

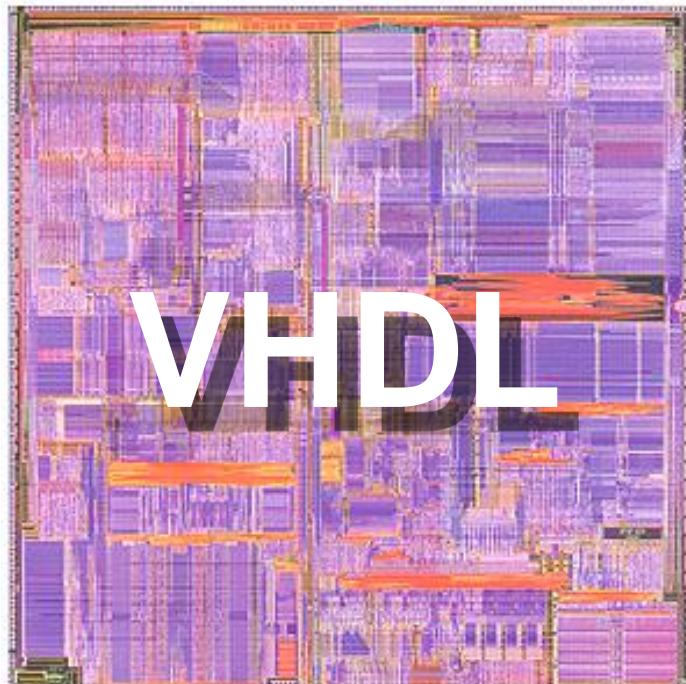
# **Diseño de Circuitos y Sistemas Electrónicos**

## **Estructura y Diseño de Circuitos Digitales**

### **Introducción al Lenguaje de Descripción de Hardware VHDL**



# Lenguaje de Descripción de Hardware VHDL



## Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

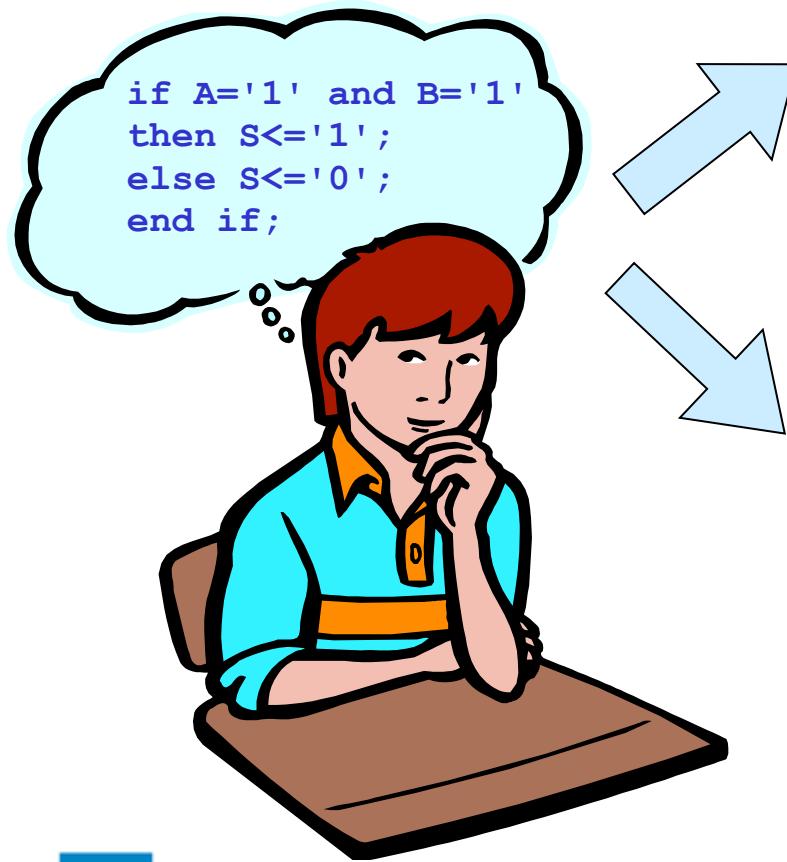
Estilos de diseño

Verificación con testbenches

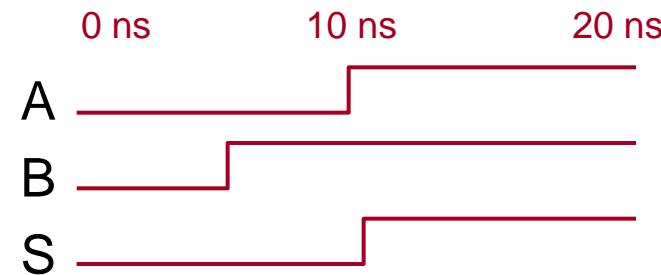


# ¿Para qué sirven los HDLs?

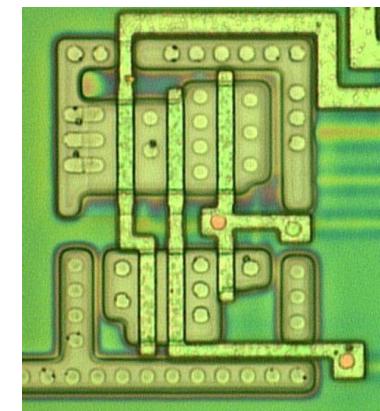
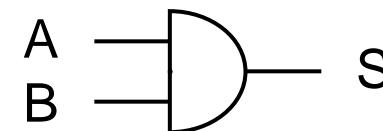
Los lenguajes de descripción HW sirven para escribir modelos de circuitos



Los modelos se pueden **simular** para comprobar que se corresponden con la funcionalidad deseada

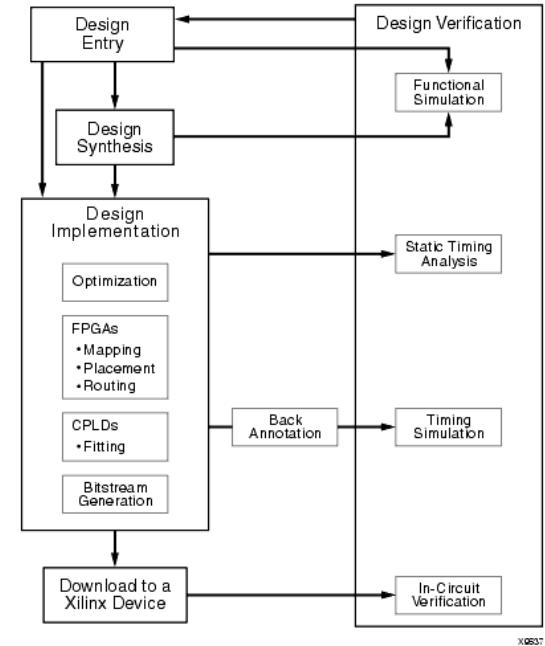


O se pueden **sintetizar** para crear un circuito que funciona como el modelo



# Uso de los HDLs

- Resumen
  - Modelado de un circuito
    - Simulación
    - Síntesis
  - Verificación del resultado de la síntesis, etc.
    - Simulación post-síntesis
    - Simulación post-layout
  - Documentación
    - Contratación
  - Verificación Funcional
    - Desarrollo de un testbench
- Ventajas adicionales
  - Lenguaje estándar
  - Independiente de la tecnología y del fabricante
    - Portable, reusable
    - Útil para desarrollo con FPGAs y con ASICs

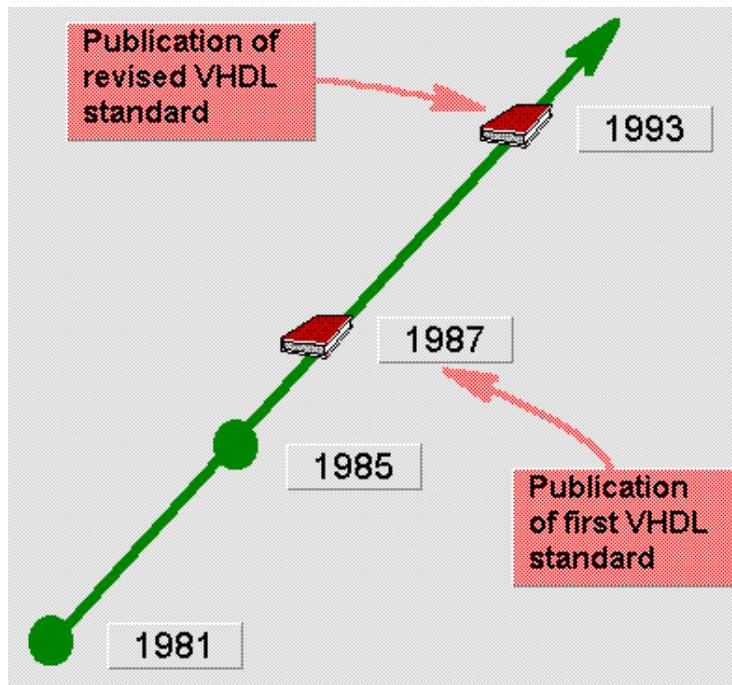


# VHDL: orígenes e historia

- 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.



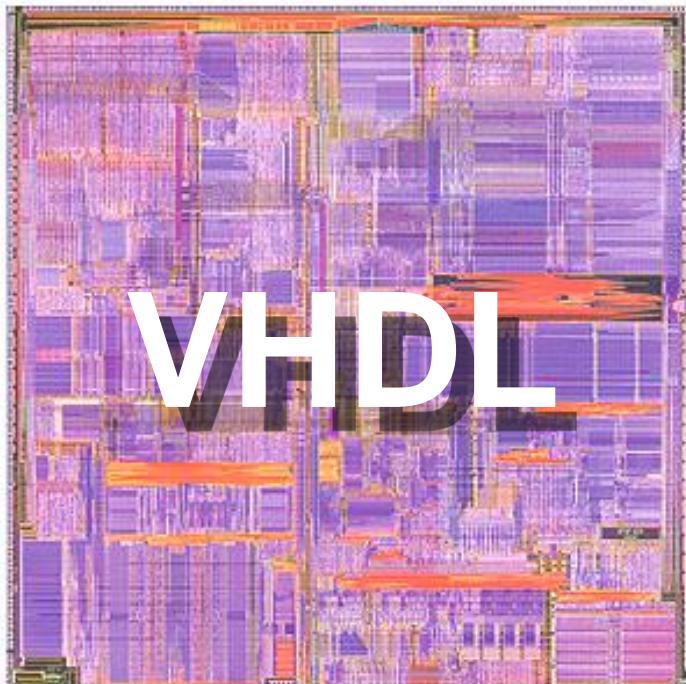
# VHDL: Evolución



- **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)
- **2000,2002**: Varias mejoras estandarizadas .
- **2008**: Última modificación de VHDL (VHDL-2008). Incorporación de sub-estándares. Mejor soporte de verificación y otras mejoras.



# Lenguaje de Descripción de Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

Diseño jerárquico

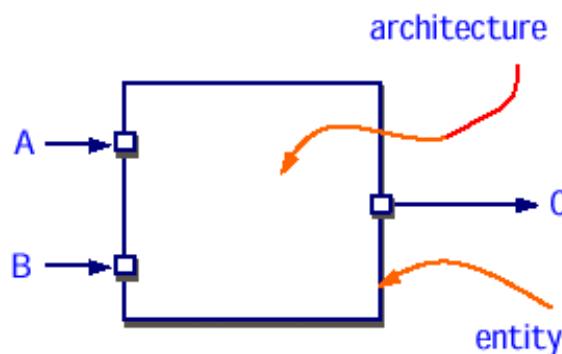
Estilos de diseño

Verificación con testbenches

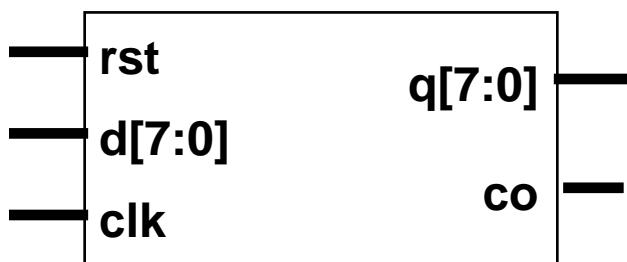


# Entidad y Arquitectura: 1<sup>er</sup> nivel de abstracción

## Abstracción: caja negra



## Interfaz: entradas y salidas



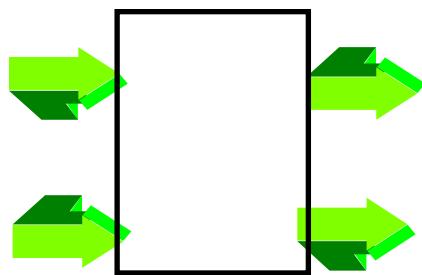
## Entidad y arquitectura

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

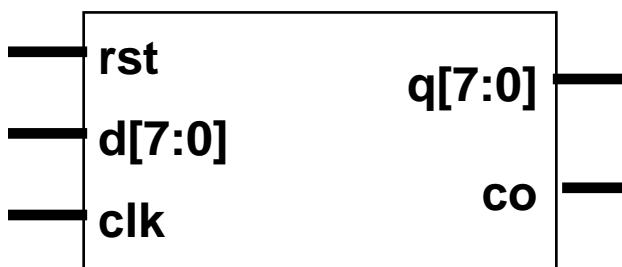


# PORTS: Puertos de una entidad

## Interfaz de dispositivo



## Ports: entradas y salidas



## Ports = Canales de Comunicación

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

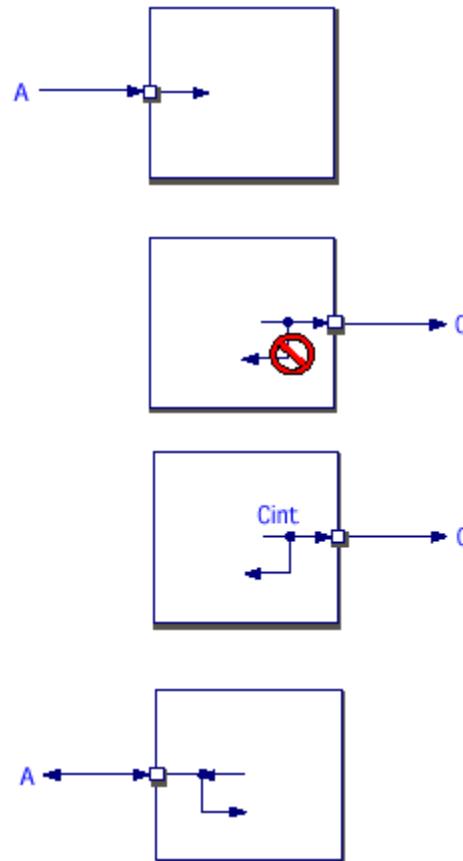
- Un **nombre**, que debe ser único dentro de la entidad.
- Modo: La dirección del flujo de datos: entrada, salida, bidireccional...
- Tipo: Los valores que puede tomar el puerto: '0', '1' o ('Z'), etc. dependen de lo que se denomina TIPO de señal.

