Lenguaje de descripción de hardware VHDL

Parte I

Martín O. Vázquez

Universidad Nacional del Centro de las Provincia de Buenos Aires

Introducción al lenguaje VHDL

¿Que es VHDL?

VHSIC (Very High Speed Integrated Circuit)
Hardware
Description
Language

Un lenguaje industrial estándar IEEE para la descripción de hardware.

Una lenguaje de alto nivel para la simulación y síntesis de circuitos.

VHDL: Objetivos

DOCUMENTACIÓN

- El código fuente del lenguaje es usado como documento

MODELADO

- Concepción sencilla de sistemas digitales. Permite la descripción de un sistema en términos de lenguaje

SIMULACIÓN

- Simulación de la descripción para comprobar su funcionamiento.

SÍNTESIS

- Generación de una *netlist* de componentes a partir del modelado VHDL

VHDL: Ventajas

- Estándar
- Soporte gubernamental
- Soporte industrial
- Portabilidad
- Capacidad de modelación
- Reusabilidad
- Independencia de los fabricantes y tecnologías
- Metodología de diseño

VHDL: Unidades de diseño

Entidad

- Define la interface externa del modelo

SIMBOLO

Configuración

- Se usa para relacionar una arquitectura con una entidad

ENTIDAD ⟷ ARQUITECTURA

Arquitectura

- Define la funcionalidad interna del modelo

ESQUEMA

Package

- Librería de componentes y funciones

DECLARACION CUERPO

Entidad y Arquitectura: 1er Nivel de abstracción

- Una unidad de hardware se visualiza como una caja negra
  - La interfaz de la caja negra está completamente definida
  - El interior está oculto
- En VHDL la caja se denomina entidad
  - La **ENTITY** describe la E/S del diseño
- Para describir su funcionamiento se asocia una implementación que se denomina arquitectura
  - La **ARCHITECTURE** describe el contenido del diseño

PORTS: Puertos de una entidad

Ports = Canales de comunicación

Cada una de las posibles conexiones se denomina un PORT y consta de:

- Un **nombre** que debe ser único dentro de la entidad
- Una **lista de propiedades** como:
  - la dirección del flujo de datos, entrada, salida, bidireccional y se conoce como **MODO** del puerto
  - los valores que puede tomar el puerto: `1´, `0´ o (`Z´), etc., los valores posibles dependen de lo que se denomina **TIPO** de señal
- Los puertos son una clase especial de señales que adicionalmente al tipo de señal añade el modo

PORTS: Modos de un puerto

Modo de los puertos

Indican la dirección y si el puerto puede leerse o escribirse dentro de la entidad

- **IN** Una señal que entra a la entidad y no sale. La señal puede ser leída pero no escrita
- **OUT** Una señal que sale de la entidad y no es usada internamente. La señal no puede ser leída dentro de la entidad
- **BUFFER** Una señal que sale de la entidad y también es realimentada dentro de la entidad. Esta señal puede ser leída
- **INOUT** Una señal que es bidireccional entrada/salida de la entidad

VHDL: Descripción de entidad

La declaración VHDL de la caja negra

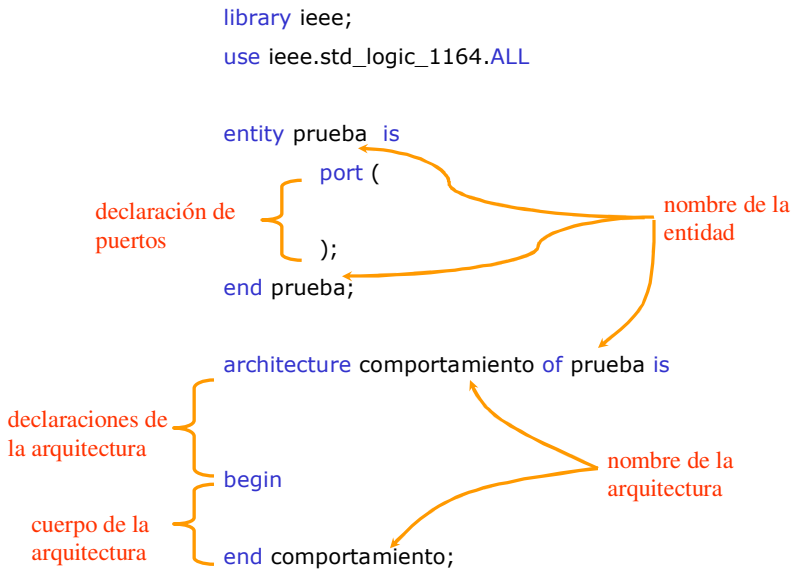
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL
ENTITY mi_componente  IS PORT (
        clk, rst :      IN std_logic;
        d:              IN std_logic_vector( 7 DOWNTO 0 );
        q:              OUT std_logic_vector( 7 DOWNTO 0 );
        co :            OUT std_logic);
END mi_componente;
```

MODO **TIPO**

Estructura de un diseño VHDL

```
ENTITY nombre_entidad IS
        genéricos
        declaración de puertos
END nombre_entidad;
ARCHITECTURE nombre_arquitectura OF nombre _entidad IS
        tipos de datos enumerados
        declaración de señales internas
        declaración de componentes
BEGIN
        sentencias de asignación de señales
        sentencias procesos (process)
        instanciación de componentes
END nombre_arquitectura;
```

Estructura de un diseño VHDL



Arquitectura

- La arquitectura describe lo que hay en la entidad, es decir el contenido de la caja negra
- Una arquitectura pude describirse con diferentes estilos de modelado
  - Estructural
  - Flujo de Datos
  - Algorítmico
- Y diferentes niveles de abstracción
  - Lógico o de puertas
  - Arquitectural o de transferencia de registros (RTL)
  - Funcional o Comportamental

Estilos de modelado: Algorítmico

- Descripciones similares a programas de software. Refleja funcionalidad del circuito mediante procesos concurrentes que poseen descripciones secuenciales del algoritmo

```
entity comparador  is
            port (A, B :in bit;
                        C:out bit);
end  comparador;
```

```
architecture algoritmo of comparador is
begin
            process
            begin
                        if  (A=B) then
                                    C <= '1';
                        else
                                    C <= '0';
                        end if ;
                        wait on A,B;
            end process ;
end algoritmo;
```

Estilos de modelado: Flujo de Datos

- Descripciones basadas en expresiones y ecuaciones que reflejan el flujo de información y las dependencias entre datos y operaciones

```
entity comparador  is
            port (A, B :in bit;
                        C:out bit);
end  comparador;
```

```
architecture flujo_datos of comparador is
begin
            C <= (A and B) or (not (A) and not (B));

end flujo_datos;
```

Estilos de modelado: Estructural

- Una unidad de alto nivel se divide en unidades de mas bajo nivel
- Descripción que contiene los componentes y las conexiones entre los mismos

