



# Tema 1: Diseño y Modelado Hardware con VHDL



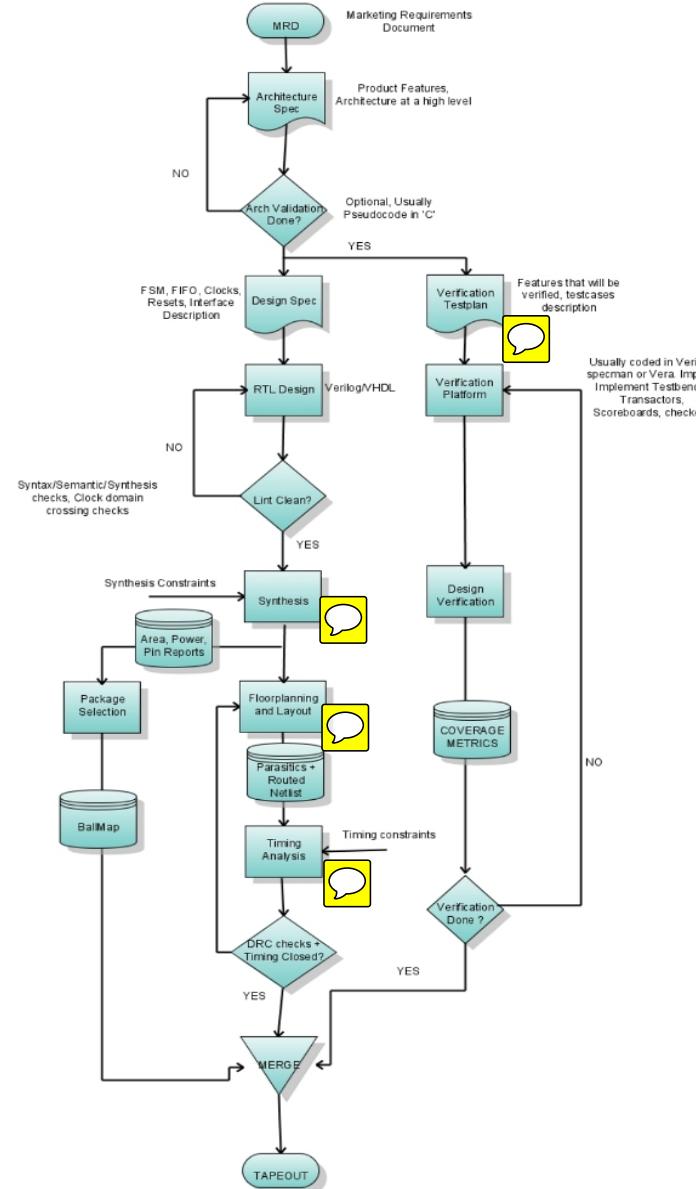
# Índice

1. Flujo de diseño
2. Lenguaje de descripción Hardware
3. Simulación con VHDL
4. Estructura de un modelo VHDL
5. Elementos básicos de VHDL
6. Máquina de estados finitos
7. Otros elementos de VHDL
8. Test-bench de simulación

# 1. Flujo de diseño



# Flujo de diseño





# Lenguaje de Descripción Hardware

- ¿Qué es HDL?
  - Lenguaje específicamente creado para el diseño de circuitos:
    - » Nivel de puerta (gate level).
    - » Nivel de comportamiento (behavioural level).
  - La estructura del lenguaje sugiere el diseño hardware.
- ¿Por qué usar HDL?
  - Poder descubrir problemas en el diseño antes de su implementación física.
  - La complejidad de los sistemas electrónicos crece exponencialmente, es necesaria una herramienta que trabaje con el ordenador.
  - Permite que más de una persona trabaje en el mismo proyecto.

*No es lo mismo un lenguaje **hardware** que un lenguaje **software***



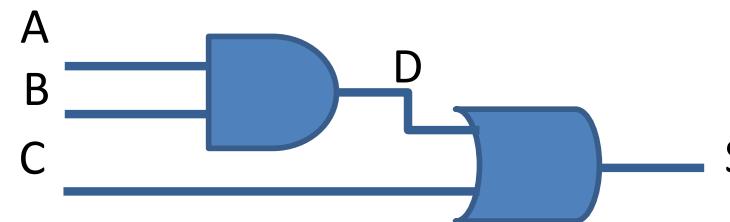
# Lenguaje de Descripción Hardware

- VHDL
  - VHSIC (Gobierno de EE.UU. 1980).
  - IEEE VHDL'87.
  - [www.vhdl.org](http://www.vhdl.org)
- Verilog
  - Desarrollado por CADENCE.
  - IEEE 1364.
  - [www.eda.org](http://www.eda.org)
- SystemVerilog
  - Descripción hardware y verificación.
  - Extensión IEEE 1364
  - [www.systemverilog.org](http://www.systemverilog.org)



# ¿Por qué HDL?

- HDL tiene que ser capaz de simular el comportamiento real del HW sin que el programador necesite imponer restricciones.



**A and B  
D or C**



# ¿Por qué HDL?

$t = 5\text{ns}$	$t = 10\text{ns}$
$A = 0$	$A = 1$
$B = 1$	$B = 1$
$C = 0$	$C = 0$

## Descripción 1

$D = A \text{ and } B;$   
 $S = D \text{ or } C;$

## Descripción 2

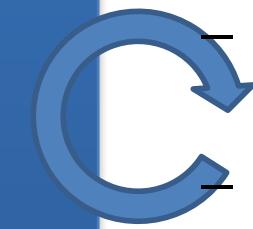
$S = D \text{ or } C;$   
 $D = A \text{ and } B;$

¿Se obtiene el mismo resultado?

