

Introducción al Lenguaje de Descripción Hardware VHDL

- **Arquitectura y tecnología de Computadores** (Informática)
- **Fundamentos de Computadores** (Teleco)

Juan González (juan.gonzalez@uam.es)

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

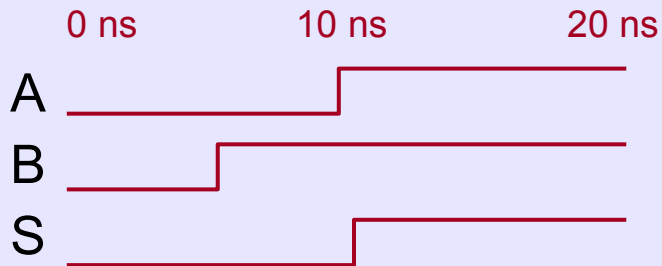
Estilos de diseño

Verificación con testbenches

¿Para qué sirve el VHDL?

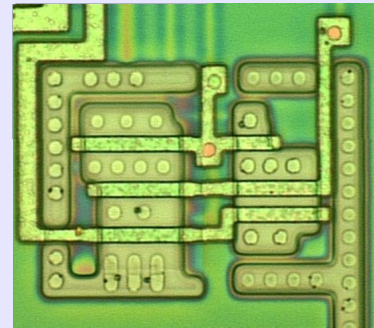
- El VHDL permite modelar **SISTEMAS DIGITALES**
- A partir de estos modelos podremos:

Simular



Comprobar que tienen la funcionalidad deseada

Sintetizar

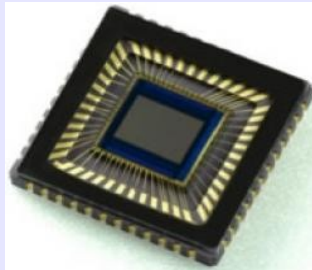


Crear un circuito que funciona como el modelo

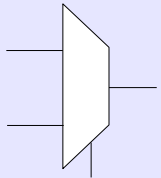
FPGAs

EDCD (Informática, 3º)
DCSE (Teleco, 4º)

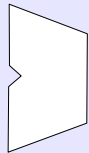
SISTEMAS DIGITALES



Microprocesador



MUX

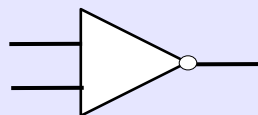
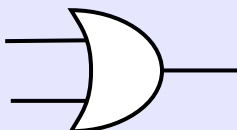
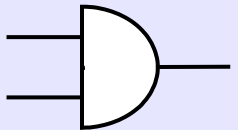


ALU



DECOD

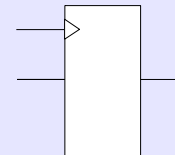
Circuitos
Combinacionales



Puertas lógicas



REG



CONT

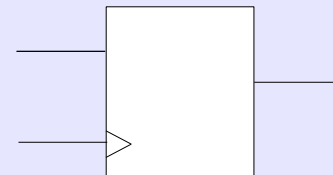


MEM



AUT

Circuitos
Secuenciales

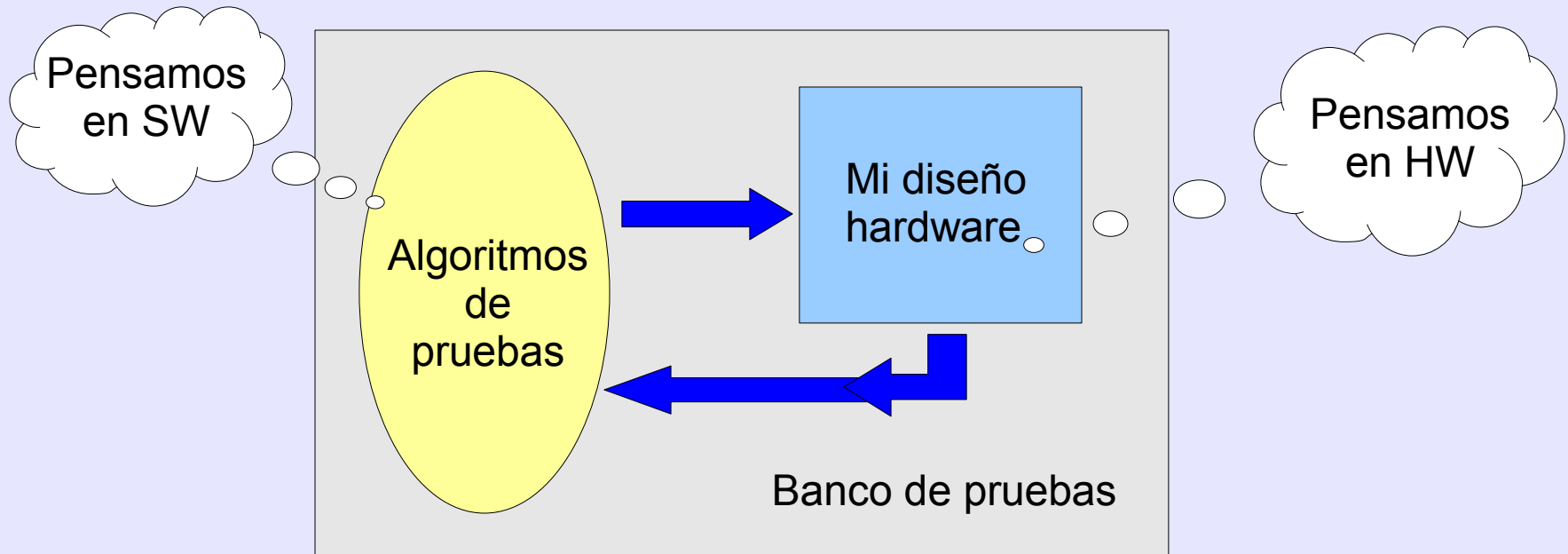


Biestables

VHDL: HW + ALGORITMOS

- Con VHDL modelamos el **HARDWARE**
- Pero VHDL permite también programar **ALGORITMOS** (Software)

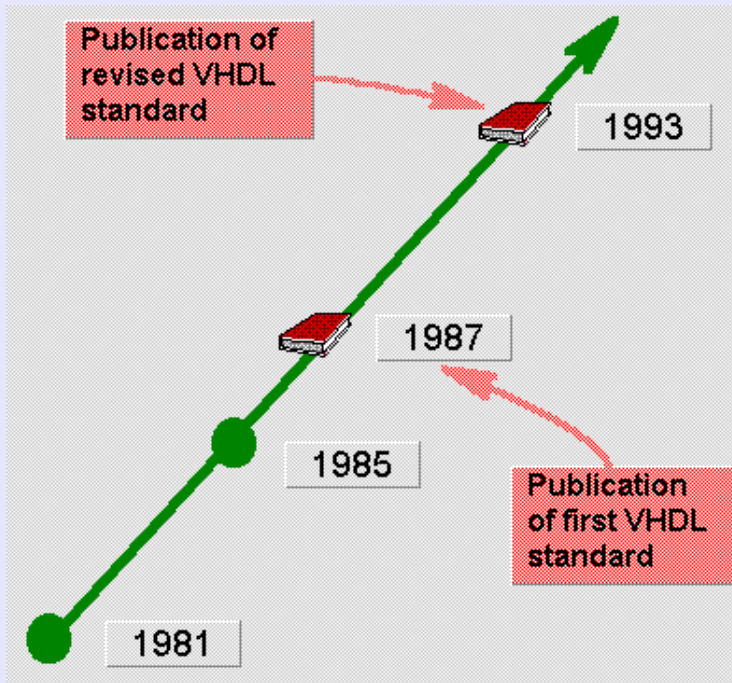
Ejemplo: Pruebas de funcionamiento



VHDL: orígenes e historia

- VHDL surge a principios de los '80 de un proyecto DARPA (Departamento de Defensa de los EE.UU.) llamado VHSIC – *Very High Speed Integrated Circuits*
- VHDL aparece como una manera de describir circuitos integrados
 - *La crisis del ciclo de vida del HW:* cada día los circuitos integrados eran más complicados, y el coste de reponerlos cada vez era mayor, porque no estaban correctamente documentados. VHDL nació como una manera estándar de documentar los circuitos
 - Al mismo tiempo, se vio que la expresividad de VHDL permitiría reducir el tiempo de diseño de los circuitos, porque se podrían crear directamente de su descripción: utilidad de la síntesis
- En 1987 el trabajo fue cedido al IEEE, y a partir de ese momento es un estándar abierto.

VHDL: Evolución



- **1980:** El departamento de defensa de los EEUU funda el proyecto para crear un HDL estándar dentro del programa VHSIC
- **1981:** Woods Hole Workshop, reunión inicial entre el Gobierno, Universidades e Industria
- **1983:** Se concedió a Intermetrics, IBM y Texas Instruments el contrato para desarrollar VHDL
- **1985:** Versión 7.2 de dominio público.
- **1987:** El IEEE lo ratifica como su estándar 1076 (VHDL-87)
- **1993:** El lenguaje VHDL fue revisado y ampliado, pasando a ser estándar 1076 '93 (VHDL-93)
- **2000:** Última modificación de VHDL

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

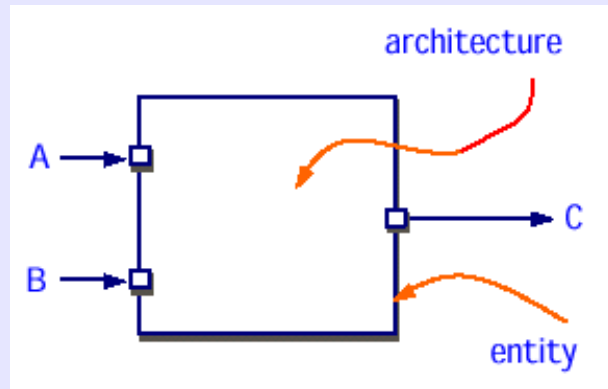
Diseño jerárquico

Estilos de diseño

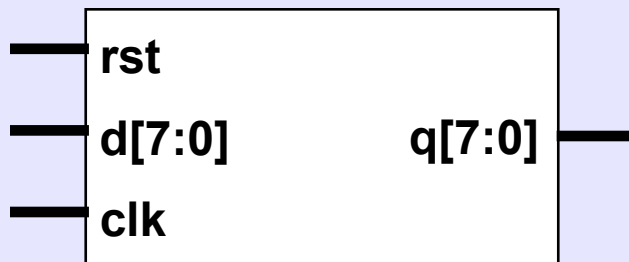
Verificación con testbenches

Entidad y Arquitectura: 1^{er} nivel de abstracción

Abstracción: caja negra



Interfaz: entradas y salidas

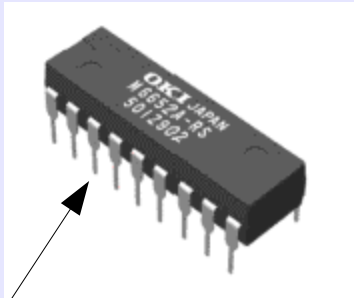


Entidad y arquitectura

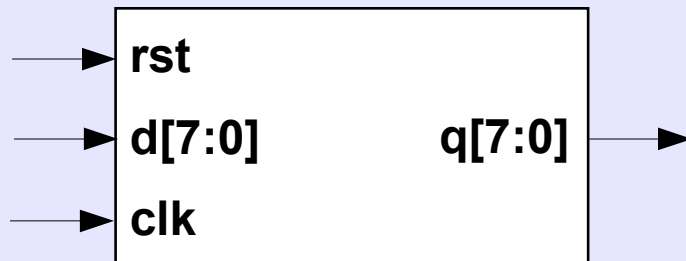
- Una unidad hardware se visualiza como una “caja negra”
 - El interfaz de la caja negra esta completamente definida.
 - El interior esta oculto.
- En VHDL la caja negra se denomina entidad
 - La ENTITY describe la E/S del diseño
- Para describir su funcionamiento se asocia una implementación que se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

PORTS: Puertos de una entidad

Interfaz de dispositivo



Ports: entradas y salidas



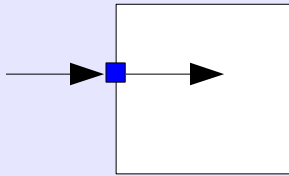
Ports = canales de comunicación

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: '0', '1' 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

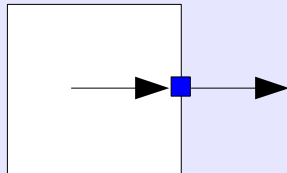
PORTS: Modos de un puerto

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



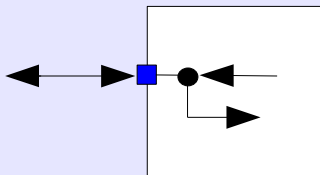
IN

Una señal que entra en la entidad y no sale. La señal puede ser leída pero no escrita.



OUT

Una señal que sale fuera de la entidad y no es usada internamente. La señal no puede ser leída dentro de la entidad.



INOUT

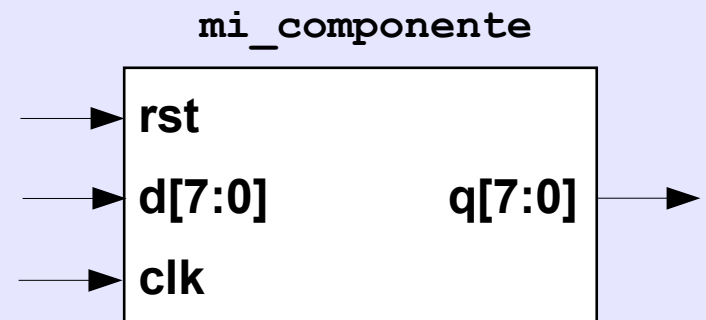
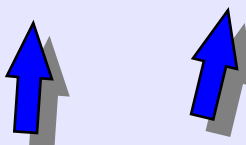
Una señal que es bidireccional, entrada/salida de la entidad.