```
entity comparador  is
            port (A, B :in bit;
                        C:out bit);
end  comparador;
```

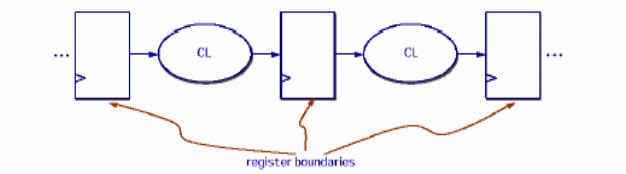
```
architecture estructural of comparador is;
            component XOR2
                        port (O: out bit; I1, I2: in bit);
            end component;

            component INV
                        port (O: out bit; I: in bit);
            end component;

            signal S: bit;
begin
            C1: XOR port map (O =>S, I1 =>A, I2 =>B);
            C2: INV port map (C, S);
end estructural;
```

Niveles de abstracción

- **Funcional o Comportamental**:
  - Un algoritmo puro consiste en un conjunto de instrucciones ejecutadas secuencialmente que realizan una tarea
  - No se detallan relojes o retrasos
  - No es sintetizable (o sintetizables en casos limitados)
- **RTL:**
  - Es la entrada para la síntesis
  - Las operaciones se realizan en un ciclo de reloj especifico
  - No se detallan retrasos
- **Lógico o de puertas:**
  - Es la salida de la síntesis
  - Expresado en término de ecuaciones lógicas o puertas y elementos de una biblioteca (genérica o específica)
  - Se incluye información de retraso para cada puerta

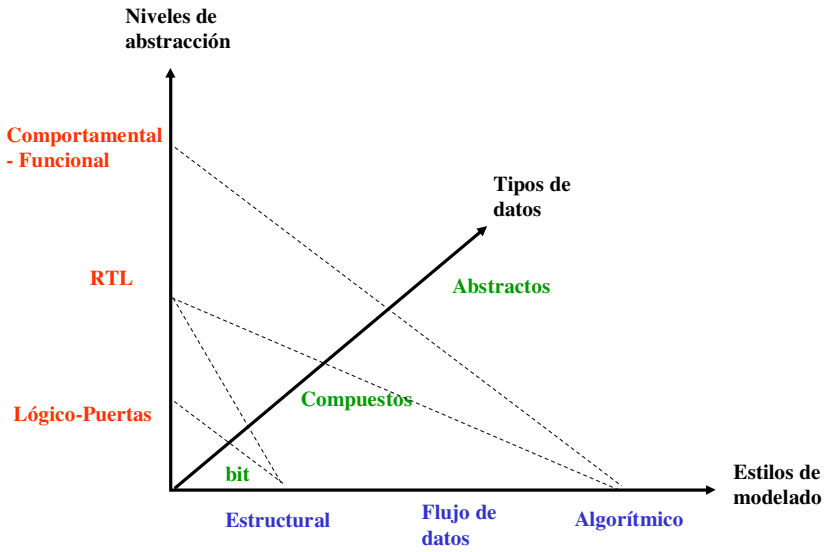
RTL: Register Transfer Level

- Flujo de datos entre registros y bloques funcionales



- Tienen en cuenta el ciclo de reloj
- Independiente de la tecnología
- Definición del sistema en términos de:
  - registros, lógica combinacional y operaciones

Nivel abstracción/Estilo de modelado



Package

Package son una manera conveniente de almacenar y utilizar la información relativa a un modelo

```
PACKAGE nombre_paquete IS
        declaración de constantes, tipos, señales,
        subprogramas, componentes, etc
END PACKAGE nombre_paquete;
```

```
PACKAGE BODY nombre_paquete IS
        declaración de tipos, señales, subprogramas...
        cuerpo de subprograma
                . . .
END PACKAGE BODY nombre_paquete;
```

Consta de dos partes:

Declaración de Package (Obligatoria) y Cuerpo de Package (Opcional)

Package

```
package filt_cmp is
        type state_type is (idle, tap1, tap2, tap3, tap4)
        component acc
                port (      x: in std_logic_vector(7 downto 0);
                            clk, first: in std_logic;
                            y: out std_logic_vector(7 downto 0));
        end component;
        function compare (variable a,b: integer): return boolean;
end filt_cmp;
```

Package Declaration

```
Package body filt_cmp is
        function compare (variable a,b: integer) is
        variable temp: boolean;
        begin
                if  (a<b) then
                        temp := true;
                else
                        temp := false;
                end if;
                return temp;
        end compare;
end filt_cmp;
```

Package Body

Tipos de datos

- VHDL es un lenguaje fuertemente tipado
  - A los objetos siempre se les asigna un tipo cuando se los declara
  - Las asignaciones sólo pueden hacerse entre objetos del mismo tipo

- Los tipos predefinidos son
  - Escalares: integer, floating point, enumerated, physical
  - Compuestos: array, record
  - Punteros: access
  - Archivos: file

Tipos básicos predefinidos

- BIT: sólo puede tomar los valores de `0´ o `1´. Para modelar señales digitales
- BIT_VECTOR: un array unidimensional (vector) de bits. Para modelar buses
- INTEGER: tipo entero. Usado como valor índice en lazos, constantes o val. genéricos
- BOOLEAN: tipo lógico. Puede tomar valor TRUE o FALSE
- REAL: tipo para números en coma flotante
- ENUMERATED: enumeración. Conjunto de valores definido por el usuario
  - Por ejemplo: `type estados is (inicio, lento, rápido)`

Usando arrays

- Los vectores se pueden definir tanto en rangos ascendentes como descendentes

```
signal a: bit_vector (0 to 3); -- rango ascendente
signal b: bit_vector (3 downto 0); -- rango descendente
        a <= "0101";  b<="0101";
```

Produce como resultado

a(0) = `0´;  a(1) = `1´; a(2) = `0´;  a(3) = `1´;

b(0) = `1´;  b(1) = `0´; b(2) = `1´;  b(3) = `0´;

- Una manera rápida de asignar valores a los vectores son los *aggregates*

```
a <= ( 0 => `1´, 1 => c and d, others => `Z´);
```

Tipo **std_logic**

- Los dos valores del tipo bit no alcanzan para modelar todos los estados de una señal digital en la realidad
- El paquete **IEEE.standard_logic_1164** define el tipo std_logic; que representa todos los estados de una señal real

U	No inicializado, valor por defecto
X	Desconocido fuerte, salida con múltiples fuentes en corto
0	Salida de una puerta con nivel lógico bajo
1	Salida de una puerta con nivel lógico alto
Z	Alta impedancia
W	Desconocido débil, terminación del bus
L	0 débil, resistencia de pull-down
H	1 débil, resistencia de pull-up
-	No importa, usado como comodín para síntesis

Tipo **std_logic**

- Para describir buses se utiliza el tipo std_logic_vector, que es un array de std_logic
- Los tipos std_logic y std_logic_vector son los estándares industriales
- Todos los valores son válidos en un simulador VHDL, sin embargo solo: `1´, `0´, `Z´, `L´, `H´ y `-´ se reconocen para la síntesis
- En el paquete **IEEE.standard_logic_1164** aparecen otros dos tipos: std_ulogic y std_ulogic_vector. Son los mismos pero sin haber pasado por la función de resolución:
  - Esta función decide cuál debe ser el valor de la señal cuando tienen dos fuentes que le asignan valores distintos.
  - Por ejemplo, si una fuente asigna un `1´ y la otra un `L´, la función de resolución dice que la señal se queda a `1´

Operadores definidos en VHDL

- **Lógicos:** and, or, not, xor y xnor
- **Relacionales:** **=** (igual), **/=** (distinto), **<** (menor), **<=** (menor o igual), **>** (mayor) y **>=** (mayor o igual)
- **Adición:** + (suma), - (resta) y & (concatenación de vectores)
- **Multiplicativos:** * (multiplicación), / (división), rem (resto) y mod (módulo)
- **Signos** (unarios): + , -
- **Desplazamiento** (bit_vector): sll, srl, sla, sra, rol y ror
- **Misceláneos:** abs (valor absoluto), ** (exponenciación), not (negación unario)

Construcciones básicas en VHDL

Procesos constituidos por instrucciones que se ejecutan y evalúan secuencialmente. Durante la ejecución de las instrucciones secuenciales de un proceso el tiempo no avanza.

Sentencias concurrentes en la arquitectura se encuentran fuera de los procesos y se evalúan concurrentemente (asignaciones concurrentes)

Instanciaciones de componentes, también se evalúan concurrentemente

Objetos disponibles en VHDL

Constante es una asociación de un nombre a un determinado valor