# VHDL



- Características
  - Descripción de la estructura del circuito
    - Descomposición en sub-circuitos
    - Interconexión de sub-circuitos
    - Comportamiento
    - Estructural
  - Permite la especificación de la funcionalidad de un circuito utilizando formas familiares de lenguajes de programación
  - Permite la simulación del circuito antes de su fabricación
    - Testear y comparar alternativas sin necesidad de prototipos hardware



## Fases simulación



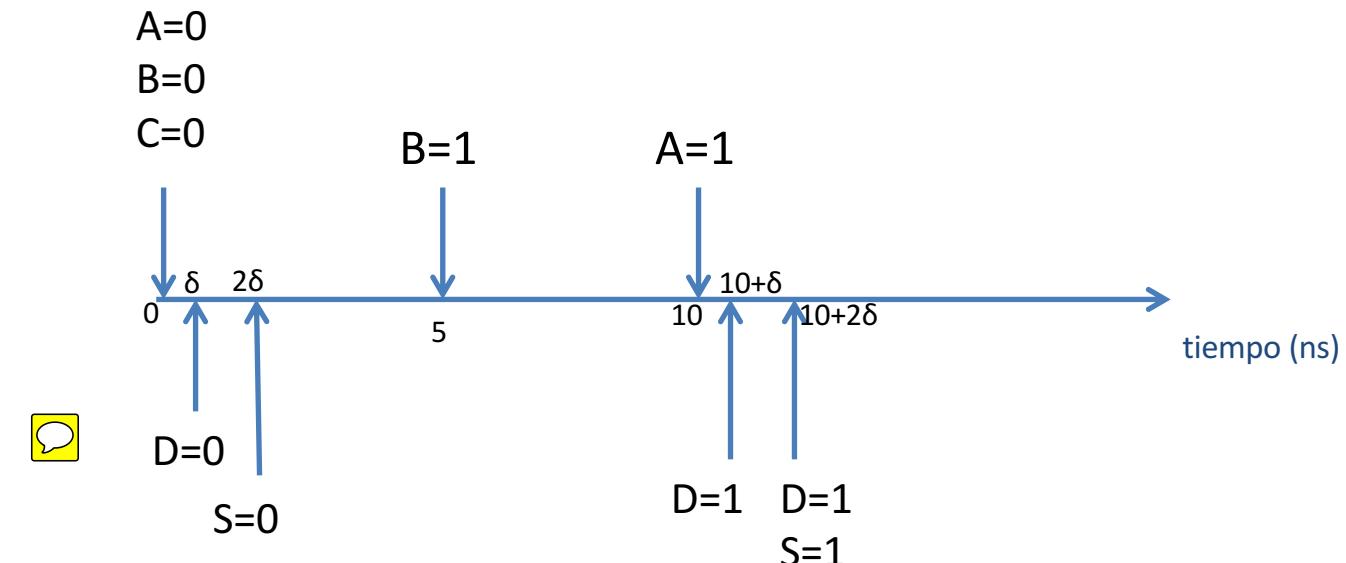
- VHDL realiza la simulación siguiendo la técnica de **simulación por eventos discretos** (*Discrete Event Time Model*).
  - Permite avanzar el tiempo a intervalos variables, en función de la planificación de ocurrencia de eventos
- La simulación consta de tres fases:
  - Fase 0: fase de inicialización donde a las señales se les asignan unos valores iniciales y se pone el tiempo a cero. La asignación se hace rellenando una lista de eventos para el instante  $t = 0$ .
  - Fase 1: todas las transiciones planificadas para ese tiempo son ejecutadas  
  - Fase 2: Las señales que se han modificado como consecuencia de las transiciones planificadas en el instante  $t$  se escriben en la lista de eventos planificándose para el instante  $t + \delta$ . Donde  $\delta$  es un instante infinitesimal.  



