Unidad 1: Diseño digital y VHDL

Escuela Politécnica Superior - UAM

Copyright © 2007 Elsevier, "Digital Design and Computer Architecture"

Índice

- **Introducción**
- Lógica combinacional
- Circuitos combinacionales
- Lógica secuencial
- Modelado estructural
- Bancos de prueba (*testbenches*)

Introducción

- *Hardware description language* (HDL): permite diseñar la funcionalidad de un circuito digital sin poner funciones lógicas ni puertas. Diversas herramientas informáticas producen o ***sintetizan*** el circuito concreto que realiza dicha funcionalidad.
- Los circuitos comerciales se diseñan con HDLs
- Los dos HDLs principales son:
  - **VHDL**
    - Desarrollado en 1981 por el Dpto. de Defensa de EEUU
    - Se convirtió en estándar IEEE (1076) en 1987
  - **Verilog**
    - Desarrollado en 1984 por Gateway Design Automation
    - Se convirtió en estándar IEEE (1364) en 1995

Simulación y síntesis

- **Diseño y síntesis**
  - Descripción comportamental del sistema electrónico, subdividiendo el circuito en components y definiendo sus interconexiones.
  - La síntesis transforma el código HDL en un circuito (*netlist*) describiendo el hardware (una lista de puertas y cables conectándolas)
- **Simulación**
  - Se aplican ciertos valores a las entradas
  - Se comprueba si las salidas son correctas
  - Mucho tiempo/dinero ahorrado por depurar en simulación y no en hardware

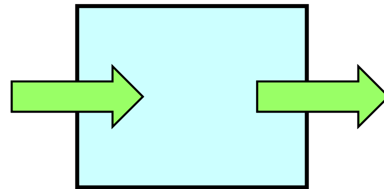
IMPORTANTE:

✓ No todos los diseños que se simulan son sintetizables.

✓ Al describir circuitos en un HDL, es vital pensar en el **hardware** que se debería generar (**no es programar en C**).

Entidad - Arquitectura

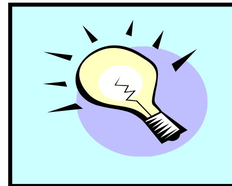
- La entidad se utiliza para hacer una descripción "caja negra" del diseño, sólo se detalla su interfaz (los puertos de entrada y salida “**ports**”)



Equivalente en C

```
float CalculaMedia(float a, float b) {
    float c;
    c = (a+b) / 2;
    return c;
}
```

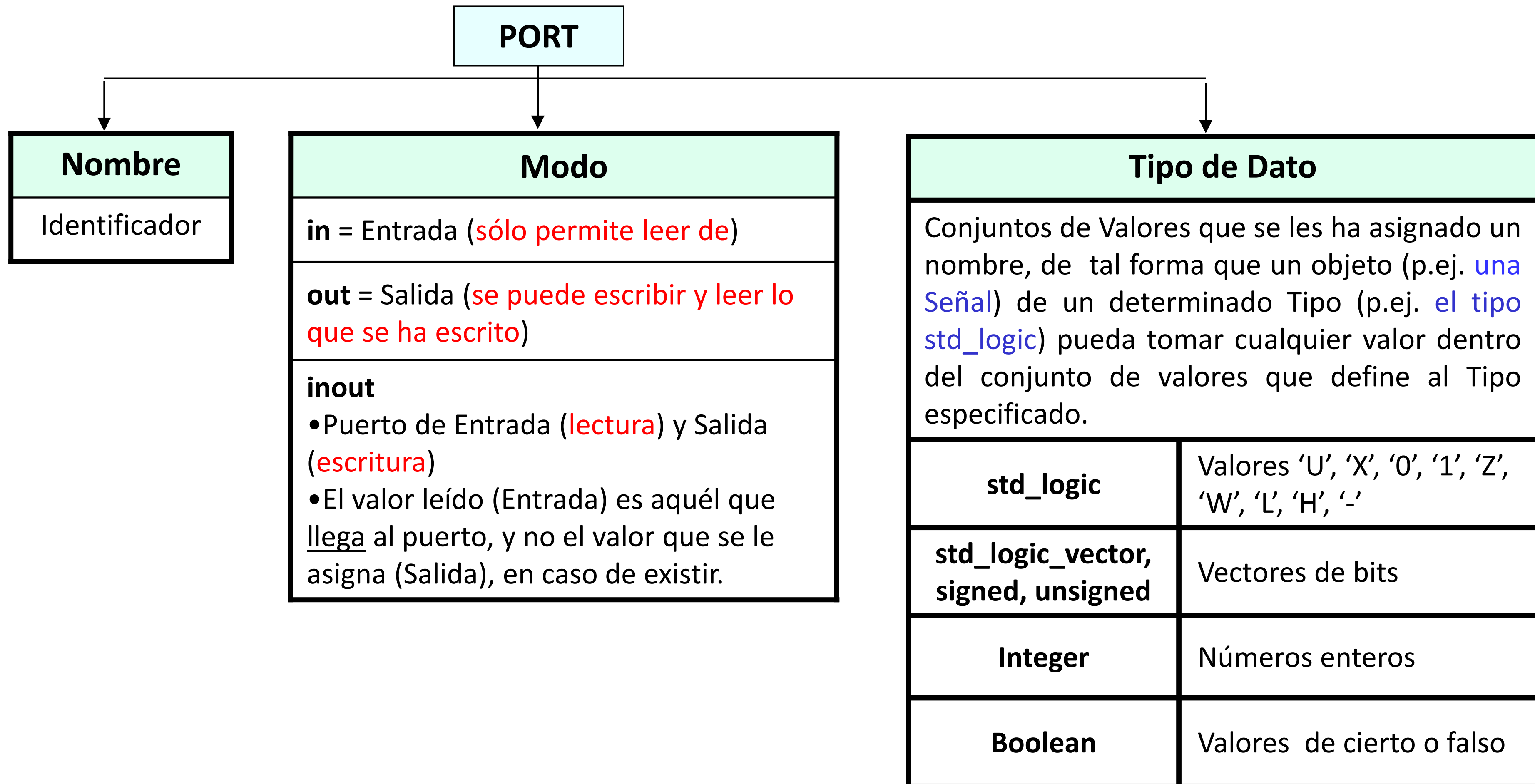
- Los contenidos del circuito se modelan dentro de la arquitectura



```
float CalculaMedia(float a, float b) {
    float c;
    c = (a+b) / 2;
    return c;
}
```

- Una entidad puede tener varias arquitecturas

PORTS: La conexión con el exterior



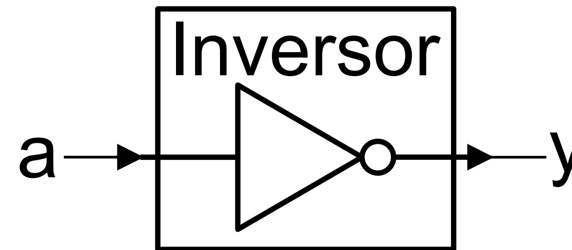
Ejemplo de código VHDL

BIBLIOTECAS

```
library IEEE;                    -- similar a declaración de .h
use IEEE.std_logic_1164.all;     -- para usar std_logic
```