```
constant pi: real := 3.1416;
```

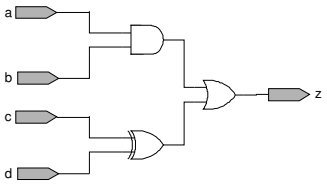
Variable es un almacenamiento local de un dato temporal visible solo en el interior de un proceso. Su valor es modificado por una asignación (**:=**)

```
variable parada: std_logic := `0´;
```

Señal es la modelización de un cable, el cual puede conectar puertos (E/S) de entidades y también conectar procesos dentro de la arquitectura. Es una forma de onda que cambia con el tiempo, por lo que su asignación (**<=**) no es inmediata, sólo tiene efecto cuando avanza el tiempo

```
signal parada: std_logic <= `0´;
```

Declaración de señales



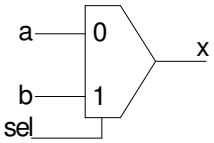
declaración de señales

```
libary ieee;
use ieee.std_logic_1164.all;
entity simple is
        port (a,b,c,d : in std_logic;
              z: out std_logic);
end simple;
architecture logica of simple is
signal x, y : in std_logic;

begin
        x <= a and b;
        y <= c xor d;
        z <= x or y;
end logica;
```

a, **b**, **c**, **d** y **z** son señales por defecto

x e **y** son señales que necesitan ser declaradas

Asignaciones concurrentes



Asignación por ecuaciones booleanas

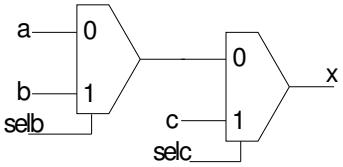
```
x <= (a and not sel) or (b and sel) ;
```

Asignación condicionales de las señales

```
x <= a when sel='0' else
     b;
```

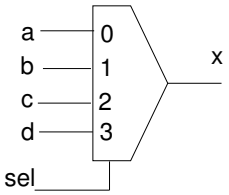
procesos implícitos

Asignaciones condicionales




```
x <= c when selc='1' else
     b when selb='1' else
     a;
```

procesos implícitos




```
with sel select
     x <= a when sel="00",
          b when sel="01",
          c when sel="10",
          d when others;
```

la cláusula others cubre todas las posibilidades que no son descritas

Procesos

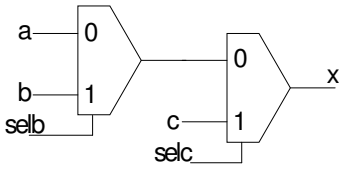
- Un modelo VHDL puede considerarse como un conjunto de procesos ejecutándose en paralelo
- Semánticamente un proceso es un bucle infinito de instrucciones secuenciales
- Por tanto, todo proceso debe tener una instrucción wait, que puede sustituirse por una lista de sensibilidad
  - **Lista de sensibilidad** es un conjunto de señales que activan el proceso cuando se produce un cambio en alguna de ellas
  - **Instrucción wait**:
    - wait on a,b (cambio en a o b),
    - wait for retraso (un cierto tiempo),
    - wait until condición

```
process                          (a, b)
   declaraciones
begin
   sentencias secuenciales
                                 wait on a,b
end process;
```

Procesos

- Un proceso puede ser Explícito o Implícito
  - Proceso Implícitos. La lista de sensibilidad se obtiene de la propia expresión:
    - Asignaciones concurrentes
    - Instanciación de componentes
  - Proceso Explícitos. La lista de sensibilidad se declara explícitamente y en su constitución se utilizan
    - Sentencias secuenciales

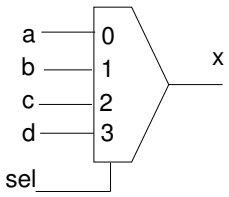
Sentencias secuenciales: IF-THEN-ELSE




```
process (selc, selb, a, b, c)
begin
        if (selc = '1') then
                x <= c;
        elsif (selb = '1') then
                x <= b;
        else
                x <= a;
        end if;
end process;
```

Con prioridad: las condiciones son evaluadas en orden de arriba hacia abajo. La primera condición *TRUE* hace que se evalúe la instrucción asociada. Si todas las condiciones son falsas se evalúa la instrucción asociada *ELSE*

Sentencias secuenciales: CASE




```
process (sel, a, b, c, d)
begin
        case sel is
                when "00" =>
                        x <= a;
                when "01" =>
                        x <= b;
                when "10" =>
                        x <= c;
                when others =>
                        x <= d;
        end case;
end process;
```

Las condiciones son evaluadas de una vez, sin prioridad. La cláusula *OTHERS* sirve para evaluar los casos no especificados.

Bucles secuenciales: LOOP

- Bucle infinito con instrucción de salida

```
[label_loop] loop
        sentencias secuenciales
end loop [label_loop];
```

- Bucle infinito con condición

```
while <condición> loop
        sentencias secuenciales
end loop;
```

- Bucle **for** iterativo

```
for <identificador> in <rango> loop
        sentencias secuenciales
end loop;
```

Asignación de señales en buses

```
signal tmp: std_logic_vector(7 downto 0);
```

- Todos los bits

```
tmp <= "10100011";
tmp <= x"A3"; -- VHDL 93
```

- Un solo bit

```
tmp(7) <= `1´;
```

- Un rango de bits

```
tmp(3 downto 0) <= "0011";
```

Uso correcto de las asignaciones a señales

- Agregados para compactar y hacer código más rápido en simulación

```
y <= others => `0´;
y <= (`0´, `1´, a xor b, a and b);
y <= (3 => a xor b, 1 =>`1´, 2 => '0', a and b);
y <= y(2 dowto 0) &`0´;
```

- No usar bucles para inicializar señales

```
for i in 0 to 4 loop
        a(i) <= `0´;
end loop;
```

```
a(i) <= others =>`0´;
```

- No usar bucles para desplazar datos

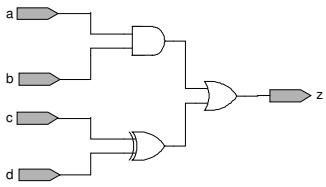
```
for i in 3 downto 0 loop
        if (i /= 0)  then
                d(i) <= d(i-1);
        else
                d(i) <= `0´;
        end if;
end loop;
```

```
d <=d(2 downto 0) & `0´;
```

Diseño concurrente en VHDL

- VHDL soporta la noción de "ejecución concurrente"
- Los métodos para describir la concurrencia dentro de una arquitectura son:
  - Instanciaciones
  - Asignaciones concurrentes
  - Procesos. Un proceso es un conjunto de sentencias secuenciales que se ejecutan según el orden.
- El orden relativo de procesos y asignaciones concurrentes dentro de una arquitectura no es relevante
- Las señales se utilizan para controlar la activación de los procesos

Instrucciones concurrentes