Los puertos son una clase especial de señales que adicionalmente al tipo de señal añade el modo (IN, OUT, etc.)



# 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 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.
- **BUFFER** Una señal que sale de la entidad y también es realimentada dentro de la entidad. Post-layout se convierte en OUT, no se recomienda.
- **INOUT** Una señal que es bidireccional, entrada/salida de la entidad.



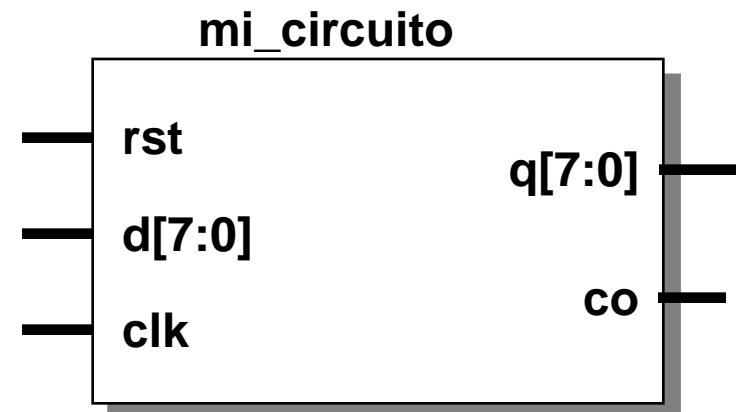
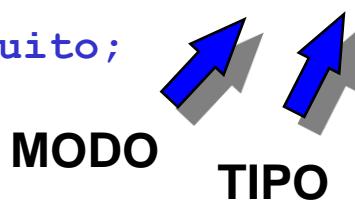
# VHDL: Declaración de entidad

La declaración VHDL de la caja negra:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY mi_circuito 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_circuito;
```



Este estándar permite el uso de una lógica multi-valorada.



# Estructura de un diseño VHDL

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
ENTITY toto IS  
PORT (  
);  
END toto;  
  
ARCHITECTURE test OF toto IS  
BEGIN  
END test;
```

declaraciones de puertos

parte declarativa de la arquitectura

cuerpo de la arquitectura

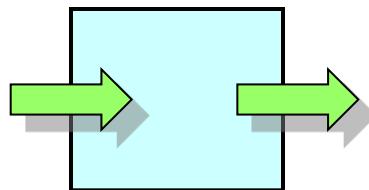
nombre de la entidad

nombre de la arquitectura

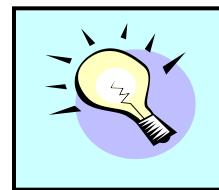


# Resumen: Entidad y Arquitecturas

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

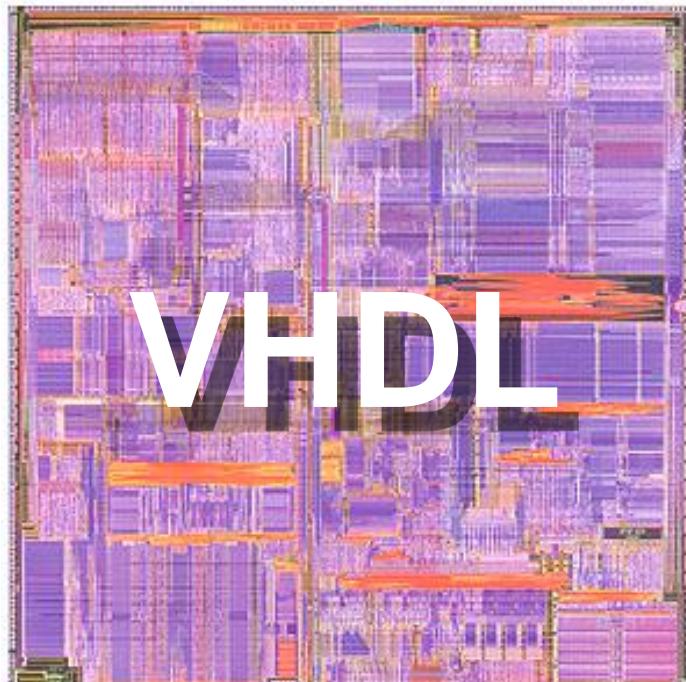


- Los contenidos del circuito se modelan dentro de la arquitectura



- Una entidad puede tener varias arquitecturas
  - Por ejemplo, la descripción de comportamiento que ha hecho el diseñador y el modelo post-layout obtenido después de implementar el chip

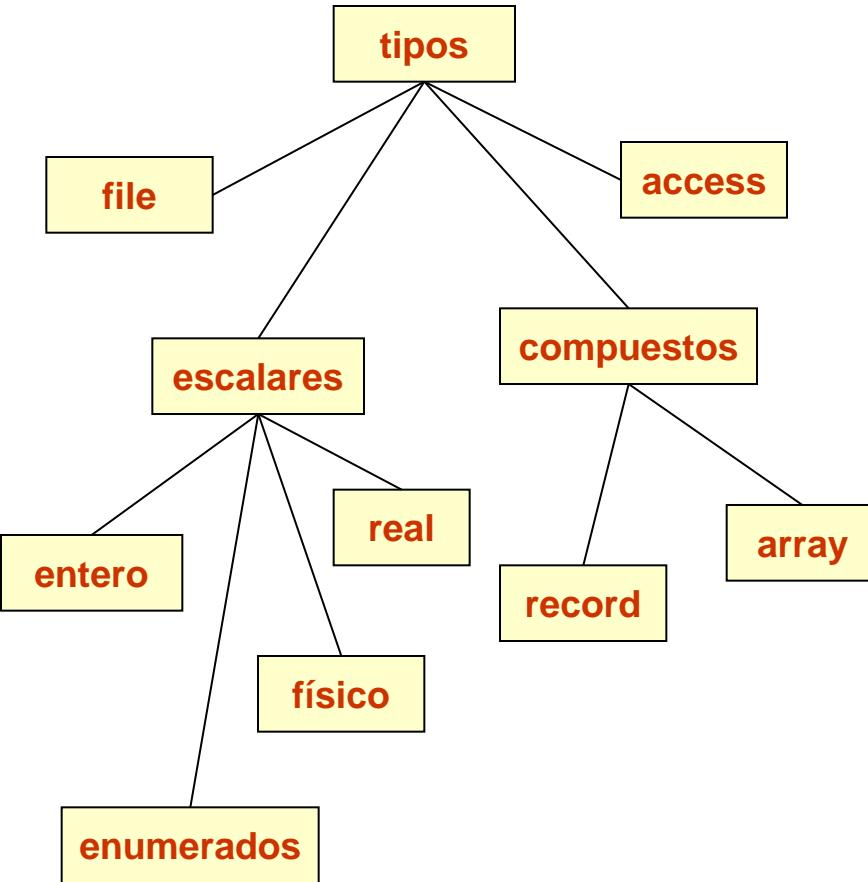
# Lenguaje de Descripción de Hardware VHDL



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
- VHDL es un lenguaje fuertemente tipado:
  - A los objetos se les asigna siempre un tipo cuando se declaran
  - 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

## Tipos IEEE-1076

- **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 índice en bucles, para constantes, valores genéricos...
- **BOOLEAN**: tipo lógico
  - Puede tomar como valores 'TRUE' o 'FALSE'
- **REAL**: tipo para números en coma flotante
- **ENUMERATED**:
  - Conjunto de valores enumerados definido por el usuario
  - Por ejemplo: **TYPE estados IS (inicio, lento, rápido)**



# 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, 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 de 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 (2)

- 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: '0', '1', 'Z', 'L', 'H' y '-' se reconocen para la síntesis.
- En el paquete IEEE.std\_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 tiene dos fuentes que le asignan valores distintos
  - Por ejemplo, si una fuente asigna un '1' y la otra una 'L', la función de resolución dice que la señal se queda a '1'



# Utilizando los tipos: señales en VHDL

- El objeto distintivo en VHDL es la señal, que se utiliza para modelar los hilos del circuito
- Puesto que modela nodos físicos, incluye información de tiempo
  - No sólo contiene unos valores ('0', '1', 'Z', etc...) sino también el tiempo en el que se toman estos valores
- Se declaran antes del begin de la arquitectura (en la parte declarativa):

```
ARCHITECTURE uam OF prueba IS
    SIGNAL s1 : STD_LOGIC;
    SIGNAL s2 : INTEGER;
BEGIN
```

- Pueden tener un valor inicial (no soportado en síntesis)

```
SIGNAL a : STD_LOGIC := '0';
```

- Para asignar valores a una señal se utiliza      <=



# Utilizando los tipos: constantes y variables

- Como en cualquier otro lenguaje, en VHDL se pueden utilizar constantes.
- Se declaran también en la parte declarativa, antes del **begin**

```
ARCHITECTURE uam OF prueba IS
    CONSTANT c1 : STD_LOGIC := '0';
    CONSTANT c2 : TIME := 10 ns;
    CONSTANT c3 : INTEGER := 5;
BEGIN
```

- Las constantes pueden ser de cualquier tipo.



# Utilizando los tipos: constantes y variables

- El tercer objeto posible en VHDL son las variables.
- Sólo almacenan valores, no entienden de tiempo.
- Visibilidad limitada, sólo dentro de un proceso y no en toda la arquitectura (a diferencia de las señales y constantes)

```
ARCHITECTURE uam OF prueba IS
  ...
BEGIN
  PROCESS
    VARIABLE r: STD_LOGIC;
  BEGIN
    ...
  END PROCESS;
```

- Se les asignan valores empleando    :=



# Usando arrays para crear buses

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

```
SIGNAL a: STD_LOGIC_VECTOR(0 TO 3);      -- i.e. rango ascendente  
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0); -- i.e. 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 y eficiente de asignar valores a vectores son los *aggregates*:

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



# Asignación de señales en buses

## Flexibilidad en la asignación de valores de 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(7 downto 4) <= "1010";`
- Notación:
  - 1 bit : comilla simple ('), STD\_LOGIC
  - múltiples bits: comilla doble ("), STD\_LOGIC\_VECTOR



# Cómo definir nuevos tipos (y usarlos)

- VHDL permite definir nuevos tipos, bien a partir de tipos enumerados, o como subconjunto de tipos ya existentes, o tipos multidimensionales.
- Las definiciones de tipos se hacen frecuentemente en la parte declarativa de la arquitectura.
- Definir un tipo como una enumeración:

```
TYPE estados IS (inactivo, operando, finalizar);  
SIGNAL mi_maquina : estados;
```

- Definir un tipo bidimensional:

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



# Operadores definidos en VHDL

- Lógicos
  - and, nand
  - or, nor
  - xor, xnor
  - not (unario)
- Relacionales
  - = igual
  - /= distinto
  - < menor
  - <= menor o igual
  - > mayor
  - >= mayor o igual
- Misceláneos
  - abs valor absoluto
  - \*\* exponenciación
- Adición
  - + suma
  - resta
  - & concatenación de vectores
- Multiplicativos
  - \* multiplicación
  - / división
  - rem resto
  - mod módulo
- Signo (unarios)
  - +,-
- Desplazamiento (bit\_vector)
  - sll, srl
  - sla, sra
  - rol, ror



# Más sobre operadores

- No todos los operadores están definidos para todos los tipos
- En particular, para los std\_logic habrá que obtenerlos de alguna librería como:
  - std\_logic\_signed
  - std\_logic\_unsigned
  - std\_logic\_arith
- 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"
```

- Los operadores de desplazamiento básicos sólo funcionan con bit\_vector. Es práctico usar concatenación con std\_logic.

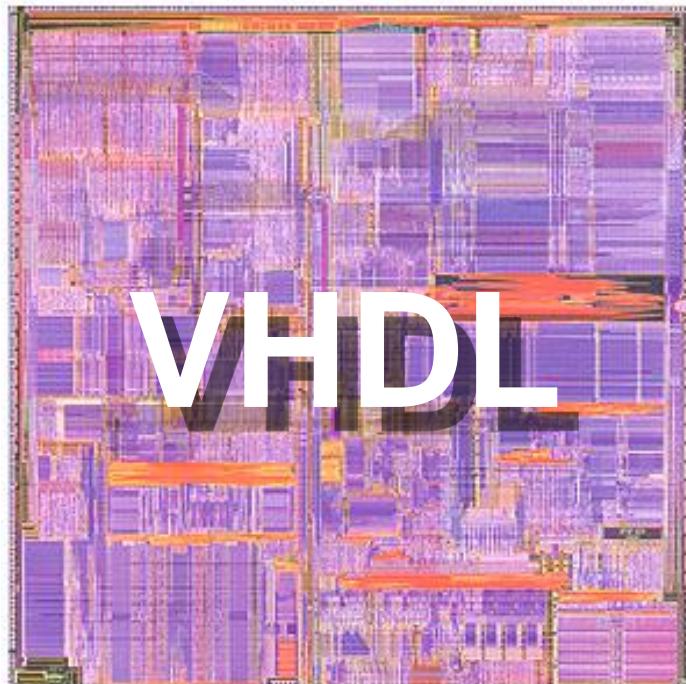


# Acerca de las librerías en VHDL

- Librerías clásicas (no estándares):
  - std\_logic\_signed
  - std\_logic\_unsigned
  - std\_logic\_arith
- Las librerías **signed** y **unsigned** se deben emplear cuando se quiere que los std\_logic\_vector estén respectivamente en complemento a 2 o en binario natural
  - Aquí está el CONV\_INTEGER
- La librería **arith** es más completa, y utiliza mayormente los tipos signed o unsigned (derivados de std\_logic\_vector)
  - Aquí está CONV\_STD\_LOGIC\_VECTOR
- Librería estándar de IEEE: **numeric\_std**
  - Pensada para trabajar con los tipos signed y unsigned
  - TO\_INTEGER, TO\_SIGNED, TO\_UNSIGNED



# Lenguaje de Descripción de 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 UAM of EJEMPLO is
```

```
-- Parte declarativa: aquí se definen:  
-- Tipos, subtipos, señales que vamos a  
-- usar, etc.
```

```
begin
```

```
-- Construcciones Concurrentes  
    -- PROCESOS  
    -- asignaciones  
    -- instanciaciones  
-- El orden en el texto no importa  
-- ya que se ejecuta concurrentemente
```