ENTIDAD

```
entity inversor is
   port (a : in std_logic;
         y : out std_logic);
end inversor;
```



ARQUITECTURA

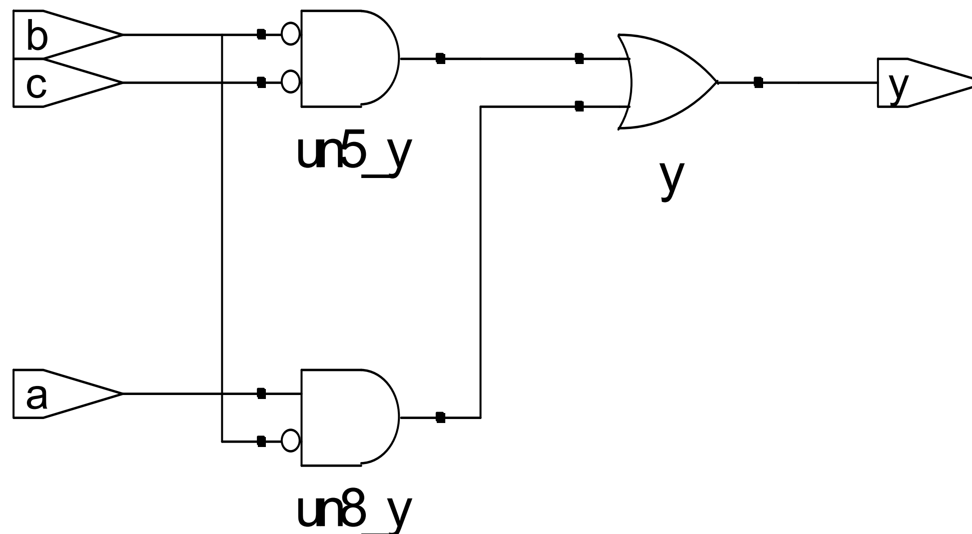
```
architecture comportamental of inversor is
begin
   y <= not a;                    -- asignación con flecha
end comportamental;
```

Síntesis de código VHDL

VHDL:

```
architecture comportamental of ejemplo is
begin
   y <= (not a and not b and not c) or (a and not b and not c) or
   (a and not b and c);
end comportamental;
```

Síntesis:



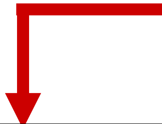
Sintaxis VHDL

- No distingue mayúsculas de minúsculas (*case insensitive*)
  - Ejemplo: `reset` y `Reset` son la misma señal.
  - Recomendación: poner siempre las mismas mayúsculas para facilitar la lectura y las búsquedas.
- Los nombres no pueden empezar por números
  - Ejemplo: `2mux` no es un nombre válido.
- Se ignoran los espacios, tabuladores, retornos de carro
- Comentarios:

```
-- Desde un guión doble hasta el final de la línea.
/* Comentario de bloque. */
```

Sintaxis en VHDL: Identificadores

IDENTIFICADORES
Nombres o etiquetas que se usan para referirse a: Constantes, Señales, Procesos, Entidades, etc.
Longitud (Número de Caracteres): Sin restricciones
Palabras reservadas por VHDL no pueden ser identificadores
En VHDL, un identificador en mayúsculas es igual que en minúsculas
Están formados por números, letras (mayúsculas o minúsculas) y guión bajo "_" con las reglas especificadas en la tabla siguiente.



Reglas para especificar un identificador	*Incorrecto*	*Correcto*
Primer carácter debe ser siempre una letra mayúscula o minúscula	4Suma	Suma4
Segundo carácter no puede ser un guión bajo (_)	S_4bits	S4_bits
Dos guiones bajos consecutivos no son permitidos	Resta__4	Resta_4_
Un identificador no puede utilizar símbolos especiales	Clear#8	Clear_8

Índice

- Introducción
- **Lógica combinacional**
- Circuitos combinacionales
- Lógica secuencial
- Modelado estructural
- Bancos de prueba (*testbenches*)

Tipo std_logic

- Los valores ‘0’ y ‘1’ del tipo bit se quedan cortos para modelar todos los estados de una señal digital en la realidad
- El paquete IEEE.std_logic_1164 define el tipo **std_logic**, que representa todos los posibles estados de una señal real:

U	No inicializado, valor por defecto
X	Desconocido fuerte, indica cortocircuito
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 de bus
L	0 débil, resistencia de pull-down
H	1 débil, resistencia de pull-up
–	No importa, usado como comodín para síntesis

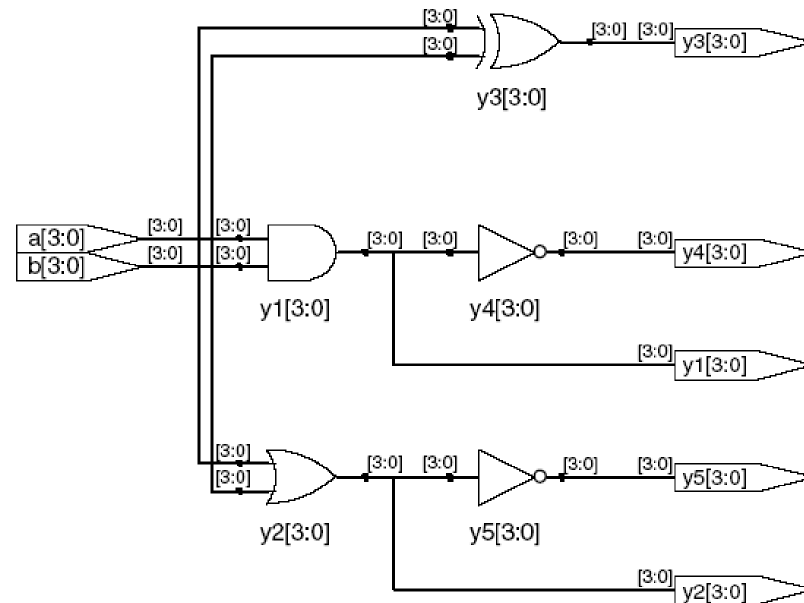
El tipo **std_logic_vector** define un *array* de bits de tipo **std_logic**.