VHDL: Declaración 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));  
END mi_componente;
```

MODO **TIPO**



Estructura de un diseño VHDL

```
library ieee;  
use ieee.std_logic_1164.all;
```

```
entity mi_componente is
```

```
  port (
```

```
  );
```

```
end mi_componente;
```

```
architecture test of mi_componente is
```

```
begin
```

```
end test;
```

Declaraciones del
puerto

Nombre de la entidad

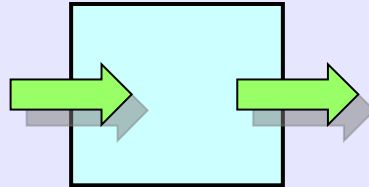
Parte declarativa

Cuerpo

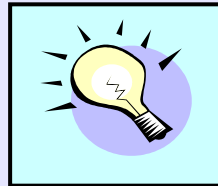
Nombre de la arquitectura

Resumen: Entidad y Arquitecturas

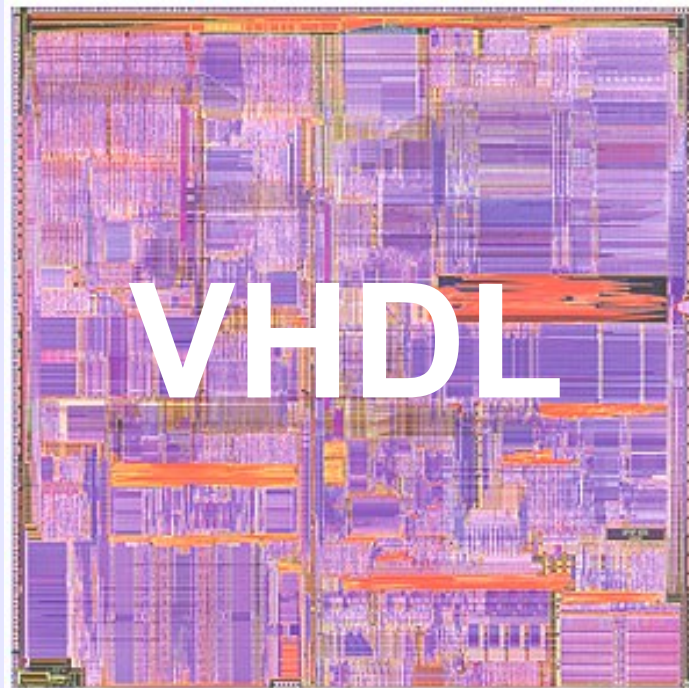
- La entidad se utiliza para hacer una descripción "caja negra" del diseño, sólo se detalla su interfaz



- Los contenidos del circuito se modelan dentro de la arquitectura



Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

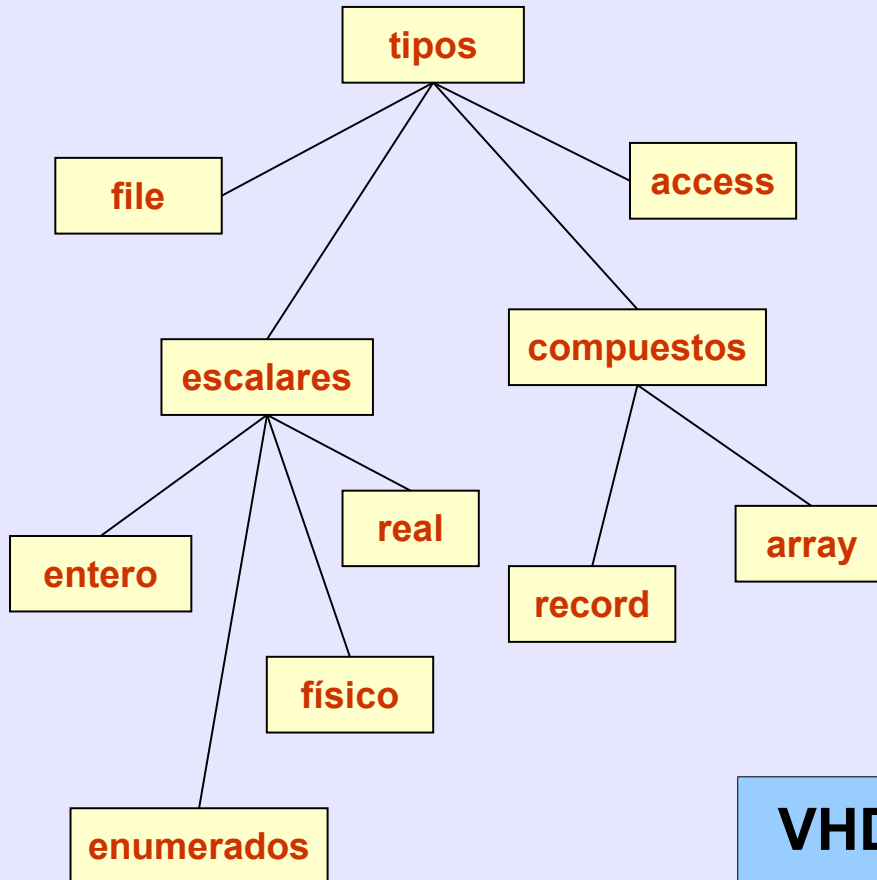
Triestados

Diseño jerárquico

Estilos de diseño

Verificación con testbenches

Tipos de datos básicos



- TIPO es la definición de los valores posibles que puede tomar un objeto
- Los tipos predefinidos son:
 - Escalares: integer
floating point
enumerated
physical
 - Compuestos: array
record
 - Punteros: access
 - Archivos: file

VHDL ES FUERTEMENTE TIPADO

Algunos tipos básicos predefinidos

- **INTEGER**: tipo entero
 - usado como valor índice en lazos, constantes o valores genéricos
- **BOOLEAN**: tipo lógico
 - Puede tomar como valores 'TRUE' o 'FALSE'
- **ENUMERATED**: Enumeración
 - Conjunto de valores definido por el usuario
 - Por ejemplo: **TYPE estados IS (inicio, lento, rapido)**

Tipos STD_LOGIC y STD_LOGIC_VECTOR

- Definidos en el paquete **IEEE.standard_logic_1164**
- Son un **estándar industrial**.
- **Los emplearemos SIEMPRE para definir los puertos de las entidades.**
- Tipo **Std_logic**: valor presente en un cable de 1 bit
- Tipo **Std_logic_vector**: para definir buses (array de std_logic)

'0' Salida de una puerta con nivel lógico bajo

'1' Salida de una puerta con nivel lógico alto

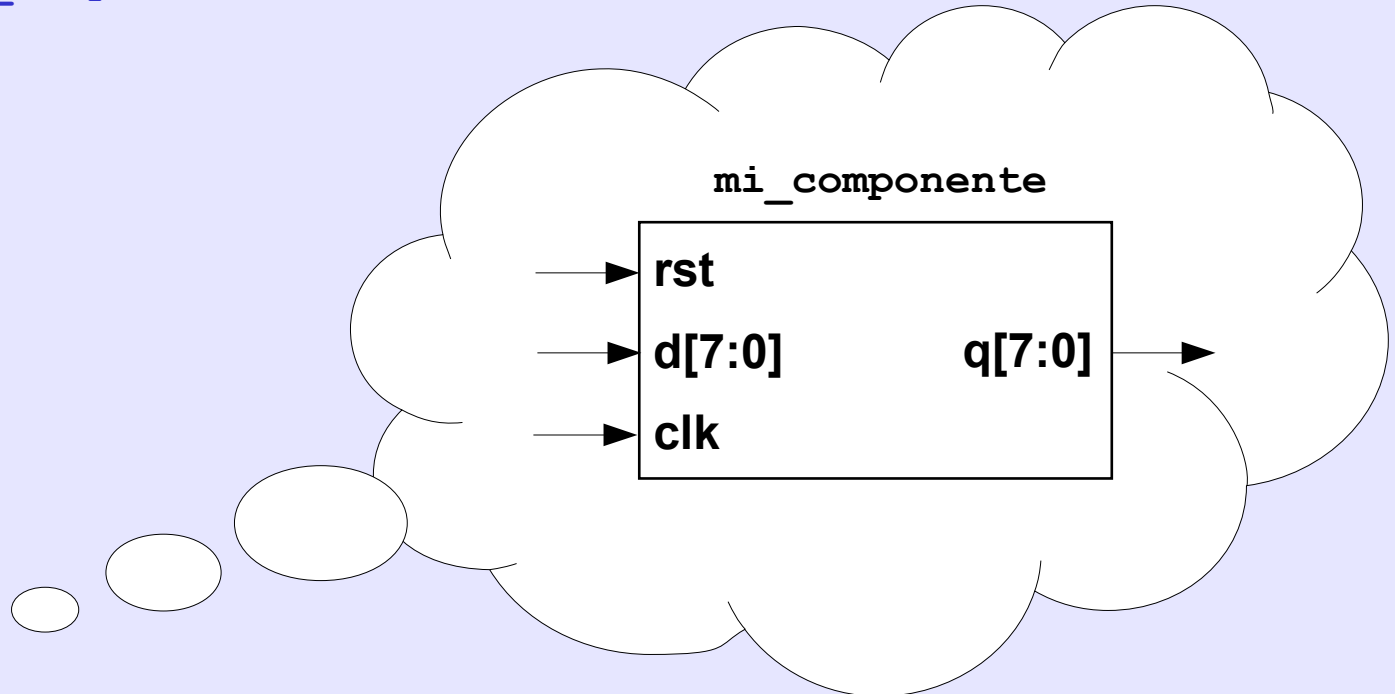
'U' No inicializado, valor por defecto.

'X' Desconocido. Debido a un CORTOCIRCUITO

'Z' Alta Impedancia

- Tiene más valores posibles, que no usaremos en el laboratorio:
'W','L','H','-'

```
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));  
END mi_componente;
```



Asignación de señales en buses

- Vamos a definir una señal de 8 bits para trabajar con ella:

```
SIGNAL tmp: STD_LOGIC_VECTOR(7 downto 0);
```

- Asignación de un valor binario: `tmp <= "10100011";`
- Asignación de un valor en hexadecimal: `tmp <= x"A3";`
- Asignación de un bit: `tmp(7) <= '1';`
- Asignación de un rango de bits: `tmp(7 downto 4) <= "1010";`
- Asignación compacta: `tmp<= (0=>'0', 1=>c and b, others=>'Z');`

- Notación:
 - 1 bit : comilla simple (')
 - multiples bits: comilla doble (")

Tipos SIGNED y UNSIGNED

- Las operaciones aritméticas estándares sólo están definidas para los tipos **signed** y **unsigned**
- Son similares a `std_logic_vector`.
- Están definidos en la librería **IEEE.numeric_std**

```
USE ieee.numeric_std.all;
```

- **Ejemplo de uso:**

- Definimos una variable de tipo unsigned, para implementar un contador:

```
VARIABLE contador: unsigned(7 downto 0);
```

- Incrementamos la variable en 1:

```
contador:=contador + 1;
```

Conversiones de tipos (I)

- VHDL es un lenguaje **FUERTEMENTE TIPADO**
- Las operaciones aritméticas estándares están definidas para los tipos **SIGNED** y **UNSIGNED**
- ...pero los puertos de las entidades se definen SIEMPRE para los tipos **STD_LOGIC** y **STD_LOGIC_VECTOR**...
- ...por tanto hay que hacer **CONVERSIONES** entre tipos
- Existen librerías **NO ESTÁNDARES** que permiten hacer operaciones directamente con el tipo **std_logic_vector**
 - `std_logic_signed`,
 - `std_logic_unsigned`,
 - `std_logic_arith`

Si se quiere hacer un código VHDL portable, conviene no usarlas

Conversiones de tipos (II)

- Usaremos estos objetos como ejemplo:

```
signal stdv: std_logic_vector(7 downto 0);  
variable uns: unsigned(7 downto 0);  
variable sig: signed(7 downto 0);  
variable entero: Integer
```

- Conversión de **signed** y **unsigned** a **std_logic_vector**:

```
stdv<=std_logic_vector(uns);  
stdv<=std_logic_vector(sig);
```

- Conversión de **std_logic_vector** a **signed** y **unsigned**:

```
uns := unsigned(stdv);  
sig := signed(stdv);
```

Conversiones de tipos (III)

- Conversión de **signed** y **unsigned** a **Integer**:

```
entero := to_integer(sig);  
entero := to_integer(uns);
```

- Conversión de **Integer** a **signed** y **unsigned**:

```
uns := to_unsigned(entero, 8);  
sig := to_signed(entero, 8);
```

- Conversión de **std_logic_vector** a **Integer** y vice-versa

```
stdv <= std_logic_vector(to_unsigned(entero, 8));  
entero := to_integer(unsigned(stdv));
```


Definición y uso de nuevos tipos

- Las definiciones de tipos se deben hacer en la parte declarativa de la arquitectura
- **Ejemplo 1.** Definición de un tipo como una enumeración para usarlo en un autómata:

```
TYPE estados IS (INACTIVO, OPERANDO, FINALIZAR);  
SIGNAL mi_maquina : estados;
```

Uso: mi_maquina<=INACTIVO;

- **Ejemplo 2.** Definición de un tipo bidimensional para implementar una memoria:

```
TYPE memoria IS ARRAY (1023 downto 0) OF  
    std_logic_vector(7 downto 0);
```

```
SIGNAL mi_memoria : memoria;
```

Uso: mi_memoria(0)<=x"AA";

Operadores definidos en VHDL

- Lógicos
 - `and`
 - `or, nor`
 - `xor, xnor`
- Relacionales
 - `=` igual
 - `/=` distinto
 - `<` menor
 - `<=` menor o igual
 - `>` mayor
 - `>=` mayor o igual
- Misceláneos
 - `abs` valor absoluto
 - `**` exponenciación
 - `not` negación (unario)
- Adición
 - `+` suma
 - `-` resta
 - `&` concatenación de vectores
- Multiplicativos
 - `*` multiplicación
 - `/` división
 - `rem` resto
 - `mod` módulo
- Signo (unarios)
 - `+, -`
- Desplazamiento (signed y unsigned)
 - `shift_right, shift_left`

Más sobre operadores

- No todos los operadores están definidos para todos los tipos
- El operador de **concatenación** se utiliza muy a menudo

```
signal a:  std_logic_vector( 3 downto 0);  
signal b:  std_logic_vector( 3 downto 0);  
signal c:  std_logic_vector( 7 downto 0);  
a <= "0011";  
b <= "1010";  
c <= a & b;      -- c ="00111010"
```

- Las funciones **shift_right()** y **shift_left()** permiten hacer desplazamientos, pero solo para los tipos **unsigned** y **signed**

```
signal a:  unsigned( 3 downto 0);  
signal b:  unsigned( 3 downto 0);  
  
a <= "0011";  
b <= shift_left(a,1);  -- b ="0110"  
b <= shift_right(a,1); -- b ="0001"
```

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

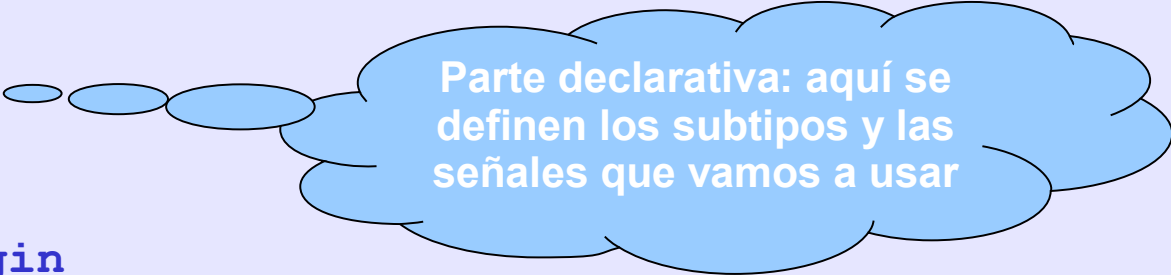
Diseño jerárquico

Estilos de diseño

Verificación con testbenches

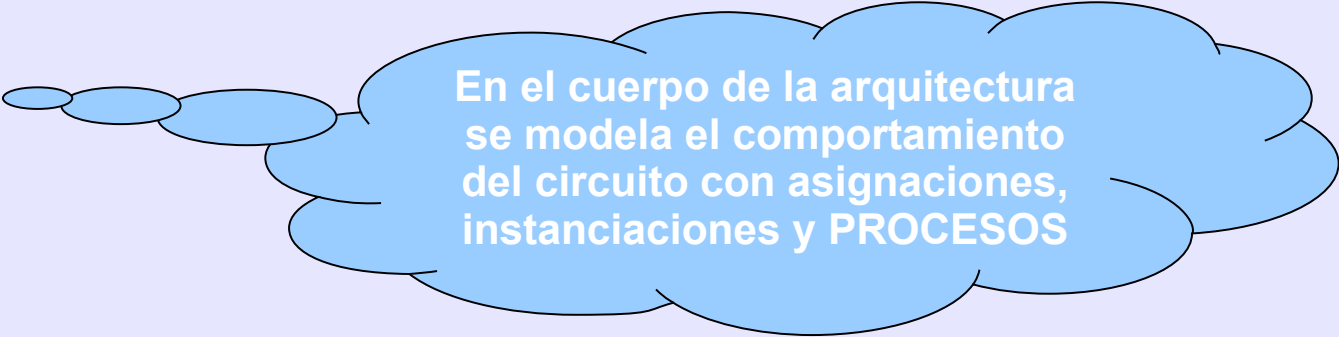
Entrando en detalle en la arquitectura

