7. INTRODUCCIÓN A VHDL

upna
Universidad
Pública de Navarra
Nafarroako
Unibertsitate Publikoa

11. INTRODUCCIÓN A VHDL

- **Historia de los lenguajes de descripción de hardware**

  Hasta la década de los 80', se aplicaba un método prácticamente manual para el diseño de los circuitos digitales

  El diseño se centraba en los niveles eléctricos para establecer parámetros y relaciones entre los componentes básicos a nivel de transistor.

  En los 80' surgen los lenguajes de descripción de hardware (hardware description language – HDL), con el objetivo de simplificar el método de diseño de circuitos digitales.

  Los lenguajes HDL son herramientas de programación formal para describir el comportamiento y la estructura de sistemas usando un esquema textual

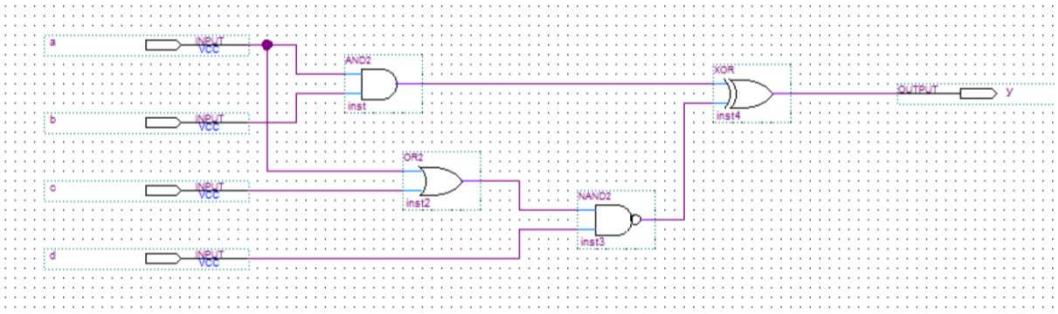
  De todos estos lenguajes se impusieron dos, que son los que se emplean actualmente: VHDL y Verilog

- **Ventajas de los HDL**

  El lenguaje es independiente de la tecnología en que se programe (el mismo modelo puede ser sintetizado en librerías de distintos vendedores). Esto permite reutilizar el diseño en diferentes componentes, como pueden ser ASICs o FPGAs, con muy poco esfuerzo.

  Posibilita comprobar el funcionamiento del sistema durante el proceso de diseño sin tener que implementar el circuito físicamente (se puede probar el sistema, rectificar y realizar cambios de diseño).

  Más sencillez para comprender la función que con un esquemático



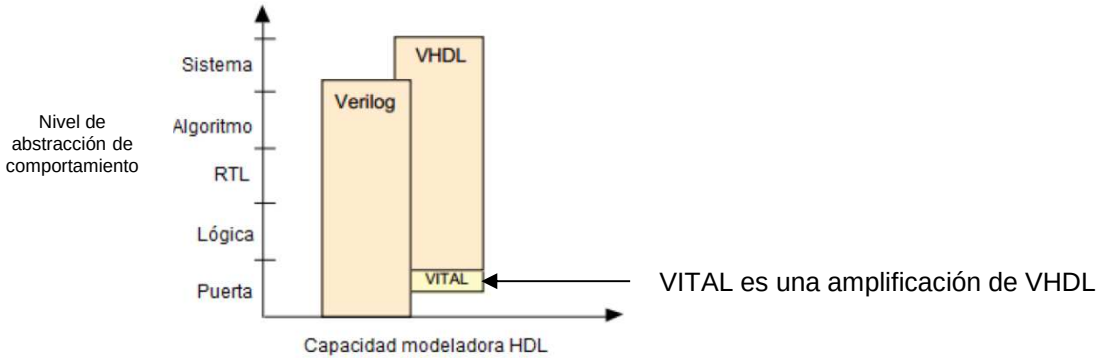

```
entity puertaslogicas is
   port( a,b,c,d: in bit;                    -- 4 entradas
         y: out bit);                        -- 1 salida
end puertaslogicas;

architecture ejemplo1 of puertaslogicas is   -- Cabecera del programa
begin                                        -- Empieza el programa
   process(a,b,c,d)                          -- Cabecera de un proceso
   begin                                     -- Empieza el proceso
     y<=(a and b)xor((a or c)nand d);        -- Sentencia para implementar el circuito lógico
         end process;                        -- Termina el proceso
end ejemplo1;                                -- Termina el programa
```

Esquemático de un circuito combinacional y su equivalente en lenguaje HDL

- **Comparativa entre VHDL y Verilog**

Implantación en universidades: si bien en Estados Unidos la implantación de ambos lenguajes en sus universidades es pareja, las universidades españolas se decantan claramente por VHDL. Por otro lado, aprender Verilog tras conocer VHDL es más sencillo que lo contrario, de ahí que en esta asignatura se opte por VHDL

Capacidad: la descripción de hardware se puede realizar con ambos lenguajes, aunque abarcan un rango ligeramente distinto de los niveles de abstracción de comportamiento (http://bit.ly/1oGGKQR)



Tipos de datos: en VHDL se pueden utilizar multitud de tipos de datos definidos por el usuario, mientras que en Verilog éstos son definidos por el propio lenguaje y no se pueden añadir más. Todo esto permite que los modelos sean más fáciles de escribir, más fáciles de leer e interpretar y evitar funciones de conversión innecesarias que estorban en el código.