```
end UAM;
```



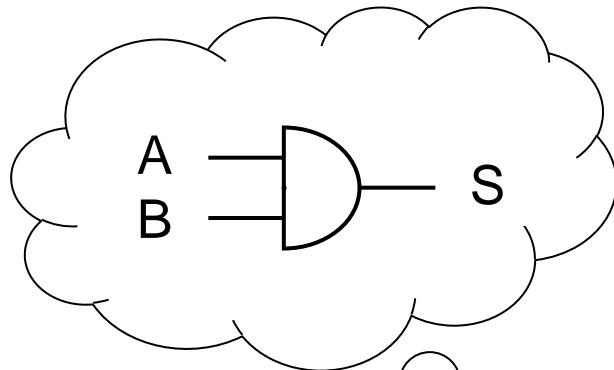
# El proceso: el elemento de diseño principal

- Un proceso describe el comportamiento de un circuito
  - Cuyo estado puede variar cuando cambian ciertas señales
  - Sentencias secuenciales: *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

Este proceso no declara nada

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

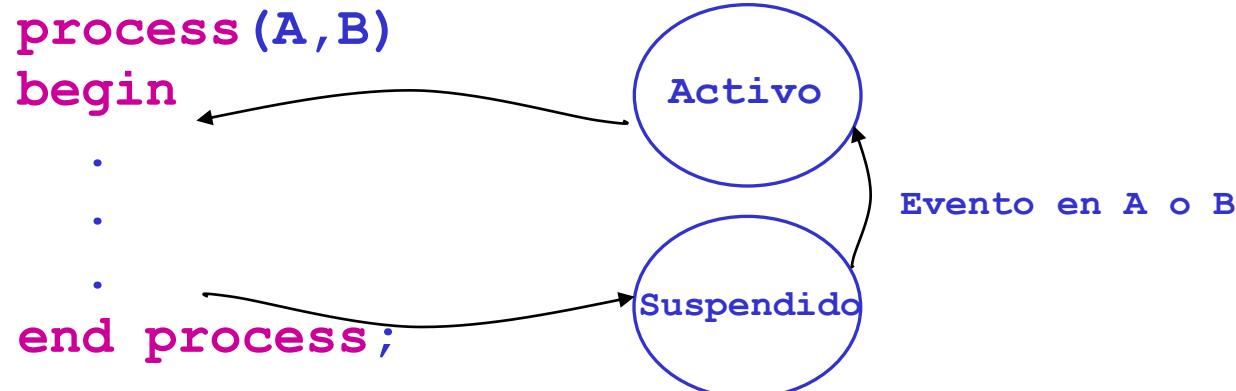
Se usa un if..then..else para describir la puerta



# Funcionamiento de los procesos

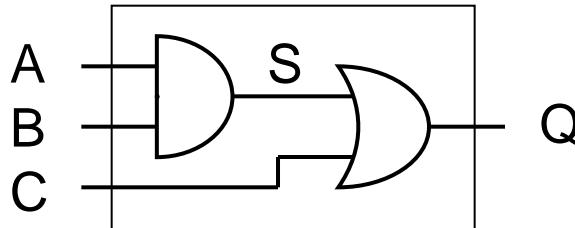
- En simulación:
  - Las sentencias en el cuerpo de un proceso se ejecutan secuencialmente.
  - La ejecución de un proceso se hace sin que avance el tiempo simulado.
  - Un proceso puede estar activo o suspendido.

Ejemplo:



# Necesidad de la concurrencia

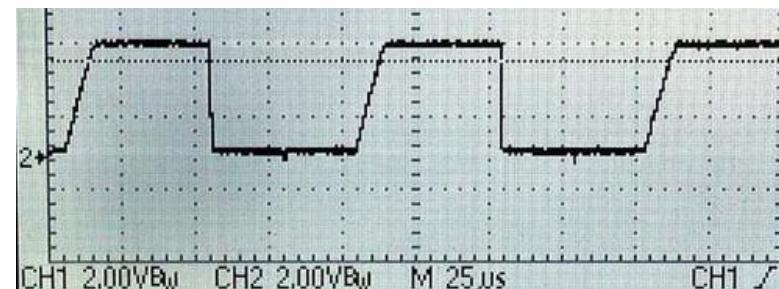
- Cuando hay dependencias entre señales:



```
process (A, B, C)
begin
  ...
  S <= A and B;
  Q <= S or C;
  ...
end process;
```

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

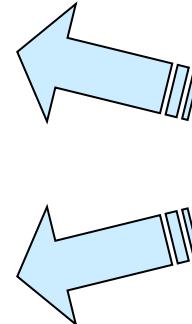
- ¿Por qué? No hay que olvidar que se trata de modelar circuitos reales, no virtuales, y las señales necesitan que transcurra el tiempo para tomar un valor:



# La solución de VHDL

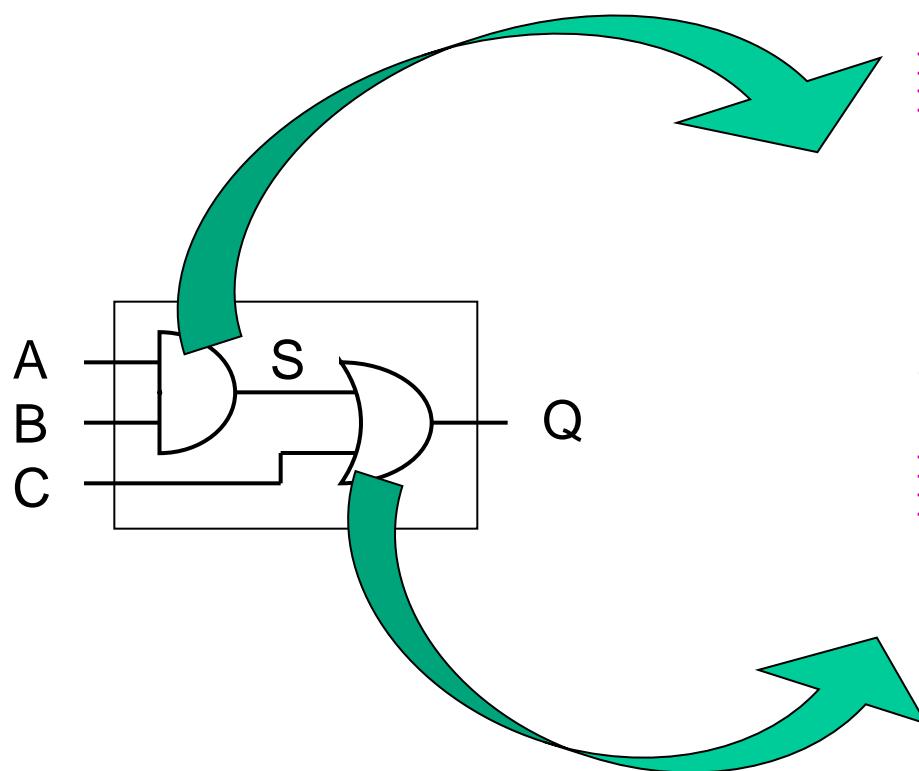
- VHDL (y en general, todos los HDLs) solucionan este problema dando soporte explícito a la concurrencia
- En VHDL, una arquitectura puede tener tantos procesos como queramos, **y todos se ejecutan concurrentemente**

```
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;
```



# Ejecución de Procesos

- Inicialización
  - $T=0$
  - Inicialización de drivers de todas las señales
  - Ejecución del cuerpo de todos los procesos
    - Normalmente se provocará cambios en señales
    - Se suspenden todos los procesos
- Simulación
  - Si hay transacciones previstas para algunos *drivers*
    - Avanzar  $T$
    - Actualizar señales
    - Reactivar los procesos sensibles a las señales que cambiaron
    - Ejecutar el cuerpo de esos procesos
      - Normalmente se provocará más cambios en señales
    - Se suspenden los procesos activados en este ciclo de simulación



# 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 3
    ...
ELSE
    ...
    instrucciones por defecto
    ...
END IF;

```

## Ejemplo: Una indicación de nivel

```

architecture example of level_ind is
begin

    ctrl: process (level) is
    begin
        if level > 60 then
            ind <= "11";
        elsif level > 40 then
            ind <= "10";
        elsif level > 20 then
            ind <= "01";
        else -- level menor o igual que 20
            ind <= "00";
        end if;
    end process ctrl;

end example;

```



# Instrucciones en procesos: CASE

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

## Ejemplo: Una ALU sencilla

```
architecture uam of alu is
begin

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

end 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)
  begin
    for i in 0 to 7 loop
      if i = CONV_INTEGER(a) then
        q(i) <= '1';
      else
        q(i) <= '0';
      end if;
    end loop;
  end process decod;

end 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:  
Control de simulación

```
while (simulando = true) loop  
    clk <= '0';  
    wait for 10 ns;  
    clk <= '1';  
    wait for 10 ns;  
end loop;  
wait;
```



# 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

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

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



# Distintas cláusulas wait

- Esperar a que ocurra una condición:

```
wait until a='1' and b='0';
```

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

```
wait on a, b, clk;
```

- Esperar un cierto tiempo:

```
wait for 100 ns;
```

- Esperar indefinidamente (matar el proceso):

```
wait;
```



# 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 suspenden todos los procesos activos.



# Las variables

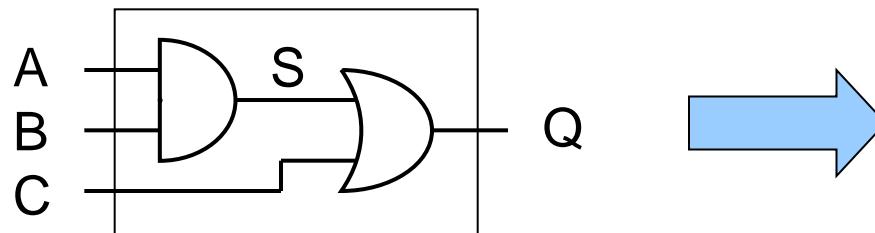
- A la hora de modelar un circuito nos puede venir bien tener un objeto cuyo valor se actualice inmediatamente
  - sin tener que esperar a que avance el tiempo, como en las señales
- La solución son las variables
  - Las variables se declaran dentro de los procesos
  - Sólo se ven dentro del proceso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

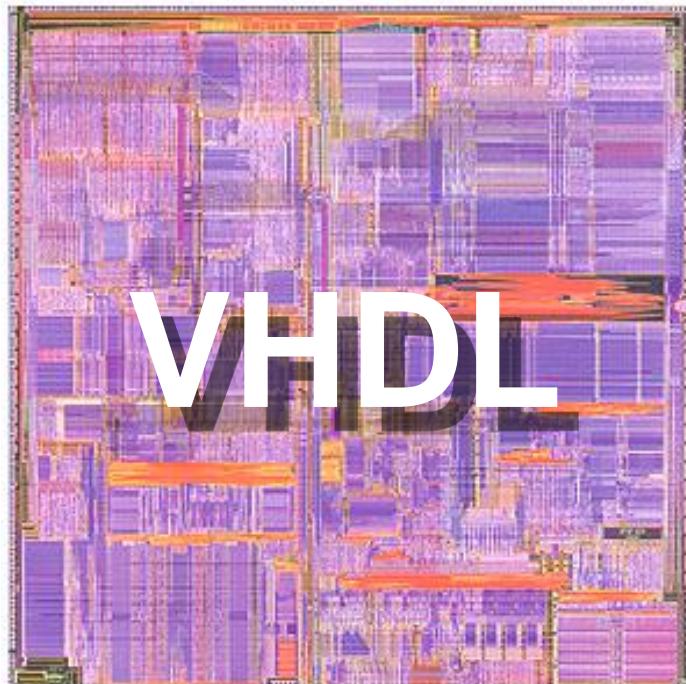


# Semántica de variables y señales

	Señales	Variables
Sintaxis	destino <= 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 de 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



# Síntesis: El problema de la memoria implícita

Las señales en VHDL tienen un estado actual y un estado futuro

En un proceso, si no se define el valor futuro de una señal determinada, se mantiene el valor actual.

=> Se sintetiza un *latch* para mantener su estado actual

## PROBLEMA

- Pueden generarse por descuido *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';
        when others =>
            null;
    end process;

```

- La 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 debe 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.
  - Dando un valor inicial o por defecto. Así siempre se recibe al menos un valor.

```
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 explícitos y asignaciones
- Hay tres tipos
  - Asignaciones simples  
`s <= (a and b) + c;`
  - Asignaciones condicionales  
`s <= a when c='1' else b;`
  - Asignaciones con selección  
`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 asignación 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 utilicemos 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 uam;
```

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



# Asignaciones concurrentes con selección

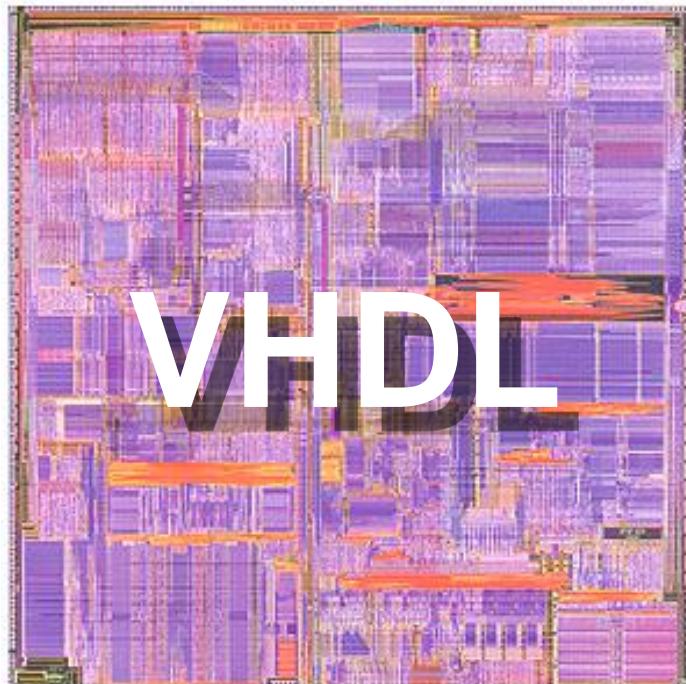
- Se le asigna un valor a una señal dependiendo del valor que tome una expresión:

```
architecture uam of decod is
begin
    with a select
        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 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 de 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)

proceso sensible  
al reloj

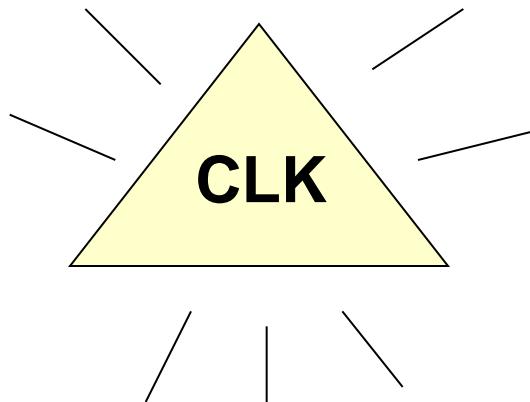
```
process (clk) .  
begin  
    • if clk'event and clk='1' then  
        q <= d;  
    • end if;  
end process;
```

no hay else,  
queremos  
inferir memoria

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



# 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



# Flip-flop con reset asíncrono

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

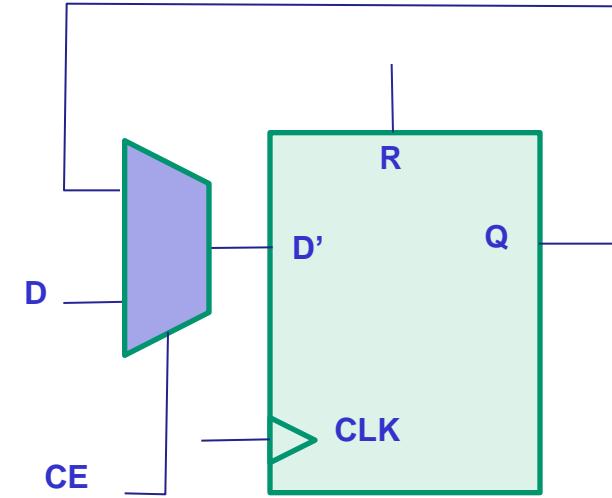


# Flip-flop con reset asíncrono y *clock enable*