Tipo unsigned y signed

- El paquete IEEE.numeric_std define los tipos **unsigned** y **signed** para representar números sin signo y con signo respectivamente. Deben usarse estos tipos de datos para realizar operaciones aritméticas. Es un paquete que partiendo del tipo **std_logic** ofrece operaciones aritméticas según el vector es con o sin signo. Es equivalente a std_logic_vector, pero ofrece dichas operaciones:
- Operaciones de cambio de signo: *" - "* y *" abs "*
- *Operaciones aritméticas: " + "* (sumar), *" - "* (restar) y *" * "* (multiplicar)
- Operaciones de comparación: *" > "* (mayor), *" < "* (menor), *" <= "* (menor igual), *" >= "* (mayor igual), *" = "* (igual) y *" /= "* (distinto)
- Desplazamientos lógicos: *" sll "* (izquierda) y *" srl "* (derecha)
- Desplazamientos aritméticos: *" sla "* (izquierda) y *" sra "* (derecha)
- Rotaciones: *" rol "* (izquierda) y *" ror "* (derecha)
- Redimensionado: *" resize "*

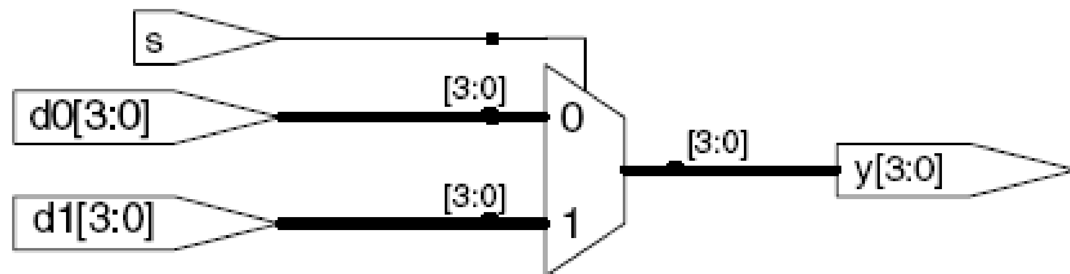
Operadores *bitwise*

```
entity puertas is
port (a, b: in std_logic_vector(3 downto 0);
   y1, y2, y3, y4, y5: out std_logic_vector(3 downto 0));
end puertas;

architecture comport of puertas is
begin
-- Pueden actuar sobre bits
-- o sobre buses
   y1 <= a and b; -- AND
   y2 <= a or b; -- OR
   y3 <= a xor b; -- XOR
   y4 <= a nand b; -- NAND
   y5 <= a nor b; -- NOR
end comport;
```



Asignación condicional: when ... else

```
entity mux2a1_4bits is
port(d0,d1 : in std_logic_vector(3 downto 0);
      s : in std_logic;
      y : out std_logic_vector(3 downto 0));
end mux2a1_4bits;

architecture comport of mux2a1_4bits is
begin
  y <= d0 when s = '0' else d1;
end comport;
```



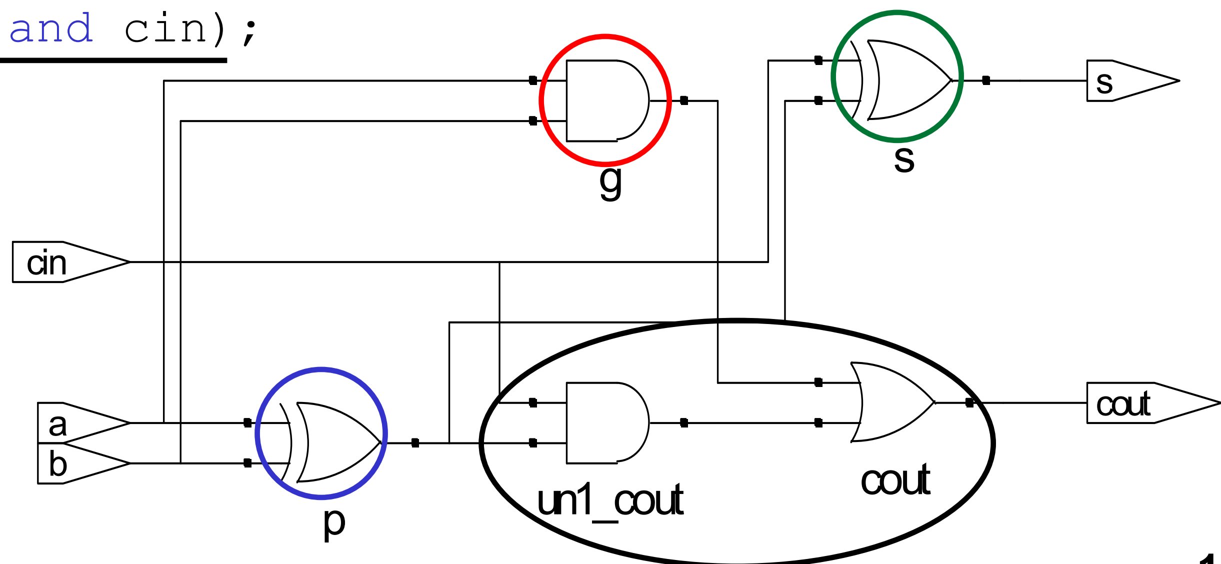
Asignación condicional: with … select

```
entity deco2a4 is
port(a : in std_logic_vector(1 downto 0);
         y : out std_logic_vector(3 downto 0));
end deco2a4;

architecture comport of deco2a4 is
begin

  with a select
        y <= "0001" when "00",
             "0010" when "01",
             "0100" when "10",
             "1000" when others;    -- último caso, siempre
                                    -- conviene incluirlo
end comport;
```

1.8

Señales internas

```
architecture comport of fulladder is

   signal p, g : std_logic;  -- La declaración de señales internas
                             -- se pone entre architecture y begin
begin
   p <= a xor b;
   g <= a and b;
   s <= p xor cin;
   cout <= g or (p and cin);
end comport;
```



Concurrencia

```
architecture comport of fulladder is
   signal p, g : std_logic;
begin
   p <= a xor b;            -- No importa el orden de las
   g <= a and b;            -- líneas, todo el hardware
   s <= p xor cin;          -- "existe" a la vez
   cout <= g or (p and cin);
end comport;
-------------------------------------------------------
architecture comport of fulladder is
   signal p, g : std_logic;
begin
   cout <= g or (p and cin); -- Es el mismo circuito, no
   s <= p xor cin;           -- importa que p se asigne en
   p <= a xor b;             -- una línea posterior porque
   g <= a and b;             -- el hardware es concurrente
end comport;
```

Precedencia de operadores

- Orden en que se resuelven los operadores si no hay paréntesis
- Recomendación: usar paréntesis

Primero

Operador	Operación
not	NOT
* / %	mult div módulo
+ –	suma resta
< <= > >=	comparar
= /=	igual distinto
and nand	AND NAND
xor xnor	XOR XNOR
or nor	OR NOR

Último

Formatos de números y bits

Formato	Nº bits	Base	Memoria
'1'	1	Binario	1
"101"	>1	Binario	101
X"AF"	>1	Hexadecimal	10101111
1	Depende	Decimal	0…00001
-2	Depende	Decimal	1…11110 (C2)
1.5	Depende	Decimal	IEEE-754

Los valores de las señales se asignan con una "flecha":

```
a_bit <= '1';
a_bus <= "101";
a_int <= 1;
```