```
x <= a and b;
y <= c xor  d after 10 ns;
z <= x or y after 5 ns;
```

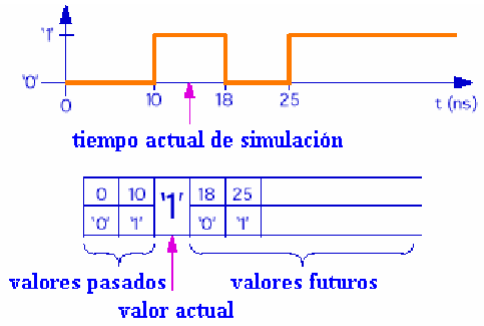
Asignaciones concurrentes

- Una instrucción se ejecuta cuando cambia una de sus entradas
- La asignación de los nuevos valores se realiza una vez ejecutadas todas las instrucciones cuyas entradas han cambiado.
- Si como consecuencia del nuevo valor cambia alguna de las entradas de otra instrucción, se ejecuta dicha instrucción

Fundamentos: Conceptos de señal

- Las señales permiten comunicación entre procesos
- Cada señal posee una historia. Los valores pasados, presente y futuros (previstos) se almacenan en forma permanente.

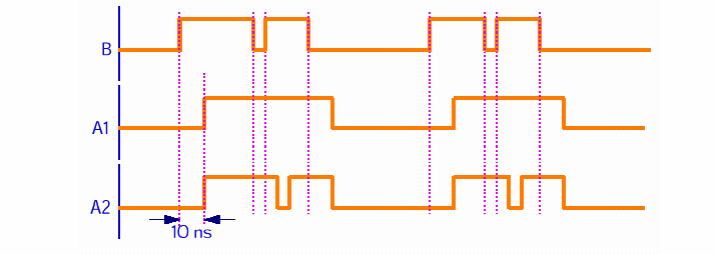
```
s <= '1' after 10 ns, '0' after 18 ns, '1' after 25 ns;
```



Fundamentos: Tipos de retardo

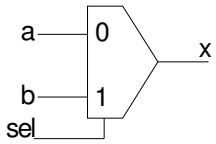
- Las señales poseen dos tipos de retardo: inercial (por defecto) y de transporte

```
A1 <= B after 10 ns;
A2 <= transport B after 10 ns;
```



Instrucciones secuenciales: El proceso

- Las sentencias dentro de un proceso son evaluadas secuencialmente
- Un proceso puede estar *activo* o *inactivo*
- Típicamente un proceso posee una lista de sensibilidad: El proceso pasa a estar *activo* cuando cambia un valor de esta lista
- También puede tener una o varias sentencias wait, que hacen que el proceso pase a *inactivo*
- El tiempo avanza cuando el proceso pasa a inactivo



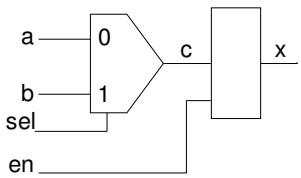

```
process (a, b, sel)
begin
        if  sel='0' then
                x <= a;
        else
                x <= b;
        end if;
end process;
```

Asignación de señales en procesos

La asignación de señales dentro de un proceso sólo se ejecutan cuando se suspende el proceso

- Las señales modelan conexiones físicas, por tanto, no sólo deben tener en cuenta el valor, sino el tiempo
- Para que un cable cambie de valor es necesario que el tiempo avance
- De la misma forma para que una señal cambie de valor es necesario que el tiempo avance
- El tiempo solo avanza cuando se suspende el proceso

Asignación de señales: end process

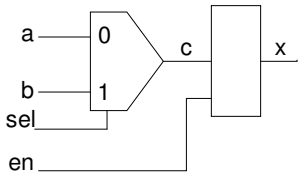


INCORRECTO:

Cuando *en*='1', a *x* se le asigna el valor previo de *c*

```
libary ieee;
use ieee.std_logic_1164.all;
entity mux2latch is
        port (a,b,sel, en : in std_logic;
                x: buffer std_logic);
end mux2latch;
architecture comp of mux2latch is
signal c : std_logic;
begin
        mux: process (a, b, sel, en)
        begin
                if (sel='0') then c<=a;
                else c<=b;
                end if;
                x <=(x and (not en)) or  (c and en);
        end process;   -- c  cambia aquí
end comp;
```

Asignación de señales: end process



CORRECTO: proceso con sentencias secuenciales y una asignación concurrente

Cuando *en*='1', a *x* se le asigna el valor actualizado de *c*

```
libary ieee;
use ieee.std_logic_1164.all;
entity mux2latch is
        port (a,b,sel, en : in std_logic;
                x: buffer std_logic);
end mux2latch;
architecture comp of mux2latch is
signal c : std_logic;
begin
        mux: process (a, b, sel)
        begin
                if (sel='0') then c<=a;
                else c<=b;
                end if;
        end process;   -- c  cambia aquí
        x <=(x and (not en)) or  (c and en);
end comp;
```

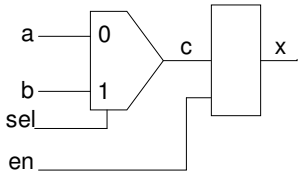
Variables

- Cuando no se puede usar una asignación concurrente, el problema anterior se soluciona utilizando una variable
  - Las variables solo pueden existir dentro de un proceso.
  - Las variables pueden ser de los mismos tipos de una señal (signal)
  - Las variables son independientes del tiempo, el valor asignado a una variable está disponible inmediatamente
  - La asignación de valores se realiza con **:=**

```
c := a and b;
```

Solución con variables

Solución usando una variable dentro de un proceso




```
architecture comp of mux2latch is
begin
        mux: process (a, b, sel)
          variable c : std_logic;
        begin
                if (sel='0') then c:=a; -- inmediato
                else c:=b;
                end if;
                x <=(x and (not en)) or  (c and en);
        end process;
end comp;
```

Semántica de variables y señales

	Señales	Variables
Sintaxis	destino <= fuente	destino := fuente
Utilidad	modelan nodos físicos del circuito	representan almacenamiento local
Visibilidad	global (comunicación entre procesos)	local (dentro de proceso)
Comportamiento	se actualizan cuando avanza el tiempo (se suspende el proceso)	se actualizan inmediatamente

Lenguaje de descripción de hardware VHDL

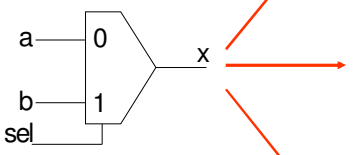
Parte II

Martín O. Vázquez

Universidad Nacional del Centro de las Provincia de Buenos Aires

Circuitos básicos: Multiplexores

Multiplexores por asignación concurrente: simple, condicional y por selección



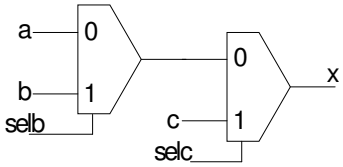

```
x <= (a and not sel) or (b and sel) ;
```

```
x <= a when sel='0' else
     b;
```

```
with sel select
     x <= a when sel='0',
          b when sel='1',
          '0' when others;
```

Circuitos básicos: Multiplexores

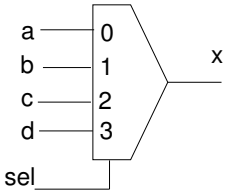
Multiplexores usando sentencias secuenciales: if-then-else. Con prioridad




```
process (selc, selb, a, b, c)
begin
          if (selc = '1') then
                    x <= c;
          elsif (selb = '1') then
                    x <= b;
          else
                    x <= a;
          end if;
end process;
```

Circuitos básicos: Multiplexores

Multiplexores usando sentencias secuenciales: case. Sin prioridad

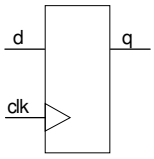


Mux 4-1