Lenguaje en que se basan: VHDL se basa en ADA y Verilog en C

Reutilización del diseño: en VHDL las funciones se pueden incluir en un paquete reutilizable en otros diseños, mientras que en Verilog no se puede. Es el concepto de librería, muy habitual en lenguajes de programación.

Procedimientos concurrentes: VHDL permite llamadas a procedimientos concurrentes y Verilog no

- **Entidad**

Es la interfaz de un dispositivo (puede ser un chip o un diseño más grande) con el exterior.

Cada señal que se declara en la entidad está referida a un puerto, lo que indica que esta señal es visible o accesible desde el exterior. En un chip los puertos son los pines del dispositivo.

La estructura de la entidad es la siguiente:

```
Entity nombre_entidad is port(
    puertos
);
end entidad;
```

Ejemplo multiplexor →

```
1  entity multi is port(
2      a, b, c, d, e, f, g, h :in bit;              --8 canales de entrada
3      enable :in bit;                              --señal enable
4      control :in bit_vector(2 downto 0);          --3 canales de control
5      s :out bit                                   --1 canal de salida
6      );
7  end multi;
```

Seguido del nombre del puerto y separado de éste por dos puntos, se debe indica el modo del puerto.

El modo describe la dirección en la cual la información es transmitida a través del puerto: in, out, buffer o inout. Si no se especifica nada, se asume que el puerto es del modo in

Descripción de los modos de VHDL:

Modo in: los datos son sólo de entrada (se usa para relojes y entradas de control)

Modo out: los datos son sólo de salida. El estado lógico en el que se encuentra no es leíble para el compilador, lo que supone una desventaja pero a la vez consume menos recursos.

Modo buffer: se parece al modo out pero permite la realimentación. Es decir, se puede conectar a una señal interna, o a un puerto de modo buffer de otra entidad (ejemplo: la salida de un contador, donde se requiere saber la salida en el momento actual para saber a salida en el momento siguiente)

Modo inout: se usa para señales bidireccionales, donde por el mismo puerto fluyen datos 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, pero se reduce la simplicidad de lectura posterior del código por otra persona, y los recursos disponibles

- **Entidad**

Además de los modos, se requiere indicar el tipo para cada señal de la entidad:

```
enable :in bit;          control :in bit_vector(2 downto 0);
```

nombre, modo, tipo (enable) — nombre, modo, tipo (control)

Se puede crear tantos tipos de señales como se desee, aunque la norma de VHDL inicialmente definía cuatro tipos nativos, que son los más utilizados:

boolean: puede tomar dos valores: verdadero (true) o falso (false)

bit: puede tomar dos valores: 0 ó 1 (o también "low" o "high", según se prefiera).

bit_vector: es un vector de bits, en el que con la palabra reservada downto o to se indica que el bit más significativo es el número más alto o el más bajo del vector, respectivamente.

```
control :in bit_vector(2 downto 0);   ⟶ el bit más significativo es control(2) y el menos significativo control(0)
control :in bit_vector(0 to 2);       ⟶ el bit más significativo es control(0) y el menos significativo control(2)
```

integer: toma valores de números enteros (-2147483647 a 2147483647). Consume muchos recursos del dispositivo de lógica programable, debido a que está creado para la simulación.

Después se han ido añadiendo más tipos nativos como:

std_ulogic y std_logic: que representan un hilo y pueden tomar los siguientes valores:

```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High     Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'); -- Don't care
```

```
TYPE std_logic IS ( 'U',  -- Uninitialized
                    'X',  -- Forcing unknown
                    '0',  -- Forcing 0
                    '1',  -- Forsing 1
                    'Z',  -- High    impedance
                    'W',  -- Weak    unknown
                    'L',  -- Weak    0
                    'H',  -- Weak    1
                    '-'); -- Don't care
```

std_ulogic_vector y std_logic_vector: son las versiones de vector de std_ulogic y std_logic

Y muchos otros: **character, real, line, natural, positive, string, text, time**

Nota final: para los nombres de los puertos no se puede utilizar como primera letra un número.

- **Arquitectura**

La arquitectura o architecture define la función del dispositivo, es decir, qué transformaciones se realizarán sobre los datos que entren por los puertos de entrada para producir la salida.

La estructura para la arquitectura es:

arquitecture *nombre_arquitectura* ***of*** *nombre_entidad* ***is***
declaraciones
begin
sentencias
end *nombre;*

Ejemplo de multiplexor tomando de la entidad 'multi' vista antes

```
1  architecture archmul of multi is            --Cabecera del programa
2  begin                                       --Comienza el programa
3    process (a, b, c, d, e, f, g, h, control, enable) --Cabecera del proceso
4      begin                                   --Comienza el proceso
5        if enable='1' then s<='0';            --Sentencia if
6          elsif enable='0' then               --Sentencia elseif
7            case control is                   --Sentencia case
8              when "000" => s <= a;
9              when "001" => s <= b;
10             when "010" => s <= c;
11             when "011" => s <= d;
12             when "100" => s <= e;
13             when "101" => s <= f;
14             when "110" => s <= g;
15             when "111" => s <= h;
16             when others => s <= '0';
17           end case;                  --Acaba sentencia case
18       end if;                        --Acaba sentencia if y elsif
19   end process;                       --Acaba el proceso
20 end archmul;                          --Finaliza el programa
```

- **Paquetes o librerías**

Una de las propiedades más interesantes de VHDL es la de poder generar y utilizar paquetes (también llamados librerías) de funciones, al estilo de C y otros lenguajes de programación.