Manipulación de bits

```
signal a : std_logic_vector(3 downto 0);
signal b : std_logic_vector(0 to 3);
...
a <= "0101";
b <= "0101";

-- es equivalente a:
a(3) <= '0';   a(2) <= '1';   a(1) <= '0';    a(0) <= '1';
b(3) <= '1';   b(2) <= '0';   b(1) <= '1';    b(0) <= '0';
```

a

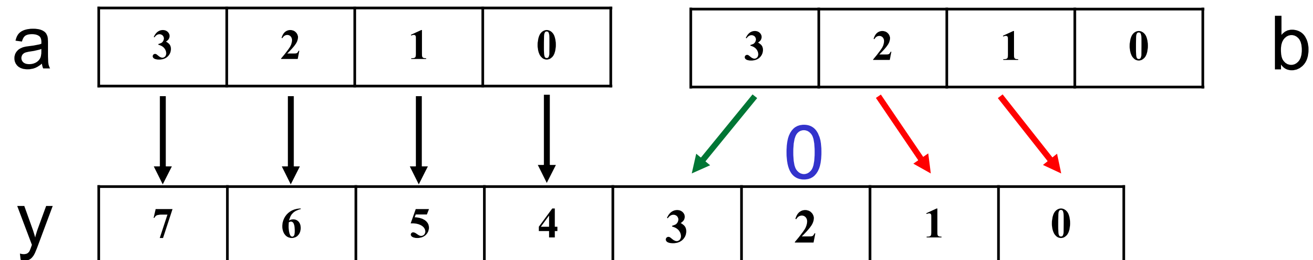
Posición	3	2	1	0
Valor	0	1	0	1

b

Posición	0	1	2	3
Valor	0	1	0	1

Manipulación de bits

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- el operador & en VHDL significa concatenar
y <= a & b(3) & '0' & b(2 downto 1);

-- es equivalente a:

y(7) <= a(3);  y(6) <= a(2);  y(5) <= a(1);   y(4) <= a(0);

y(3) <= b(3);  y(2) <= '0';   y(1) <= b(2);   y(0) <= b(1);
```



Manipulación de bits

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- Se pueden hacer todo tipo de agrupaciones parciales
-- siempre que coincidan el número de bits
y(7 downto 4) <= a;
y(3) <= b(3);
y(2) <= '0';
y(1 downto 0) <= b(2 downto 1);

-- es el mismo resultado para la "señal y" que en la
-- transparencia anterior
```

```
y <= a & b(3) & '0' & b(2 downto 1);
```

Manipulación de bits

```
signal y : std_logic_vector(7 downto 0);
signal a, b : std_logic_vector(3 downto 0);
...
-- en VHDL se permite hacer asignaciones por índice en
-- lugar de orden (izquierda a derecha) bit a bit (aggregates)
   y <= (3 downto 2 => a(2), 4 => b(1), 5 => '1',
         others => '0');       -- others es "resto de bits"

-- es equivalente a:
   y(7) <= '0'; y(6) <= '0'; y(5) <= '1'; y(4) <= b(1);
   y(3) <= a(2); y(2) <= a(2); y(1) <= '0'; y(0) <= '0';

-- muy utilizado para la inicialización de buses:
   y <= (others => '0'); -- poner todos los bits a '0'
```

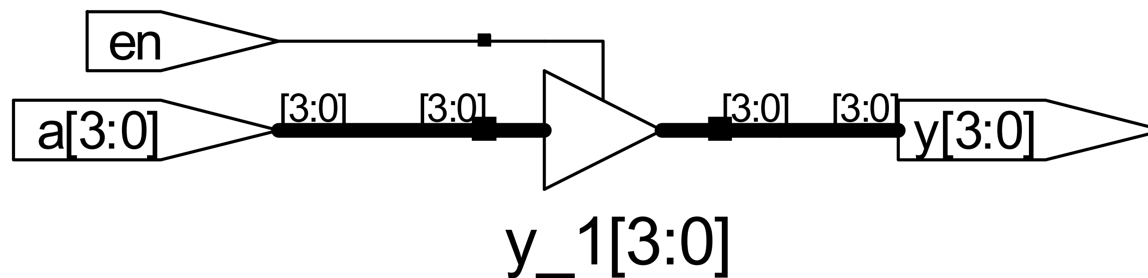
1.3, 1.9

Z: alta impedancia

VHDL:

```
signal y, a: std_logic_vector(3 downto 0);
signal en : std_logic;
...
y <= (others => 'Z') when en = '0' else a;
```

Síntesis:



Índice

- Introducción
- Lógica combinacional
- **Circuitos combinacionales**
- Lógica secuencial
- Modelado estructural
- Bancos de prueba (*testbenches*)

Circuitos combinacionales

- La mayoría de los circuitos combinacionales se pueden definir con asignaciones concurrentes (vistas hasta ahora)

- También hay código secuencial (se ejecuta línea tras línea, como en el software, sin concurrencia, así que **el orden del código sí es importante**) pero se encapsula en "procesos" `process` (aparte de funciones y procedimientos que no vemos)

- Dentro de procesos (y sólo dentro de ellos) se pueden utilizar `if`, `case`, `for` y `while`

Process

Estructura del “process”:

```
architecture nombreArq of nombreEnt is
   -- parte declarativa, señales internas de la arquitectura
Begin
-- Los procesos, uno o varios, aparecen entre begin y end de
-- architecture
[ETIQ:] process(lista sensibilidad) -- la etiqueta es opcional
   -- parte declarativa, variables pero no señales, de uso
   -- interno en el proceso
  begin
  -- código;
   end process [ETIQ];
end nombreArq;
```

Cada vez que cambia alguna señal de la lista sensibilidad, se ejecuta secuencialmente ***código***. A partir del estándar VHDL-2008, se puede escribir ***all*** en la lista, evitando escribir las señales

if

```
if condicion_1 then
  -- sec instr 1
[elsif condicion_2 then
  -- sec instr 2]
[elsif condicion_3 then
  -- sec instr 3]
[else
  -- instr por defecto]
end if;
```

Similar a la asignación condicional when…else

Ejemplo

```
CTRL: process(all) -- o (nivel)
begin
  if nivel > 60 then
    a <= "11";
  elsif nivel > 40 then
    a <= "10";
 elsif nivel > 20 then
    a <= "01";
  else
    a <= "00";
  end if;
end process CTRL;
```

Cuidado: si es lógica combinacional SIEMPRE tiene que haber "else"

case

```
case expresion is
  when caso_1 =>
    -- sec instr 1
  when caso_2 =>
    -- sec instr 2
  when others =>
    -- instr por defecto
end case;
```

Similar a la asignación condicional with…select

Ejemplo

```
MUX: process(all)
     --o (sel, a, b, c, d)
begin
  case sel is
    when "11" => y <= d;
    when "10" => y <= c;
    when "01" => y <= b;
    when others => y <= a;
  end case;
end process MUX;
```

Cuidado: siempre tienen que estar todos los caminos. Para no olvidarse ninguno, que el último sea "`when others`"

1.2, 1.10

for

```
[ETIQ:] for indice in rango loop
  -- sec instr
end loop [ETIQ];
```

Ejemplo

```
AND8: process(all)
begin
  for i in 0 to 7 loop
    y(i) <= a(i) and b(i);
  end loop;
end process AND8;
```

```
AND8: process(all)
begin
  for i in 7 downto 0 loop
    y(i) <= a(i) and b(i);
  end loop;
end process AND8;
```

```
                 -- Hubiera sido más fácil porque
y <= a and b;    -- and es operador bitwise
                 -- además no sería necesario un proceso
```

while

```
[ETIQ:] while condicion loop
  -- sec instr
end loop [ETIQ];
```

Ejemplo

```
process(…)
begin
    while a = '1' loop
       …
       …
       …
    end loop;
end process;
```

When/else y with/select dentro de proceso

```
process(all)
begin
    y <= a and b when sel = '0' else c;
        …
end process;


process(all)
begin
    with sel select
    y <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when others;
    …
end process;
```

¿Concurrente o secuencial?