```

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;

```



**¡No es más que lógica combinacional añadida!**  
 No despistarse: recordar el elemento básico,  
 el FF D, con o sin reset asíncrono.



# Ejemplo: Un contador de 8 bits

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

como q es una salida,  
no se puede leer... se  
define la señal counter



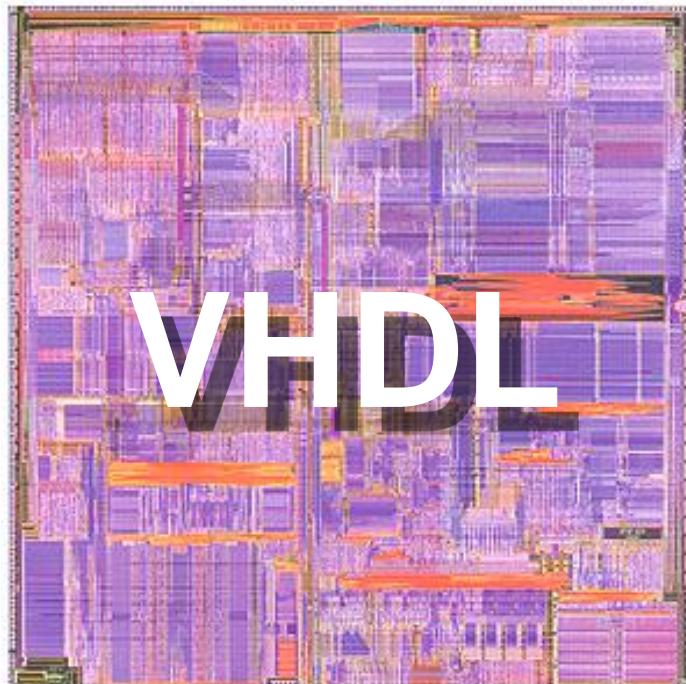
# Ejemplo: Un registro de desplazamiento

```
process(clk, rst_n)
begin
    if rst_n='0' then
        dout <= "00000000";
    elsif rising_edge(clk) then
        if ce='1' then
            if load='1' then
                dout <= din;
            else
                dout <= dout(6 downto 0) & sin;
            end if;
        end if;
        end if;
    end process;
```

importante: uso  
de  
concatenación



# Lenguaje de Descripción de Hardware VHDL



Introducción

La entidad y la arquitectura

Tipos de datos

Los procesos

Circuitos combinacionales

Circuitos secuenciales

Máquinas de estados

Triestados

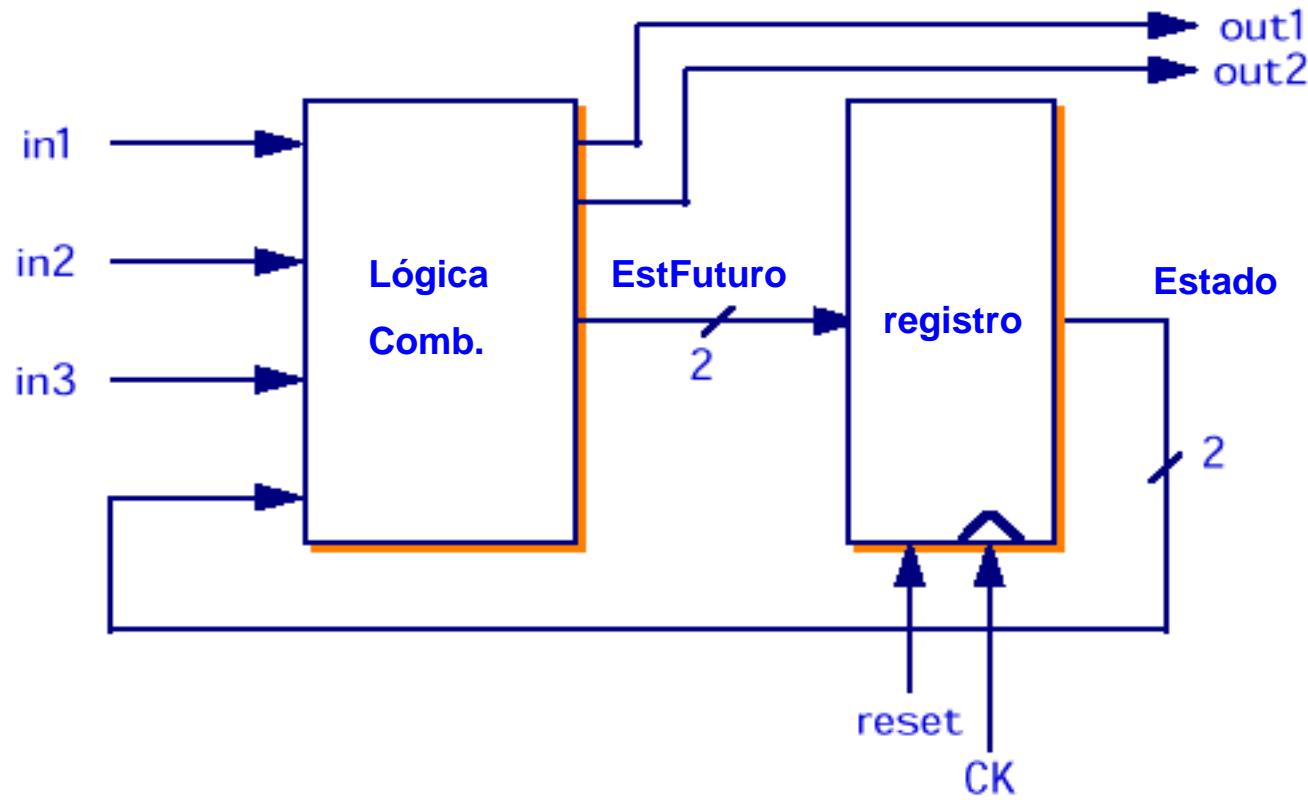
Diseño jerárquico

Estilos de diseño

Verificación con testbenches



# Metodología: Diseño circuitos secuenciales

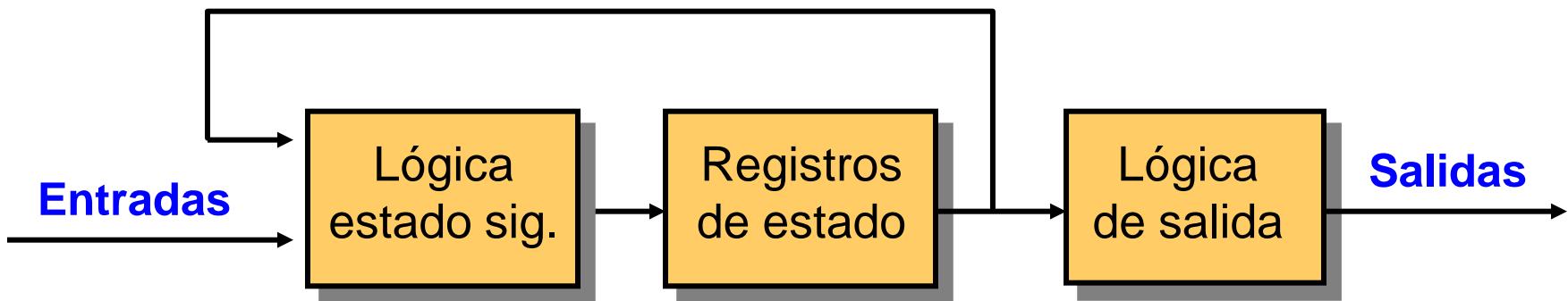


# Implementación de una FSM de Moore (1)

- Salidas decodificadas a partir del valor de los estados.

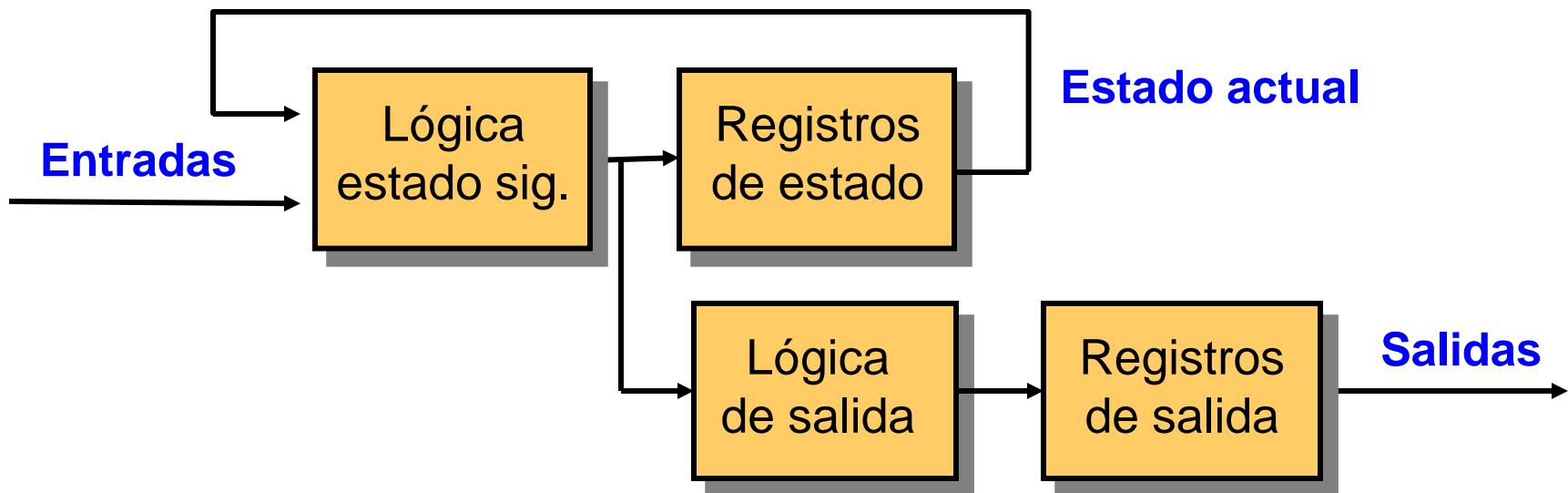
## 1. Decodificación Combinacional

- Las salidas se decodifican a partir del estado actual
- Salidas = función(estado\_actual)



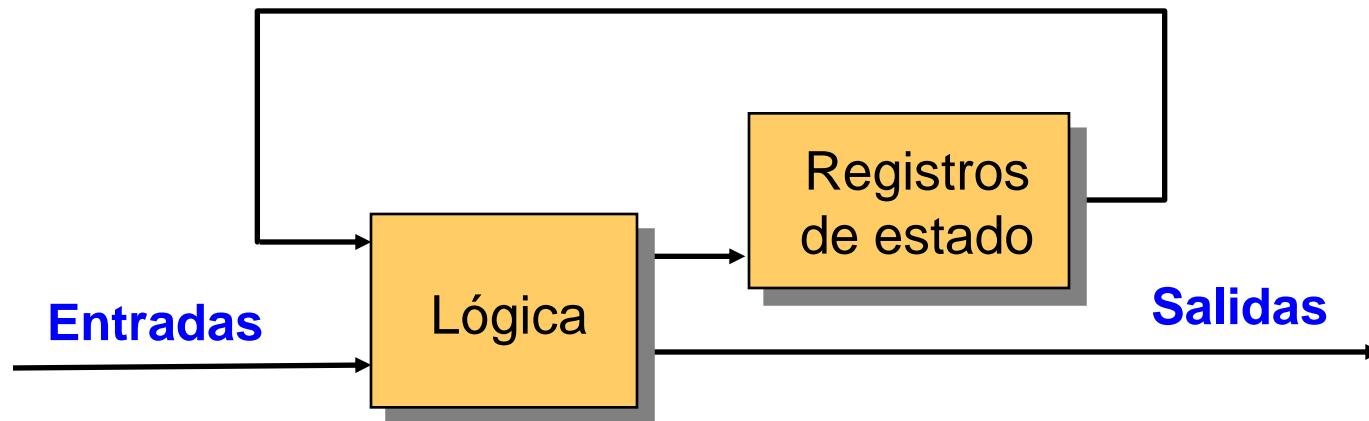
# Implementación de una FSM Moore (2)

- Salidas decodificadas a partir del valor de los estados.
  2. Decodificación con salidas registradas
    - La decodificación de las salidas se realiza en paralelo con la decodificación del siguiente estado.
    - Salidas = función(estado\_anterior, entradas)



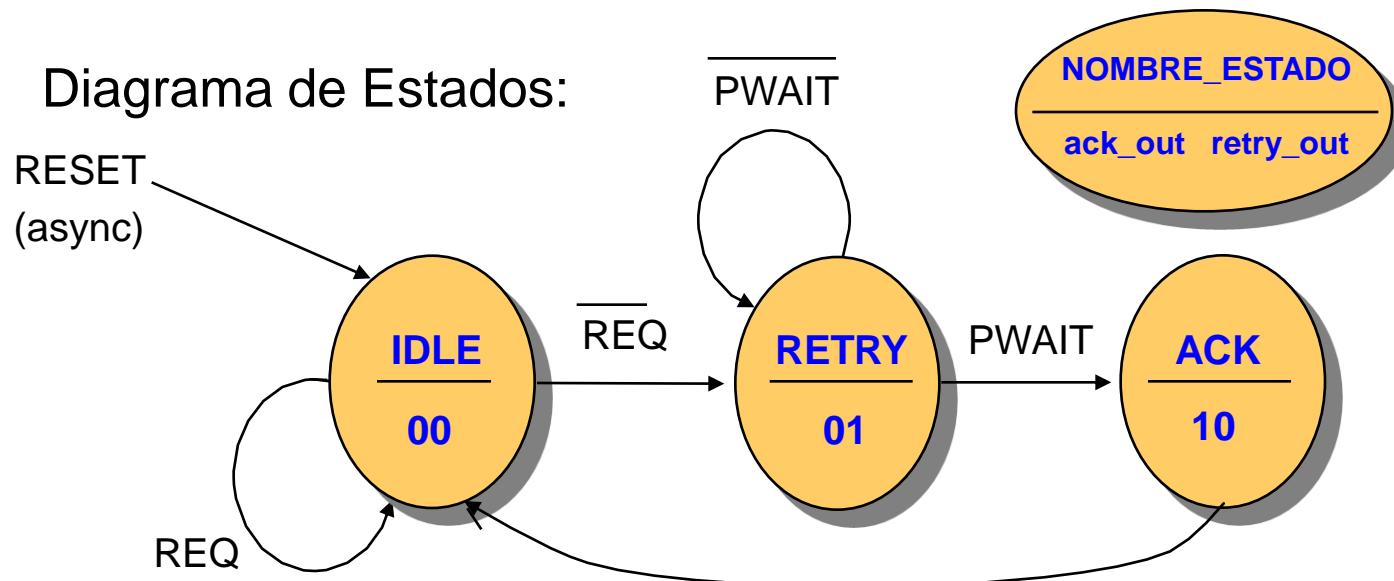
# FSM de Mealy

- Las salidas cambian por un cambio de estado o por un cambio en el valor de las entradas
  - Hay que tener cuidado con los caminos combinacionales de la entrada a la salida: posibles violaciones de timing.



# Ejemplo: Generador de “wait states”

- Diagrama de Estados:



- Declaración de la Entidad:

```

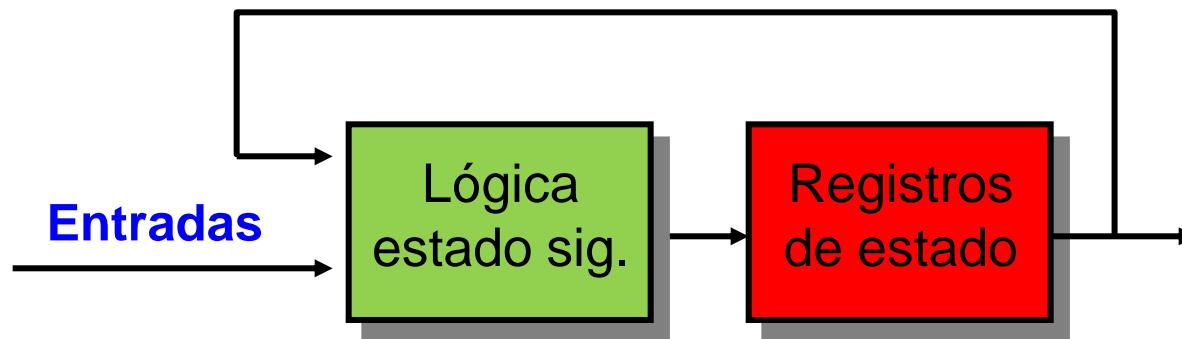
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY maq IS PORT (
    clock, reset: IN std_logic;
    req, pwait: IN std_logic;
    retry_out, ack_out: OUT std_logic);
END maq;
  
```



# Codificación combinacional + secuencial

- Se suele considerar una buena práctica de diseño el separar explícitamente:
  - El proceso combinacional que determina el siguiente estado
  - El proceso secuencial que define el registro de estado



- Esto es independiente del tipo de implementación: Moore/Mealy, salidas registradas/combinacionales, esquema one-hot ...



# Ejemplo: combinacional + secuencial

```

ARCHITECTURE arch_comb_seq OF maq IS
    TYPE fsm_states IS (IDLE, RETRY, ACK);
    SIGNAL state      : fsm_states; -- register
    SIGNAL next_state : fsm_states; -- combinational logic

BEGIN
    fsm_comb: PROCESS (state, req, pwait)
    BEGIN
        next_state <= state; -- default state = current
        CASE state IS
            WHEN IDLE    => IF req = '0' THEN next_state <= RETRY; END IF;
            WHEN RETRY   => IF pwait='1' THEN next_state <= ACK;    END IF;
            WHEN ACK     => next_state <= IDLE;
            WHEN OTHERS  => next_state <= IDLE;
        END CASE;
    END PROCESS fsm_comb;

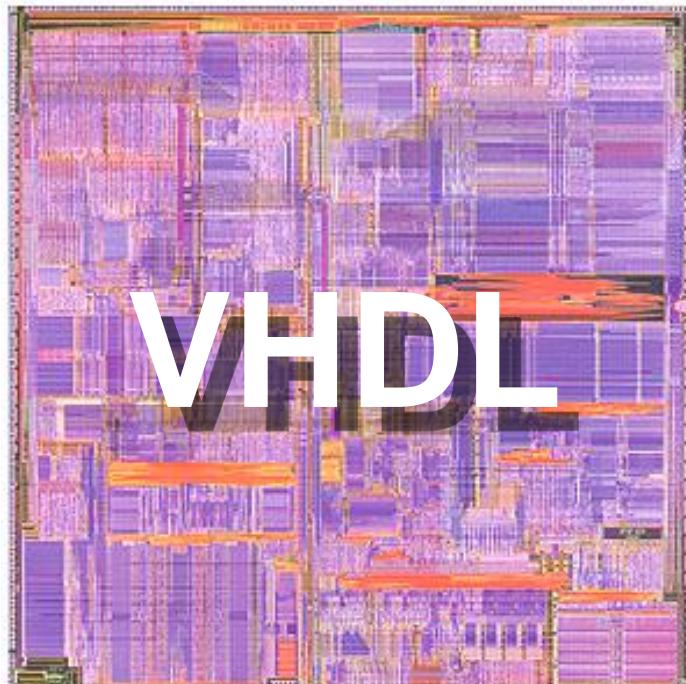
    fsm_seq: PROCESS (clock, reset)
    BEGIN
        IF  reset = '1' THEN
            state <= IDLE;
        ELSIF clock'EVENT AND clock = '1' THEN
            state <= next_state;
        END IF;
    END PROCESS fsm_seq;

    RETRY_out <= '1' WHEN (state = RETRY) ELSE '0';
    ACK_out   <= '1' WHEN (state = ACK)   ELSE '0';
END arch_comb_seq;

```



# Lenguaje de Descripción de 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

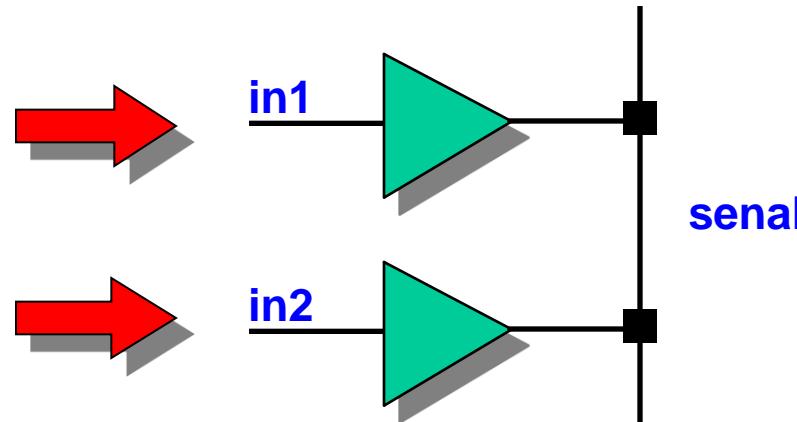


# Concepto de driver de una señal

- El driver es el elemento que da valores a una señal
- Para cada señal a la que se le asigna un valor dentro de un proceso se crea un driver
  - Independientemente de cuántas 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

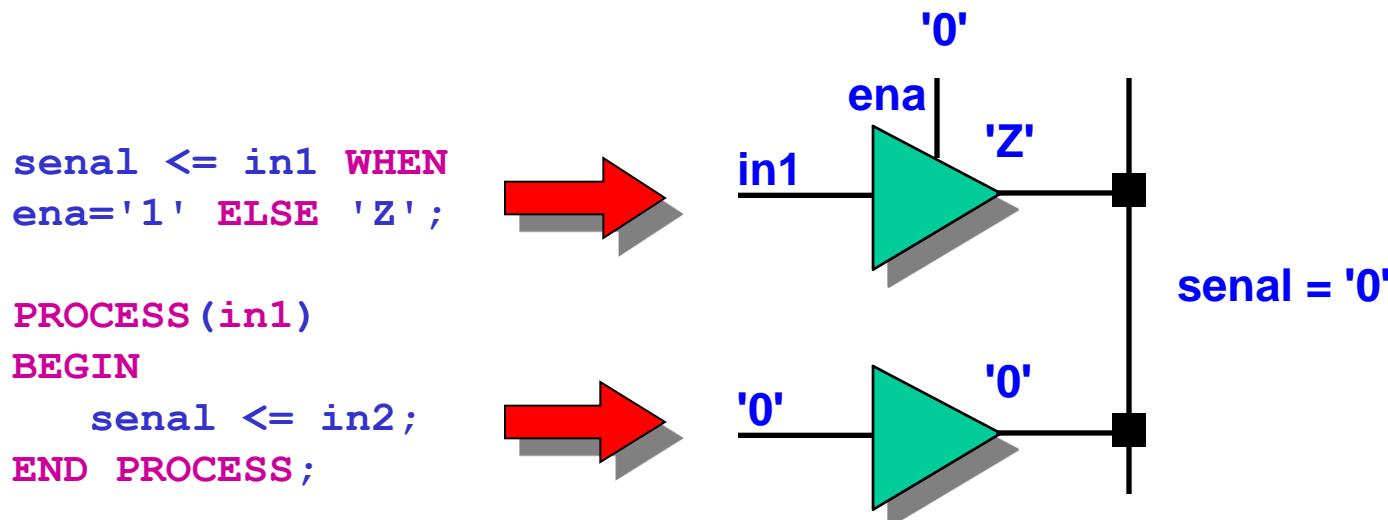
```
PROCESS (in1)
BEGIN
  senal <= in1;
END PROCESS;

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'
  - Vale para el tipo std\_logic y sus derivados
- Igual que ocurre en la realidad, el estado de la señal lo fijará el driver que no esté en alta impedancia



- Lo normal es usarlo exclusivamente en el top de una FPGA



# Inferencia de triestado

La función de resolución usa esta tabla...

Ver PACKAGE std\_logic\_1164

```
CONSTANT resolution_table : stdlogic_table := (
  --
  -- | U X 0 1 Z W L H - | |
  --
  ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
  ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
  ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
  ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);
```



# Ejemplos de inferencia de buffers triestado

- Con asignación condicional:

```
t <= a WHEN ena_a= '0' ELSE 'Z';
```

- Con un proceso explícito:

```
PROCESS (ena_a, a)
BEGIN
    IF (ena_a = '0') THEN
        t <= a;
    ELSE t <= 'Z';
END PROCESS;
```



# Señales bidireccionales

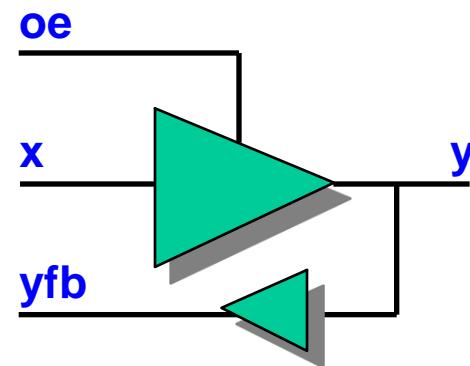
- En este caso la señal tiene drivers externos, fuera de la entidad

```

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

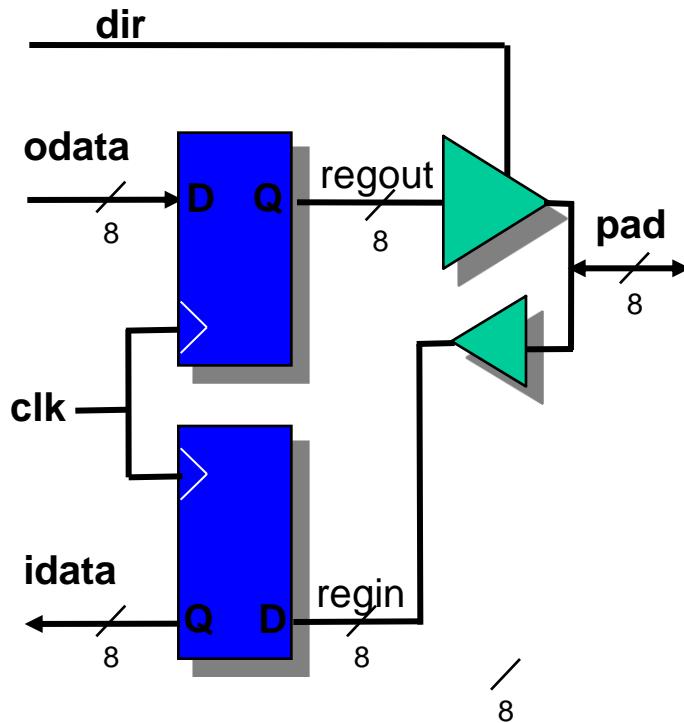
ARCHITECTURE simple OF bufoe IS
BEGIN
    y <= x WHEN oe='1' ELSE 'Z';
    yfb <= y;
END simple;

```



# Ejemplo con señales bidireccionales

- Puerto I/O de propósito general (simplificado)



```

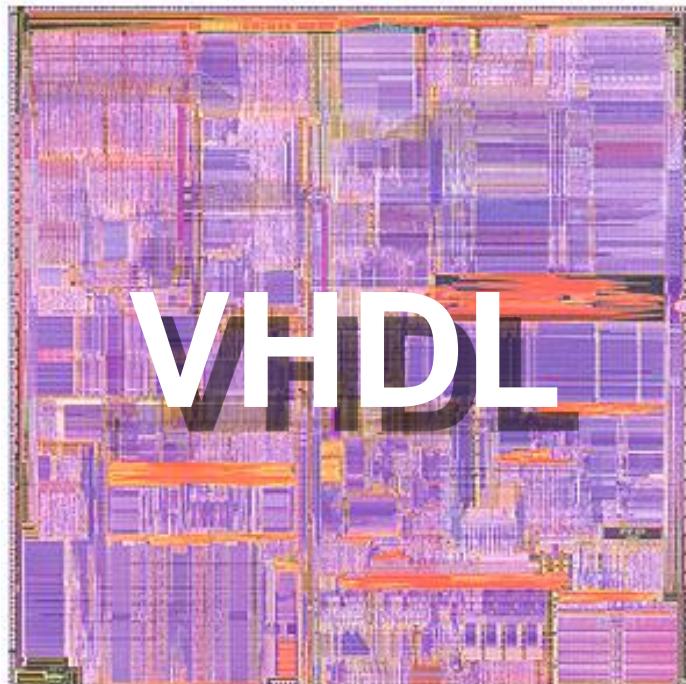
ENTITY ejbidir IS PORT (
    clk,dir : IN std_logic;
    odata   : IN std_logic_vector(7 downto 0);
    idata   : OUT std_logic_vector(7 downto 0);
    pad     : INOUT std_logic_vector(7 downto 0));
END ejbidir;

ARCHITECTURE simple OF ejbidir IS
    SIGNAL regin,regout: std_logic_vector(7 downto 0);
BEGIN
    pad <= regout WHEN dir='1' ELSE "ZZZZZZZZ";
    regin <= pad;
    PROCESS(clk)
    BEGIN
        IF rising_edge(clk) THEN
            regout <= odata;
            idata  <= regin;
        END IF;
    END PROCESS;
END simple;