## Ejemplo I

- Ejemplo I →  $D \leq A \text{ and } B;$   
 $S \leq D \text{ or } C;$

Valores iniciales  
A=U  
B=U  
C=U  
D=U  
S=U

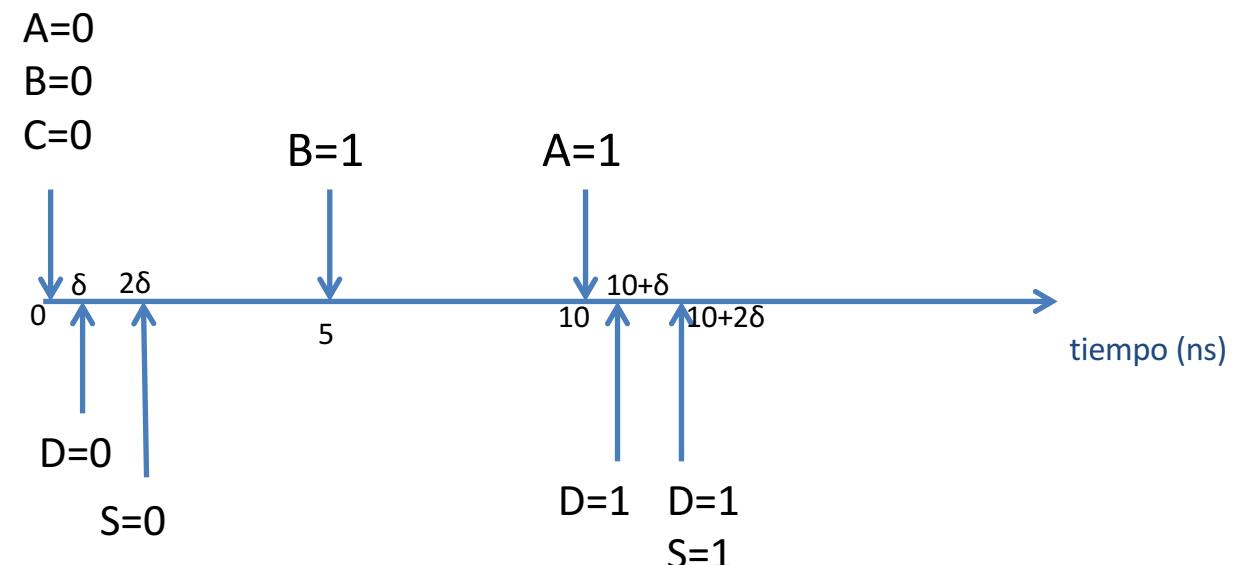




## Ejemplo II

- Ejemplo II →  $S \leq D \text{ or } C;$   
 $D \leq A \text{ and } B;$

Valores iniciales  
A=U  
B=U  
C=U  
D=U  
S=U





## Ejemplo III

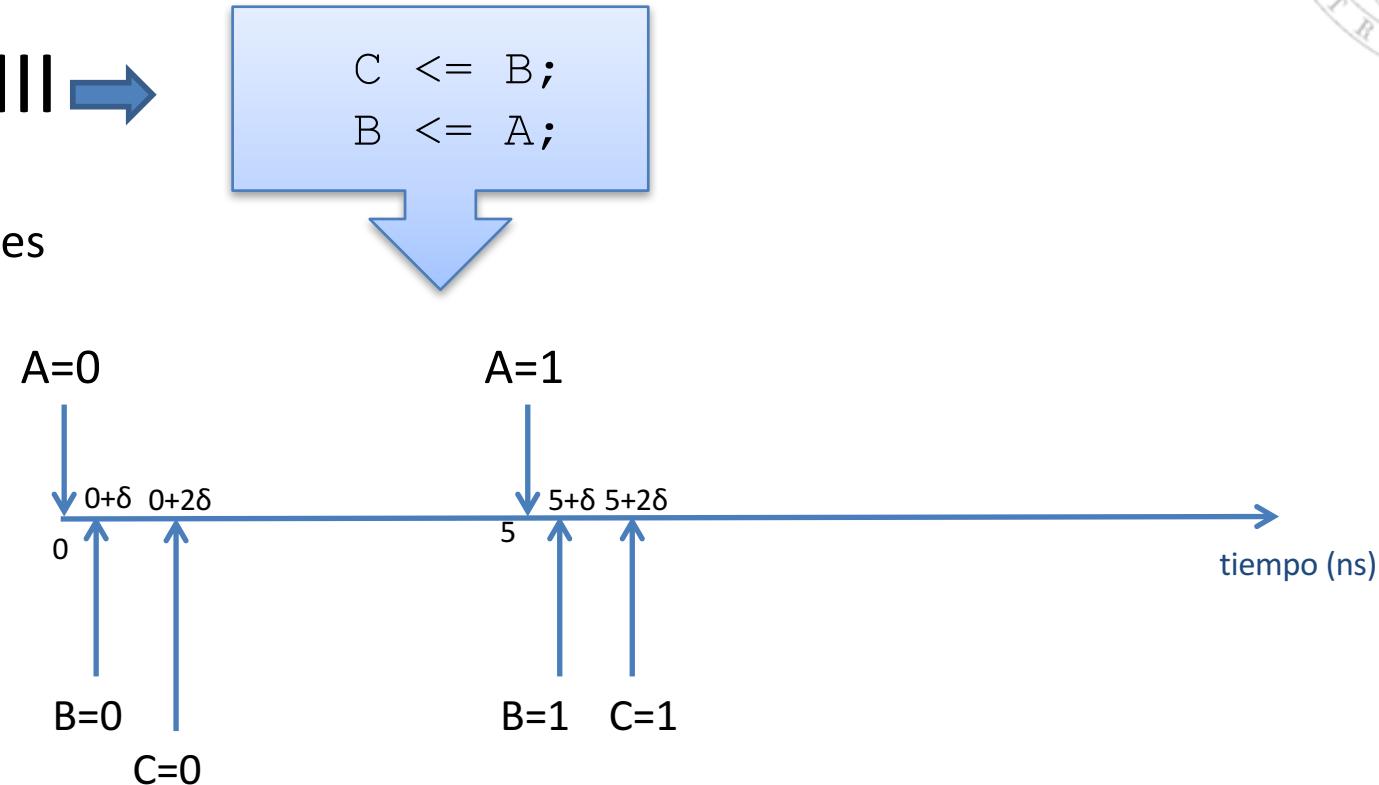
- Ejemplo III →

Valores iniciales

$A=U$

$B=U$

$C=U$



$t(ns)$	0	$0 + \delta$	$0 + 2\delta$	No hay más cambios	5	$5 + \delta$	$5 + 2\delta$	No hay más cambios
A	0	0	0		1	1	1	
B	U	0	0		0	1	1	
C	U	U	0		0	0	1	

### 3. Simulación con VHDL

toc



## Ejemplo IV

### Ejemplo III →

Valores iniciales

A=U

B=U

C=U

A=0

C=0

0+δ

0

B=0

```
A <= C xor B;  
B <= A;
```

C=1

5

...