- El hardware es concurrente. ¿Qué sentido tiene un código secuencial?

```
…
a <= '0';
…
a <= '1';
…
```

a=¿? a=X

```
Process(all)
begin
    …
    a <= '0';
    …
    a <= '1';
end process;
```

a=¿? a='1'

Actualización de señales en procesos

- El tiempo "**se detiene**" cuando se ejecuta un proceso. Las señales que se escriben reciben un nuevo valor después de acabar el proceso o, si lo hay, después de un wait.

```
Process(all)
begin
    …
    a <= '0';
    b <= not a;
    …
end process;
```

Señal	Antes del proceso	Después del proceso
a	'1'	'0'
b	?	'0'

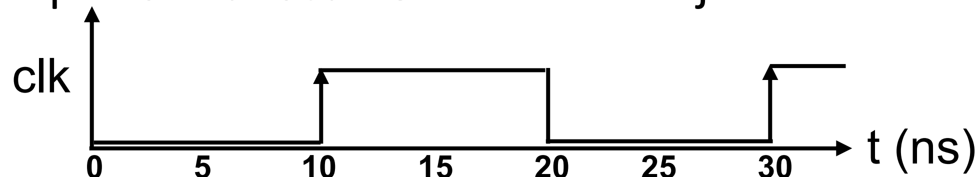
Actualización de señales en procesos

- El tiempo "**se detiene**" cuando se ejecuta un proceso. Las señales que se escriben reciben un nuevo valor después de acabar el proceso o, si lo hay, después de un wait.

```
Process                        -- sin lista de sensibilidad
begin
     while simulando = '1' loop
        clk <= '0';        ← '?'
        wait for 10 ns;              ← '0'
        clk <= '1';                          ← '0'
        wait for 10 ns;
     end loop;                                       ← '1'
     wait ;    -- evita que se ejecute el while de forma
               -- continuada cuando simulando = '0'
end process;
```

Señal	Antes del proceso	Antes 1er wait	Después 1er wait	Antes 2º wait	Después 2º wait
clk	?	?	'0'	'0'	'1'

El tiempo "**si transcurre**" durante la ejecución de un *wait*



Nota: Un *wait* no tiene sentido en hardware. Sólo se puede simular.

Sentencias condicionales y bucles: Resumen

- Fuera y dentro de proceso
  - ✓ When – else …..else – ;
  - ✓ With – select ……when others ;

- Dentro de proceso
  - ✓ If ** then – ; elsif – ; else – ; end if ;
  - ✓ Case ** is when – ; ….when others – ; end case;
  - ✓ For ** in ** downto/to ** loop – ; end loop ;
  - ✓ While ** loop – ; end loop;

1.16

Lógica combinacional: ¿Dentro o fuera de proceso?

- Excepto en el uso de bucles, cualquier código combinacional escrito en proceso puede escribirse fuera de proceso (sentencias concurrentes) y viceversa.

- Escribir un proceso requiere más código pero permite más flexibilidad en la escritura del código (uso de if y case, sobreescritura de señales, etc).

Índice

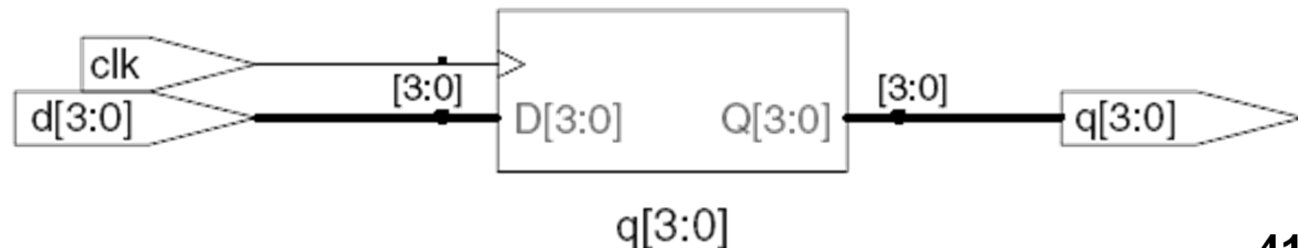
- Introducción
- Lógica combinacional
- Circuitos combinacionales
- **Lógica secuencial**
- Modelado estructural
- Bancos de prueba (*testbenches*)

Lógica secuencial

- En VHDL los flip-flops (o señales registradas) se describen siempre igual:
  - **Con un `process`**
- Otras descripciones pueden conseguir simulaciones equivalentes, pero no generan el mismo hardware

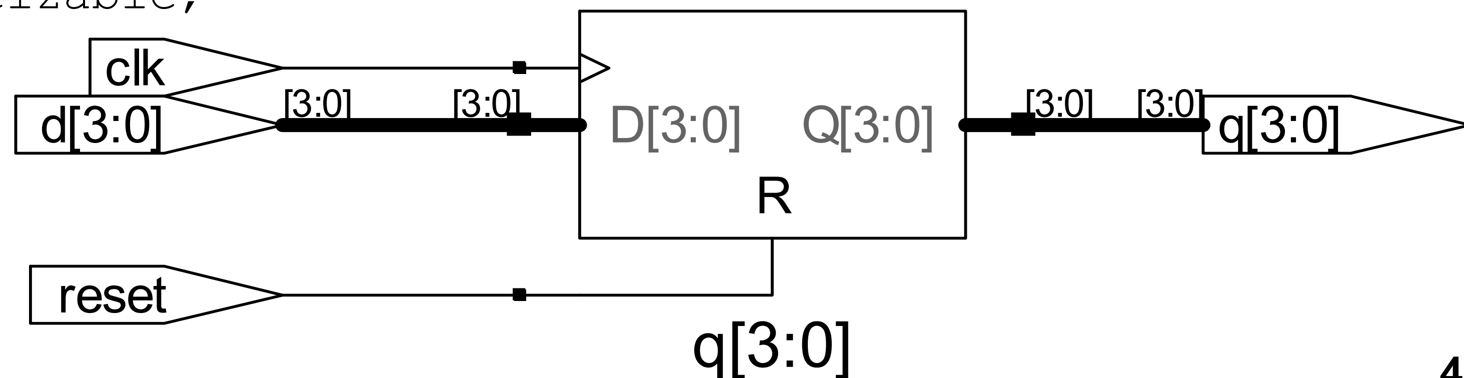
Flip-Flop tipo D

```
entity flop is
port(Clk : in std_logic;
     D : in std_logic_vector(3 downto 0);
     Q : out std_logic_vector(3 downto 0));
end flop;

architecture sintetizable of flop is
begin

REG: process(all)
  begin
    if Clk = '1' and Clk'event then  --if rising_edge(Clk) then
      Q <= D;
    end if;
  end process;

end sintetizable;
```