```



# Lenguaje de Descripción de Hardware VHDL

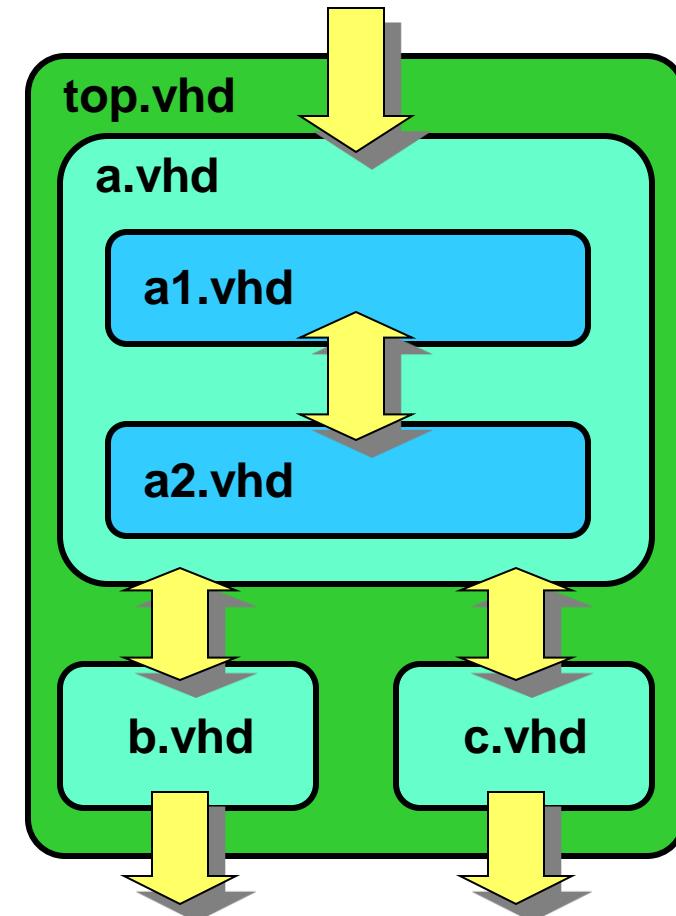


- Introducción
- La entidad y la arquitectura
- Tipos de datos
- Los procesos
- Circuitos combinacionales
- Circuitos secuenciales
- Máquinas de estados
- Triestados
- Diseño jerárquico
- Estilos de diseño
- Verificación con testbenches

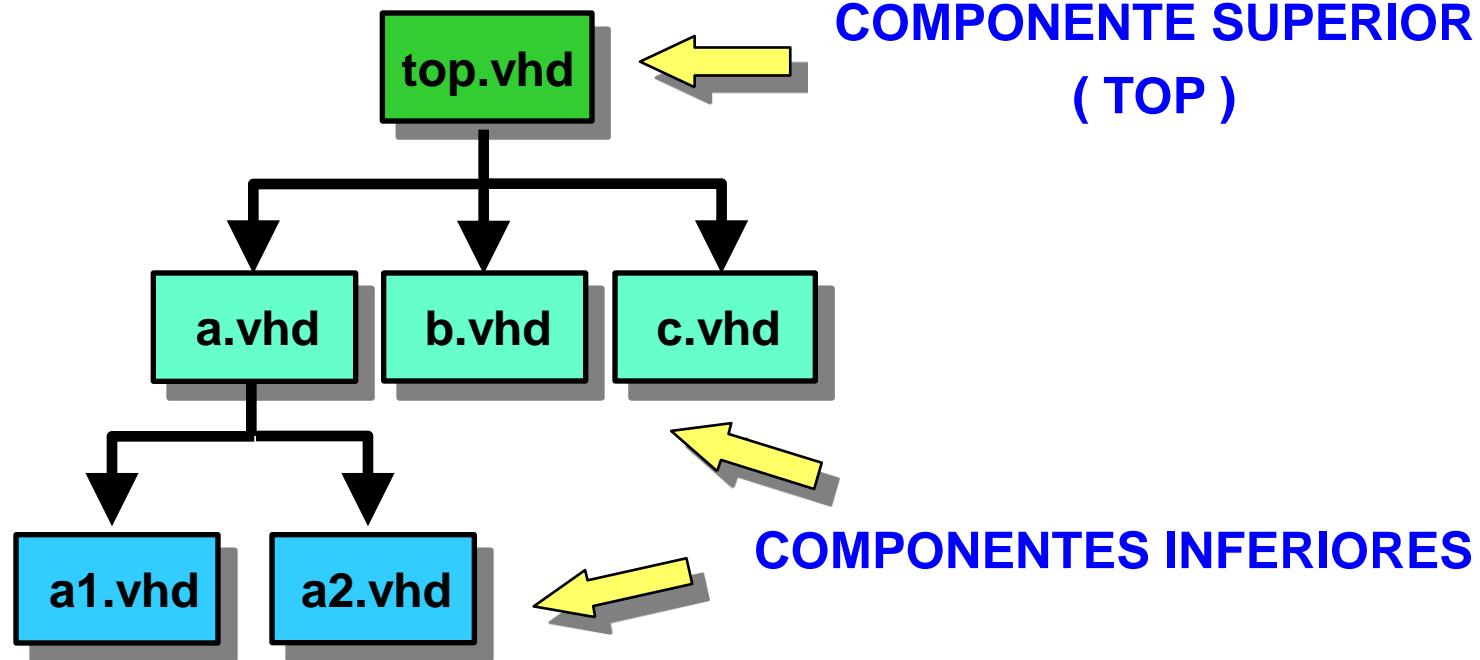


# Diseño jerárquico

- 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*.



# Árbol de jerarquías



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



# 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;
```



# Declaración 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 **también** se puede hacer en un package
  - Para declarar el componente, sólo habrá que importar el package.
  - Opción interesante cuando se declaran varios componentes a modo de “biblioteca” y la declaración de los componentes no aporta nada al lector del código.



# Ejemplo de diseño jerárquico (I):

## Descripción del componente de nivel inferior

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux2to1 IS PORT (
    a, b, sel: IN std_logic;
    c: OUT std_logic);
END mux2to1;

ARCHITECTURE rtl OF mux2to1 IS
BEGIN
    c <= (a AND NOT sel) OR
          (b AND sel);
END rtl;

```

Declaración componente.  
Mismos puertos



```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux4to1 IS PORT (
    i0, i1, i2, i3: IN std_logic;
    sel: IN std_logic_vector (1 downto 0);
    z: OUT std_logic);
END mux4to1;

ARCHITECTURE structural OF mux4to1 IS

COMPONENT mux2to1 PORT (
    a, b, sel: IN std_logic;
    c: OUT std_logic);
END COMPONENT;

SIGNAL c1,c2 : std_logic;

BEGIN
    m0: mux2to1 PORT MAP
        (a=>i0, b=>i1, sel=>sel(0), c=>c1);
    m1: mux2to1 PORT MAP
        (a=>i2, b=>i3, sel=>sel(0), c=>c2);
    m2: mux2to1 PORT MAP
        (a=>c1, b=>c2, sel=>sel(1), c=>z);
END structural;

```

Instanciación



# Ejemplo de diseño jerárquico (II): componente inferior

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

ENTITY mux2to1 IS PORT (
    a, b, sel: IN std_logic;
    c: OUT std_logic);
END mux2to1;
```

Descripción del componente  
de nivel inferior

```
ARCHITECTURE archmux2to1 OF mux2to1 IS
BEGIN
    c <= (a AND NOT sel) OR (b AND sel);
END archmux2to1;
```

---

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

PACKAGE mymuxpkg IS
COMPONENT mux2to1 PORT (
    a, b, sel: IN std_logic;
    c: OUT std_logic);
END COMPONENT;
END mymuxpkg;
```

Creación del package

Declaración del componente  
Mismos puertos



# Ejemplo de diseño jerárquico (II): top-level

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY toplevel IS PORT (s: IN std_logic;
    p, q, r: IN std_logic_vector(2 DOWNTO 0);
    t:         OUT std_logic_vector(2 DOWNTO 0));
END toplevel;

```

Declaración del componente en un package

```

USE WORK.mymuxpkg.ALL;

```



```

ARCHITECTURE archtoplevel OF toplevel IS
    SIGNAL i: std_logic_vector(2 DOWNTO 0);
BEGIN
    m0: mux2to1 PORT MAP (a=>i(2), b=>r(0), sel=>s, c=>t(0));
    m1: mux2to1 PORT MAP (c=>t(1), b=>r(1), a=>i(1), sel=>s);
    m2: mux2to1 PORT MAP (i(0), r(2), s, t(2));
    i <= p AND NOT q;
END archtoplevel;

```

Asociación por nombre



Asociación posicional  
(menos recomendable)

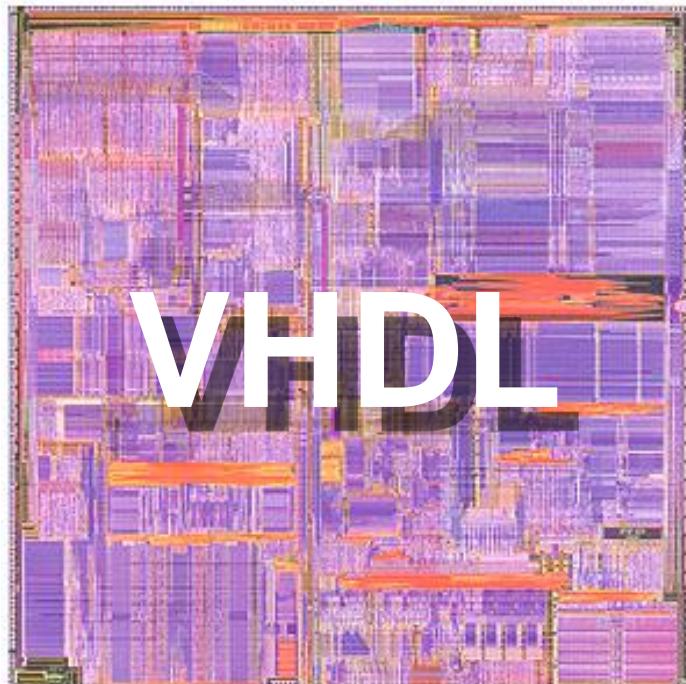



# ¿Qué son los packages (paquetes)?

- Es el mecanismo que tiene VHDL para guardar construcciones que se van a reutilizar en varios diseños
  - Declaraciones de componentes, subprogramas, constantes...
- Consta de dos partes:
  - Declaración del paquete
    - Declaraciones de señales y constantes
    - Declaraciones de componentes
    - Definiciones de tipos
    - Declaraciones de subprogramas
  - Cuerpo del paquete
    - Cuerpo de los subprogramas
    - Cualquier otra declaración que se desea que permanezca privada al paquete
- Los paquetes se agrupan en librerías



# Lenguaje de Descripción de 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



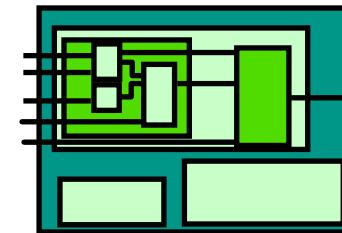
# Estilos de arquitecturas

- Arquitectura estructural

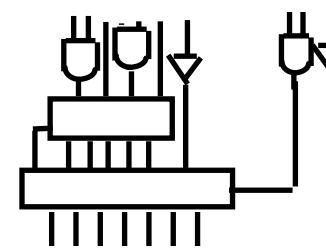


```

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



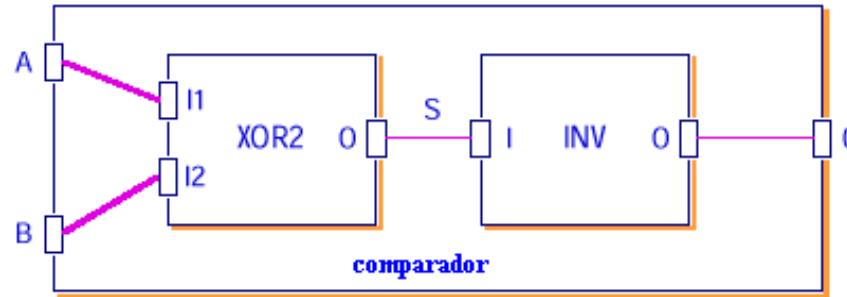
- Arquitectura “comportamental”: RTL



- Arquitectura “comportamental”:  
Algorítmica / modelo funcional



# Estilos de descripción: Estructural



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

```

architecture structural 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 : XOR2 port map (O => S, I1 => A, I2 => B);
    C2 : INV port map (C, S);
end structural;

```



# Estilos de descripción: Comportamiento



- Nivel abstracto de descripción, usando la expresividad de las construcciones de VHDL
- Definición de QUÉ HACE el modelo y NO CÓMO SE CONSTRUYE.
- Es sintetizable siempre que sea RTL

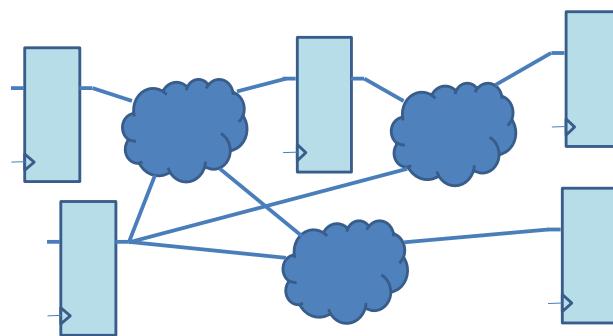
```
architecture comportamental of comparador is
begin
  process
    begin
      if (A = B) then
        C <= '1';
      else
        C <= '0';
      end if;
      wait on A, B;
    end process;
end comportamental;
```



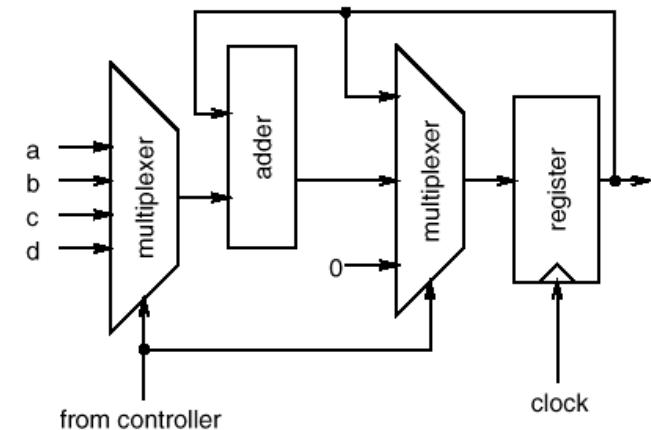
# Estilos de descripción: RTL

## RTL: Register Transfer Level

- Descripción de comportamiento que usa sólo transferencia de datos entre registros, atravesando lógica combinacional.