```
architecture test of mi_componente is
```



Parte declarativa: aquí se definen los subtipos y las señales que vamos a usar

```
begin
```



En el cuerpo de la arquitectura se modela el comportamiento del circuito con asignaciones, instanciaciones y PROCESOS

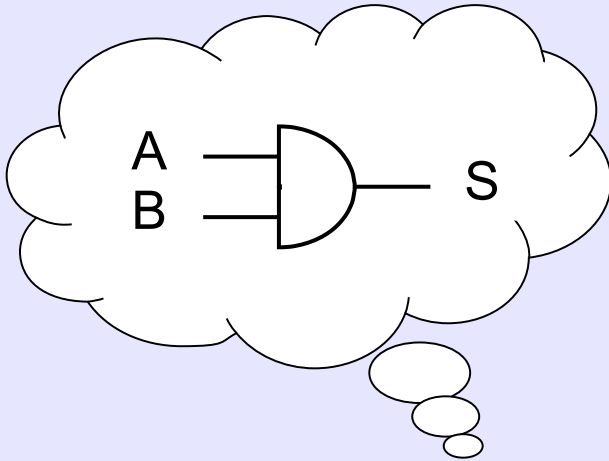
```
end UAM;
```

El proceso: el elemento de diseño principal

- Un proceso describe el comportamiento de un circuito
 - Cuyo estado puede variar cuando cambian ciertas señales
 - Utilizando construcciones muy expresivas: *if..then..else*, *case*, bucles *for* y *while*, etc...
 - Y que además puede declarar variables, procedimientos, etc...

```
process(lista de señales)
...
  parte declarativa (variables, procedimientos, tipos, etc...)
...
begin
...
  instrucciones que describen el comportamiento
...
end process;
```

Ejemplo: Descripción de una puerta AND



La lista de sensibilidad tiene las señales A, B porque cualquier cambio en las entradas puede variar el estado de la puerta

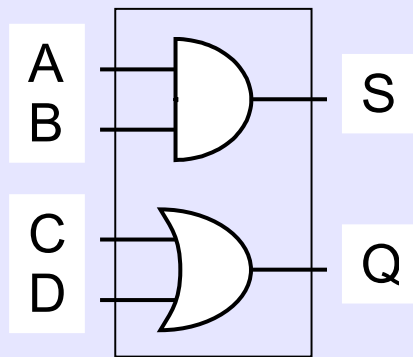
El proceso no declara nada

```
process (A,B)
begin
    if A='1' and B='1' then
        S <= '1';
    else
        S <= '0';
    end if;
end process;
```

Se usa un if..then..else para describir la puerta

El problema de la concurrencia del HW

El HW es inherentemente concurrente,
los circuitos coexisten físicamente en el tiempo



*El chip tiene dos puertas que
funcionan simultáneamente*

Este HW **no** se puede modelar
en un lenguaje secuencial
como C:

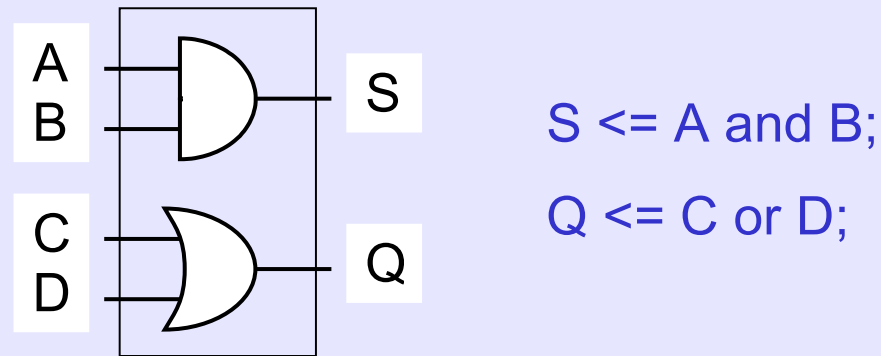
$S = A \& B;$

$Q = C \mid D;$

Ambas puertas funcionan al
mismo tiempo, **¡no una antes
de la otra!**

Concurrencia: Una posible solución

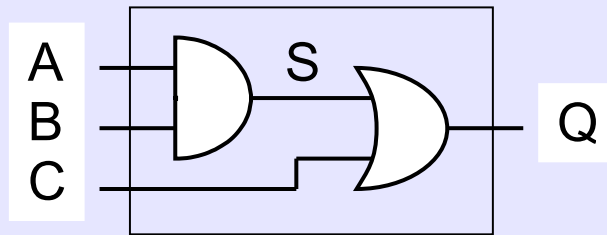
- La solución al problema anterior es que aunque la ejecución sea secuencial, las instrucciones no tarden ningún tiempo en ejecutarse:



- De esta manera la aunque una instrucción se ejecuta después de la otra, como las dos se evalúan en el mismo instante, desde el punto de vista de la modelización del circuito ambas puertas están funcionando simultáneamente
- Esta es la solución por la que opta VHDL (y Verilog)

Necesidad de la concurrencia

- Sin embargo, esta solución ya no vale con este circuito:

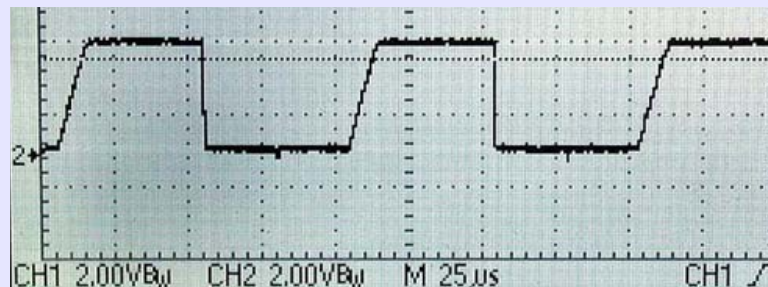


$S \leq A \text{ and } B;$

$Q \leq S \text{ or } C;$

!Q no toma el valor correcto porque no se da tiempo para que se actualice S!

- ¿Por qué? No hay que olvidar que se trata de modelizar circuitos reales, no virtuales, y las señales necesitan que transcurra el tiempo para tomar un valor:



La solución de VHDL

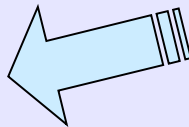
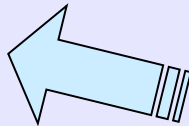
- VHDL (y en general, todos los HDLs) solucionan este problema dando soporte explícito a la concurrencia
- En VHDL, una arquitectura puede tener tantos procesos como queramos, **y todos se ejecutan concurrentemente**

```
architecture ...  
...  
begin
```

```
    process (...)  
    ...  
end process;
```

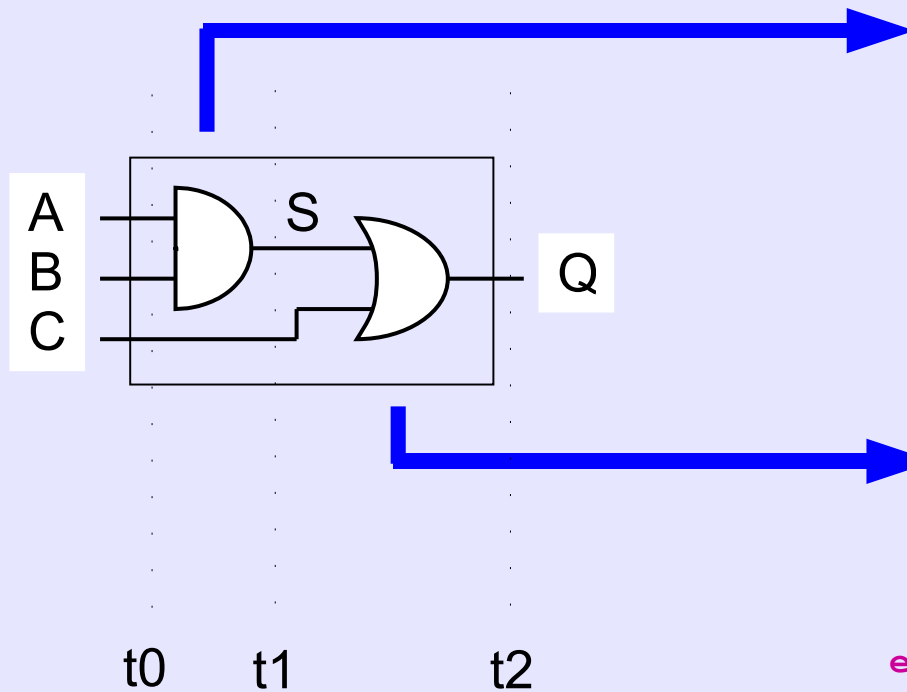
```
    process (...)  
    ...  
end process;
```

```
end ...;
```



***Los procesos se ejecutan
concurrentemente***

Dos procesos en paralelo como solución



```
architecture uam of ejemplo is
...
begin
```

```
process (A,B)
begin
  if A='1' and B='1' then
    S <= '1';
  else
    S <= '0';
  end if;
end process;
```

```
process (C,S)
begin
  if C='1' then
    Q <= '1';
  else
    Q <= S;
  end if;
end process;
```

```
end uam;
```

Procesos: Recapitulando

- Los procesos se disparan (su código se ejecuta) cuando cambia alguna de las señales en su lista de sensibilidad
- Las instrucciones dentro del proceso se ejecutan secuencialmente, una detrás de otra, pero sin dar lugar a que avance el tiempo durante su ejecución
- El tiempo sólo avanza cuando se llega al final del proceso
- Las señales modelan hilos del circuito, y como tales, sólo pueden cambiar de valor si se deja que avance el tiempo
- Una arquitectura puede tener tantos procesos como queramos, y todos se van a ejecutar en paralelo
- Esta es la manera que tiene VHDL de expresar la concurrencia inherente al hardware

Instrucciones en procesos: IF..THEN..ELSE

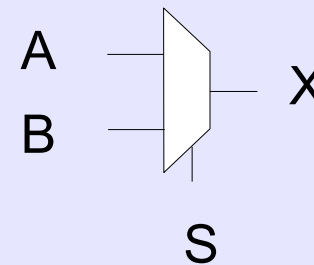
```
IF condicion_1 THEN
    ...
    secuencia de instrucciones 1
    ...
ELSIF condicion_2 THEN
    ...
    secuencia de instrucciones 2
    ...
ELSIF condicion_3 THEN
    ...
    secuencia de instrucciones 1
    ...
ELSE
    ...
    instrucciones por defecto
    ...
END IF;
```

Ejemplo: Un multiplexor

```
process (A,B,S)
begin

    if S = '1' then
        X <= A;
    else
        X <= B;
    end if;

end process;
```



Instrucciones en procesos: CASE

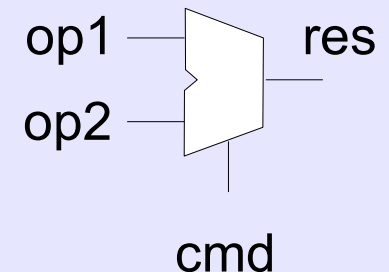
Ejemplo: Una ALU sencilla

```
CASE expresion IS
  WHEN caso_1 =>
    ...
    secuencia de instrucciones 1
    ...
  WHEN caso_2 =>
    ...
    secuencia de instrucciones 2
    ...
  WHEN OTHERS =>
    ...
    instrucciones por defecto
    ...
END CASE;
```

```
architecture uam of alu is
begin
```

```
alu : process (op1, op2, cmd) is
begin
  case cmd is
    when "00" =>
      res <= op1 + op2;
    when "01" =>
      res <= op1 - op2;
    when "10" =>
      res <= op1 and op2;
    when "11" =>
      res <= op1 or op2;
    when others =>
      res <= "XXXXXXXXX";
  end case;
end process alu;
```

```
end architecture uam;
```



Instrucciones en procesos: Bucle FOR

```
[etiqueta] FOR identificador IN rango LOOP
    ...
    instrucciones secuenciales
    ...
END LOOP [etiqueta];
```

```
architecture uam of decoder is
begin
```

```
    decod : process (a) is
begin
```

```
    for i in 0 to 7 loop
```

```
        if i = to_integer(unsigned(a)) then
```

```
            a(i) <= '1';
```

```
        else
```

```
            a(i) <= '0';
```

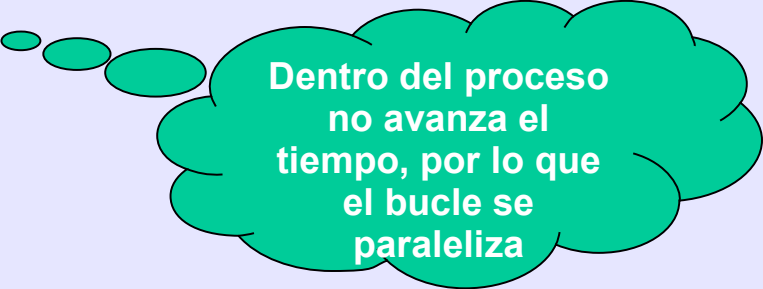
```
        end if;
```

```
    end loop;
```

```
end process decod;
```

```
end architecture uam;
```

Ejemplo:
Decodificador de 3 a 8



Dentro del proceso
no avanza el
tiempo, por lo que
el bucle se
paraleliza

Instrucciones en procesos: Bucle WHILE

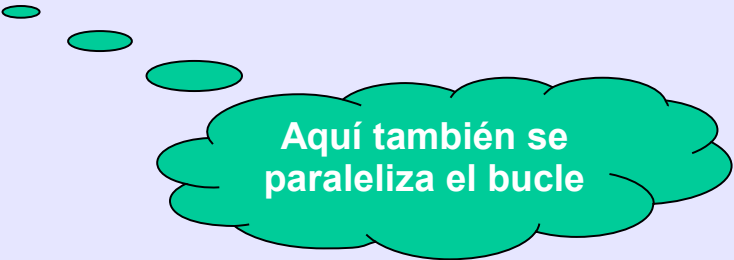
```
[etiqueta] WHILE condicion LOOP
    ...
    instrucciones secuenciales
    ...
END LOOP [etiqueta];
```

Ejemplo:
Búsqueda en una tabla