```
process (sel, a, b, c, d)
begin
        case sel is
                when "00" =>
                        x <= a;
                when "01" =>
                        x <= b;
                when "10" =>
                        x <= c;
                when others =>
                        x <= d;
        end case;
end process;
```

Registros

- Existen dos métodos para crear flip-flops
  - Instanciación del componente flip-flop desde una librería
  - Utilizar un proceso sensible al flanco de reloj



Registro: Instanciación de componentes

- Ejemplo flip-flop tipo D

```
libary ieee;
use ieee.std_logic_1164.all;
entity ffd is
        port (d, clk : in std_logic;
                q: out std_logic);
end ffd;

use ieee.rtlpkg.all;
architecture instffd of ffd is
begin
        flipflop:  dff port map (d,clk,q);
        -- componente dff definido en  rtlpkg
end instffd;
```

Registro: Descripción basada en proceso

- Ejemplo flip-flop tipo D

```
libary ieee;
use ieee.std_logic_1164.all;
entity ffd is
        port (d, clk : in std_logic;
                q: out std_logic);
end ffd;

architecture compffd of ffd is
begin
        flipflop: process (clk)
        begin
                if (clk´event and clk='1') then
                        d <= q;
                end if;
        end process;
end compffd;
```

Registro: Descripción basada en proceso

El sintetizador infiere que debe crear registro para la señal *q* basándose en:

- El elemento es únicamente sensible a la señal de reloj clock
- La expresión ***clock´event and clock='1'*** implica que la asignación de la señal se realiza en el flanco de subida de reloj
  - Se sintetiza un elemento síncrono
- La especificación incompleta de la sentencia ***IF-THEN*** por faltar la cláusula ***ELSE*** implica que si la condición ***clock´event and clock='1'*** no se satisface (no hay flanco de subida), *q* debe mantener su valor hasta la siguiente asignación (memoria implícita)

Ejemplos básicos de codificación

Tipos de registro

- Flip Flop D

```
ff: process (clk)
begin
        if (clk´event and clk='1') then
                output <= input;
        end if;
end process;
```

- Flip Flop D con set asíncrono

```
ff_SA: process (clk, st)
begin
        if (st='1') then
                output <= others => '1';
        elsif (clk´event and clk='1') then
                output <= input;
        end if;
end process;
```

Ejemplos básicos de codificación

Tipos de registro

- Flip Flop D con reset síncrono

```
ff_RS: process (clk)
begin
      if (clk´event and clk='1') then
          if (rst= '1') then
              output <= others => '0';
          else
              output <= input;
          end if;
      end if;
end process;
```

- Flip Flop D con reset asíncrono

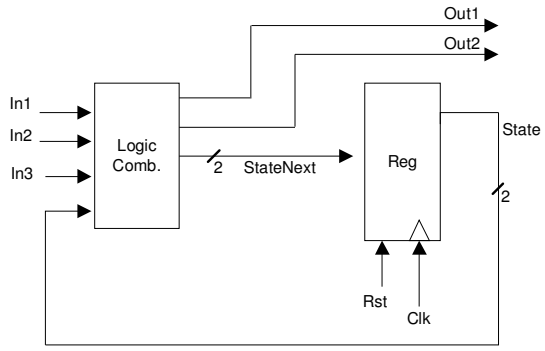
```
ff_SA: process (clk, rst)
begin
      if (rst='1') then
            output <= others => '0';
      elsif (clk´event and clk='1') then
            output <= input;
      end if;
end process;
```

Ejemplos básicos de codificación

Contadores

```
PCont: process (clk, rst)
begin
      if (rst='1') then
            cont <= (others => '0');
      elsif (clk´event and clk='1') then
            cont <= cont + 1;
      end if;
end process;
```

Metodología: Diseño de circuitos secuenciales



Metodología: Diseño de circuitos secuenciales

Máquinas de estados: FSM

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
architecture rtl of fsm is
  type TypeState is (St0, St1, St2, St3);
  signal State, StateNext :TypeState;
  signal in1, in2, in3: bit;
  signal out1, out2: bit;
  signal rst, clk: bit;

            . . .
```

Metodología: Diseño de circuitos secuenciales

Máquinas de estados: FSM

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
begin
LogComp: process (State, in1, in2, in3)
          begin
            case State is
                when St0 => out1 <= '0';
                            out2 <= '0';
                            StateNext <= St1;
                when St1 => out1 <= '1';
                            if (in1='1') then
                              StateNext <= St2;
                            else
                              StateNext <= St3;
                            end if;
                when St2 =>
                            . . .
                when St3 =>
                            . . .
            end case;
          end process;
```

Metodología: Diseño de circuitos secuenciales

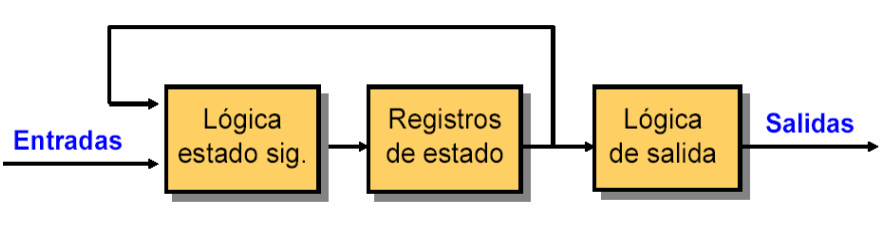
Máquinas de estados: FSM

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
PReg: process (clk, rst)
begin
      if (rst='1') then
            State <= St0;
      elsif (clk´event and clk='1') then
            State <= StateNext;
      end if;
end process;
```

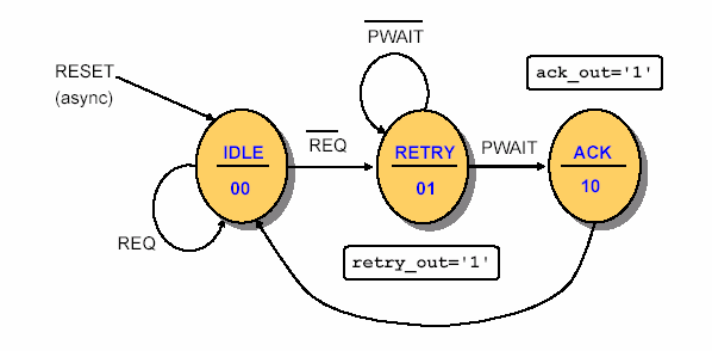
FSM: Máquinas de Moore

- FSM MOORE: Una máquina de estados en la que las salidas cambian solo cuando cambia el estado
- Las salidas son decodificadas a partir del valor de los estados



Ejemplo: Generador “wait states”

Diagrama de estados



Ejemplo: Declaración de la entidad

- La declaración de entidad es la misma para todas las implementaciones

```
libary ieee;
use ieee.std_logic_1164.all;

entity Moore is port (
            rst, clk : in std_logic;
            req, pwait : in std_logic;
            retry_out, ack_out: out std_logic);
end Moore;
```

Ejemplo: Solución 1

- Salidas combinacionales decodificadas a partir de los estados

```
architecture  archMoore of Moore is
        type TypeStates is (idle, retry, ack);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                elsif (clk´event and clk='1') then
                        case wait_gen is
                                when idle =>
                                        if  req='0'  then wait_gen <= retry;
                                        else wait_gen <= idle;
                                        end if;
```

Ejemplo: Solución 1

```
                                        when retry =>
                                                if  pwait='1'  then wait_gen <= ack;
                                                else wait_gen <= retry;
                                                end if;

                                        when ack =>  wait_gen <= idle;

                                        when others => wait_gen <= idle;
                                end case;
                        end if;
                end process;

                retry_out <= '1' when (wait_gen=retry) else '0';
                ack_out <= '1' when (wait_gen=ack) else '0';

end archMoore;
```