- Tiene en cuenta el ciclo de reloj 
- Independiente de la tecnología
- Definición del sistema en términos de:
  - registros
  - lógica combinacional



# Estilos de descripción: RTL

- Recordatorio: estructuras básicas RTL en VHDL:

## PROCESO SECUENCIAL

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

- Sólo Clock y Reset en lista de sensibilidad
- Respetar estrictamente el formato del if

## PROCESO COMBINACIONAL

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

- Todas las entradas deben estar en la lista de sensibilidad
- Se debe asignar siempre (en todos los casos) a la salida un valor



# Estilos de descripción: Algorítmico / Modelo funcional

- Descripción de comportamiento completamente independiente del hardware, sin especificar necesariamente relojes ni separando entre bloques combinacionales y secuenciales.

```
instrucion := memoria(PC);
opcode := instrucion (31 downto 28)
rs1 := instrucion(27 downto 24); rs2 := instrucion(23 downto 20);
rd := instrucion(19 downto 16); offset := instrucion(15 downto 0);
case opcode is
  when load => rd := memoria(rs1 + offset);
  when store => memoria(rs + offset) <= rd;
  when add => rd := rs1 + rs2;
  ...
  ...
```

- Equivalente a un modelo que se podría crear en cualquier otro lenguaje de alto nivel, por ejemplo C.
- Sólo es útil para simulación.



# Niveles de abstracción

VHDL permite describir circuitos electrónicos a distintos niveles de abstracción, de acuerdo con los estilos de descripción comentados anteriormente

- **Algorítmico:**

- Un algoritmo puro consiste en un conjunto de instrucciones ejecutadas secuencialmente que realizan una tarea.
- No se detallan necesariamente relojes o retrasos.
- No sintetizable (o sintetizable en casos limitados).

- **RTL**

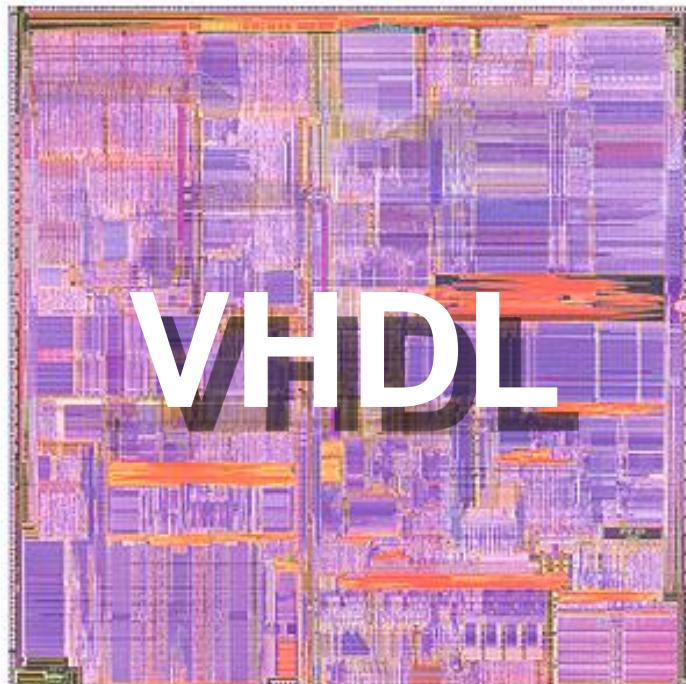
- Es la entrada para la síntesis.
- Las operaciones se realizan en un ciclo de reloj específico.
- No se detallan retrasos.

- **Nivel de puerta**

- Es la salida de la síntesis.
- Netlist de puertas e instanciaciones de una librería tecnológica.
- Se puede incluir información de retraso para cada puerta.

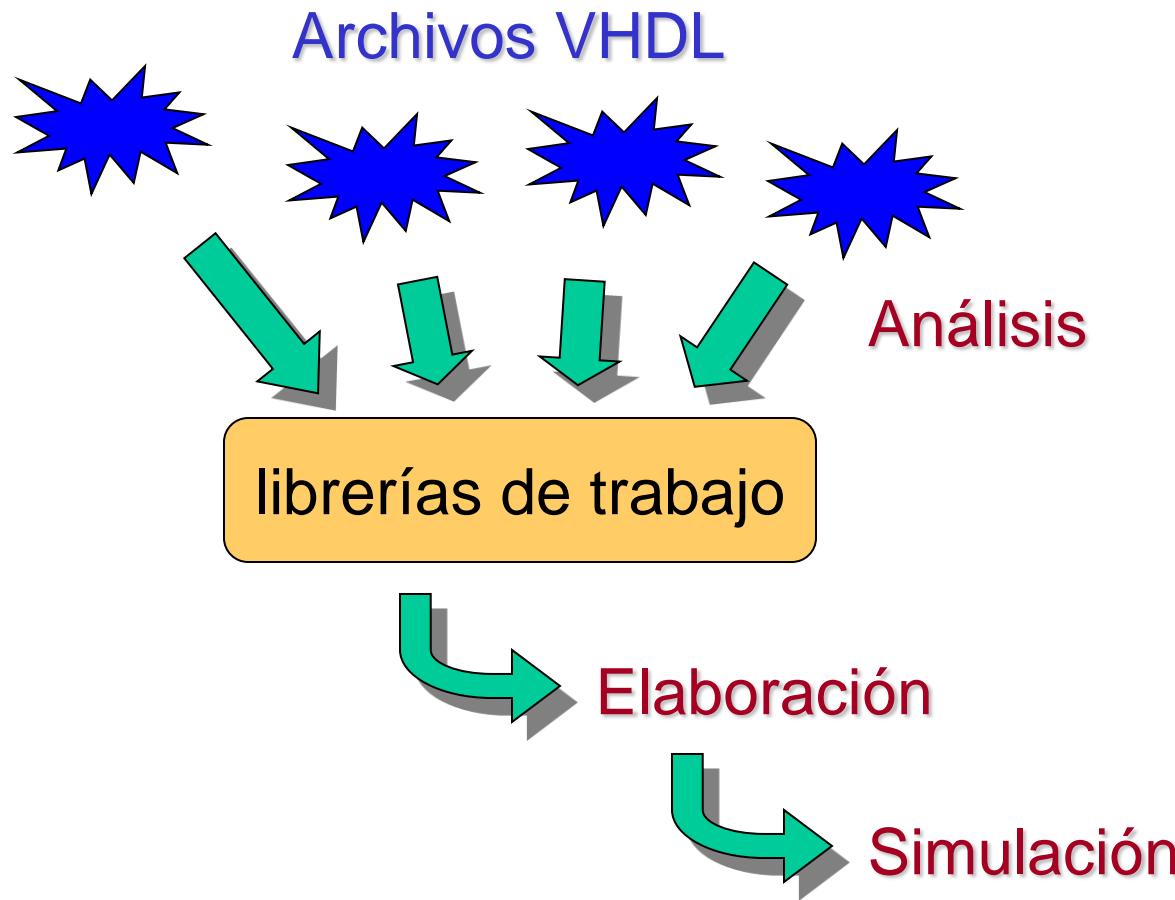


# Lenguaje de Descripción de Hardware VHDL



- 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



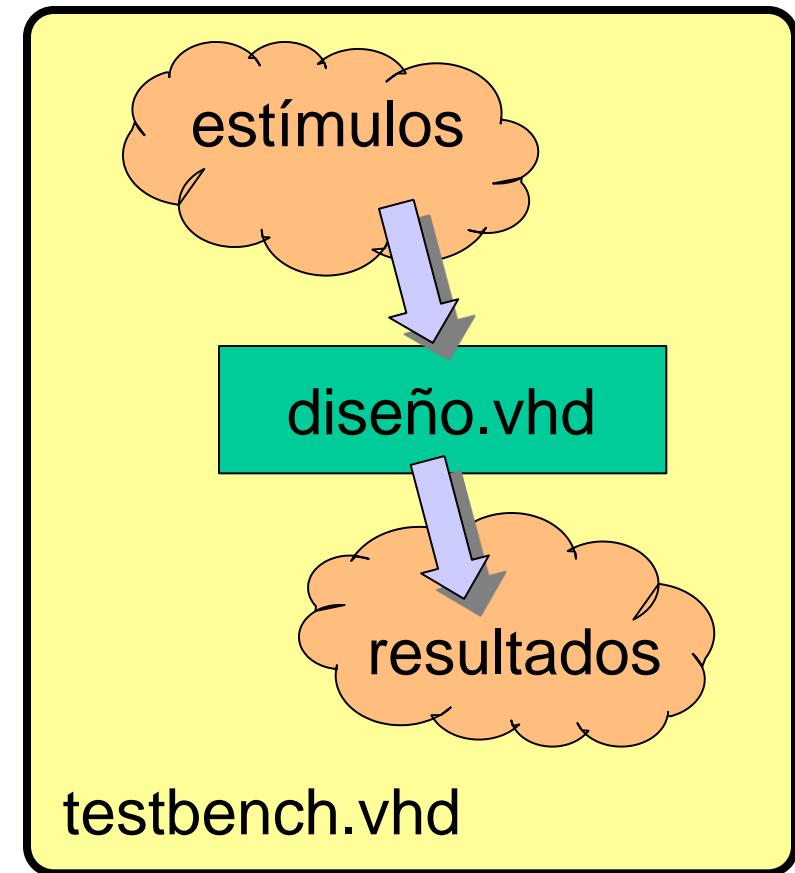
# Verificación con testbenches

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

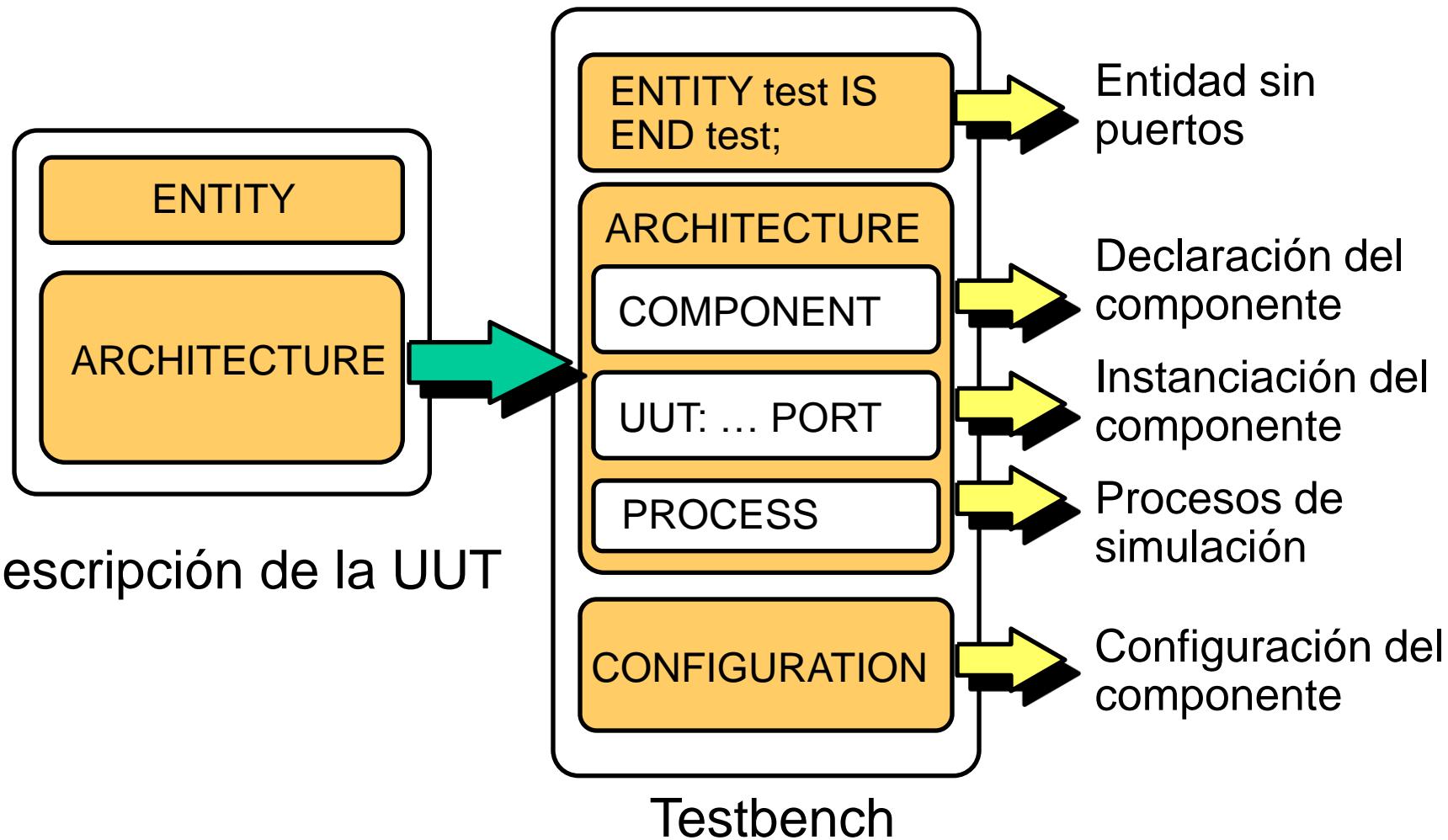


# Cómo hacer un testbench

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



# Instanciando la unidad bajo test (UUT)



# Generando estímulos

- Dar valores a las señales conectadas a las entradas de la UUT
  - En síntesis no tiene sentido el tiempo absoluto (tecnología)
  - En los testbenches controlamos el tiempo de generación de los estímulos.
- Asignación concurrente

```
senal <= '1',
  '0' AFTER 20 ns,
  '1' AFTER 30 ns;
```

- Asignación secuencial

```
senal <= '1';
WAIT FOR 20 ns; . .
senal <= '0';
WAIT FOR 30 ns;
senal <= '1';
```