La estructura para las librerías se compone de dos partes (análogo a C):

- La declaración del paquete: una visión externa y simplificada del paquete (como una caja negra)
- La definición del paquete: donde se detalla el contenido de cada elemento del paquete

```
package nombre_del_package is
    -- declaración de procedures
    -- declaración de funciones
    -- declaración de tipos, etc...
end nombre_del_package;
```

a) Declaración

```
package body nombre_del_package is
    -- definición de procedures
    -- definición de funciones
    -- definición de tipos, etc.
end nombre_del_package;
```

b) Definición

Ejemplo:

```
1  package ee530 is
2      constant maxint: integer := 16#ffff#;
3      type arith_mode_type is (signed, unsigned);
4      function minimum(constant a,b: in integer) return integer;
5  end ee530;
6
7  package body ee530 is
8      function minimum (constant a,b: integer) return integer is
9        variable c: integer; -- local variable
10          begin
11             if a < b then
12                 c := a;  -- a is min
13             else
14                 c := b;  -- b is min
15             end if;
16             return c;  -- return min value
17         end;
18 end ee530;
19
20 use work.ee530.all
```

Declaración

Definición

Se hace visible el paquete: work indica el directorio de trabajo, ee530 es el nombre del paquete y all indica que después de esta sentencia se puede utilizar todo el contenido del paquete ee530

- **Paquetes o librerías**

Además de la crear librerías y utilizarlas, también se puede importar librerías estándar o realizadas por otros programadores. En el siguiente ejemplo (http://bit.ly/1oL60pc) se combinan ambas cosas:

```
1  library IEEE;                 --se indica que IEEE es un nombre de librería
2  use IEEE.std_logic_1164.all;  --se hace visible la librería IEEE.std_logic_1164
3  use IEEE.std_logic_arith.all; --se hace visible la librería IEEE.std_logic_arith
4
5  --declaración del paquete:
6  package test_pkg is
7  type t1 is
8      record
9          a :std_logic_vector(11 downto 0);
10         b :std_logic_vector(15 downto 0);
11         c :std_logic_vector(3 downto 0);
12     end record;
13 function add (a2 : t1; b2: t1) return t1;
14 end test_pkg;
15
16 --definición del paquete:
17 package body test_pkg is
18 function  add (a2 : t1; b2: t1) return t1 is
19 variable sum : t1;
20 begin
21 sum.a:=a2.a xoR b2.a;
22 sum.b:=a2.b xoR b2.b;
23 sum.c:=a2.c xoR b2.c;
24 return sum;
25 end add;
26 end test_pkg;
```

Declaración y definición del paquete test_pkg

```
1  library IEEE;                  -- se indica que IEEE es un nombre de librería
2  use IEEE.STD_LOGIC_1164.ALL;   -- se hace visible la librería IEEE.STD_LOGIC_1164
3  use IEEE.STD_LOGIC_ARITH.ALL;  -- se hace visible la librería IEEE.STD_LOGIC_ARITH
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;-- se hace visible la librería IEEE.STD_LOGIC_UNSIGNED
5
6  library work;                  -- se indica que work es un nombre de librería (el directorio del proyecto VHDL)
7  use work.test_pkg.all;         -- se hace visible la librería test_pkg, creada por nosotros
8
9  entity test is
10 port (clk : in std_logic;
11       a1 : in t1;
12       b1 : in t1;
13       c1: out t1
14     );
15 end test;
16
17 architecture Behavioral of test is
18 begin
19 process(clk)
20  begin
21 if(clk'event and clk='1') then
22 c1<=add(a1,b1);
23 end if;
24 end process;
25 end Behavioral;
```

Programa general utilizando librerías estándar y la test_pkg

- **Identificadores**

Conjunto de caracteres dispuestos de forma adecuada y siguiendo unas normas propias del lenguaje.

Reglas a tener en cuenta:

- Deben empezar con un carácter alfabético.

- VHDL identifica indistintamente tanto las mayúsculas como las minúsculas, pudiéndose por ejemplo emplear por igual el identificador "sumador" o "SUMADOR"

- Los identificadores pueden contener caracteres numéricos del '0' al '9', sin que éstos puedan aparecer al principio.

- No puede usarse como identificador una palabra reservada por VHDL.

- **Símbolos especiales**

Además de las palabras reservadas, VHDL utiliza algunos símbolos especiales.

Ejemplos:

- El símbolo "+" se representa la operación suma

- El símbolo "- -" se usa para los comentarios del usuario (el programa salta el contenido después de este símbolo)

- El símbolo ";" se emplea al final de una sentencia de código para indicar que esta ha terminado

El resto de símbolos especiales son: - / ( ) . , : & ' < > = | # <= => :=

- **Tipos de datos**

Para introducir un tipo de datos nuevo, se emplea la palabra reservada type seguida del nombre del nuevo tipo de datos, de la palabra reservada 'is', y de los elementos que contiene:

```
1   type longitud_maxima is range 10 to 50
2   type estado is (q0, q1, q2);
```

a) Introducción de dos tipos de datos: estado y longitud_maxima

```
3     variable est: estado;
4   port (entrada: in estado;
5         salida: out longitud_maxima);
```

b) Declaración de variable 'est' de tipo estado y de señales entrada y salida de tipo estado y longitud máxima