Ejemplo: Solución 2

- Salidas registradas decodificadas desde el valor de los estados

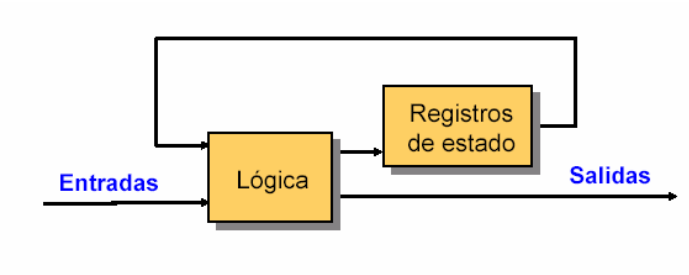
```
architecture  archMoore of Moore is
        type TypeStates is (idle, retry, ack);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                        retry_out <= '0';
                        ack_out <= '0';
                elsif (clk´event and clk='1') then
                        retry_out <= '0';
```

Ejemplo: Solución 2

```
case wait_gen is
    when idle => if req='0' then wait_gen <= retry;
                    retry_out <= '1';
                    ack_out <= '0';
                 else
                    wait_gen <= idle;
                    ack_out <= '0';
                 end if;
    when retry => if pwait='1' then wait_gen <= ack_out;
                    ack_out <= '1';
                  else
                    wait_gen <= retry_out;
                    retry_out <= '1';
                    ack_out <= '0';
                  end if;
    when ack => wait_gen <= idle;
                ack_out <= '0';
    when others => wait_gen <= idle;
                   ack_out <= '0';
  end case;
end if;
```

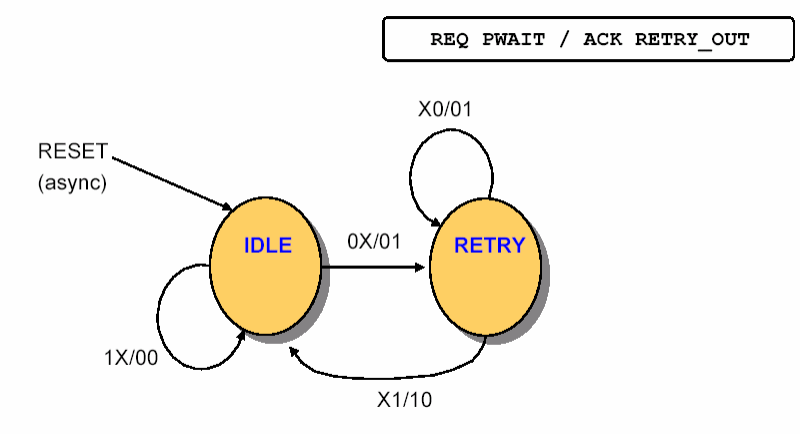
FSM de Mealy

- Las salidas cambian por un cambio de estado o por un cambio en el valor de las entradas
  - Hay que tener mucho cuidado con las entradas asíncronas



Ejemplo: Generador “wait states”

Diagrama de estados



Ejemplo: Solución

```
architecture  archMealy of Mealy is
        type TypeStates is (idle, retry);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                elsif (clk'event and clk='1') then
                        case wait_gen is
                                when idle => if  req='0'  then wait_gen <= retry;
                                             else wait_gen <= idle;
                                             end if;

                                when retry => if  pwait='1'  then wait_gen <= idle;
                                              else wait_gen <= retry;
                                              end if;
```

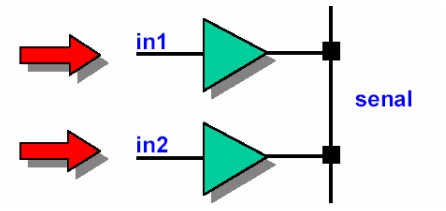
Ejemplo: Solución

```
                                        when others => wait_gen <= idle;
                                end case;
                        end if;
                end process;

                retry_out <= '1' when (wait_gen=retry and pwait='0') or
                                      (wait_gen=idle and req='0')  else '0';

                ack_out <= '1' when (wait_gen=retry and pwait='1') else '0';

end archMealy;
```

Concepto de *driver* de una señal

- El *driver* es el elemento que da valores a una señal
- Para cada señal que se le asigna un valor dentro de un proceso se crea un *driver* para esa señal
  - Independientemente de cuanto veces se le asigne valor a una señal, se crea un único *driver* por proceso
  - Tanto para procesos explícitos como implícitos
  - Cuando hay múltiples *drivers* se usa una función de resolución

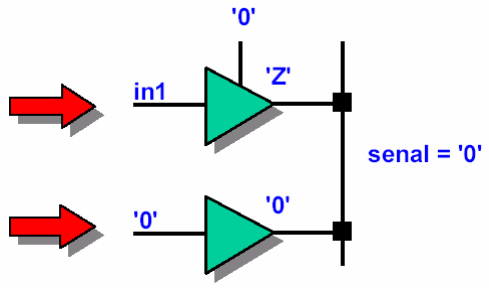
```
process (in1)
begin
        senal <= in1;
end process;

senal <= in2;
```



Inferencia triestado

- Cuando se quiere que un *driver* de la señal se quede en alta impedancia, se le asigna a la señal el valor 'Z'
  - Es válido para el tipo std_logic
- Igual que ocurre en la realidad, el estado de la señal lo fijará el *driver* que no esté en alta impedancia

```
senal <= in1 when ena='0'
                else 'Z';

senal <= '0';
```



Ejemplos de inferencia de buffer triestado

- Con asignación condicional

```
a_out <= a when enable_a='1' else 'Z';
b_out <= b when enable_b='1' else 'Z';
```

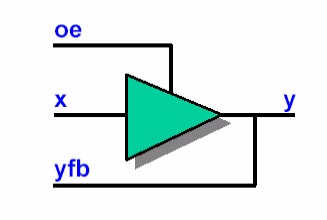
- Con un proceso

```
process (enable_a, a_out)
begin
        if (enable_a = '0') then
                a_out <= a;
        else a_out <= 'Z';
end process;
```

Soluciones bidireccionales

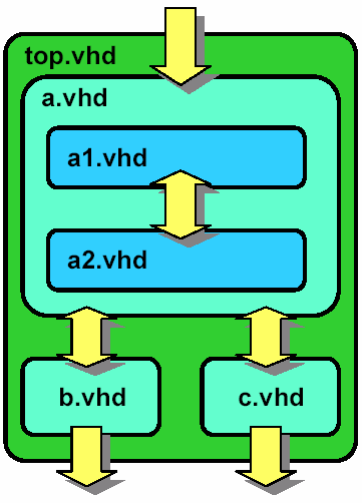
- En este caso la señal tiene *drivers* externos, fuera de la entidad

```
entity bufoe is port  (
            x, oe : in std_logic;
            y: inout std_logic;
            yfb : out std_logic);
end bufoe;

architecture simple of bufoe is
begin
            y <= x when oe='1'  else 'Z';
            yfb <= y;
end simple;
```



Diseño jerárquico

- Componentes pequeños son utilizados por otros más grandes
- Es fundamental para la reutilización de código
- Permite mezclar componentes creados con distintos métodos de diseño: VHDL, Verilog, esquemáticos
- Genera diseños más legibles y más portables
- Necesario para estrategias de diseño *top-down* o *botton-up*