- Típicamente siguiendo los ciclos de reloj

```
WAIT UNTIL rising_edge (Clk);
senal <= '1';
```



# Observando señales con assert

- Assert se usa para comprobar si se cumple una condición
  - Equivalente a **IF (not condición)**

- Sintaxis

```
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 si no se incluye SEVERITY)
  - FAILURE



# Características adicionales de assert

- El simulador puede configurarse para indicar a partir de qué nivel de error se parará la simulación
- Se pueden mostrar en el REPORT valores de señales:

```
ASSERT q=d
REPORT "Valor erroneo: " & std_logic'image(q)
SEVERITY error;
```

- Se utiliza el atributo predefinido de VHDL 'image, que pasa de cualquier valor, de un tipo escalar, a una representación en forma de string
- Generalmente se usa dentro de procesos (instrucción secuencial), pero también se puede usar como concurrente
  - Se chequea la condición continuamente, y en el momento en que deja de cumplirse, se escribe el mensaje



# Algoritmo básico para los testbenches

- Algoritmo elemental de verificación:
  - Dar valores a las señales de entrada a la UUT
  - Esperar con WAIT
  - Comprobar los resultados con ASSERT
  - Volver a dar valores a las señales de entrada a la UUT
  - y repetir...



# Ejemplo de código

```
ARCHITECTURE tb_arch OF dff_tb IS

COMPONENT dff PORT (...) END COMPONENT;
SIGNAL d, c, q : std_logic;

BEGIN

UUT : dff PORT MAP (d => d, c => c, q => q );

PROCESS
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;
```



# Ejemplo de código

- Se pueden dar valores dependiendo de los resultados

```
senal <= "00"; WAIT FOR 10 ns;  
IF salida /= '0' THEN  
    REPORT "Intento otra vez"; senal <= "01";  
ELSE  
    REPORT "Ahora pruebo con bit uno"; senal <= "10";  
END IF;  
WAIT FOR 10 ns;
```

- Usar los bucles para hacer pruebas sistemáticas

```
FOR i IN 0 TO 255 LOOP  
    FOR j IN 0 TO 255 LOOP  
        sumando1 <= conv_std_logic_vector(i,N);  
        sumando2 <= conv_std_logic_vector(j,N);  
        WAIT FOR 10 ns;  
        ASSERT conv_integer(suma)=(i+j)  
            REPORT "ha fallado la suma" SEVERITY ERROR;  
    END LOOP;  
END LOOP;
```



# Procedimientos y funciones

- VHDL permite definir procedimientos (subrutinas)

```
PROCEDURE nombre (clase parametro : dir tipo; ...) IS
    {declaraciones}
BEGIN
    {instrucciones secuenciales}
END PROCEDURE nombre;
```

- La clase de los parámetros puede ser:
  - VARIABLE, CONSTANT, SIGNAL
- Y 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.
- Existe otro tipo de subrutina que devuelve un valor, la función:  
`FUNCTION nombre (parametros...) RETURN tipo IS`

*Interesantes 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 PROCEDURE send_clock_edge;

BEGIN

UUT : dff PORT MAP (d => d, c => c, q => q );

PROCESS
BEGIN
    c <= '0'; d <= '0';
    send_clock_edge(c);
    ASSERT q=d REPORT "falla" SEVERITY FAILURE;
END PROCESS;

END tb_arch;
```



# Acceso a archivos

- La simulaciones pueden leer archivos, por ejemplo con vectores de entrada.
- Ejemplos:
  - Simulación de un multiplicador que escribe los resultados en un archivo de texto
  - Testbench para un microprocesador que lee un programa en ensamblador de un archivo, lo ensambla y lo ejecuta
- Acceso básico: paquete std.textio
  - Archivos de texto
  - Acceso línea a línea: READLINE, 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(archivo_estimulos, linea);
```

- Leer un campo de una línea

```
VARIABLE opcode : string(2 downto 1);  
...  
read (linea, opcode);
```

- Escribir

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



# Configuración

```
CONFIGURATION <identifier> OF <entity_name> IS
    FOR <architecture_name>
    END FOR;
END; (1076-1987 version)
END CONFIGURATION; (1076-1993 version)
```

Aplicándolo a un componente particular:

**FOR** instance\_name : comp\_name **USE** ...;

Aplicándolo a todas las instancias:

**FOR ALL:** comp\_1 **USE ENTITY**  
**WORK.**entity\_name(architecture\_name);

**Una configuración es una unidad de diseño:**

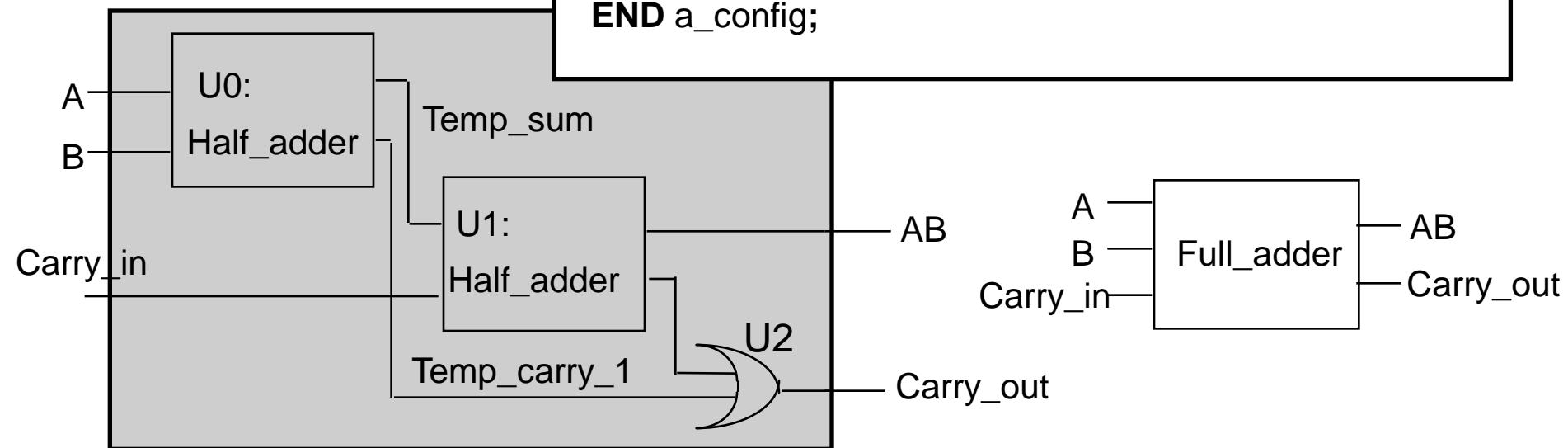
- Que se usa para realizar asociaciones dentro de los modelos  
 Asociar entidad con arquitectura.  
 En la instanciación de un componente asociarlo a una entidad y su arquitectura.
- Muy utilizada en entornos de simulación  
 Proporciona una manera rápida y flexible de probar distintas alternativas del diseño
- Limitada o no soportada en entornos de síntesis.



# Configuración: Ejemplo

Ejemplo: Sumador

```
CONFIGURATION a_config OF Full_adder IS
  FOR structural
    FOR all: Half_adder
      USE ENTITY work.Half_adder(algorithmic);
    END FOR;
    FOR U2: or_gate
      USE ENTITY work.or_gate(behavioral);
    END FOR;
  END FOR;
END a_config;
```



# Verificación de un diseño escrito en VHDL

## 1. Simulación funcional

- RTL / behavioral sintetizable
- Emplea construcciones estándar VHDL
- Sin tiempos

## 2. Simulación post-síntesis

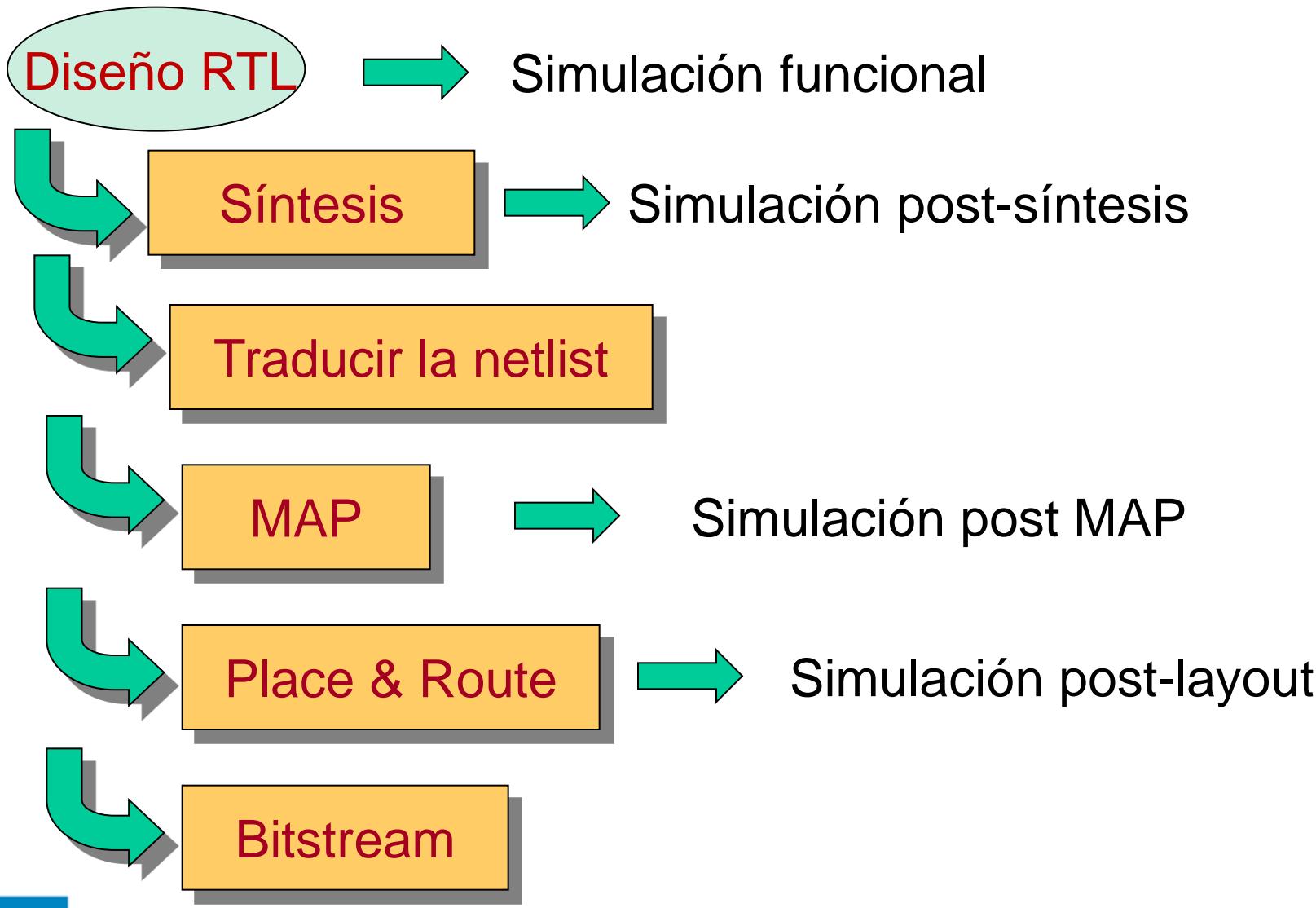
- Estructural
- Librería de síntesis, elementos básicos de diseño para la FPGA
- Sin tiempos

## 3. Simulación post-map, post-layout

- Estructural
- Librería del fabricante (Xilinx), elementos reales de la FPGA
- Con tiempos si se carga el archivo SDF (Standard Delay Format)



# Simulaciones en el flujo de diseño de FPGAs



# Testbench para un diseño VHDL

- El testbench debe ser el mismo para todas las simulaciones en las diferentes etapas de diseño
- Las simulaciones se asociarán con sus respectivas configuraciones, se pondrá como top-level la configuración que nos interese en cada momento

```
CONFIGURATION funcional OF dfftb IS
  FOR dfftbatch
    FOR uut:dff USE ENTITY WORK.dff(behavioral);
    END FOR;
  END FOR;
END funcional;
```

**FUNCIONAL**



```
CONFIGURATION postlayout OF dfftb IS
  FOR dfftbatch
    FOR uut:dff USE ENTITY WORK.dff(structure);
    END FOR;
  END FOR;
END postlayout;
```

**POST-LAYOUT**

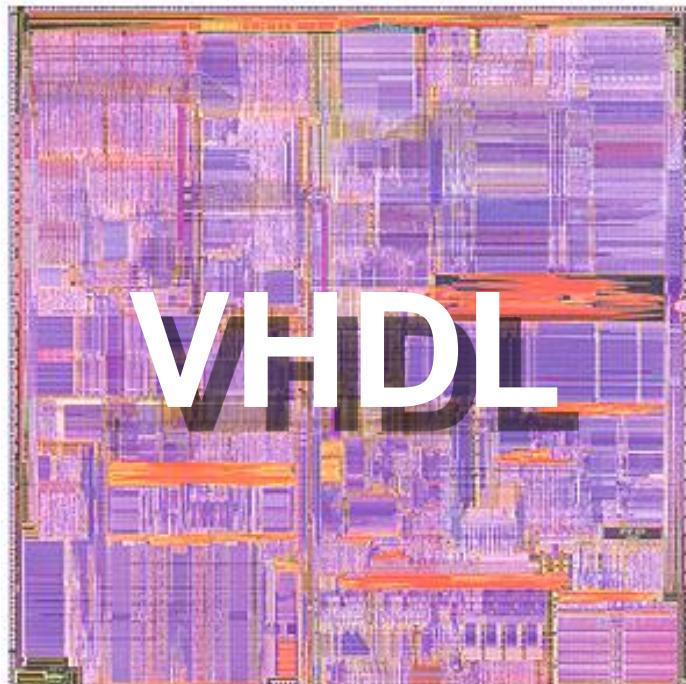


# Modelo post-layout

- Una vez implementado el diseño, se crea un modelo post-layout del diseño en la FPGA mediante el comando *netgen* (integrado en los procesos ejecutados con el interfaz gráfico de ISE)
  - **xxx\_timesim.vhd**
- El modelo usa la librería SIMPRIM
- Los tiempos se anotan en un archivo SDF separado
  - **xxx\_timesim.sdf**
- Toda la FPGA está modelada en estos dos archivos
- El simulador puede cargar (o no) el fichero SDF
  - Primera simulación, rápida, ver si el comportamiento es correcto
  - Segunda simulación, detallada, ver si los tiempos se respetan
- Pueden aparecer en la entidad los nodos GSR/GTS (opcional)
  - Se puede arreglar reconfigurando los ports en CONFIGURATION



# Lenguaje de Descripción de 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