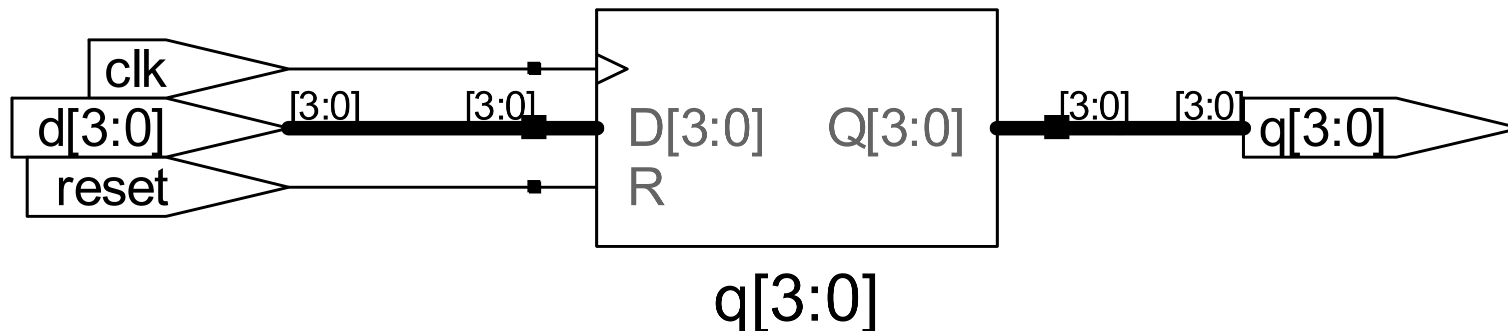
Flip-Flop tipo D con Reset asíncrono

```
-- Se añade la entrada Reset en la entidad

architecture sintetizable of flop is
begin
  process(all)
    begin
      if Reset = '1' then
        Q <= (others => '0');
      elsif Clk = '1' and Clk'event then
        Q <= D;
      end if;
  end process;

end sintetizable;
```



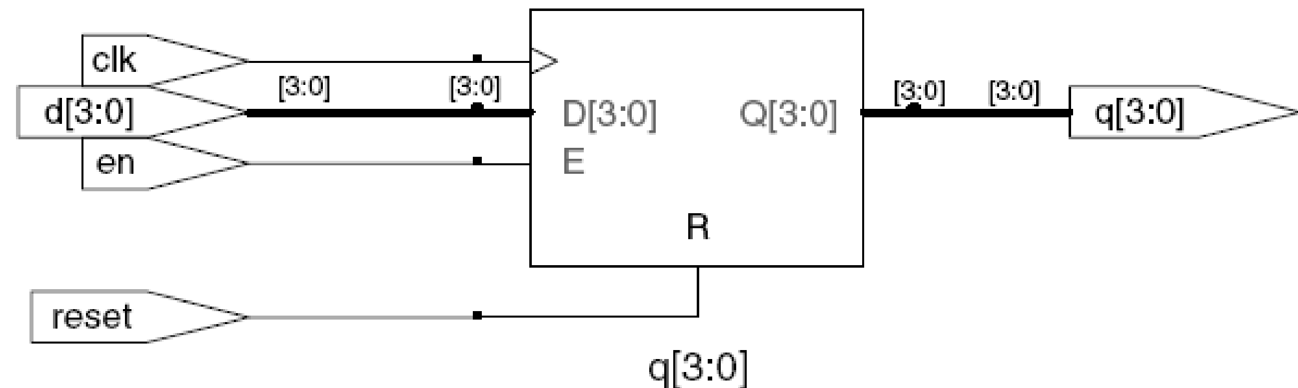
Flip-Flop tipo D con Reset síncrono

```
architecture sintetizable of flop is
begin
  process(all)
  begin
    if Clk = '1' and Clk'event then
      if Reset = '1' then
        Q <= (others => '0');
      else
        Q <= D;
      end if;
    end if;
  end process;
end sintetizable;
```



Flip-Flop tipo D con Enable

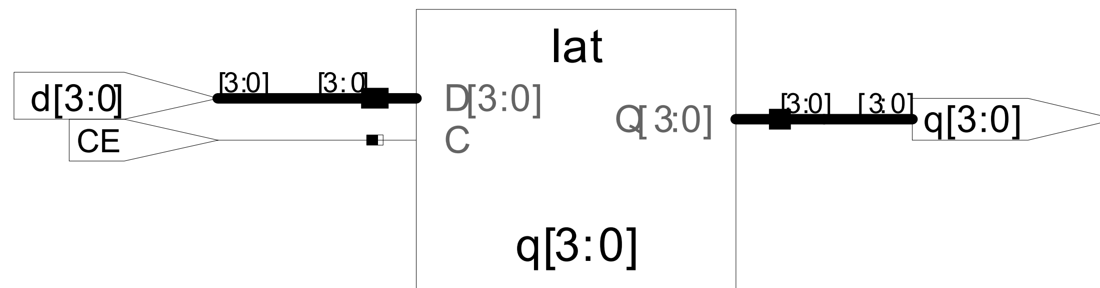
```
-- Se añade la entrada En
architecture sintetizable of flop is
begin
  process(all)
  begin
    if Reset = '1' then
      Q <= (others => '0');
    elsif Clk = '1' and Clk'event then
      if En = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end sintetizable;
```



1.11, 1.14

Latch (cerrojo)

```
architecture sintetizable of latch is
begin
  process(all)
  begin
    if CE = '1' then -- sin flanco, por nivel
      Q <= D;
    end if;
  end process;
end sintetizable;
```



Latches indeseados

```
process(all)
begin
	if a = "00" then
		b <= "11";
	elsif a = "01" then
		b <= "10";
	elsif a = "10" then
		b <= "00";
	end if;
end process;
```

¿Cuánto vale b cuando a="11"?