```
architecture uam of buscar is
begin

    busca: process(valor)
    begin
        encontrado <= '0'; pos := 0;
        while valor /= tabla(pos) or pos < 100 loop
            pos := pos + 1;
        end loop;
        if pos < 100 then
            encontrado <= '1';
        end if;
    end process;

end architecture uam;
```



Aquí también se
paraleliza el bucle

Bucles con next y exit

- En VHDL se pueden crear bucles infinitos

```
[etiqueta] LOOP
    ...
    instrucciones secuenciales
    ...
END LOOP [etiqueta];
```

- Todos los bucles pueden tener una condición de salida

```
exit [etiqueta] [when condicion];
```

- Con la instrucción *next* termina inmediatamente la iteración actual y se pasa a la siguiente

```
next [etiqueta] [when condicion];
```

Procesos: Dos opciones de funcionamiento

Las instrucciones se ejecutan hasta que se llega al final, y entonces se suspende el proceso

```
process(lista de señales)  
...  
begin  
...  
instrucciones secuenciales  
...  
end process;
```

El proceso se dispara cuando cambia alguna de estas señales

Las instrucciones se ejecutan hasta que se llega al wait, y en ese punto se suspende el proceso

```
process  
...  
begin  
...  
instrucciones secuenciales  
...  
wait...  
...  
instrucciones secuenciales  
...  
end process;
```

El proceso se dispara inmediatamente

Cuando se deja de cumplir la condición de espera, la ejecución continúa

Al llegar al final, se empieza otra vez por el principio

Distintas cláusulas wait

- La que más usaremos en las prácticas para hacer bancos de pruebas:

Suspender el proceso durante un tiempo:

Ej. `wait for 10 ns;`

Suspender el proceso hasta que ocurra una condición:

Ej. `wait until rising_edge(clk);`

Finalizar un proceso en el banco de pruebas: `wait;`

- Otras formas de utilización:

- Esperar a que cambie alguna de las señales de una lista:

`wait on a, b, clk;`

Equivalente a emplear
lista de sensibilidad

Asignación de valores a señales

- No olvidar...

Las asignaciones a señales dentro de procesos sólo se ejecutan cuando se suspende el proceso

- No es un dogma de fe, tiene su explicación...
 - Las señales modelan conexiones físicas, y por tanto, no sólo deben tener en cuenta el valor, sino también el tiempo
 - Para que un cable cambie de valor hace falta que el tiempo avance
 - De la misma forma, para que una señal cambie de valor hace falta que el tiempo avance
 - El tiempo sólo avanza cuando se suspende el proceso

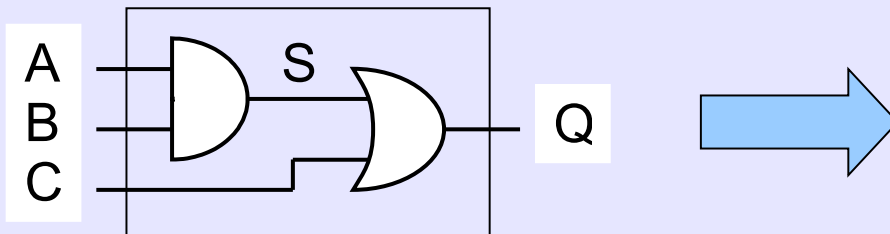
Las variables

- A la hora de modelar un circuito nos puede venir bien un tener un objeto cuyo valor se actualice inmediatamente
 - sin tener que esperar a que avance el tiempo, como en las señales
- La solución son las variables
 - Las variables se declaran dentro de los procesos
 - Sólo se ven dentro del proceso que las ha declarado
 - Toman el valor inmediatamente, son independientes del tiempo

```
process (a,b,c)
...
    variable v : std_logic;
...
begin
...
    v := a and b or c;
...
end process;
```

Solución con variables

El problema de la actualización de la señal S tiene muy fácil solución con una variable



```
architecture uam of ejemplo is  
...  
begin
```

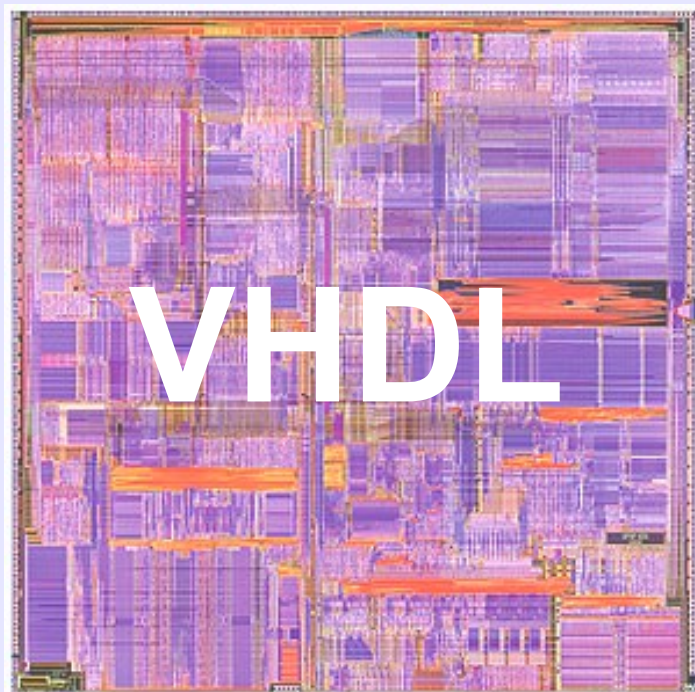
```
    process (A,B,C)  
        variable S : std_logic;  
    begin  
        S := A and B;  
        if C='1' then  
            Q <= '1';  
        else  
            Q <= S;  
        end if;  
    end process;
```

```
end uam;
```

Semántica de variables y señales

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

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

Estilos de diseño

Verificación con testbenches

Modelar lógica combinacional con procesos

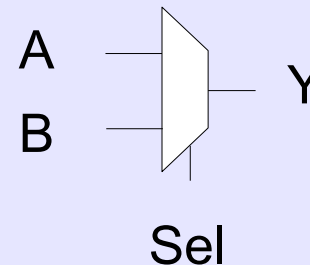
```
architecture uam of mux is  
begin
```

```
    process (a,b,sel)  
    begin  
        if sel='1' then  
            y <= a;  
        else  
            y <= b;  
        end if;  
    end process;
```

```
end uam;
```

Se debe asignar
siempre (en
todos los casos)
a la salida un
valor

Todas las entradas
deben estar en la
lista de sensibilidad



El problema de la memoria implícita

- CAUSA
 - las señales en VHDL tienen un estado actual y un estado futuro
- EFECTOS
 - En un proceso, si el valor futuro de una señal no puede ser determinado, se mantiene el valor actual.
 - Se sintetiza un latch para mantener su estado actual
- VENTAJAS
 - Simplifica la creacion de elementos de memoria
- DESVENTAJAS
 - Pueden generarse latches no deseados,p.ej. cuando todas las opciones de una sentencia condicional no están especificadas

Un problema con la memoria implícita

- Diseñar un circuito de acuerdo a esta tabla de verdad

A	S
00	1
01	1
10	0
11	don't care

```
process (a)
begin
  case a is
    when "00" =>
      res <= '1';
    when "01" =>
      res <= '1';
    when "10" =>
      res <= '0';
  end process;
```

- Solución es incorrecta, por no poner el caso "11" no significa "don't care", simplemente está guardando el valor anterior, está generando un latch

Reglas para evitar la memoria implícita

- Para evitar la generación de latches no deseados
 - Se deber terminar la instrucción IF...THEN...ELSE... con la cláusula ELSE
 - Especificar todas las alternativas en un CASE, definiendo cada alternativa individualmente, o mejor terminando la sentencia CASE con la cláusula WHEN OTHERS... Por ejemplo,

```
CASE decode IS
    WHEN "100" => key <= first;
    WHEN "010" => key <= second;
    WHEN "001" => key <= third;
    WHEN OTHERS => key <= none;
END CASE;
```

Asignaciones concurrentes

- Las asignaciones concurrentes son asignaciones de valores a señales, fuera de proceso, que permiten modelar de una manera muy compacta lógica combinacional
 - Funcionan como procesos (son procesos implícitos) y se ejecutan concurrentemente con el resto de procesos y asignaciones
- Hay tres tipos
 - Asignaciones simples
 - Asignaciones condicionales
 - Asignaciones con selección

```
s <= (a and b) + c;
```

```
s <= a when c='1' else b;
```

```
with a+b select  
s <= d when "0000",  
e when "1010",  
'0' when others;
```

Asignaciones concurrentes simples

- A una señal se le asigna un valor que proviene de una expresión, que puede ser tan compleja como queramos

```
s <= ((a + b) * c) and d;
```

- Esta expresión es completamente equivalente a este proceso:

```
process (a,b,c,d)
begin
    s <= ((a + b) * c) and d;
end process;
```

- Se pueden utilizar todos los operadores que queramos, tanto los predefinidos como los que importemos de las librerías

Asignaciones concurrentes condicionales

- A la señal se le asigna valores dependiendo de si se cumplen las condiciones que se van evaluando:

```
architecture uam of coder is
begin
    s <= "111" when a(7)='1' else
        "110" when a(6)='1' else
        "101" when a(5)='1' else
        "100" when a(4)='1' else
        "011" when a(3)='1' else
        "010" when a(2)='1' else
        "001" when a(1)='1' else
        "000";
end architecture uam;
```

- Por su ejecución en cascada es similar al IF..THEN..ELSE
- Pueden generarse problemas de memoria implícita si no se pone el último e/se

Asignaciones concurrentes con selección

- Se le asigna un valor a una señal dependiendo del valor que tome una expresión:

```
architecture uam of decod is
begin
  with a sel
    s <= "00000001" when "000",
        "00000010" when "001",
        "00000100" when "010",
        "00001000" when "011",
        "00010000" when "100",
        "00100000" when "101",
        "01000000" when "110",
        "10000000" when others;
end architecture uam;
```

- Por su ejecución en paralelo (balanceada) es similar a un CASE
- Se pueden dar problemas de memoria implícita si no se pone el último *when others*

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

Estilos de diseño

Verificación con testbenches

El fundamento: Modelo del flip-flop D

también vale
`rising_edge(clk)`

```
process (clk)
begin
-  if clk'event and clk='1' then
        q <= d;
    end if;
end process;
```

proceso sensible
al reloj

no hay else,
queremos
inferir memoria

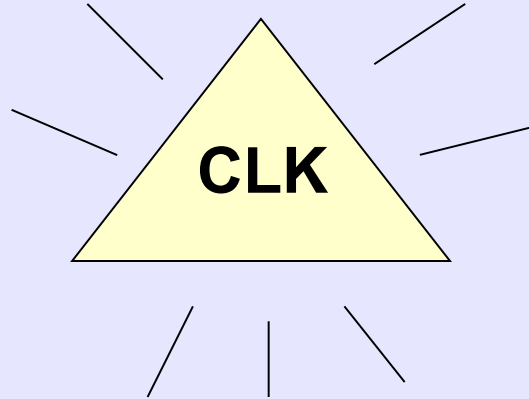
cambia el reloj y
es ahora 1 ...
hay un flanco de
subida

Flip-flop con reset asíncrono y *clock enable*

```
process (clk,rst)
begin
    if rst='1' then
        q <= '0';
    elsif clk'event and clk='1' then
        if ce='1' then
            q <= d;
        end if;
    end if;
end process;
```

- Otro circuito fundamental.
- El reset debe estar en la lista de sensibilidad porque es asíncrono, tiene efecto independientemente del reloj.
- En los circuitos secuenciales, la lista de sensibilidad debe estar compuesta como mucho por el reloj y el reset (si es asíncrono).

El axioma del diseño síncrono



El reloj es único y está en todos los flip-flops del diseño

- No se pueden usar dos relojes en el sistema
- Todas las señales asíncronas se deben muestrear (pasar por un flip-flop D) nada más entrar al sistema
- No se deben poner puertas en el reloj, si se necesita deshabilitar la carga de un flip-flop utilizar la habilitación de reloj

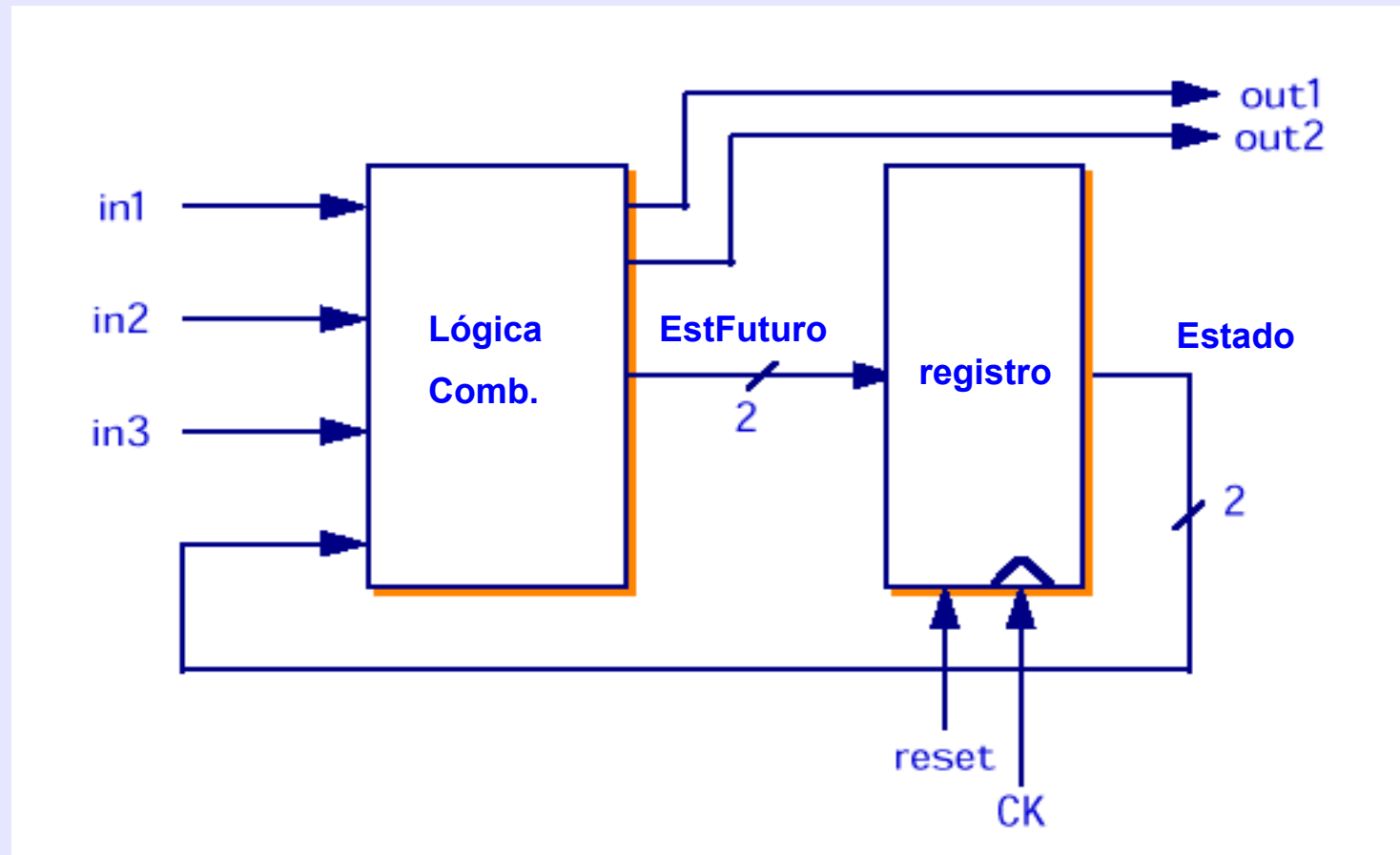
Ejemplo: Un contador de 8 bits