A=1

B=1

A=0

B=0

A=1

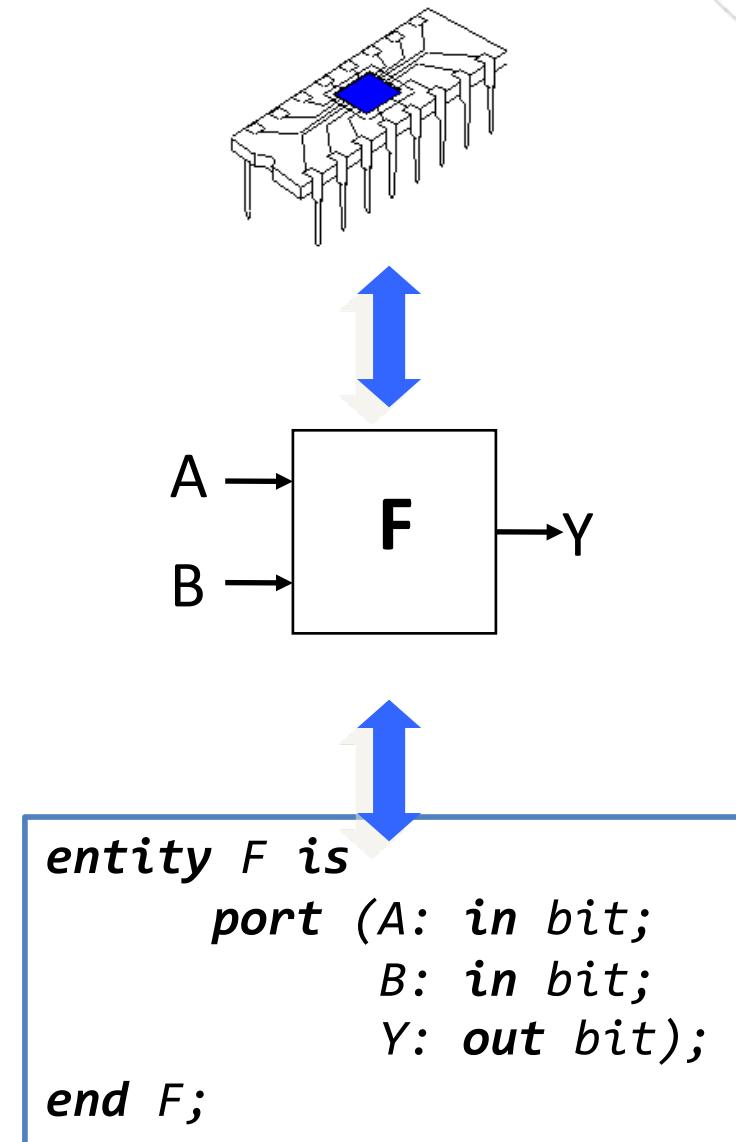
B=1

tiempo (ns)

Oscila indefinidamente

# ¿Qué es?

- Descripción de un sistema digital:
  - Un sistema digital está descrito por sus entradas y sus salidas, donde las salidas dependen de las entradas



# Entidad y arquitectura



- Los modelos VHDL están formado por 2 partes:

```
entity nombre is
generic (lista de parámetros);
port (lista de puertos de entrada y salida);
end nombre;
```

```
architecture circuito of nombre is
-- señales
begin
-- programacion
end architecture circuito;
```

# Entidad y arquitectura



- Los modelos VHDL están formado por 2 partes:
  - Entity
    - Define externamente al circuito o subcircuito
    - Nombre y número de puertos, tipos de datos de entrada y salida
    - Tienes toda la información necesaria para conectar tu circuito a otros circuitos
  - Architecture 
    - Define internamente el circuito
    - Señales internas, funciones, procedimientos, constantes ...

# Entidad



```
entity nombre_entidad is
    port (lista puertos de entrada y salida);
end nombre_entidad;
```

## Lista puertos de entrada-salida

nombre Puerto: tipo de puerto tipo de señal; ...

*entradaA: in bit\_vector(7 downto 0);*





# Arquitectura (I)

```
architecture arch_name of nombre_entidad is
```

- tipos
- señales
- componentes



}

Parte declarativa arquitectura

```
begin
```

- sentencias concurrentes
- procesos
- componentes

```
end architecture arch_name;
```

```
<etiqueta> : process (lista de sensibilidad)  
begin  
  -- código de descripción  
end process;
```

# Arquitectura (II)



- El código VHDL propiamente dicho se escribe dentro de ***architecture***.
- Cada ***architecture*** va asociada a una ***entity*** y se indica en la primera sentencia.
- A continuación, y antes de ***begin*** se definen:
  - Señales
  - Tipos
  - Componentes: otros circuitos ya definidos y compilados de los cuales conocemos su interfaz en VHDL (su ***entity***).
- Desde ***begin*** hasta ***end*** escribiremos todas las sentencias propias de VHDL
  - No todas pueden utilizarse en cualquier parte del código.



# Tipos de datos (I)



- Built-in
  - Tipos enteros y reales
    - INTEGER**
    - NATURAL**
    - REAL**
  - Tipos físicos
    - TIME**
    - RESISTENCIA**
  - Tipos enumerados: conjunto de valores posibles.

# Tipos de datos (II)



- Built-in
  - Tipos enumerados
    - BIT**: {'0', '1'}
    - BOOLEAN** : {TRUE, FALSE}
    - CHARACTER** : {ascii}
    - STD\_LOGIC** : {'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'}



## Tipos de datos (III)

- Definidos por diseñador
  - Pueden definirse tipos adicionales:
    - Enumerados
    - Sub-tipos: subconjunto de tipos

nuevo\_tipo ::= **type** identificador **is** definición\_tipo; 

# Señales y variables



- Dentro de la arquitectura:
  - Señales:
    - Representan elementos de memoria o conexiones.
    - Los puertos de una entidad.
    - En la arquitectura antes del BEGIN, lo cual nos permite realizar conexiones entre diferentes módulos.
- Dentro de los procesos:
  - Variables
    - Se utilizan como índices (instrucciones de bucle o modelar componentes).
    - Las variables NO representan conexiones o estados de memoria.

señal <= valor



variable := valor



# Sentencias concurrentes (I)

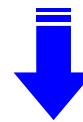


- Siempre fuera de los procesos.
- Aparecen en cualquier punto del programa (después del begin de la arquitectura).
- Es lógica combinacional pura.
- Siempre tienen un *else final* o un *when others*.