Utilización de componentes

- Antes de poder usar un componente, se debe declarar
  - Especificar sus puertos (PORT)
  - Especificar parámetros (GENERIC)
- Una vez instanciado el componente, los puertos de la instancia se conectan a las señales del circuito usando PORT MAP
- Los parámetros se especifican usando GENERIC MAP
- La declaración de los componentes se puede hacer en un PACKAGE
  - Para declarar el componente, sólo habrá que importar el PACKAGE
  - La declaración de los componentes no aporta nada al lector

Ejemplo de diseño jerárquico: Top-Level

```
entity mux4to1 is port ( a,b,c,d : in std_logic;
                         sel : in std_logic_vector(1 downto 0);
                        o : out std_logic);
end mux4to1;

use  work.mymuxpkg.all;

architecture  InstComp of mux4to1 is
          signal r: std_logic_vector(1 downto 0);
begin
          mux0: mux2to1 port map  (i0 => a, i1 => b, s => sel(0), o => r(0));
          mux1: mux2to1 port map  (i0 => c, i1 => d, s => sel(0), o => r(1));
          mux2: mux2to1 port map  (r(0), r (1), sel(1), o);
end  InstComp;
```

Declaración del componente en un package

Asociación por nombre

Asociación por posición

Ejemplo de diseño jerárquico: componente inferior

```
entity mux2to1 is port ( i0, i1, s : in std_logic;
                         o : out std_logic);
end mux2to1;

architecture  archmux2to1 of mux2to1 is
begin
            o <= (i0 and not s) or (i1 and s);
end  archmux2to1;
```

Descripción del componente de nivel inferior

```
package mymuxpkg is
    component mux2to1 port (
                                i0,i1,s : in std_logic;
                                o : out std_logic);
    end component;
end mymuxpkg;
```

Creación del paquete

Instanciación repetitiva y condicional:GENERATE

- La instrucción GENERATE permite generar código iterativamente o condicionalmente
- Generación iterativa: Ideal para circuitos repetitivos como arreglos de componentes

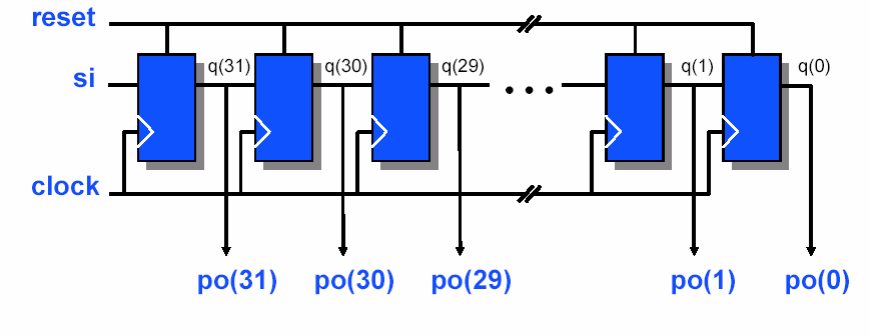
```
etiqueta: for <parámetro> in <rango> generate
            sentencias concurrentes
end generate;
```

- Generación condicional: Menos usada, por ejemplo para el primer elemento de un arreglo

```
etiqueta: if <condición> generate
            sentencias concurrentes
end generate;
```

Ejemplo de estructura repetitiva

Conversor serie paralelo de 32 bits



Ejemplo: Solución con GENERATE

```
entity sipo is port ( clk, rst, si : in std_logic;
                      po : out std_logic_vector(31 downto 0));
end sipo;

use  work.mypkg.all;

architecture  archsipo of sipo is
        signal q: std_logic_vector(31 downto 0);
begin
        gen: for i in 0 to 30 generate
                nxt: dsrff port map (clk, rst, q(i+1), q(i));
        end generate;
        beg: dsrff port map (clk, rst, si, q(31));
        po <= q;
end archsipo;
```

Configuración

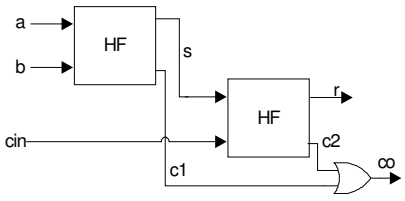
- Una configuración es una unidad de diseño
- Se usan para realizar asociaciones dentro de los modelos:
  - Asociar entidad con arquitectura
  - En la instanciación de un componente asociarlo a una entidad y arquitectura
- Muy utilizados en entornos de simulación: Una manera rápida y flexible de probar distintas alternativas de diseño
- Limitada o no soportada en entorno de síntesis

```
configuration <identificador> of <nombre_entidad> is
        for <nombre_arquitectura>
        end for
end configuration;
```

Aplicándolo a un componente en particular: for *nombre_instancia*: *nom_comp* use ...

Aplicándolo a todas las instancias: for all: *nom_comp* use entity *work.nom_entidad* (*nom_arq*)

Configuración: Ejemplo




```
entity FullAdder is port (a,b,cin: in std_logic;
                          r,co: out std_logic);
end FullAdder;
architecture structural of FullAdder is
        signal s,c1,c2: std_logic;
begin
        hf0: HalfAdder port map (a,b,s,c1);
        hf1: HalfAdder port map (cin,s,r,c2);
        u2:  or_gate port map (c1,c2,co);
end structural;
```

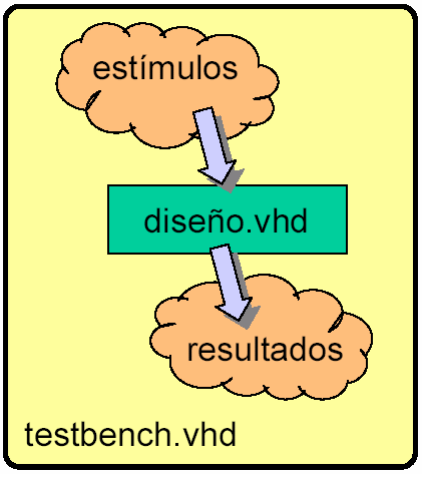
```
configuration a_config of FullAdder is
    for structural
        for all: HalfAdder use entity work.HalfAdder (algorithmic);
        end for;
        for u2: or_gate  use entity work.or_gate (behavioral);
        end for;
    end for;
end a_config;
```

Verificación con TestBenches

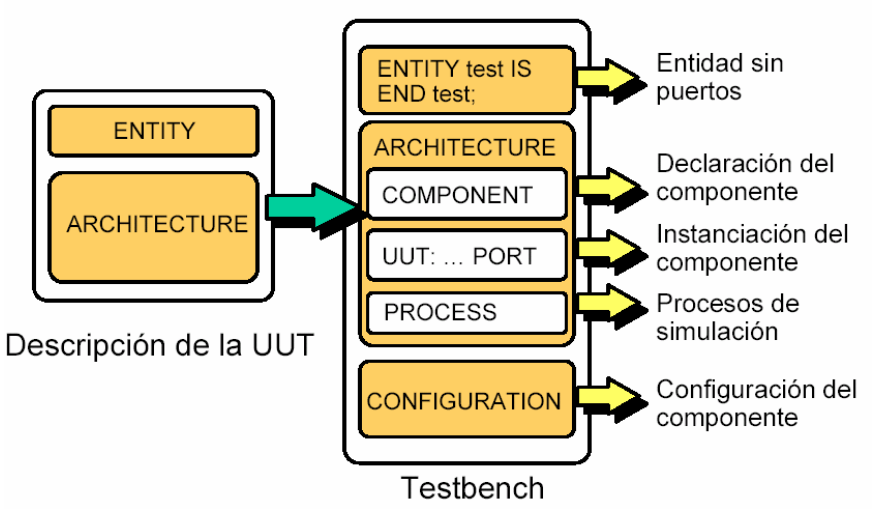
- Un diseño sin verificación no está completo:
  - Existen muchas maneras de verificar, la más utilizada es el banco de pruebas: TestBench
- Simular básicamente es:
  - Generar estímulos
  - Observar resultados
- Un TestBench es un código VHDL que automatiza estas dos operaciones
- Los TestBenches no se sintetizan
  - Se puede utilizar un VHDL algorítmico
  - Usualmente con algoritmos secuenciales

¿Cómo hacer un TestBench?