b mantiene el valor anterior => Latch

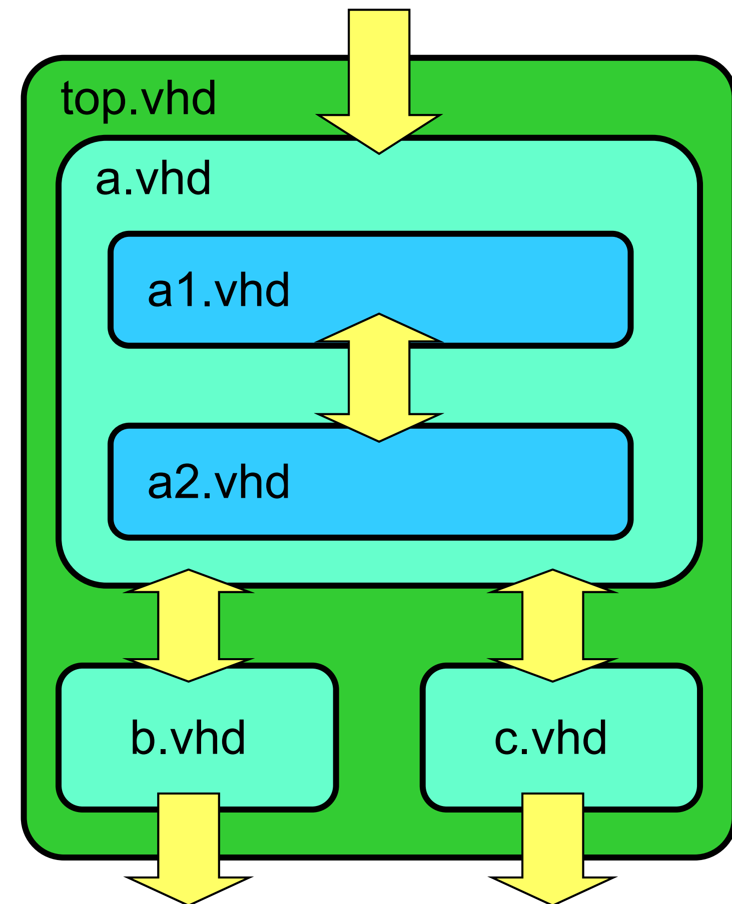
Atención: se generan latches no deseados cuando no se asigna valor por todos los caminos. En este curso no se utilizan latches, pero un código puede generarlos sin querer. Si el hardware sintetizado en prácticas incluye latches, se considerará como un error, sobre todo si se quiere diseñar un combinacional.

Índice

- Introducción
- Lógica combinacional
- Circuitos combinacionales
- Lógica secuencial
- **Modelado estructural**
- Bancos de prueba (*testbenches*)

Modelado estructural

- Los componentes básicos se utilizan como elementos de otros más grandes.
- Es fundamental para la reutilización de código.
- Permite mezclar componentes creados con distintos métodos de diseño:
  - Esquemáticos
  - VHDL, Verilog
- Genera diseños más legibles y más portables.
- Necesario para estrategias de diseño *top-bottom* o *bottom-up.*



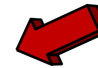
Cómo instanciar un componente

```
entity top is
port
( ... );
end top;
```

```
architecture jerarquica of top is
  signal s1, s2 : std_logic;
  component a
  port
  (entrada: in std_logic;
    salida: out std_logic);
  end component;
begin
  u1: a
  port map
  (entrada => s1,
   salida => s2);
end jerarquica;
```

Ejemplo de diseño jerárquico: componente inferior

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY miand2 IS PORT (
  x, y: IN std_logic;
  z: OUT std_logic);
END miand2;

ARCHITECTURE archand2 OF miand2 IS
BEGIN
    z <= x AND y;
END archand2;
```



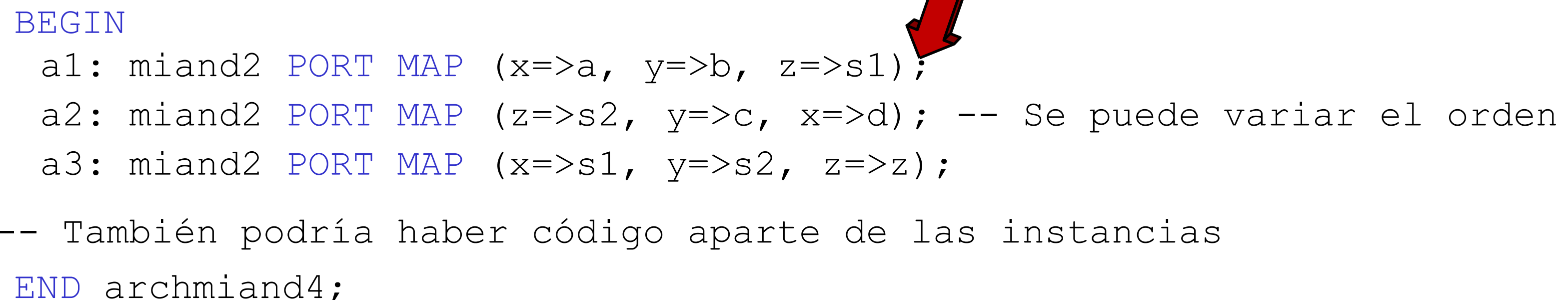
Descripción del componente de nivel inferior:
"Puerta AND 2 entradas"

Ejemplo de diseño jerárquico: top-level

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY miand4 IS PORT (
    a, b, c, d: IN std_logic;
        z: OUT std_logic);
END miand4;

ARCHITECTURE archmiand4 OF miand4 IS
```

```
COMPONENT miand2 PORT (
  x, y: IN std_logic;
  z: OUT std_logic);
END COMPONENT;

SIGNAL s1, s2: std_logic;
```

Instanciación del componente. Asociación por nombre



```
BEGIN
 a1: miand2 PORT MAP (x=>a, y=>b, z=>s1);
 a2: miand2 PORT MAP (z=>s2, y=>c, x=>d); -- Se puede variar el orden
 a3: miand2 PORT MAP (x=>s1, y=>s2, z=>z);
-- También podría haber código aparte de las instancias
END archmiand4;
```

Índice

- Introducción
- Lógica combinacional
- Circuitos combinacionales
- Lógica secuencial
- Modelado estructural
- **Bancos de prueba (*testbenches*)**

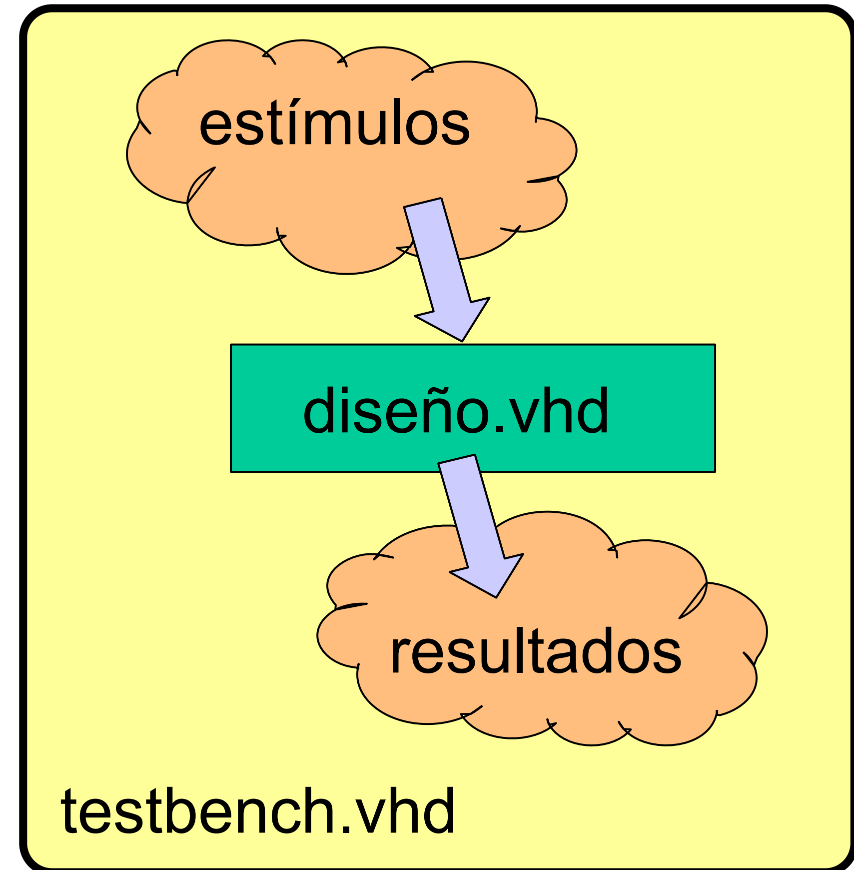
Bancos de prueba (*test-benches*)