## Sentencias concurrentes (II)



```
signal_name <= valor_1 when condición1 else  
    valor_2 when condición2 else  
    ...  
    valor_i when condicióni else else ← OBLIGATORIO  
    otro_valor;
```



```
salida <= "00" when entrada = "0001" else  
    "01" when entrada = "0010" else  
    "10" when entrada = "0100" else  
    "11";
```

# Procesos



## ■ Procesos: sentencias concurrentes “vitaminadas”

```
<etiqueta> : process (lista de sensibilidad)
begin
    -- sentencias secuenciales
    -- sentencias condicionales
    -- bucles
end process
```

Sólo se ejecutan las sentencias que se encuentran dentro del process si alguna de las señales de la lista de sensibilidad ha cambiado de valor

- Sólo se ejecutan las sentencias que se encuentran dentro del proceso si alguna de las señales de la lista de sensibilidad ha cambiado de valor.
- Código secuencial.



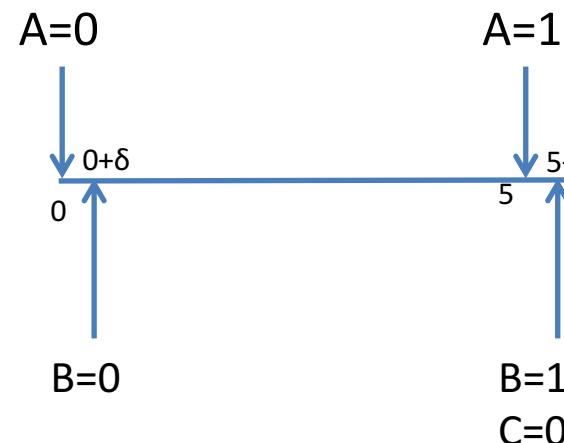
# Simulación de un proceso (I)

Valores iniciales

$A=U$

$B=U$

$C=U$



```
p_wire : process (A)
begin
    B <= A;
    C <= B;
end process;
```

tiempo (ns)

$t$ (ns)	0	$0 + \delta$	No hay más cambios	5	$5 + \delta$	No hay más cambios
A	0	0		1	1	
B	U	0		0	1	
C	U	U		U	0	



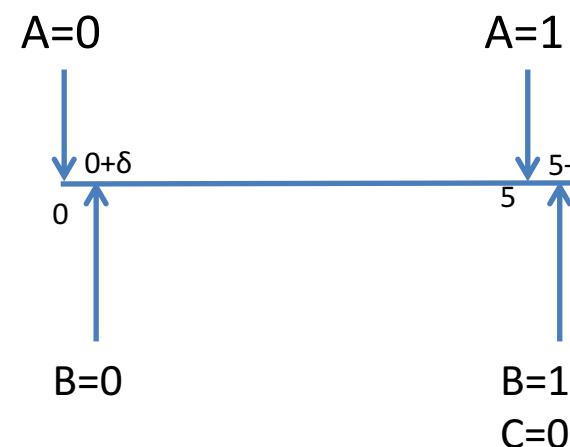
# Simulación de un proceso (II)

Valores iniciales

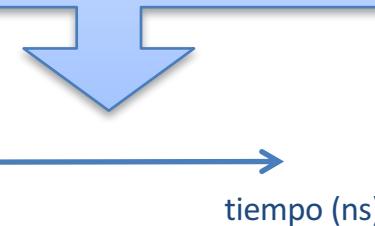
$A=U$

$B=U$

$C=U$

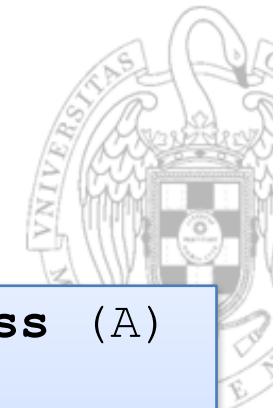


```
p_wire : process (A)
begin
    C <= B;
    B <= A;
end process;
```



tiempo (ns)

$t$ (ns)	0	$0 + \delta$	No hay más cambios	5	$5 + \delta$	No hay más cambios
A	0	0		1	1	
B						
C						



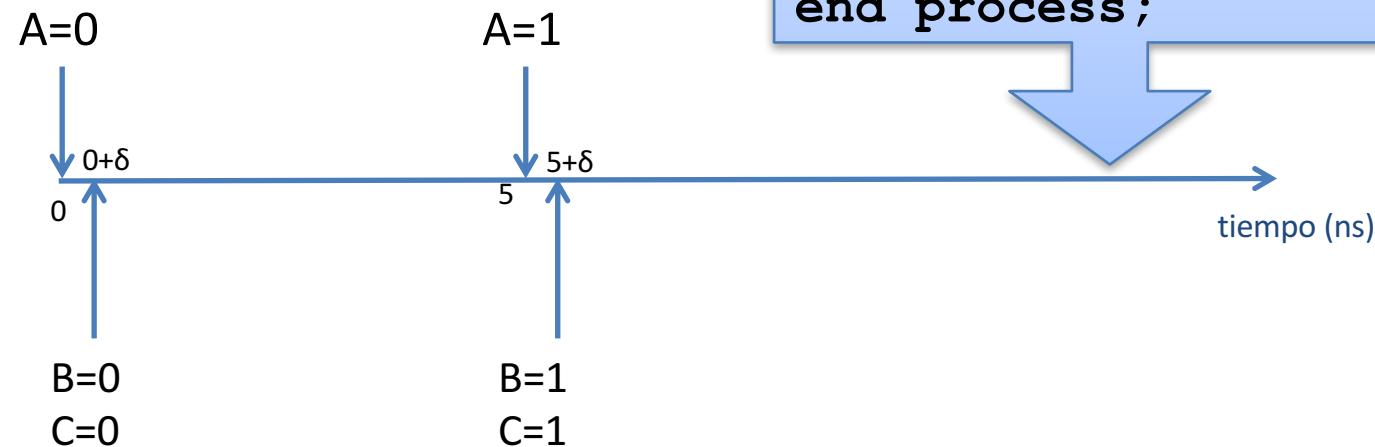
# Simulación de un proceso (III)