```
process(clk,rst)
    variable q_temp : unsigned(7 downto 0);
begin
    if rst='1' then
        q_temp := (others => '0');
    elsif rising_edge(clk) then
        if ce='1' then
            if up='1' then
                q_temp := q_temp + 1;
            else
                q_temp := q_temp - 1;
            end if;
        end if;
    end if;
    q <= std_logic_vector(q_temp);
end process;
```

Ejemplo: Un registro de desplazamiento

```
process (rst, clk)
    variable q_temp : std_logic_vector(7 downto 0);
begin
    if rst='1' then
        q_temp:="00000000";
    elsif rising_edge(clk) then
        if ce='1' then
            if load='1' then
                q_temp:=din;
            else
                q_temp:=q_temp(6 downto 0) & sin;
            end if;
        end if;
    end if;
    q<=q_temp;
end process;
```

Metodología: Diseño circuitos secuenciales



Metodología: Diseño circuitos secuenciales

Maquinas 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 de
definición de salidas

```
architecture uam of ejemplo is
    type t_estado is (E0, E1, E2, E3);
    signal Estado, EstFuturo : t_estado;
    signal in1, in2, in3 : std_logic;
    signal out1, out2 : std_logic;
    signal CK, reset : std_logic;
    ...
```


Metodología: Diseño circuitos secuenciales

Maquinas de estados: FSM

Utilización de subtipos:

Definicion de Estados

Tres Bloques Funcionales

Lógica combinacional:
Decision de cambio de
estado

Registros: Mantienen el
estado.

Logica combinacional de
definición de salidas

```
registro: process(reset,clk)
begin
    if reset='1' then
        Estado <= E0;
    elsif rising_edge(clk) then
        Estado <= EstFuturo;
    end if;
end process registro;
```

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

Estilos de diseño

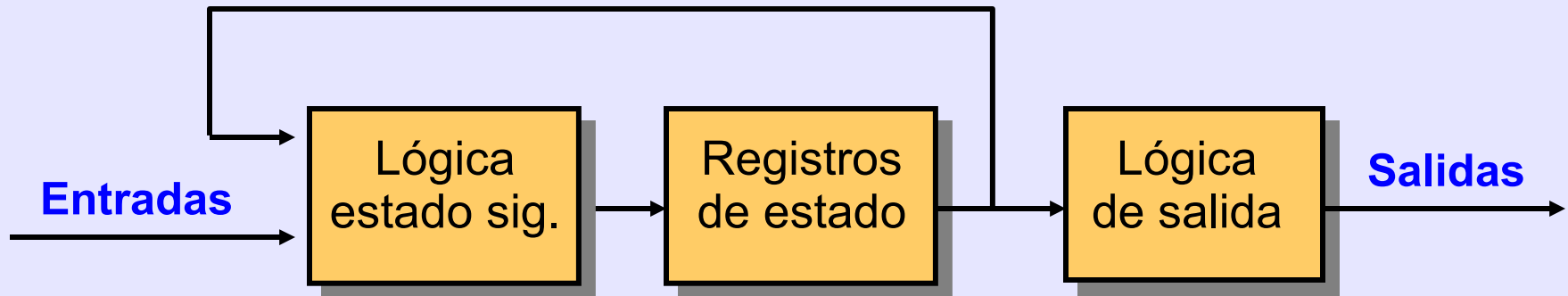
Verificación con testbenches

FSM: Maquinas de Moore

- FSM MOORE: Una maquina de estados en la que las salidas cambian solo cuando cambia el estado
- Las posibles implementaciones son:
 - **Asignación arbitraria del valor de los estados**
 - Las salidas se decodifican a partir de los estados
 1. Decodificación combinacional.
 2. Decodificación registrada.
 - **Asignación específica de los valores de estado**
 - Las salidas pueden ser codificadas directamente en los estados
 - Codificación one-hot

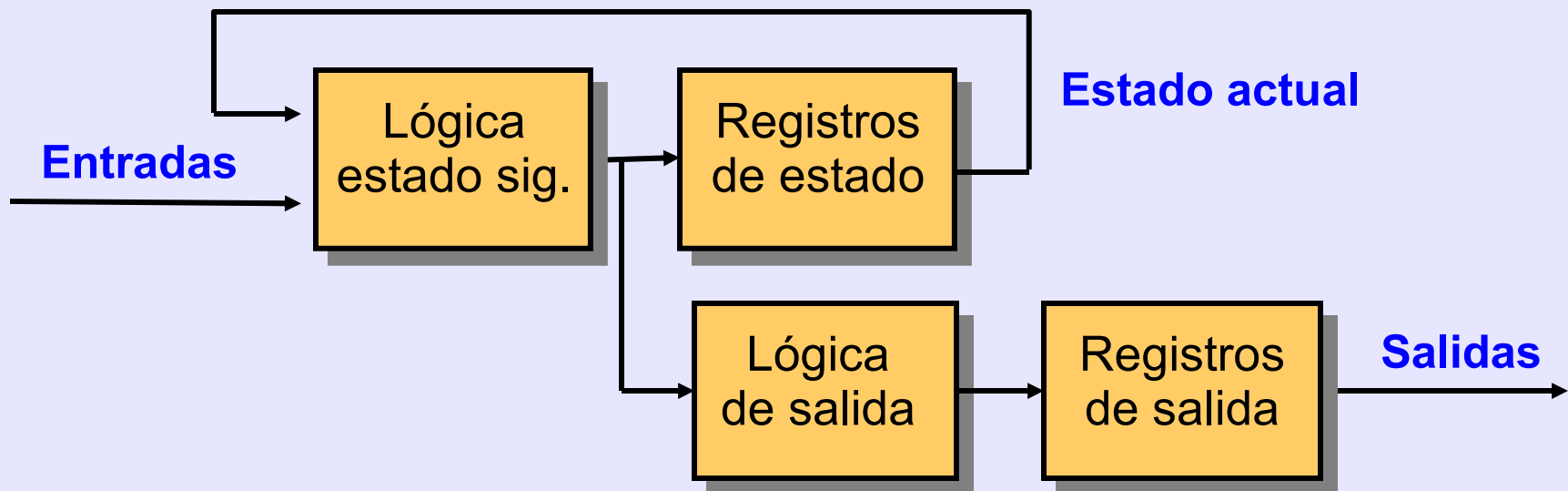
Implementación de una FSM de Moore (1)

- Salidas decodificadas a partir del valor de los estados.
 1. **Decodificación Combinacional**
 - Las salidas se decodifican a partir del estado actual
 - $\text{Salidas} = \text{función}(\text{estado_actual})$



Implementación de una FSM Moore (2)

- Salidas decodificadas a partir del valor de los estados.
 1. **Decodificación con salidas registradas**
 - La decodificación de las salidas se realiza en paralelo con la decodificación del siguiente estado.
 - $\text{Salidas} = \text{función}(\text{estado_anterior}, \text{entradas})$

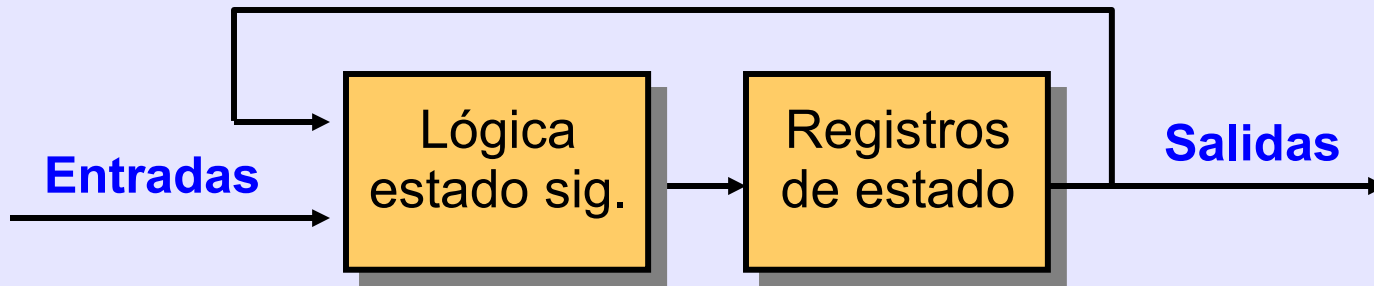


Implementación de una FSM Moore (3)

- Salidas codificadas en los bits de los estados

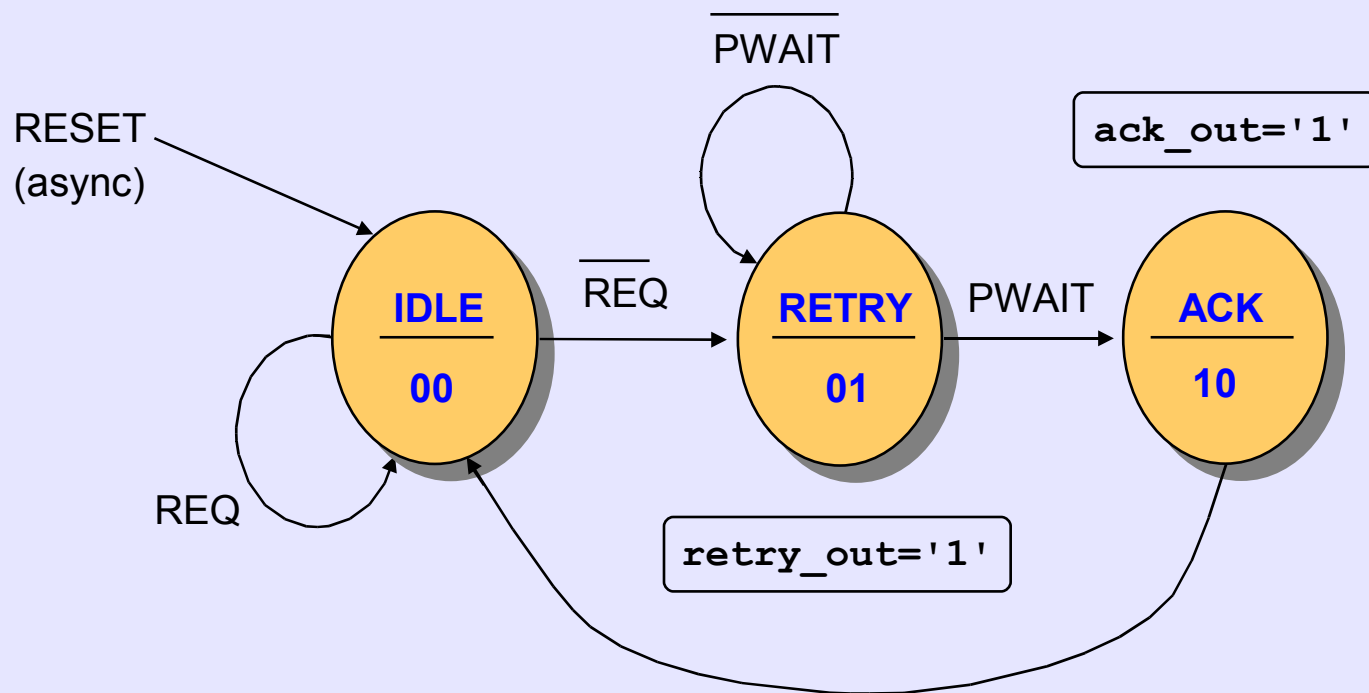
Estado	Salida 1	Salida 2	Codif. Estados
s1	0	0	00
s2	1	0	01
s3	0	1	10

Nota: Los dos bits del estado son utilizados como salida



Ejemplo: Generador de “wait states”

- Diagrama de Estados:



Ejemplo: Declaración de la entidad

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

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
  
ENTITY maq IS PORT (  
    clock, reset: IN std_logic;  
    req, pwait: IN std_logic;  
    retry_out, ack_out: OUT std_logic);  
END maq;
```

Ejemplo: Solución 1

- Salidas combinacionales decodificadas a partir de los estados

```

ARCHITECTURE archmoore1 OF maq IS

    TYPE fsm_states IS (idle, retry, ack);
    SIGNAL wait_gen : fsm_states;

BEGIN
    fsm: PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            wait_gen <= idle;  -- asynchronous reset

        ELSIF clock'EVENT AND clock = '1' THEN
            CASE wait_gen IS

                WHEN idle => IF req = '0' THEN wait_gen <= retry;
                           ELSE wait_gen <= idle;
                           END IF;
            END CASE;
        END IF;
    END PROCESS;
END archmoore1;

```

Ejemplo: Solución 1 (cont.)

```

    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 fsm;

retry_out <= '1' WHEN (wait_gen = retry) ELSE '0';
ack_out   <= '1' WHEN (wait_gen = ack)   ELSE '0';

END archmoore1;

```

Ejemplo: Solucion 2

- Salidas registradas decodificadas desde el valor de los estados

```

ARCHITECTURE archmoore2 OF maq IS

    TYPE fsm_states IS (idle, retry, ack);
    SIGNAL wait_gen: fsm_states;

BEGIN
    fsm: PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            wait_gen <= idle;
            retry_out <= '0';
            ack_out <= '0';

            ELSIF clock'EVENT AND clock = '1' THEN
                retry_out <= '0'; -- asignacion por defecto
            
```

Ejemplo: Solución 2 (cont.)

```

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;
                    ack_out  <= '1';
                    ELSE wait_gen <= retry;
                    retry_out <= '1';
                    ack_out  <= '0';
                    END IF;

  WHEN ack    =>    wait_gen <= idle;
                    ack_out  <= '0';

  WHEN OTHERS =>    wait_gen <= idle;
                    ack_out  <= '0';  -- para evitar latch

END CASE;

END IF;
END PROCESS fsm;
END archmoore2;

```

Ejemplo: Solución 3

- Salidas codificadas en el valor de los estados

```
ARCHITECTURE archmoore3 OF maq IS
```

```
    SIGNAL wait_gen: std_logic_vector(1 DOWNTO 0);
    CONSTANT idle:   std_logic_vector(1 DOWNTO 0) := "00";
    CONSTANT retry:  std_logic_vector(1 DOWNTO 0) := "01";
    CONSTANT ack:    std_logic_vector(1 DOWNTO 0) := "10";
```

```
BEGIN
```

```
    fsm: PROCESS (clock, reset)
```

```
    BEGIN
```

```
        IF reset = '1' THEN
```

```
            wait_gen <= idle;
```

```
        ELSIF clock'EVENT AND clock = '1' THEN
```