- **Palabras reservadas**

abs	operator, absolute value of right operand. No () needed.
access	used to define an access type, pointer
after	specifies a time after NOW
alias	create another name for an existing identifier
all	dereferences what precedes the .all
and	operator, logical "and" of left and right operands
architecture	a secondary design unit
array	used to define an array, vector or matrix
assert	used to have a program check on itself
attribute	used to declare attribute functions
begin	start of a **begin end** pair
block	start of a block structure
body	designates a procedure body rather than declaration
buffer	a mode of a signal, holds a value
bus	a mode of a signal, can have multiple drivers
case	part of a case statement
component	starts the definition of a component
configuration	a primary design unit
constant	declares an identifier to be read only
disconnect	signal driver condition
downto	middle of a range 31 downto 0
else	part of "if" statement, if cond then ... else ... end if;
elsif	part of "if" statement, if cond then ... elsif cond ...
end	part of many statements, may be followed by word and id
entity	a primary design unit
exit	sequential statement, used in loops
file	used to declare a file type
for	start of a for type loop statement
function	starts declaration and body of a function
generate	make copies, possibly using a parameter
generic	introduces generic part of a declaration
group	collection of types that can get an attribute
guarded	causes a wait until a signal changes from False to True
if	used in "if" statements
impure	an impure function is assumed to have side effects
in	indicates a parameter in only input, not changed
inertial	signal characteristic, holds a value
inout	indicates a parameter is used and computed in and out
is	used as a connective in various statements
label	used in attribute statement as entity specification
library	context clause, designates a simple library name
linkage	a mode for a port, used like **buffer** and **inout**
literal	used in attribute statement as entity specification
loop	sequential statement, loop ... end loop;
map	used to map actual parameters, as in port map
mod	operator, left operand modulo right operand
nand	operator, "nand" of left and right operands
new	allocates memory and returns access pointer
next	sequential statement, used in loops
nor	operator, "nor" of left and right operands
not	operator, complement of right operand
null	sequential statement and a value
of	used in type declarations, **of** Real ;
on	used as a connective in various statements
open	initial file characteristic
or	operator, logical "or" of left and right operands
others	fill in missing, possibly all, data
out	indicates a parameter is computed and output
package	a design unit, also package body
port	interface definition, also port map
postponed	make process wait for all non postponed process to suspend
procedure	typical programming procedure
process	sequential or concurrent code to be executed
pure	a pure function may not have side effects
range	used in type definitions, **range** 1 **to** 10;
record	used to define a new record type
register	signal parameter modifier
reject	clause in delay mechanism, followed be a time
rem	operator, remainder of left operand divided by right op
report	statement and clause in assert statement, string output
return	statement in procedure or function
rol	operator, left operand rotated left by right operand
ror	operator, left operand rotated right by right operand
select	used in selected signal assignment statement
severity	used in assertion and reporting, followed by a severity
signal	declaration that an object is a signal
shared	used to declare shared objects
sla	operator, left operand shifted left arithmetic by right op
sll	operator, left operand shifted left logical by right op
sra	operator, left operand shifted right arithmetic by right
srl	operator, left operand shifted right logical by right op
subtype	declaration to restrict an existing type
then	part of if condition then ...
to	middle of a range 1 to 10
transport	signal characteristic
type	declaration to create a new type
unaffected	used in signal waveform
units	used to define new types of units
until	used in wait statement
use	make a package available to this design unit
variable	declaration that an object is a variable
wait	sequential statement, also used in case statement
when	used for choices in case and other statements
while	kind of loop statement
with	used in selected signal assignment statement
xnor	operator, exclusive "nor" of left and right operands
xor	operator, exclusive "or" of left and right operands

http://bit.ly/1psjQgg

- **Expresiones y operadores**

Los más habituales son los siguientes:

		Tipo de operador	Tipo de resultado
Operadores lógicos	AND, NOT, OR, NAND, NOR, XOR	booleano	booleano
Operadores relacionales	= / > <= > >=	cualquiera	booleano
Operadores aritméticos	+ - * / ** MOD REM ABS	integer, real, signal	integer, real, signal
Operadores de concatenación	&	array	array

Ejemplo de sumador de 4 bits con acarreo utilizando expresiones y operadores http://bit.ly/1qqKVFU

```
1  architecture archisumador of sumador is
2  begin
3  process (a,b,cin)
4    variable aux:std_logic_vector(4 downto 0);
5    begin
6    aux:=('0' & a) + ('0' & b);
7  if cin='1' then aux:=aux+1;
8  elsif cin='0' then null;
9    end if;
10   sum<=aux;
11   end process;
12   end archisumador;
```

En este ejemplo se aprecia una concatenación del elemento "a" (que previamente debe de haber sido definido como un std_logic_vector), con un "0", dando como resultado un array en el que la primera posición la ocupa el "0" y después va el elemento "a"
Este ejemplo también sirve para entender la utilización del operador suma "+", que incrementa la variable “aux” en caso de que “cin” valga uno.

- **Atributos**

Son elementos que complementan a las variables.