- Código HDL escrito para comprobar que un módulo HDL funciona: el *device under test* (dut), o *unit under test* (uut)
- No sintetizable
- Tipos de *testbenches*:
  - Simple
  - Con auto-comprobación

Cómo hacer un *testbench*

1. Instanciar el diseño que vamos a verificar
   - El *testbench* será el nuevo top-level
   - Será una entidad sin ports

2. Escribir el código que:
   - Genera los estímulos
   - Observa los resultados
   - Informa al usuario



Ejemplo

Vamos a comprobar que funciona:

$$y = a \cdot b$$

VHDL

```
entity MiAnd is
  port (a, b : in std_logic;
        y : out std_logic);
end MiAnd;

architecture comportamental of MiAnd is
begin
  y <= a and b;
end comportamental;
```

Testbench (parte 1, instanciación)

```
entity testbench1 is -- no hay entradas ni salidas (ports)
end;

architecture test of testbench1 is
   component MiAnd            -- declaración del uut
   port (a, b: in std_logic;
        y: out std_logic);
   end component;
   -- Señales para conectar todos los puertos del uut
   signal a, b, y: std_logic;  -- Pueden ser nombres distintos
begin
   uut: MiAnd port map(
        a => a,
        b => b,
        y => y);
```

Testbench (parte 2, gen. estímulos)

```
-- continuación código anterior
-- Los estímulos se generan en un process
process        -- No hay lista de sensibilidad porque hay wait
begin
   a <= '0'; b <= '0';
   wait for 10 ns;
   a <= '0'; b <= '1';
   wait for 10 ns;
   a <= '1'; b <= '0';
   wait for 10 ns;
   a <= '1'; b <= '1';
   wait for 10 ns;
   wait;       -- "cuelga" este proceso, si no vuelve a empezar
end process;
end; -- end del architecture
```

El simulador debe analizar el valor de la señal 'y' en cada caso y comprobar el correcto funcionamiento

Comprobación automática

Para comprobar resultados se utiliza la sentencia assert

```
assert condicion report "Texto" severity nivel;
```

Verifica que `condicion` se cumple. Si no, saca `Texto` por el log del simulador y genera una excepción del `nivel` que se haya especificado.

Dependiendo del `nivel` (note, warning, error, failure), el simulador parará o no (configurable por usuario).

Ejemplo auto-comprobación MiAnd

```
process -- No hay lista de sensibilidad porque hay wait
begin
   a <= '0'; b <= '0';
   wait for 10 ns;
   assert y = '0' report "Falla 00" severity error;
   a <= '0'; b <= '1';
   wait for 10 ns;
   assert y = '0' report "Falla 01" severity error;
   a <= '1'; b <= '0';
   wait for 10 ns;
   assert y = '0' report "Falla 10" severity error;
   a <= '1'; b <= '1';
   wait for 10 ns;
   assert y = '1' report "Falla 11" severity error;
   wait; -- "cuelga" este proceso, si no vuelve a empezar
end process;
```

Testbenches de circ. secuenciales

El reloj se genera normalmente en un proceso aparte:

```
process
begin
  Clk <= '0';
  wait for CICLO/2; -- CICLO es una constante
  Clk <= '1';
  wait for CICLO/2;
end process; -- al no haber wait final es cíclico
```

Sería mejor generarlo sólo mientras una señal auxiliar se mantenga activa, para poder acabar la simulación, ver ejemplo de código `while`.

Ejemplo para un contador

```
process -- No hay lista de sensibilidad porque hay wait
begin
   Reset <= ‘1’; -- Activar siempre el reset al comienzo
   wait for CICLO;
   Reset <= ‘0’;     -- Se desactiva para que pueda avanzar
   for i in 0 to 255 loop
    assert to_integer(Cuenta) = i
       report “Falla en “ & to_string(i)
          severity error; -- Cuenta es unsigned
    wait for CICLO;       -- El reloj se genera en paralelo con
                          -- otro proceso, ver pág. anterior
   end loop;              -- Cierra el bucle for
  wait; -- “cuelga” este proceso
end process;
```

Listas de sensibilidad: testbenches

- Los procesos de los testbenches pueden no tener listas de sensibilidad.
- Para ello es necesario que haya alguna sentencia wait.
- Si no la hubiera, el simulador se bloquearía, ya que no podría avanzar el tiempo de simulación.

Constantes

- Como en cualquier otro lenguaje, en VHDL se pueden utilizar constantes.
- Se declaran también en la parte declarativa, entre architecture y begin, y se deben inicializar:

```
architecture ejemplo of prueba is
    constant C1 : std_logic_vector(3 downto 0) := “0101”;
    constant C2 : integer := 5;
    constant CICLO : time := 10 ns;
begin
```

- Las constantes pueden ser de cualquier tipo.

Conversiones de tipos

- VHDL es fuertemente tipado, no hay cast automático:

```
architecture ejemplo of prueba is
    signal s1 : std_logic_vector(3 downto 0);
    signal s2 : integer;
    signal s3 : signed(3 downto 0);
    signal s4 : unsigned(3 downto 0);
begin
    s1 <= s2; -- Todas incorrectas en VHDL
    s2 <= s1;
    s3 <= s1;
    s4 <= s1;
    s2 <= s3;
    s2 <= s4;
```

Conversiones de tipos

- Para convertir entre enteros, std_logic_vector y unsigned/signed se deben usar las siguientes funciones:

		HASTA			
		integer	signed	unsigned	std_logic_vector
DESDE	integer		to_signed()	to_unsigned()	No hay conversión directa
	signed	to_integer()		unsigned()	std_logic_vector()
	unsigned	to_integer()	signed()		std_logic_vector()
	std_logic_vector	No hay conversión directa	signed()	unsigned()	

Unidad 1: Diseño digital y VHDL

Escuela Politécnica Superior - UAM

Copyright © 2007 Elsevier, “Digital Design and Computer Architecture”