Ejemplo: Solución 3 (cont.)

```

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 <= ack;
                  ELSE wait_gen <= retry;
                  END IF;

    WHEN ack    => wait_gen <= idle;
    WHEN OTHERS => wait_gen <= idle;
END CASE;

END IF;
END PROCESS fsm;

retry_out <= wait_gen(0);
ack_out   <= wait_gen(1);

END archmoore3;

```


FSM: Codificación One-hot

- Un estado por flip-flop
 - En FPGAs
 - reduce la lógica de cálculo de estado siguiente
 - y por tanto, menos profundidad de lógica
 - permitiendo máquinas muy rápidas (>100MHz)
 - En CPLDs
 - reduce el número de términos producto
 - eliminando, si los hubiera, expansiones de productos, y mejorando por tanto la velocidad
 - pero usa muchas más macroceldas, y el beneficio nunca es tan evidente como en FPGAs

Ejemplo: Solución One-hot

```

ARCHITECTURE archmoore4 OF maq IS

    TYPE fsm_states IS (idle, retry, ack);
    ATTRIBUTE enum_encoding: string;
    ATTRIBUTE enum_encoding OF fsm_states : TYPE IS "001 010 100";
    SIGNAL wait_gen: fsm_states;

BEGIN
    fsm: PROCESS (clock, reset)
    BEGIN
        IF reset = '1' THEN
            wait_gen <= idle;

        ELSIF clock'EVENT AND clock = '1' THEN
            CASE wait_gen IS
                WHEN idle =>      IF req = '0' THEN wait_gen <= retry;
                                   ELSE wait_gen <= idle;
                                   END IF;
            END CASE;
        END IF;
    END PROCESS;
END archmoore4;

```

Ejemplo: Solución One-hot (cont.)

```

    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 fsm;

-- Decodificación de salidas
retry_out <= '1' WHEN (wait_gen = retry) ELSE '0';
ack_out   <= '1' WHEN (wait_gen = ack)   ELSE '0';

END archmoore4;

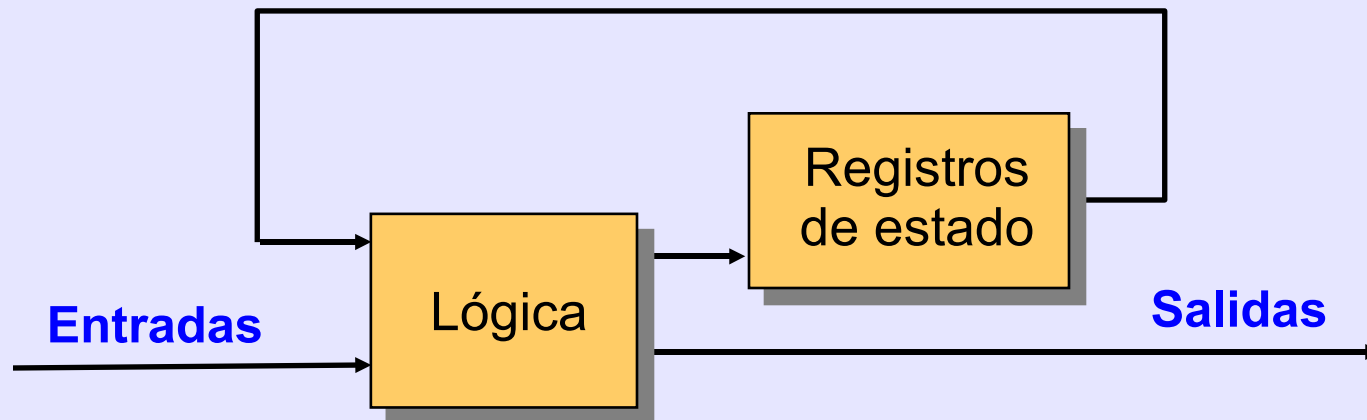
```

Resumen FSM Moore

- Salidas decodificadas de los bits de estado
 - Mayor flexibilidad en el proceso de diseño
 - Utilizando tipos enumerados se permite que la asignación de los estados se realice durante la compilación.
- Salidas codificadas en los bits de estado
 - Asignación manual del valor de los estados
 - La salida se obtiene directamente de los registros
 - Se reduce le número de registros
 - Lógica adicional más compleja
- Codificación One-Hot
 - Logica de siguiente estado mas sencilla
 - Mejora la velocidad
 - Necesita mas registros

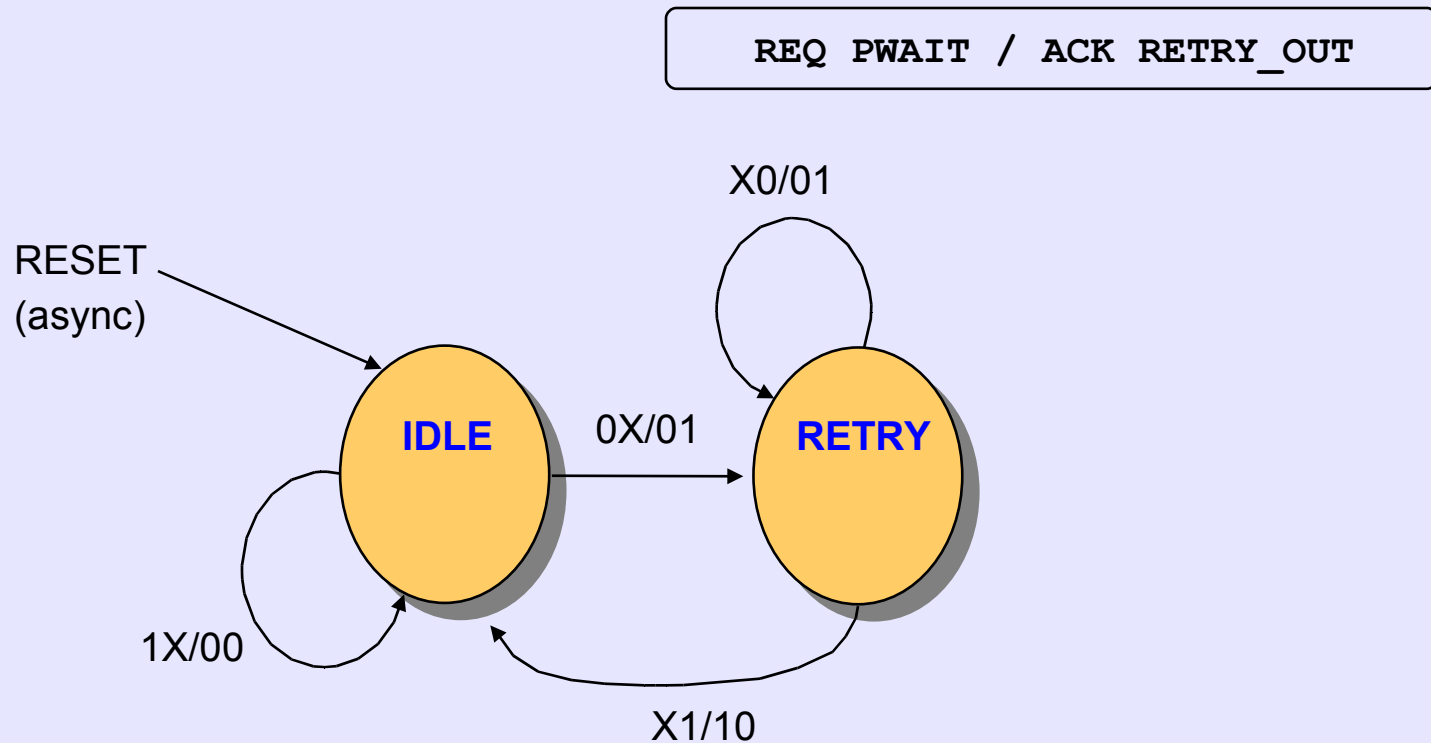
FSM de Mealy

- 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 de “wait states”

- Diagrama de estados:



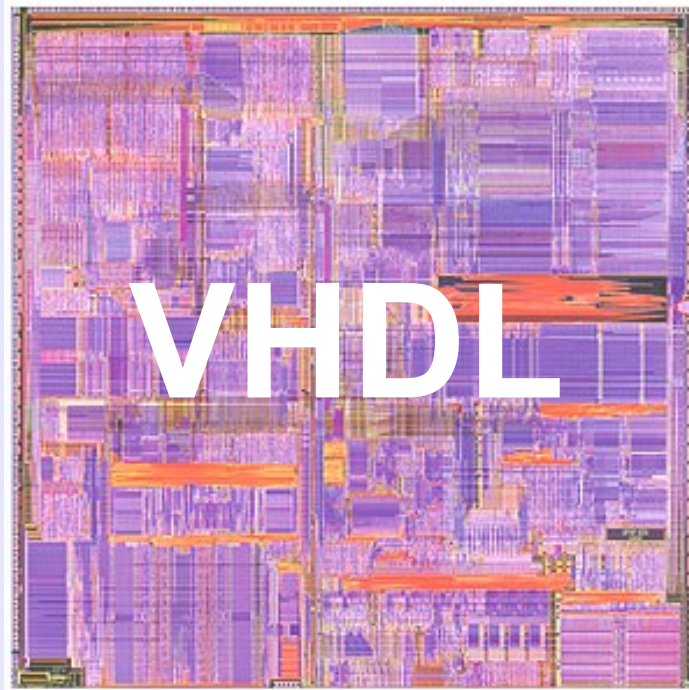
Ejemplo: Solución

```

ARCHITECTURE archmealy1 OF maq IS
  TYPE fsm_states IS (idle, retry);
  SIGNAL wait_gen: fsm_states;
BEGIN
  fsm: PROCESS (clock, reset)
  BEGIN
    IF reset = '1' THEN
      wait_gen <= idle;
    ELSIF clock'EVENT AND clock = '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;
        WHEN OTHERS    => wait_gen <= idle;
      END CASE;
    END IF;
  END PROCESS fsm;
  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 archmealy1;

```

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

Estilos de diseño

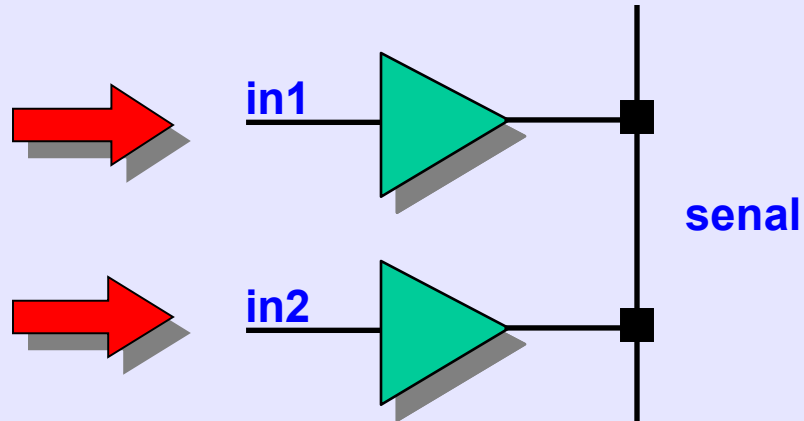
Verificación con testbenches

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 cuantas veces se le asigne un valor a la señal, se crea un único driver por proceso
 - Tanto para procesos explícitos como implícitos
 - Cuando hay múltiples drivers se usa la función de resolución

```
PROCESS (in1)  
BEGIN  
    senal <= in1;  
END PROCESS;
```

```
senal <= in2;
```

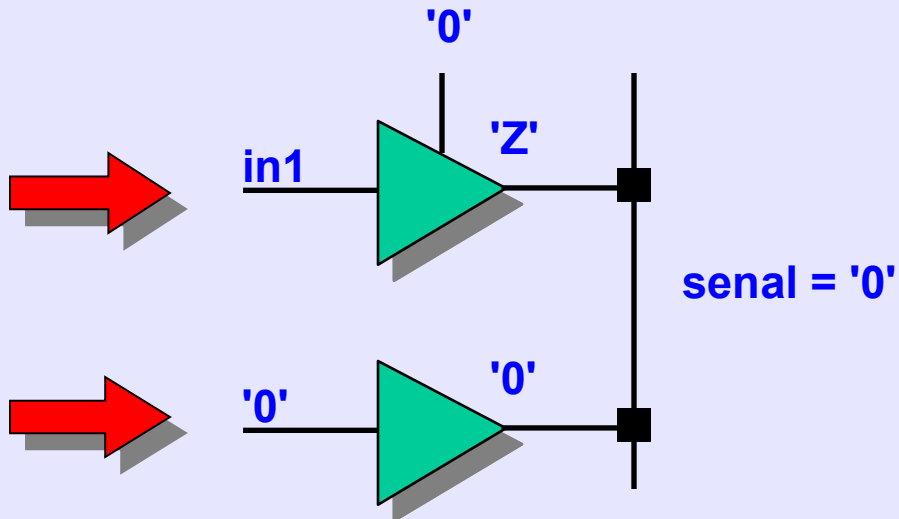


Inferencia de triestado

- Cuando se quiere que un driver de una señal se quede en alta impedancia, se le asigna a la señal el valor 'Z'
 - Sólo vale si 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='1' ELSE 'Z';
```

```
PROCESS(in1)
BEGIN
    senal <= '0';
END PROCESS;
```



Ejemplos de inferencia de buffers 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 (ena_a, a)  
BEGIN  
    IF (sel_a = '0') THEN  
        t <= a;  
    ELSE t <= 'Z';  
END PROCESS;
```

Señales bidireccionales

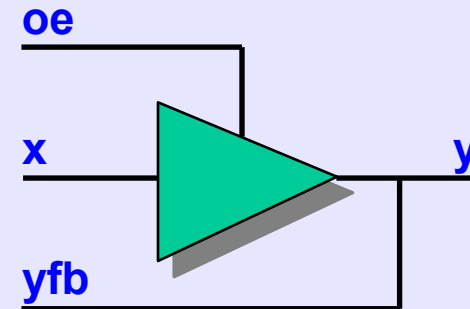
- En este caso la señal tiene drivers externos, fuera de la entidad

```

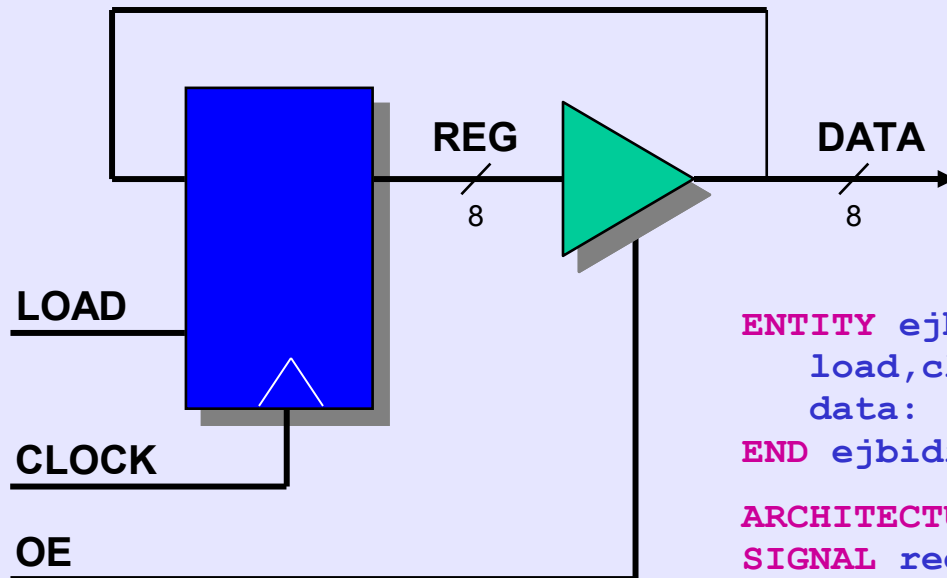
ENTITY bufoe IS PORT (
    x:    IN std_logic;
    oe:    IN std_logic;
    y:    INOUT std_logic;
    yfb: OUT std_logic);
END bufoe;

ARCHITECTURE simple OF bufoe IS
BEGIN
    y <= x WHEN oe='1' ELSE 'Z';
    yfb <= y;
END simple;