En el siguiente ejemplo, suponiendo que entrada es un vector, pueden utilizar los siguientes atributos:

```
if entrada'left='0' then ...     --Si el elemento más a la izquierda de entrada es '0', entonces se ejecuta lo que sigue al then
if entrada'right='1' then ...    --Si el elemento más a la derecha de entrada es '1', entonces se ejecuta lo que sigue al then
if entrada'length=5 then ...     --Si la longitud del vector entrada es 5, entonces se ejecuta lo que sigue al then.
if entrada'event=5 then ...      --Si hay un cambio en el nivel lógico de la señal entrada, entonces se ejecuta lo que sigue al then.
```

- **Programación secuencial y concurrente**

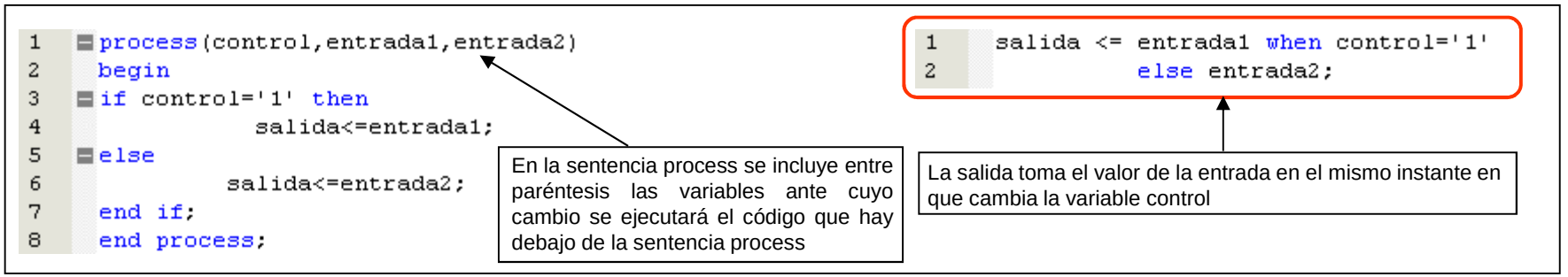
Los lenguajes de programación con los que se trabaja cuando alguien se inicia en ingeniería (Pascal, C, etc.) son de tipo SDL (software design logic), y su funcionamiento habitual es de tipo secuencial.

El funcionamiento secuencial consiste en que 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. De ahí que resulte de vital importancia la disposición de las mismas dentro del código fuente.

El funcionamiento concurrente se basa en la ejecución de código en paralelo (dos o más partes del programa se ejecutan a la vez)

VHDL combina la programación secuencial con la concurrente, lo que permite una mayor eficacia a la hora de ejecutar el programa y se adecua más al funcionamiento de un circuito, donde si se produce un cambio en una parte del mismo, éste se traduce en una variación (en algunos casos casi instantánea) de otras partes.

En el siguiente ejemplo se comparan ambos modelos de programación, el concurrente y el secuencial:



Programación secuencial (izquierda) y concurrente (derecha) de una asignación condicional

Como este capítulo de la asignatura es una introducción se hará lo posible por centrarnos sólo en la programación secuencial, dejando la concurrente para un curso más avanzado.

A continuación se verán los elementos básicos de la programación secuencial

- ## Sentencias secuenciales

A diferencia Pascal o C, donde existen variables y constantes, en VHDL existe un elemento nuevo que son las señales.

Tanto las variables como las señales se encargan de almacenar datos que se les introducen en el programa. Sin embargo, se diferencian en que las señales son elementos de tipo global (se pueden emplear en cualquier parte del programa), mientras que las variables tienen ámbito local, dentro de un process.

Process: estructuras concurrentes constituidas por sentencias de ejecución secuencial, del mismo tipo que los SDL (lenguajes de descripción de software) que nos llevan a emplear VHDL como si de otro lenguaje común se tratara.

Una segunda diferencia entre las variables y las señales es la asignación:

Asignación a una señal

Se puede asignar un valor a una señal, siempre que haya sido declarada en el apartado de declaración de puertos de la entidad, o cuando haya sido creada dentro de un process.

En la asignación a una señal se emplea el operador <=, estando la señal a asignar a la izquierda y el valor que debe tomar a la derecha:

```
1   signal1 <= signal2 + signal3;
```

Si la asignación está fuera de un process, el cambio es concurrente, es decir, la asignación está siempre activa, y se actualiza instantáneamente. Pero si la asignación está en un process, es secuencial (la señal no cambia su valor hasta que se ha evaluado el proceso en el cual se incluye):

```
1  process
2  begin
3  a <= b;       --La señal a tendrá el valor b
4  b <= a;       --La señal b tendrá el valor a
5  wait on a,b;
6  end process; --Se acualizan los cambios AQUÍ
```

En este ejemplo, las dos señales intercambian sus valores, ya que cuando se ejecuta la segunda (b<=a), el valor de la señal a no ha cambiado todavía aunque esté la sentencia a<=b, ya que ninguna señal cambia de valor hasta que se hayan evaluado todas las órdenes de un proceso, y es en ese momento cuando a y b toman el valor que se les ha indicado tomar.

- **Sentencias secuenciales**

Asignación a una variable

Las variables tienen su ámbito de actuación dentro de un proceso o en un área secuencial de un subprograma, y no retiene sus valores después del subprograma. Un uso típico es como índice en un bucle.

