VHDL 2: Describiendo la funcionalidad

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Acknowledgement to Fernando Muñoz,
Universidad de Sevilla

Contexto docente

BT01: Introducción a VHDL y su aplicación en FPGAs

- Tema 1: Estructura de un fichero VHDL
- Tema 2: Describiendo la funcionalidad
- Tema 3: Diseño de circuitos síncronos
- Tema 4: Simulación con testbenches

Conocimientos previos requeridos:

- Electrónica digital básica (puertas lógicas y biestables)
- Tema 1: Estructura de un fichero VHDL

Objetivos de aprendizaje

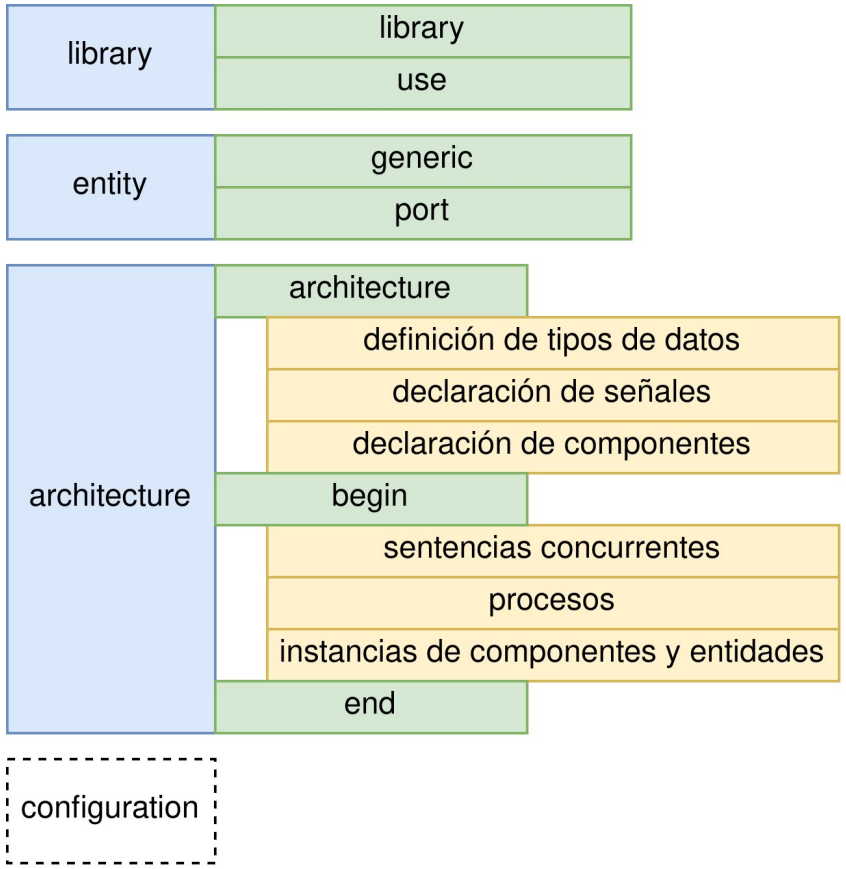
- Entender cómo se describe la funcionalidad en un fichero VHDL
- Comprender el concepto de concurrencia en VHDL
- Revisar las palabras clave más utilizadas y en qué contextos pueden usarse
- Aprender a describir circuitos combinacionales de diferentes maneras
- Conocer el uso de la sentencia `process`
- Aprender qué es la instancia de componentes y entidades y cuándo utilizar cada una

Contenido

- Repaso
- Descripción de una arquitectura
- Concurrencia
- Process
- Diseño de circuitos combinacionales
- Latches
- Ejemplo de diseño combinacional
- Instancias de componentes y entidades