- Instanciar el diseño que vamos a verificar
  - El TestBench será el nuevo top-level
  - Será una entidad sin *ports*
- Escribir el código que:
  - Genera los estímulos
  - Observa los resultados
  - Informa al usuario



Instanciando la unidad bajo test (UUT)



Generando estímulos

- Dar valores a las señales que van hacia las entradas de la UUT
  - En síntesis no tiene sentido el tiempo
  - En los testbench el tiempo es el concepto principal

- Asignación concurrente

```
s <= '1',
     '0' after 20 ns,
     '1' after 30 ns;
```

- Asignación secuencial

```
s <= '1';
wait for 20 ns,
s <= '0';
wait for 30 ns,
s <= '1';
```

Observando señales con Assert

- assert se usa para comprobar si se cumple una condición: equivalente a ***IF*** (***not condicion***)

```
assert <condicion> report string severity nivel;
```

- Tras report se añade una cadena de texto que se muestra si no se cumple la condición
- severity puede tener cuatro niveles
  - NOTE
  - WARNING
  - ERROR (nivel por defecto sino se incluye severity)
  - FAILURE

Algoritmo básico para los Testbenches

Algoritmo elemental de verificación:

- Dar valores a las señales de entrada a la UUT
- Esperar con wait for
- Comprobar los resultados con assert
- Volver a dar valores a las señales de entrada a la UUT
- y repetir ...

Ejemplo de código

```
entity dff_tb is
end dff_tb;

architecture  tb_arch of dff_tb is
        component dff is port (...) end component;
        signal d,c,q: std_logic;
begin
        uut: dff port map (d =>d, clk => c, q=>q);

        prcess
        begin
                c <= '0'; d <= '0';
                wait for 10 ns;
                c <= '1';
                wait for 10 ns;
                assert q=d report "falla" severity failure;
        end process;
end tb_arch;
```

Procedimientos

- VHDL permite definir procedimientos (subrutinas)

```
procedure  nombre (clase parámetro: dir tipo, ... ) is
        declaraciones
begin
        instrucciones secuenciales
end nombre;
```

- La clase de los parámetros pueden ser: variable, constant, signal
- La dirección: in, inout, out
- Los procedimientos se pueden declarar en la arquitectura o en un proceso, y se llaman desde un proceso o concurrentemente

Interesante para encapsular tareas repetitivas en la simulación

Ejemplo de código

```
architecture  tb_arch of dff_tb is
        (...)
        procedure send_clock_edge(signal c: out std_logic) is
        begin
                c <= '0'; wait for 10 ns;
                c <= '1'; wait for 10 ns;
        end send_clock_edge;
begin
        uut: dff port map (d =>d, clk => c, q=>q);
        prcess
        begin
                d <= '0';
                send_clock_edge(c);
                assert q=d report "falla" severity failure;
        end process;
end tb_arch;
```

Acceso a archivos

- Las simulaciones más potentes trabajan sobre archivos
- Por ejemplo:
  - Simulación de un multiplicador que escribe los resultados en un archivo de texto
  - Testbench para un microprocesador que lee un programa ensamblador de un archivo, lo ensambla y lo ejecuta
- Acceso básico: **std.textio**
  - Archivos de texto
  - Acceso línea a línea: readline y writeline
  - Dentro de una línea los campos se procesan con: read y write
- Acceso específico para std_logic: **ieee.std_logic_textio**

Instrucciones para acceder a archivos

- Especificar el archivo

```
file archivo_estimulos: text is in "Stim.txt";
```

- Leer una línea

```
variable  linea: line;
          . . .
readline (archivos_estimulos, linea);
```

- Leer un campo de una línea

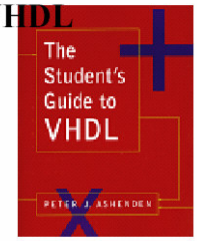
```
variable  opcode: string (2 downto 0);
          . . .
read (linea, opcode);
```

- Escribir

```
write (linea, resultado);
writeline (archivo_resultados, linea);
```

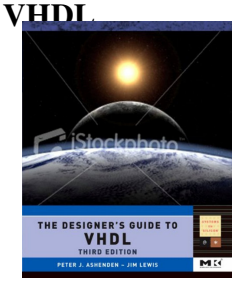
VHDL: Bibliografía

The Student`s Guide to VHDL



Peter J. Ashenden
Ed: Morgan Kaufmann
ISBN 1558605207

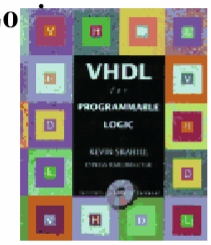
The Designer's Guide to VHDL



Peter Ashenden – Jim Lewis
Ed:Morgan Kaufmann
ISBN: 978-0120887859

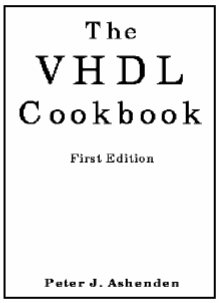
VHDL: Bibliografía

VHDL for Programmable Lo



Kevin Skahill
Ed: AddisonWesley
ISBN 0 201- 89573-0

VHDL Cookbook



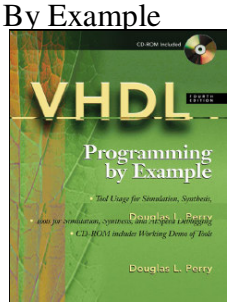
infaut.et.uni-magdeburg.de/~ks/VHDL/

VHDL: Bibliografía

VHDL 2008 Just the New Stuff

Peter Allenden – Jim Lewis
Ed:Morgan Kaufmann
ISBN 978-0-12-374249-0

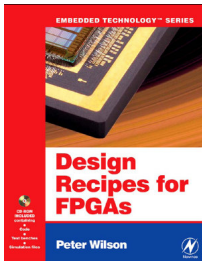
VHDL Programming By Example



Douglas L. Perry
Ed: McGraw-Hill
ISBN: 978-0-07-1409544

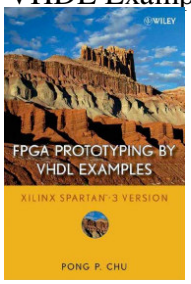
VHDL: Bibliografía Avanzada

Design Recipes for FPGAs



Peter Wilson
Ed: Elsiever
ISBN: 978-0-7506-6845-3

FPGA Prototyping By VHDL Examples



Pong P. Chu
Ed: Wiley-Interscience
ISBN 978-0-470-18531-5

VHDL: Enlaces de interés

http://fpgalibre.sourceforge.net/
Portal argentino de herramientas y recursos libres para desarrollar con FPGAs

http://www.opencores.org/
Repositorio de cores gratuitos sintetizables.

http://www.xilinx.com/
Uno de los principales fabricantes de FPGAs.

http://www.vhdl-online.de/tutorial/
http://esd.cs.ucr.edu/labs/tutorial/
Tutoriales online de VHDL