Para la asignación a una variable se emplea el operador :=

```
1    a     := b;       --a toma el valor de b
```

A diferencia de las señales, la asignación se produce instantáneamente, sin esperar a que termine el process. Así, el ejemplo que se utilizó para las señales no produce el mismo efecto en variables:

```
1    a := b;      -- a toma el valor b
2    b := a;      -- b toma el valor de a, pero a había cogido el valor de b, luego b y a se quedan con el valor de b
```

Para hacer el intercambio de datos entre a y b hay que ejecutar entonces el siguiente código:

```
1    temp := a;       --temp toma el valor de a
2    a    := b;       --a toma el valor de b
3    b    := temp;    --b toma el valor de temp (que era a)
```

Sentencia if-elsif-else

La construcción if-elsif-else se emplea para elegir, de entre una serie de condiciones, un conjunto de sentencias que ejecutará el programa. Su estructura es la siguiente:

if *(condición 1)* ***then***
haz una primera cosa;
elsif *(condición 2)* ***then***
haz una segunda cosa
else
haz una tercera cosa
end *if;*

Ejemplo →

```
14   if (L='0' and M='0' and N='1') then
15        F3<='1';
16   elsif (L='1' and M='1') then
17        F3<='1';
18   else
19        F3<='0';
20   end if;
```

En este ejemplo, ante las condiciones impuestas por L,M y N, F3 toma el valor ‘0’ o el valor ‘1’

- **Sentencias secuenciales**

Sentencia case

Se usa para especificar una serie de acciones según el valor dado de una señal de selección. La diferencia entre case e if se puede ver en http://bit.ly/1lJt0KL. La estructura case es la siguiente:

case (señal a evaluar) is
when (valor 1) => haz una primera cosa;
when (valor 2) => haz una segunda cosa;
...
when (último valor) => haz una última cosa;
end case;

Ejemplo tomado del multiplexor visto anteriormente →

```
case control is                          --Sentencia case
  when "000" => s <= a;
  when "001" => s <= b;
  when "010" => s <= c;
  when "011" => s <= d;
  when "100" => s <= e;
  when "101" => s <= f;
  when "110" => s <= g;
  when "111" => s <= h;
  when others => s <= '0';
end case;                          --Acaba sentencia case
```

Sentencia loop

Se usa para ejecutar un grupo de sentencias un número determinado de veces y presenta dos versiones:

for: se ejecuta un número específico de iteraciones basado en el valor de una variable

while: se ejecuta una operación mientra una condición de control sea cierta

Ejemplo programado con for en dos versiones (con contador ascendente y descendente) y con while:

```
1  process (A)
2  begin
3      Z <= "0000";
4      for I in 0 to 3 loop
5          if (A = I) then
6              Z(I) <= '1';
7          end if;
8      end loop;
9  end process;
```

a) Bucle for con variable I incremental

```
1  process (A)
2  begin
3      Z <= "0000";
4      for I in 3 downto 0 loop
5          if (A = I) then
6              Z(I) <= '1';
7          end if;
8      end loop;
9  end process;
```

b) Bucle for con variable I decremental

```
1  process (A)
2      variable I :
3        integer range 0 to 4;
4  begin
5      Z <= "0000";
6      I := 0;
7      while (I <= 3) loop
8        if (A = I) then
9          Z(I) <= '1';
10       end if;
11       I := I + 1;
12     end loop;
13 end process;
```

c) Bucle while

- **Sentencias secuenciales**

Sentencia exit

Se emplea en un bucle, permitiendo salir del mismo si se alcanza una condición fijada por nosotros.

A continuación se muestra un ejemplo, que es equivalente al visto para explicar los bucles for y while, pero con la orden de salida cuando 'I' valga 4 gobernada por la sentencia exit:

```
1  process (A)
2    variable I :
3      integer range 0 to 4;
4   begin
5     Z <= "0000";
6     I := 0;
7     L1: loop
8       exit L1 when I = 4;
9       if (A = I) then
10         Z(I) <= '1';
11      end if;
12      I := I + 1;
13    end loop;
14  end process;
```

Sentencia next

Como la exit, se enmarca en un bucle, y sirve para saltar una o más de las ejecuciones programadas:

```
1  process (A)
2   begin
3       Z <= "0000";
4       for I in 0 to 3 loop
5           if I=0 then next;
6           else
7               if (A = I) then
8                   Z(I) <= '1';
9               end if;
10          end if;
11      end loop;
12  end process;
```

En este ejemplo, se está saltando la asignación del elemento 0 de Z y por tanto si A=0, al final Z valdrá 0000 y no 1000 como hacía el bucle for inicialmente

- ## Sentencias secuenciales

Sentencia wait

La sentencia wait se usa para suspender un proceso si éste no tiene lista sensitiva (variables de entrada con cuyo cambio se ejecuta el código). En dicho caso el process pasa a ejecutarse en todo momento, y el wait hace la función de que barrera indicando que no se ejecutará el código que viene después hasta que una de las variables de la sentencia wait cambie.

Para entender mejor esto, basta con decir que los dos códigos siguientes son equivalentes.

```
1  process (a,b,c)
2    begin
3      x <= a and b and c;
4    end process;
```

```
1  process
2    begin
3      x <= a and b and c;
4      wait on a,b,c;
5    end process;
```

Sentencia wait until

Es el equivalente a los eventos en programación en C. Se espera a que una variable cambie para ejecutar el código correspondiente. Una aplicación típica son los biestables que ante un cambio en la señal de reloj, ejecutan una acción.

El siguiente ejemplo es el del biestable D por flanco de subida (cuando el reloj clk pasa a 1 se actualiza su salida)

```
1  architecture ejemplo of ffd is
2  begin
3  process begin
4      wait until (clk='1');
5        q <= d;
6    end process;
7    end ejemplo;
```

- **Descripción de comportamiento o behavioural**

Incluye las sentencias y órdenes típicas de un lenguaje de programación tipo C o Pascal (if, then, case,...), sin importarnos cómo quedará la distribución de puertas lógicas dentro de la PLD.

Sólo requiere un proceso que incluya toda la estructura secuencial.

