicializa cualquier variable ó señal que se hayan declarado explícitamente en el programa.

b. **Ciclo de Simulación:** El simulador comienza la ejecución de todos los procesos e instrucciones concurrentes. Sin embargo, como es imposible que eso sea simultáneamente, se puede suponer que lo realiza utilizando una "matriz de eventos temporales" y "una matriz de sensibilidad de señal". Así cada instrucción concurrente es equivalente a un proceso.
   Ejecución de tiempo cero: Todos los procesos son preparados para su ejecución, y se selecciona uno de ellos; a partir de allí, se ejecutan todas las instrucciones secuenciales, incluyendo todos los eventos repetitivos especificados en la descripción. Al finalizar la ejecución de este proceso, se selecciona otro, y así sucesivamente hasta terminar la ejecución de todos los establecidos, completando un ciclo de simulación.

oooOOOooo