```



Ejemplo con señales bidireccionales



```

ENTITY ejbidir IS PORT (
    load, clock, oe:    IN std_logic;
    data:              INOUT std_logic);
END ejbidir;

ARCHITECTURE simple OF ejbidir IS
    SIGNAL reg: std_logic_vector(7 downto 0);
BEGIN
    data<=reg WHEN oe='1' ELSE "ZZZZZZZZ";
    PROCESS(clk) BEGIN
        IF rising_edge(clk) THEN reg<=data;
        END IF;
    END PROCESS;
END simple;

```

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

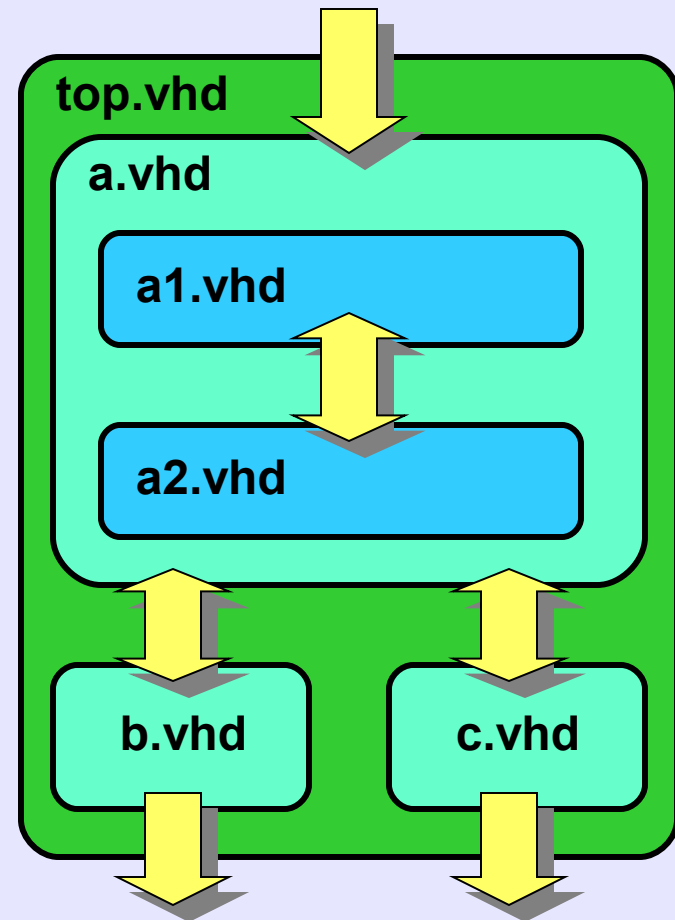
Diseño jerárquico

Estilos de diseño

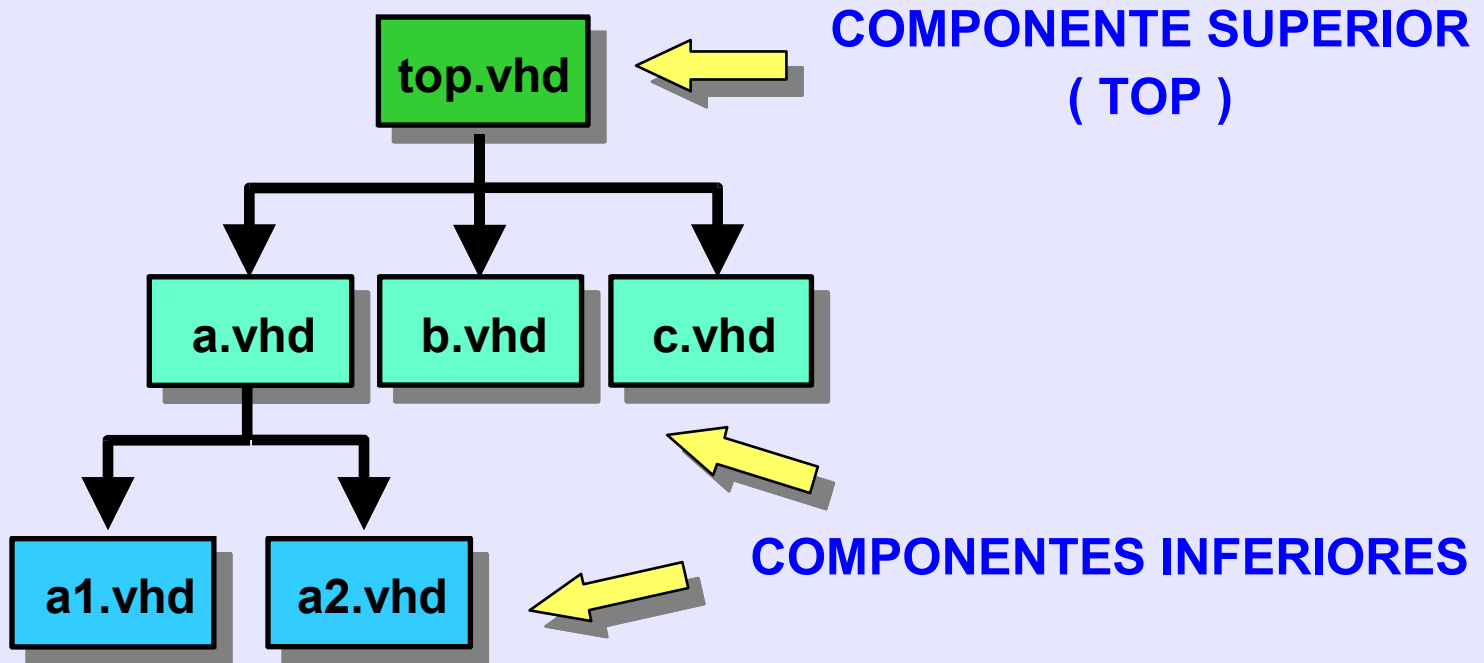
Verificación con testbenches

Diseño jerárquico

- Componentes pequeños son utilizados como elementos de otros más grandes
- Permite reutilizar código
- Diseños más legibles y portables



Árbol de jerarquías



- Cada componente de la jerarquía es un archivo VHDL, con:
 - Entidad
 - Arquitectura

Ejemplo de instanciación de componentes

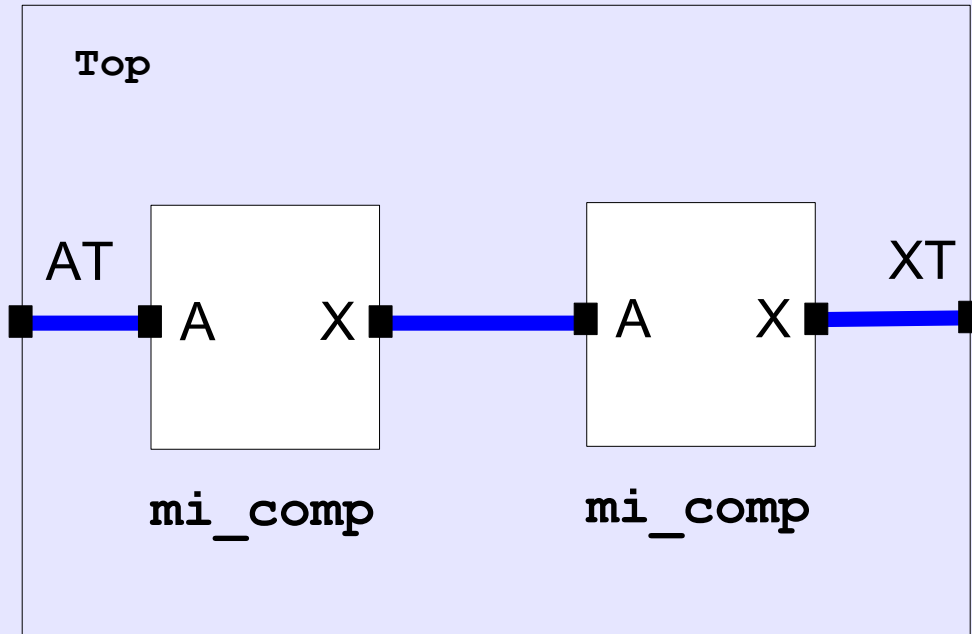
- Queremos diseñar esta entidad:

```
ENTITY Top IS PORT (  
    AT: IN  std_logic;  
    XT: OUT std_logic;  
END Top;
```



Ejemplo de instanciación de componentes

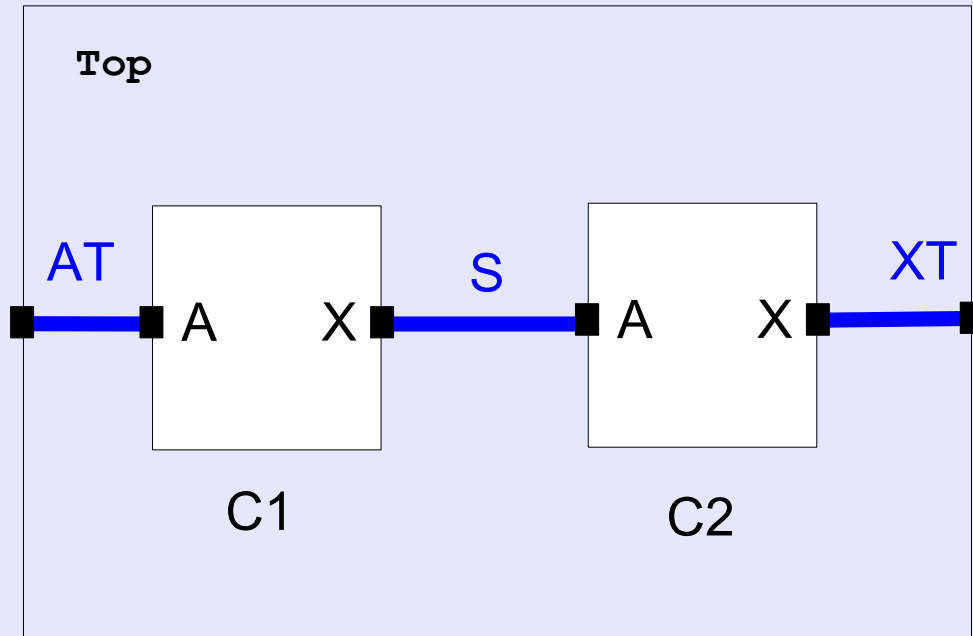
- Y la podemos implementar a partir de la conexión en serie de los componentes `mi_comp`



```
ENTITY Top IS
  PORT (
    AT: IN  std_logic;
    XT: OUT std_logic);
END Top;
```

```
ENTITY mi_comp IS
  PORT (
    A: IN  std_logic;
    X: OUT std_logic);
END mi_comp;
```

Ejemplo de instanciación de componentes



Architecture test of Top IS

Component mi_comp

```
PORT (  
  A: IN  std_logic;  
  X: OUT std_logic);
```

Signal S : std_logic;

Begin

```
C1:mi_comp PORT MAP  
  (A=>AT, X=>S)
```

```
C2:mi_comp PORT MAP  
  (A=>S, X=>XT)
```

END test;