El ejemplo del multiplexor, visto anteriormente, sigue descripción de comportamiento.

En el siguiente ejemplo se muestra el código para un comparador cuya entity se llama compa:

```
1  entity compa is port (  --Cabecera de la entidad, cuyo nombre es compa
2  a,b: in bit_vector(3 downto 0);
3  igual: out bit;
4
5      --a y b son las entradas de cuatro bits
6      --igual es la salida de un sólo bit
7
8  );
9  end compa;  --Se finaliza la entidad con la palabra clave end y el nombre de la misma (compa).
```

```
1  architecture behavioral of compa is
2  begin
3  comp: process (a, b)
4     begin
5      if a= b then
6       igual<='1';
7      else
8       igual<='0';
9      end if;
10    end process comp;
11 end behavioral;              --Finaliza el programa
```

Ejemplo extraído de http://bit.ly/WaKymm

- **Descripción de flujo de datos o dataflow**

  La información se transfiere de señal a señal y de la entrada a la salida sin el uso de asignaciones secuenciales, sino concurrentes. Es decir, en este estilo no se pueden usar los procesos.

  El comparador descrito de forma behavioral o de comportamiento se puede escribir usando el estilo dataflow de cualquiera de las dos formas siguientes (la entity es la misma que en descripción de comportamiento):

```
1  architecture dataflow1 of compa is
2  begin
3      igual<='1' when (a=b) else '0';
4  end dataflow1;  --Finaliza el programa
```

```
1  architecture dataflow2 of compa is
2  begin
3      igual<= not(a(0) xor b(0))
4          and not(a(1) xor b(1))
5          and not(a(2) xor b(2))
6          and not(a(3) xor b(3));
7  end dataflow2; --Finaliza el programa
```

Ejemplos extraídos de http://bit.ly/WaKymm

Se ha podido comprobar que el número de líneas de código en ambos casos es inferior que con la descripción de comportamiento

- **Descripción estructural o structural**

  Se describe un "netlist" de VHDL, en el que los componentes son conectados y evaluados instantáneamente mediante señales.

  No se suele usar este estilo únicamente en una arquitectura ya que resulta muy lioso y difícil de modificar, siendo de verdadera utilidad cuando es preciso crear una estructura grande y se desea descomponerla en partes para manejarla mejor, y para hacer la simulación de cada parte más sencilla.

  Se suele requerir el uso de señales auxiliares, y además paquetes y librerías accesorios, que se deben declarar al comienzo de la entidad.

  El código de comparador mediante descripción estructural se puede expresar así (la entity es la misma que mediante descripción de comportamiento y de flujo de datos):

```
1  architecture struct of compa is
2    signal x: bit_vector(0 to 3);
3  begin
4    u0: xnor2 port map (a(0),b(0),x(0));
5    u1: xnor2 port map (a(1),b(1),x(1));
6    u2: xnor2 port map (a(2),b(2),x(2));
7    u3: xnor2 port map (a(3),b(3),x(3));
8    u4: and4 port map (x(0),x(1),x(2),x(3),igual);
9    end struct;          --Finaliza el programa
```

  Ejemplo extraído de http://bit.ly/WaKymm

- **Descripción mixta**

  Se combinan dos o más de los estilos descritos

- **Conclusión final**

  Dado que en esta asignatura sólo se va a ver una introducción a VHDL nos conformaremos con realizar ejercicios con descripción de comportamiento

Ejemplos con descripción de comportamiento

- **Puerta OR de dos entradas**

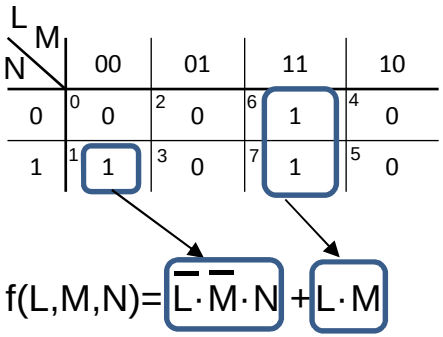
```
1  entity or2 is port (
2     a, b : in bit;                          --2 canales de entrada
3     c: out bit                           --1 canal de salida
4     );
5   end entity  or2 ;
6
7  architecture arch_or2   of   or2   is              --Cabecera del programa
8    begin                               --Comienza el programa
9           c <= a or b;
10  end architecture arch_or2 ;                 --Finaliza el programa
```

- **Puerta OR de dos entradas y NOT en el mismo código**

```
1  entity or_not is port (
2     a,b : in bit;                        --2 canales de entrada
3     c_or, c_not : out bit                --2 canales de salida
4     );
5   end and_or_not;
6
7  architecture arch_aon of and_or_not is  --Cabecera del programa
8    begin                                 --Comienza el programa
9      a_o_x: process (a, b)               --Cabecera de un proceso
10       begin                             --Empieza el proceso
11         c_or <= a or b;
12         c_not <=  not a;
13     end process a_o_x ;                 --Acaba el proceso
14  end arch_aon;                          --Finaliza el programa
```