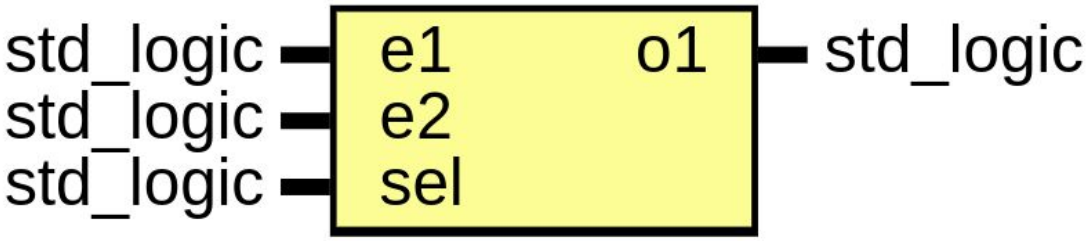
Repaso

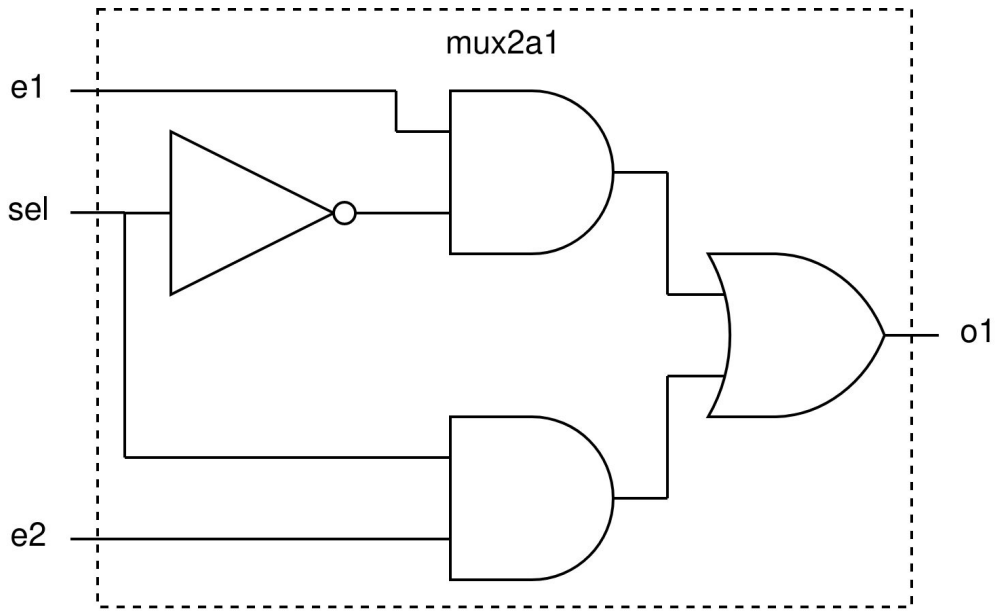
- Library
  - Inclusión de librerías y paquetes
- Entity
  - Descripción de caja negra
- Architecture
  - Descripción de la funcionalidad
- Configuration
  - Normalmente no se utiliza



Repaso

```vhdl
-- Multiplexor de dos entradas
library ieee;
use ieee.std_logic_1164.all;

entity mux2a1 is
  port (
    e1 : in  std_ulogic;
    e2 : in  std_ulogic;
    sel: in  std_ulogic;
    o1 : out std_ulogic
  );
end mux2a1;

architecture mux2a1_arch of mux2a1 is
begin
  process(e1,e2, sel)
  begin
    if (sel='0') then
      o1 <= e1;
    else
      o1 <= e2;
    end if;
  end process;
end mux2a1_arch;
```





Describir una arquitectura

3 formas distintas:

- Órdenes concurrentes
- Procesos (`process`)
- Reutilizando `entities` o `components` ya desarrollados

Se pueden combinar como se deseen en la misma `architecture`

Concepto de concurrencia

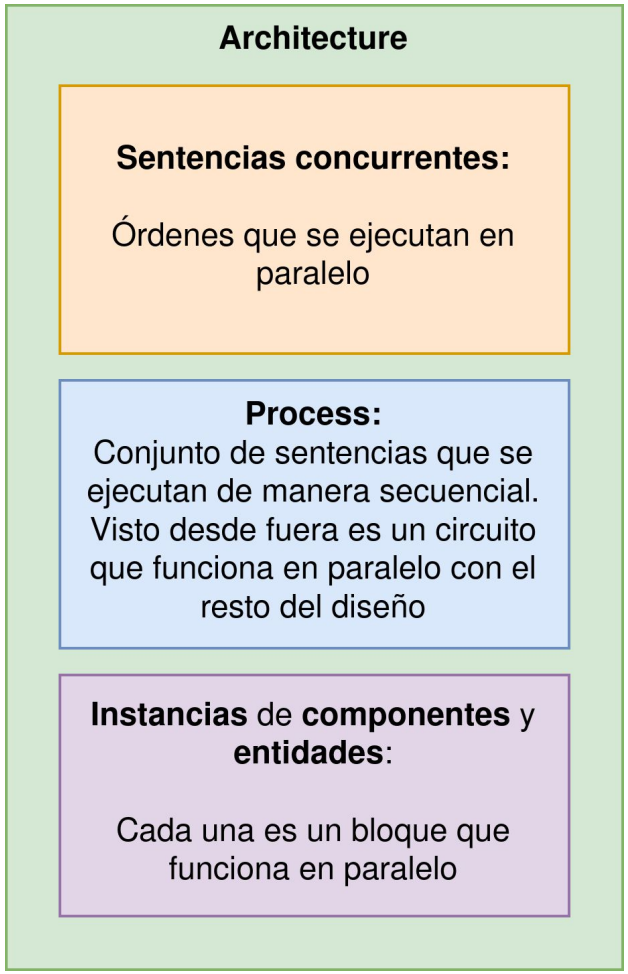
Diccionario de la lengua española:

Concurrencia: *"Coincidencia, concurso simultáneo de varias circunstancias."*

Sinónimos: *coincidencia, conjunción, convergencia, confluencia, simultaneidad, concomitancia.*

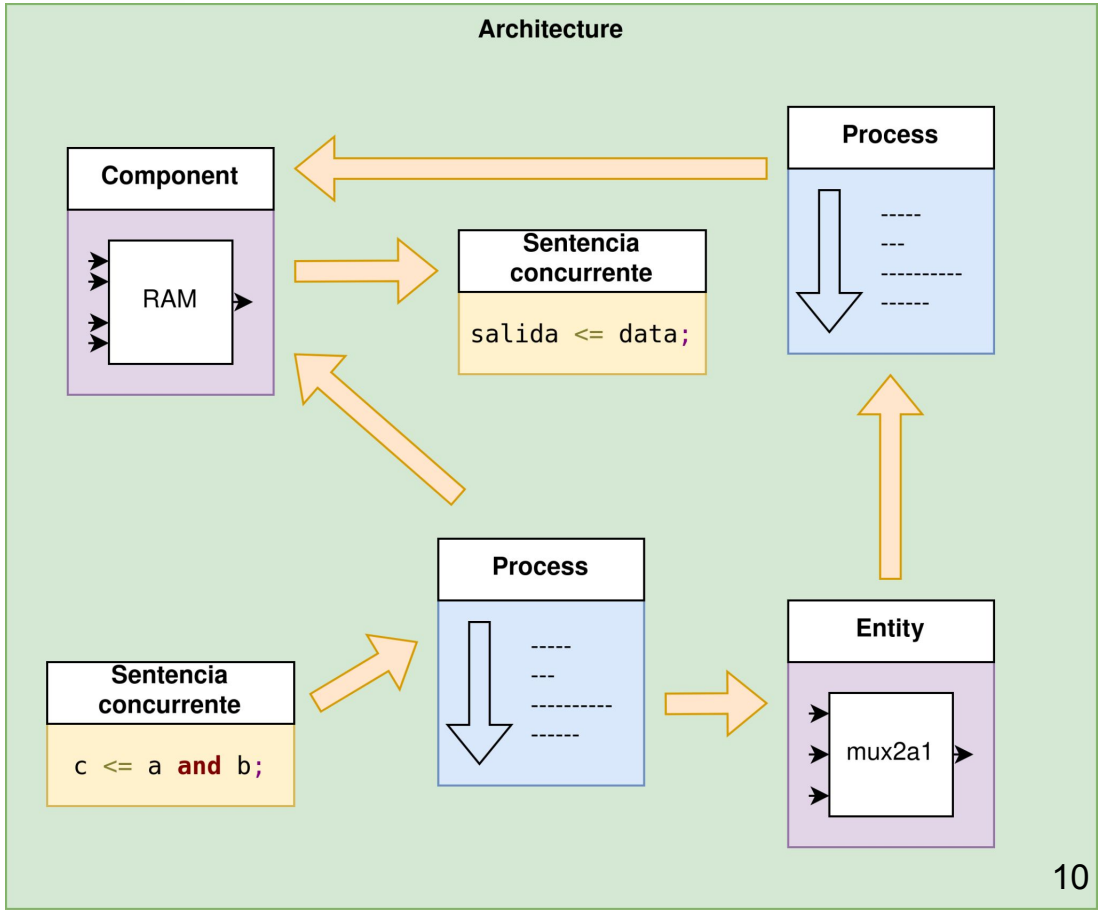
Concurrencia en una arquitectura VHDL

- Por naturaleza, todos los elementos de un circuito funcionan de forma concurrente
- Todos los elementos de la arquitectura se ejecutan de forma paralela
- El orden de las sentencias no es importante
- Una instancia de componente o de entidad se entiende como otra sentencia concurrente
- Un process se comporta externamente como una sentencia concurrente