Valores iniciales

A=U

B=U

C=U





# Procesos

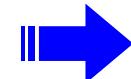
- Sentencias condicionales (I)
  - VHDL permite utilizar otro tipo de sentencias condicionales más parecidas a los lenguajes de programación usados
  - Se utilizan siempre dentro de una estructura process
  - Como veremos mas adelante sentencias condicionales incompletamente descritas dan lugar a HW secuencial



# Procesos

- Sentencias condicionales (II):

```
if cond_1 then  
    -- sequential statements  
elsif cond-n then  
    -- sequential statements  
else  
    -- sequential statements  
end if;
```



```
if a = b then  
    c <= a or b;  
elsif a<b then  
    c <= b;  
else  
    c <= "0000";  
end if;
```

# Procesos



- Todas las sentencias condicionales van dentro de un process  
¿por qué?
- Todos los process tienen lista de sensibilidad

```
process (a, b)
begin
    if a = b then
        c <= a or b;
    elsif a<b then
        c <= b;
    else
        c <= "0000";
    end if;
end process;
```

<b>t = 5 ns</b>	<b>t = 10 ns</b>	<b>t = 15 ns</b>
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c	c	c

Según avance el curso se observarán las consecuencias *malignas* de la utilización de los process



# Procesos

- Todas las sentencias condicionales van dentro de un process  
¿por qué?
- Todos los process tienen lista de sensibilidad

```
process (a, b)
begin
    if a = b then
        c <= a or b;
    elsif a<b then
        c <= b;
    end if;
end process;
```

<b>t = 5 ns</b>	<b>t = 10 ns</b>	<b>t = 15 ns</b>
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c	c	c

Según avance el curso se observarán las consecuencias *malignas* de la utilización de los process



# Procesos

- Todas las sentencias condicionales van dentro de un process  
¿por qué?
- Todos los process tienen lista de sensibilidad

```
process (a)
begin
    if a = b then
        c <= a or b;
    elsif a<b then
        c <= b;
    else
        c <= "0000";
    end if;
end process;
```

<b>t = 5 ns</b>	<b>t = 10 ns</b>	<b>t = 15 ns</b>
a <= "0010"	a <= "0010"	a <= "0010"
b <= "0010"	b <= "0110"	b <= "0001"
c	c	c

Según avance el curso se observarán las consecuencias *malignas* de la utilización de los process



# Procesos

- Sentencias condicionales (III):

```
case expression is
    when choice_1 => ... --seq. statements;
    when choice_n => ... --seq. statements;
    when others => ... --seq. statements;
end case;
```

↓

```
case RGB is
    when "111" => r <= 0;
    when "100" => r <= 1;
    when "110" => r <= 1;
    when others => r <= 0;
end case;
```



# Procesos

## ■ Bucles

```
[label:] loop  
    sequence-of-statements  
    -- use exit statement to get out  
end loop [label];
```

```
[label:] for var in range loop  
    sequence-of-statements  
end loop [label];
```

```
[label:] while cond loop  
    sequence-of-statements  
end loop [label];
```

# Procesos



## ■ Wait

[label:] wait [sensitive clause] [condition clause];

- Se usa en procesos, procedimientos y funciones.
- Tres tipos:
  - wait for <timeout clause, time delay>
  - wait until <condition>
  - wait on <sensitive clause, event>
- Ejemplos:
  - wait for 10 ns;
  - wait until clk='1';
  - wait on in1;



# Atributos

- $S'DELAYED(t)$  is the signal value of S at time now - t .
- $S'STABLE$  is true if no event is occurring on signal S.
- $S'STABLE(t)$  is true if no even has occurred on signal S for t units of time.
- $S'QUIET$  is true if signal S is quiet. (no event this simulation cycle)
- $S'QUIET(t)$  is true if signal S has been quiet for t units of time.
- $S'TRANSACTION$  is a bit signal, the inverse of previous value each cycle S is active.
- $S'EVENT$  is true if signal S has had an event this simulation cycle.
- $S'ACTIVE$  is true if signal S is active during current simulation cycle.
- $S'LAST_EVENT$  is the time since the last event on signal S.
- $S'LAST_ACTIVE$  is the time since signal S was last active.
- $S'LAST_VALUE$  is the previous value of signal S.

# Expresiones secuenciales



- Expresiones secuenciales:

Deben aparecer  
SIEMPRE dentro de un  
process (clk, ...)

```
signal'event  
signal'last_event  
signal'last_value
```

```
if clk'event and clk = '1';
```



# Ejemplos

- Biestables, registros ...  
*clk'event and clk='1' flanco de subida*  
*clk'event and clk='0' flanco de bajada*
- Biestable D disparado por flanco:  

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



# Operadores básicos

- Operadores basicos:

*abs*  
\*, /, mod, rem  
+, -  
&  
and, or, nand, nor  
...