- **Utilización de condiciones: implementación de una tabla de verdad**

L	M	N	y=f(X_1,X_2,X_3)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1




```
1   -- library declaration
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4   -- entity
5   entity my_ckt_f3 is
6   port ( L,M,N : in std_logic;
7   F3 : out std_logic);
8   end my_ckt_f3;
9   -- architecture
10  architecture f3_2 of my_ckt_f3 is
11  begin
12      process (L,M,N)
13          begin
14              if (L='0' and M='0' and N='1') then
15                  F3<='1';
16              elsif (L='1' and M='1') then
17                  F3<='1';
18              else
19                  F3<='0';
20              end if;
21      end process;
22  end f3_2;
```

a) Programación con condiciones

```
1   -- library declaration
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4   -- entity
5   entity my_ckt_f3 is
6   port ( L,M,N : in std_logic;
7   F3 : out std_logic);
8   end my_ckt_f3;
9   -- architecture
10  architecture f3_2 of my_ckt_f3 is
11  begin
12  F3<=((NOT L) AND (NOT M) AND N) OR (L AND M);
13  end f3_2;
```

b) Programación más sencillo con ejemplo extraído de Low-Carb VHDL Tutorial – Brian Mealy

- **Utilización de bucles: comprobador de paridad iterativo**

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   entity parity is
4     port (a : in std_ulogic_vector;
5           y : out std_ulogic);
6   end entity parity;
7
8   architecture iterative of parity is
9   begin
10    process (a) is
11      variable even : std_ulogic;
12    begin
13      even := '0';
14      for i in a'range loop
15        if a(i) = '1' then
16          even := not even;
17        end if;
18      end loop;
19      y <= even;
20    end process;
21  end architecture iterative;
```

Ejemplo extraído de Digital System Design with VHDL – Mark Zwolinski

Explicación:

Se va mirando cada elemento a(i) del vector a en el bucle (loop) de la arquitectura

Inicialmente la variable even vale 0, y cada vez que se detecta un elemento a(i)=1, se invierte la variable even. Es decir, si vale 0, se pone a 1 y si vale 1 se pone a 0.

El resultado se deposita en la variable ‘y’ que es la variable de salida de la entity ‘parity’

- **Implementación de una máquina de estados – problema de la cerradura digital**

Planteamiento: el código que se presenta a continuación, corresponde a una máquina de Mealy, que representa una cerradura digital que se abrirá por medio de una combinación de ceros y unos. La cerradura tiene las siguientes características:

- La combinación es '0-1-1', donde el uno es el primer valor de la secuencia que entra y el cero el último.
- Si la cadena introducida es correcta, se obtiene señal de salida que causa la apertura de la cerradura.
- Una vez que la cerradura está abierta, el introducir '0-0', hace que la cerradura se cierre. El control se resetea entonces a su estado inicial.

Solución inicial: tal y como dice el enunciado, la secuencia correcta para la apertura de la cerradura es: '0-1-1'; para cerrar la puerta se ha de introducir, estando la misma abierta, dos ceros seguidos.

Se considera que: z = 0 ⇨ puerta cerrada.

z = 1 ⇨ puerta abierta.

Los estados del sistema son:

q0: Estado Inicial. No ha llegado ningún uno.

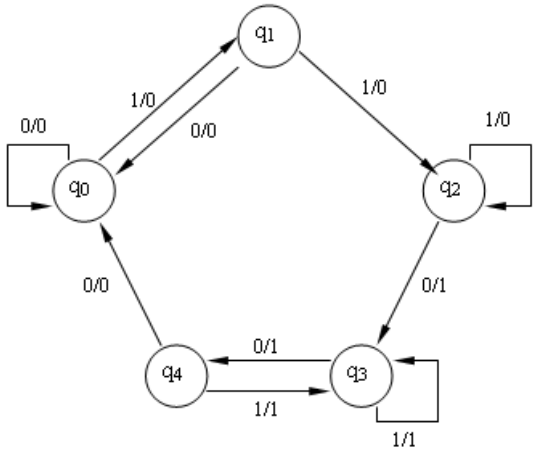
q1: Ha llegado '1'.

q2: Ha llegado la secuencia '1-1'.

q3: Ha llegado la secuencia '0-1-1'.

q4: Ha llegado '0' con la puerta abierta.

Tabla de comportamiento y diagrama de estados



- **Implementación de una máquina de estados – problema de la cerradura digital**

  Solución final con VHDL:

```
1  entity cerradura is port(
2     x, clock : in bit;
3     z : out bit
4     );
5  end cerradura;
6
7  architecture arch_cerradura of cerradura is
8     type tipo_estado is (q0, q1, q2, q3, q4);
9     signal estado_actual, estado_siguiente: tipo_estado;
10    begin
11       process (estado_actual, x)
12       begin
13          case estado_actual is
14           when q0 =>
15             if x = '0' then
16                z <= '0';
17                estado_siguiente <= q0;
18             else
19                z <= '0';
20                estado_siguiente <= q1;
21             end if;
22           when q1 =>
23             if x = '0' then
24                z <= '0';
25                estado_siguiente <= q0;
26             else
27                z <= '0';
28                estado_siguiente <= q2;
29             end if;
30          when q2 =>
31             if x = '0' then
32               z <= '1';
33                estado_siguiente <= q3;
34             else
35                z <= '0';
36                estado_siguiente <= q2;
37             end if;
38          when q3 =>
39             if x = '0' then
40                z <= '1';
41                estado_siguiente <= q4;
42             else
43                z <= '1';
44                estado_siguiente <= q3;
45             end if;
46          when q4 =>
47             if x = '0' then
48                z <= '0';
49                estado_siguiente <= q0;
50             else
51                z <= '1';
52                estado_siguiente <= q3;
53             end if;
54           end case;
55       end process;
56       process
57       begin
58         wait until clock'event and clock = '1';
59         estado_actual <= estado_siguiente;
60       end process;
61 end arch_cerradura;
```