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

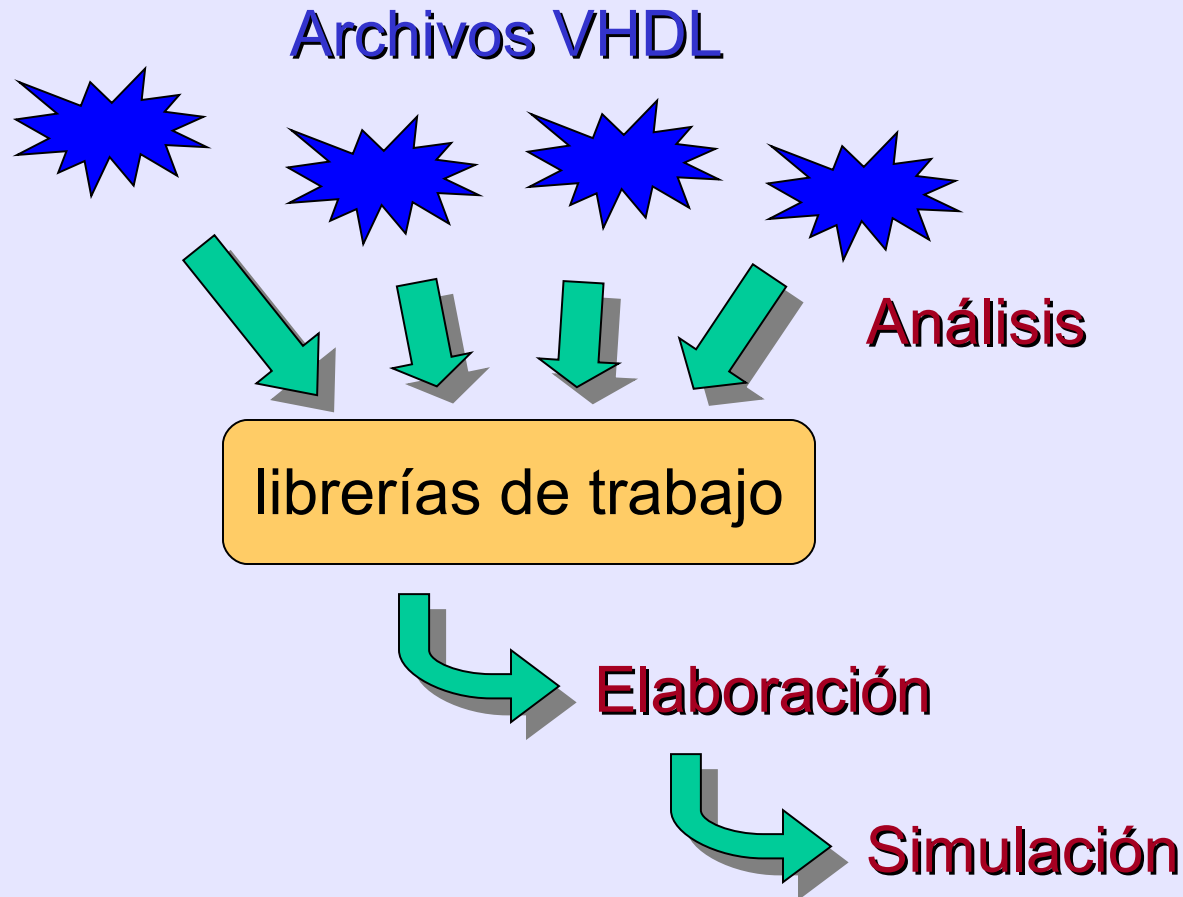
Triestados

Diseño jerárquico

Estilos de diseño

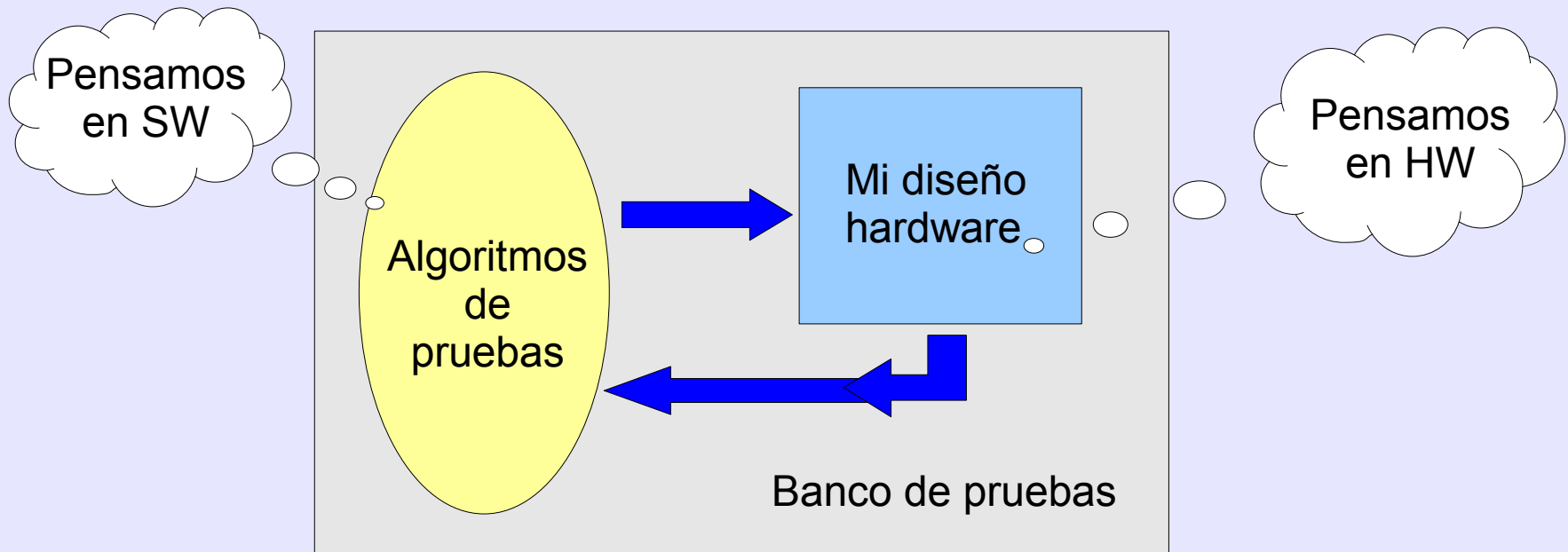
Verificación con testbenches

Pasos de la simulación



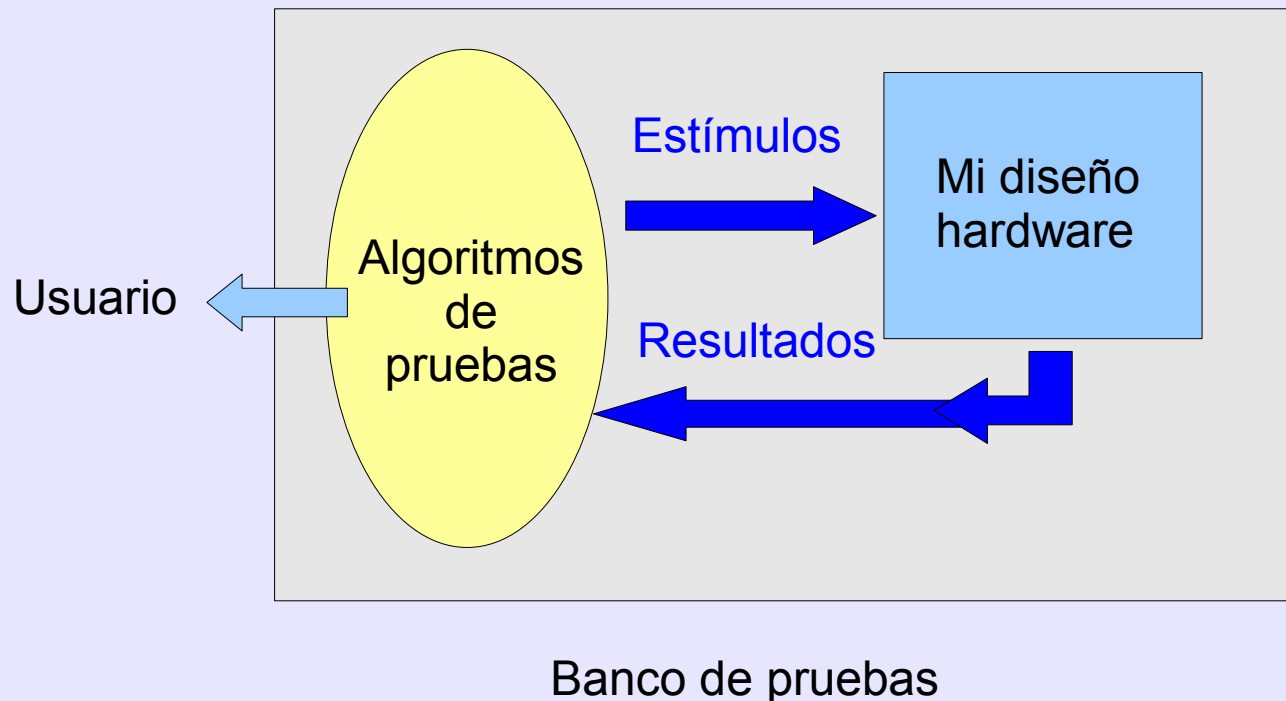
Bancos de pruebas

- Hay que hacer un banco de pruebas (testbench) para cada componente diseñado.
- La simulación de un componente consiste en:
 - Generar unos estímulos
 - Observar los resultados



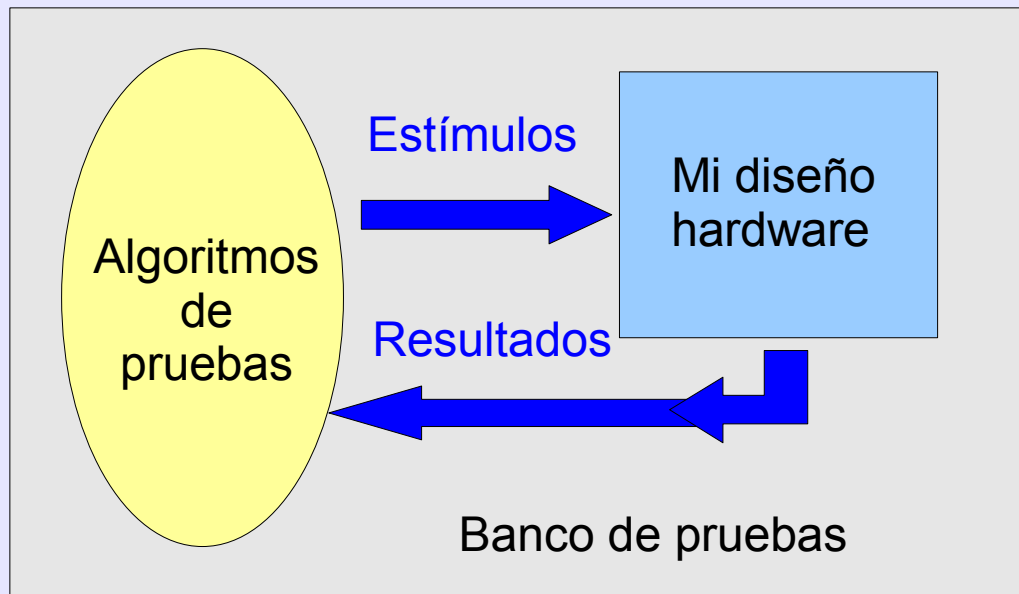
Como hacer un banco de pruebas (I)

1. Instanciar el diseño que vamos a verificar
2. Escribir un proceso (o procesos) para **generar** los estímulos, **observar** el resultado e **informar** al usuario



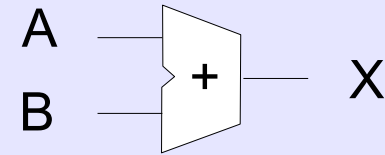
Como hacer un banco de pruebas (II)

- Es un diseño jerárquico donde el banco de pruebas ocupa el primer nivel
- El banco de pruebas es una ENTIDAD sin puertos



Generando estímulos

- El proceso de pruebas generará estímulos de manera secuencial
- Se asignan valores a las entradas
- Se espera un tiempo a obtener la respuesta
- Ej. Sumador de 8 bits



```
A <= x"03";
```

```
B <= x"02";
```

```
WAIT FOR 20 ns;
```

→ Dar valores a las entradas del sumador

```
A<= x"AA";
```

```
B<= x"20";
```

```
WAIT FOR 30 ns;
```

→ Esperar un tiempo
(el proceso se suspende)

Comprobando las respuestas

- Las respuestas del circuito a probar se comprueban con la instrucción ASSERT

```
ASSERT condicion REPORT string SEVERITY nivel;
```

- Significa: “Asegurar que se cumple la condición”
 - Si se cumple, el proceso continúa ejecutándose
 - Si NO se cumple, se informa al usuario “Report” y se indica en nivel de gravedad del error
- Los niveles de SEVERITY son:
 - NOTE
 - WARNING
 - ERROR (nivel por defecto si no 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...

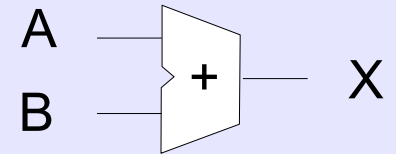


Ejemplo: sumador (I)

```
process
begin
  A <= x"01";
  B <= x"01";
  WAIT FOR 10 ns;
  ASSERT X = x"02" REPORT "Falla" SEVERITY FAILURE;

  A<= x"FF";
  B<= x"01";
  WAIT FOR 10 ns;
  ASSERT X = x"00" REPORT "Falla" SEVERITY FAILURE;

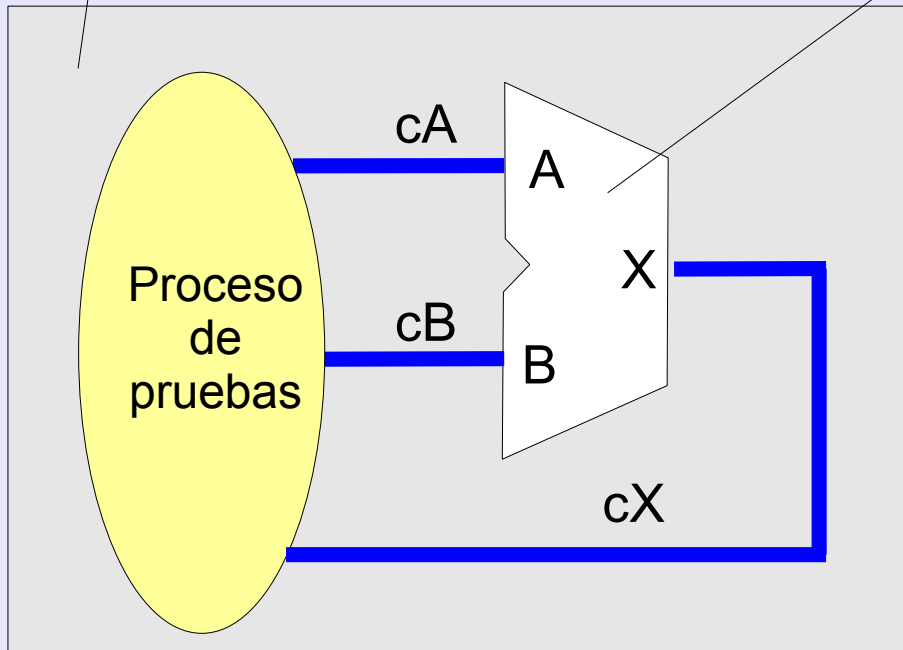
  WAIT;
end process;
```



Ejemplo: sumador (II)

```
ENTITY sumador_tb IS  
END sumador_tb;
```

```
ENTITY sumador IS PORT (  
  A: IN  std_logic_vector(7 downto 0);  
  B: IN  std_logic_vector(7 downto 0);  
  X: OUT std_logic_vector(7 downto 0);  
END sumador;
```



Elementos:

- Entidad top-level: `sumador_tb`
- Componente a probar: `sumador`
- Proceso de pruebas
- Cable cA
- Cable cB
- Cable cX

Ejemplo: sumador (III)

```
ENTITY sumador_tb IS  
END sumador_tb;
```

```
Architecture test of sumador_tb IS
```

```
    Component sumador
```

```
    PORT (
```

```
        A: IN  std_logic_vector(7 downto 0);
```

```
        B: IN  std_logic_vector(7 downto 0);
```

```
        X: OUT std_logic_vector(7 downto 0));
```

```
    Signal cA : std_logic_vector(7 downto 0);
```

```
    Signal cB : std_logic_vector(7 downto 0);
```

```
    Signal cX : std_logic_vector(7 downto 0);
```

```
Begin
```

```
    --Cuerpo: en la página siguiente :-)
```

```
END test;
```

Ejemplo: sumador (IV)

```
begin
```

```
    sumador PORT MAP (A=>cA, B=>cB, X=>cX)
```

```
process
```

```
    begin
```

```
        cA <= x"01";
```

```
        cB <= x"01";
```

```
        WAIT FOR 10 ns;
```

```
        ASSERT cX = x"02" REPORT "Falla" SEVERITY FAILURE;
```

```
        cA<= x"FF";
```

```
        cB<= x"01";
```

```
        WAIT FOR 10 ns;
```

```
        ASSERT cX = x"00" REPORT "Falla" SEVERITY FAILURE;
```

```
        WAIT;
```

```
    end process;
```

```
end test;
```

Ejemplo: sumador (V)

- Es mejor hacer una prueba sistemática. Probamos todos los casos posibles:

```
process
```

```
begin
```

```
  for i in 1 to 255 loop  
    for j in 1 to 255 loop
```

```
      cA <= std_logic_vector(to_unsigned(i,8));  
      cB <= std_logic_vector(to_unsigned(j,8));  
      WAIT FOR 10 ns;
```

```
      ASSERT cX = std_logic_vector(to_unsigned(i+j,8))  
        REPORT "Falla" SEVERITY FAILURE;
```

```
    end loop;  
  end loop;
```

```
end process;
```


Vamos a ver algunos ejemplos

Lenguaje de Descripción Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

Estilos de diseño

Verificación con testbenches