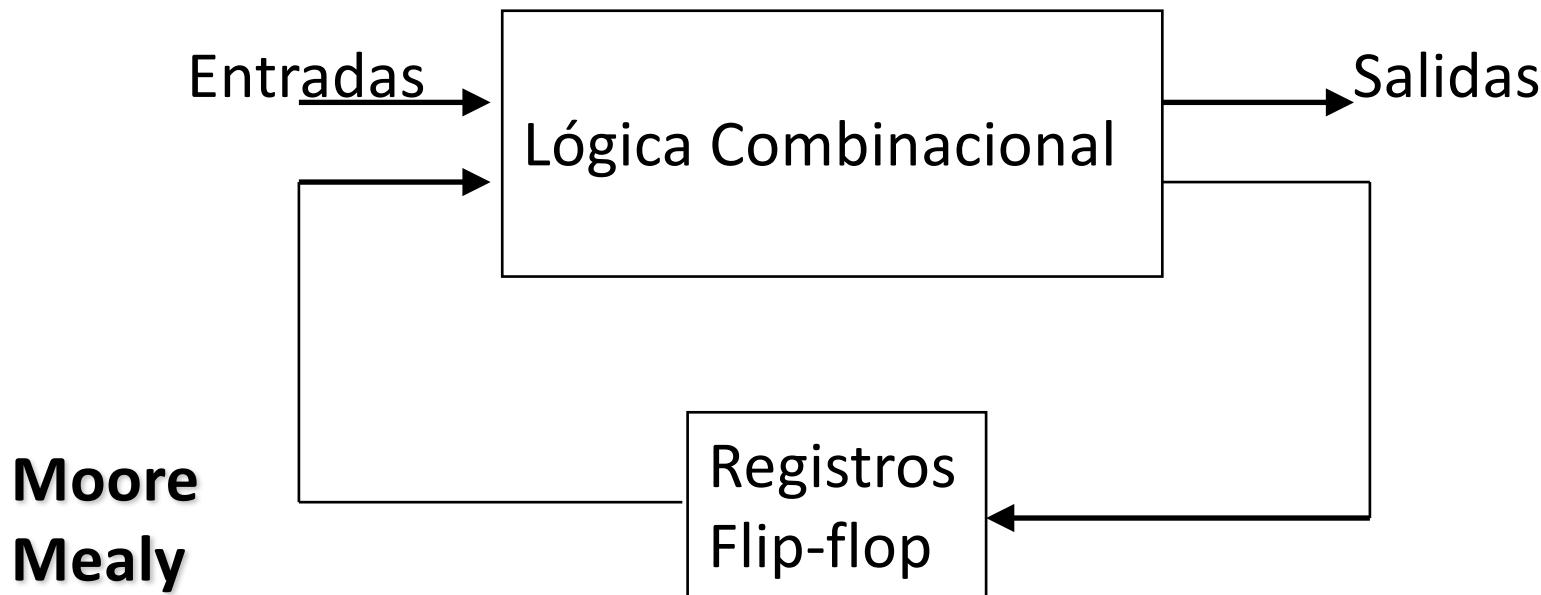


```
y<=(x1 and x2) or d(0);  
y(1) <= x1 and not x2;  
  
y: bit_vector(1 downto 0);  
y <= x1&x2;
```

# Máquina de estados finita (FSM)



- Todo circuito secuencial se divide en un circuito combinacional que implementa la salida del circuito y la transición al siguiente estado y en unos elementos de almacenamiento.



# Máquina de estados finita (FSM)



```
architecture rtl of FSM is  
internal signal declarations;  
begin
```

```
    p_reg_async: process (clk, rst)  
    begin  
        vhdl indicando los flip-flops  
    end process síncrono;
```

```
    p_comb: process (sensitivity list)  
    begin  
        vhdl indicando la lógica combinacional  
    end process comb;
```

```
end rtl;
```

# Plantilla de código



```
architecture arch_name of entity_name is
    component component_name
        generic (...);
        port (...);
    end component;
    signal declarations;
begin
    <etiqueta> : component_name
        generic map (parameter_value)
        port map (io_name)
    end arch_name;
```

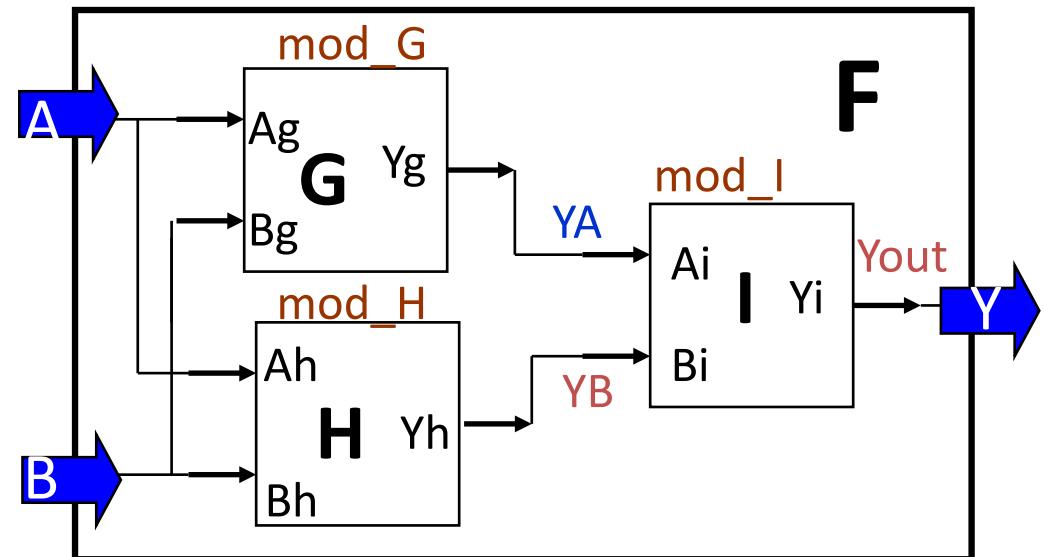
## 7. Descripción estructural

toc

```
architecture struct of f is
    component g
        port (ag: in bit;
               bg: in bit;
               yg: out bit);
    end component;
    component h
        port (ah: in bit;
               bh: in bit;
               yh: out bit);
    end component;
    component i
        port (ai: in bit;
               bi: in bit;
               yi: out bit);
    end component;
    signal ya, yb, yout: bit;
begin
```