Concurrencia en una arquitectura VHDL

- Todas las órdenes dentro de un process se interpretan de manera secuencial, pero externamente se comporta como un circuito concurrente más
- Los diferentes elementos se comunican utilizando señales (`signal`)

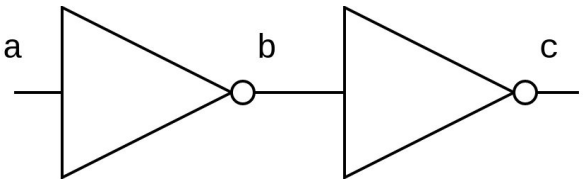


Process: ¿Por qué utilizarlos?

- Si bien en un circuito todo funciona de manera concurrente, no queremos hacerlo todo con sentencias concurrentes
- A medida que crece la complejidad del circuito el resultado se volvería inmanejable

```
b <= not a;
c <= not b;
```

síntesis

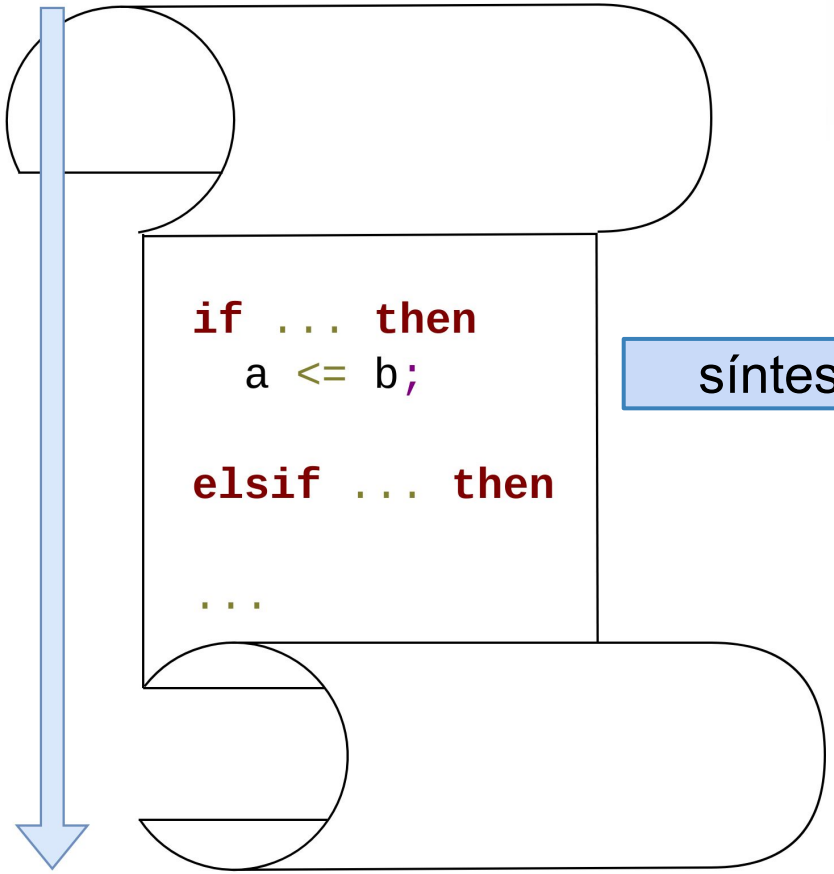


síntesis

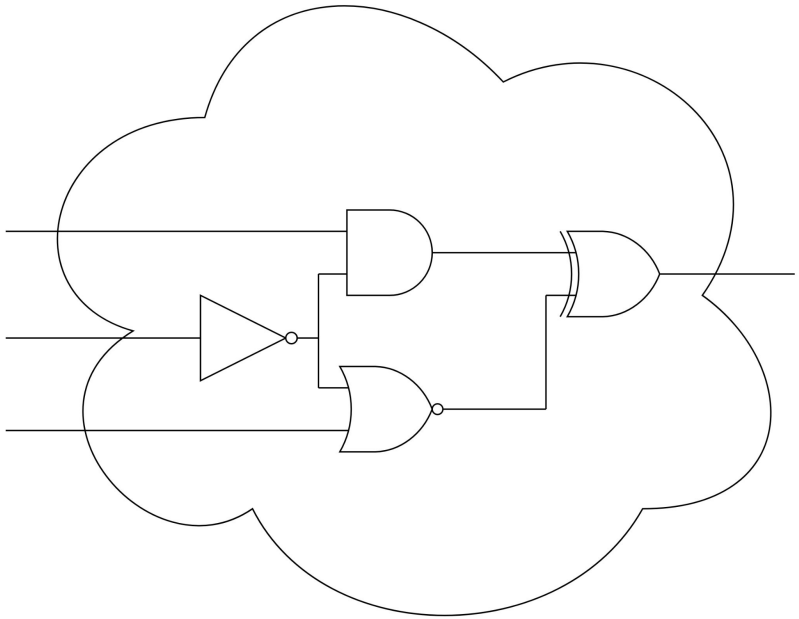
```
c <= not b;
b <= not a;
```

- Imagina un fichero con 1000 sentencias concurrentes: ¿por dónde empezamos a leer?

Process: ¿Qué son?



síntesis



Secuencia de sentencias que se convierte en un circuito

¿Qué son?

Process: "Una sentencia concurrente que describe comportamiento. Un process es en sí mismo una sentencia concurrente, pero contiene sentencias secuenciales, que se ejecutan en serie de arriba abajo. Un proceso se ejecuta en tiempos controlados por la lista de sensibilidad o por sentencias wait."

- The VHDL Golden Reference Guide, Doulos

Unidad básica de ejecución

La etiqueta es opcional, pero muy recomendable cuando tenemos más de un process

Lista de sensibilidad

```
nombre: process(e1, e2, sel)
  variable ..............
begin
  if (sel='0') then
    o1 <= e1;
  else
    o1 <= e2;
  end if;
end process;
```

Declaración de variables (opcional)

Comportamiento

Unidad básica de ejecución

```
nombre: process(e1,e2, sel)
  variable ..................
begin
  if (sel='0') then
    o1<= e1;