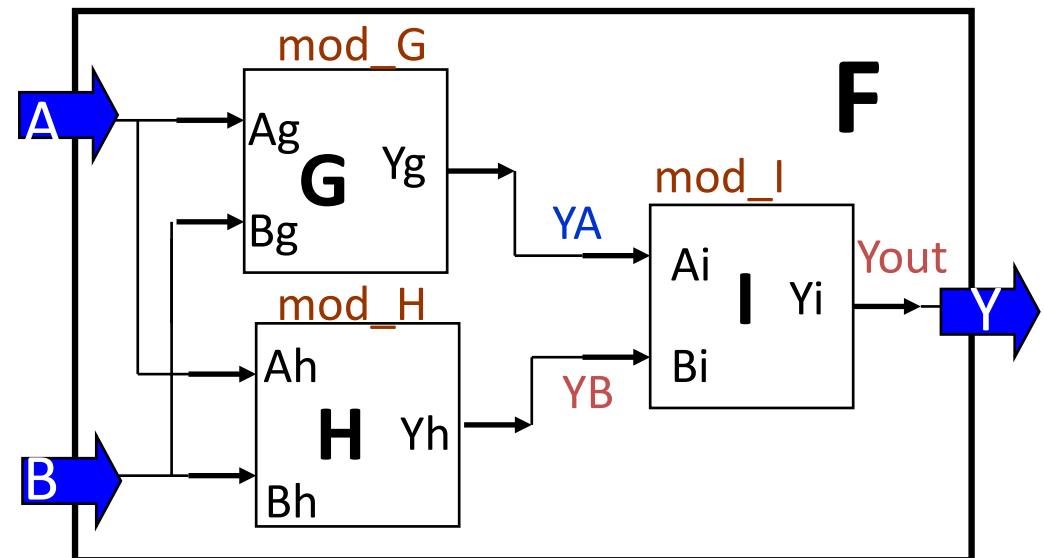
...

# Ejemplo



```
begin
    mod_g: g port map (
        ag => a,
        bg => b,
        yg => ya);
    mod_h: h port map (
        ah => a,
        bg => b,
        yh => yb);
    mod_i : i port map (
        ai => ya,
        bi => yb,
        yi => yout);
    salida: y<=yout;
end architecture struct;
```

# Ejemplo

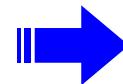




# Generate

- **GENERATE:** Las secuencias de generación de componentes permiten crear una o más copias de un conjunto de interconexiones, lo cual facilita el diseño de circuitos mediante descripciones estructurales.

```
for indice in rango generate
    -- instrucciones concurrentes
end generate;
```

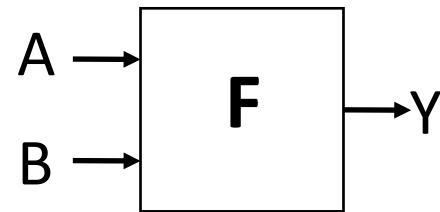
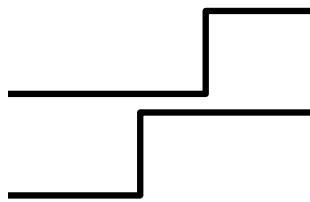


```
component comp
    port( x: in bit; y: out bit);
end component;
...
signal a, b: bit_vector(0 to 7);
...
gen: for i in 0 to 7 generate
    u: comp port map (a(i),b(i));
end generate gen;
```



# ¿Qué es?

- Para conocer si nuestro diseño funciona correctamente tendremos que introducir unos estímulos a las entradas y comprobar que las salidas obtenidas son las esperadas



Test-bench que reproduce los estímulos

```
entity test-bench  
end test-bench;  
...
```

Diseño a verificar

```
entity F is  
port (A, ...)  
end F;  
...
```

Salida gráfica de simulación ofrecida por la herramienta

# Estructura básica



```
entity F is
    port (A, B: in bit; Y out bit);
end F;
```

Diseño a verificar

```
entity test-bench
end test-bench;
```

i) Se crea una entidad de simulación sin puertos de entrada ni de salida

```
architecture behaviour of test_bench
component F is
    port (A, B: in bit; Y out bit);
end component F;
signal A, B, Y: bit;
begin
```

ii) Se añade como *component* la *entity* del diseño a verificar

...

iii) Se definen tantas señales como puertos de la *entity* del diseño a verificar

# Estructura básica



```
...
begin
    uut: F port map(
        A => A,
        B => B,
        Y => Y);
    tb : process
begin
    A<='0';
    B<='0';
    wait for 100 ns;
    B<='1';
    wait for 100 ns;    -- instante 200 ns
    A<='1';
    wait for 100 ns;    -- instante 300 ns
    ...
    wait;    -- espera para siempre
end process tb;
end;
```

iv) Se instancia el *component* igualando las señales internas a las entradas

v) Se crea el *process* de simulación.  
NO tiene lista de sensibilidad

vi) Se definen los valores iniciales de las entradas

vi) Se definen los valores de las entradas en los siguientes instantes de tiempo



# clk & rst

- Como implementar una entrada periódica: clk

```
clk_process :process
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;
```

- Como este *process* no acaba con un *wait (sin for)*, se repetirá indefinidamente