  else
    o1<= e2;
  end if;
end process;
```

- Todos los bloques `process` (salvo una excepción que veremos en el próximo tema) son sensibles a sus entradas, al igual que las puertas lógicas son sensibles a sus entradas
- Todos los bloques `process` se ejecutan en paralelo, como las puertas lógicas de un circuito

Lista de sensibilidad

- El proceso se ejecuta en su totalidad cuando cambia cualquier señal de su lista de sensibilidad

```
proc1: process (a,b,c)
begin
  x <= a and b and c;
end process;
```

```
proc2: process (a,b)
begin
  x <= a and b and c;
end process;
```

- Desde el punto de vista de simulación, estos dos procesos son muy diferentes
- El proceso proc2 no es sintetizable (faltaría un latch para c que fuera sensible a ambos flancos de a y b), por tanto, dependiendo del software utilizado:
  - El programa de síntesis añadirá la señal c a la lista de sensibilidad para poder sintetizar el circuito, dando un warning al usuario, o bien
  - El programa de síntesis producirá un mensaje de error

Lista de sensibilidad

- **Conclusión**: en un código VHDL, para síntesis de un proceso combinacional, en la lista de sensibilidad deben incluirse todos los objetos (señales y puertos) que afecten a la evaluación del proceso
  - Todos los objetos que estén dentro de la condición de sentencias `if` / `elsif` o `case`
  - Todos los objetos que estén a la derecha de una asignación

- Un proceso sin lista de sensibilidad se estaría ejecutando siempre, por tanto el simulador no avanzaría

- Si una señal de la lista de sensibilidad cambia el proceso es evaluado de nuevo, sucesivamente hasta que todas las señales/puertos a las que es sensible se estabilicen

Asignación de señales

- Todas las expresiones de un proceso se resuelven con el **valor actual** de las señales (o puertos)
  - Entendido como el valor que tienen cuando entramos en el process
- Las asignaciones a señales y puertos se hacen efectivas **cuando termina** la evaluación del proceso
  - Es decir, cuando se llega al `‘end process’`
  - También se hacen efectivas si se llega a una sentencia `‘wait’`, pero estas no se suelen usar en VHDL sintetizable
    - (Más información en el Tema 4)
- Todas las señales serán actualizadas con el **último valor asignado** dentro del proceso
- El proceso vuelve a evaluarse si se ha modificado alguna señal de su lista sensible
- Todos los procesos cuyas señales a las que es sensible hayan sido modificadas y todas las órdenes concurrentes se ejecutan hasta que se alcanza un estado de equilibrio
  - Es decir, hasta que no haya más cambios

Asignación de señales

Diferentes opciones para implementar un mux2a1:

```
process(e1,e2, sel)
begin

  if (sel='1') then
    o1<= e2;
  else
    o1<= e1;
  end if;

end process;
```

```
process(e1, e2, sel)
begin

  o1<= e1;
  if (sel='1') then
    o1<= e2;
  end if;


end process;
```

Uso de un valor por defecto

```
process(e1, e2, sel)
begin

  if (sel='1') then
    o1<= e2;
  end if;
  o1<= e1;


end process;
```

o1 siempre será igual a e1

Diseño de circuitos combinacionales

Dos opciones:

- Utilizando sentencias concurrentes
- Utilizando procesos

Se pueden combinar!

Sentencias concurrentes

Sentencias concurrentes:

- Asignaciones: `b <= a;`
- Operaciones lógicas: `c <= a and (not b);`
- Operaciones aritméticas: `f <= d + e;`
- `when … else`
- `with … select`

Asignación simple

```
obj_destino <= obj_origen;
```

- Deben ser del mismo tipo (`integer`, `std_ulogic`, etc.)
  - Recordemos que VHDL es de tipado duro
- `obj_destino` puede ser:
  - Un **`signal`**
  - Un **`port`** de dirección **`out`**, **`inout`** (o **`buffer`**)
- `obj_origen` puede ser:
  - Un **`signal`**
  - Un **`port`** de dirección **`in`**, **`inout`** (o **`buffer`**)

- Pueden usarse **dentro y fuera de un process**

Operaciones lógicas y aritméticas

```
obj_destino <= expresión(objetos_origen);
```

- La expresión debe tomar un valor compatible con el tipo de dato de `obj_destino`

Ejemplos:

- `b <= not a;`
- `f <= (c and (not d)) or e;`
- `j <= (g + h)*i;`
- `c <= a and (xnor b);`
- `f <= d + e;`

- Pueden usarse **dentro y fuera de un process**

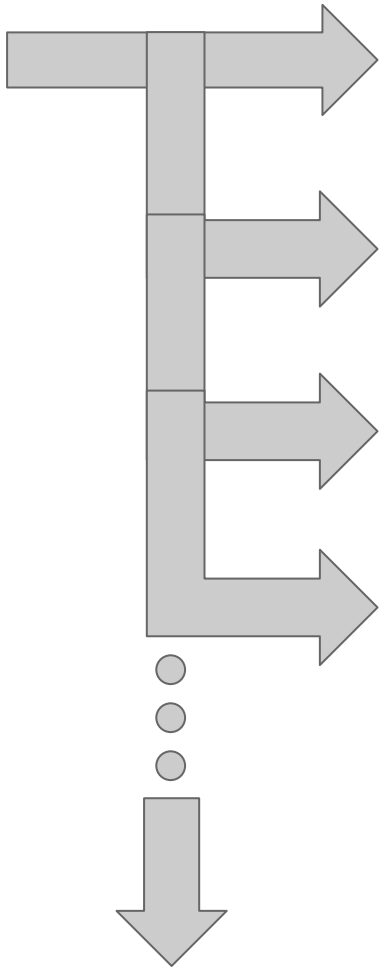
Ejemplo: mux2a1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux2a1 is
  port ( e1:  IN std_ulogic;
         e2:  IN std_ulogic;
         sel: IN std_ulogic;
         o1: OUT std_ulogic);
end mux2a1;

architecture mux2a1_arch of mux2a1 is
begin

  o1 <= (e1 and not(sel)) or (e2 and sel);

end mux2a1_arch;
```

Sentencias concurrentes

When… else:

```
d <= (not a) when e="01" else
           b when e="10" else
           c;

obj1 <= expr1 when cond1 else
        expr2 when cond2 else
        <...>
        exprN;
```



- No pueden usarse **dentro de un process**

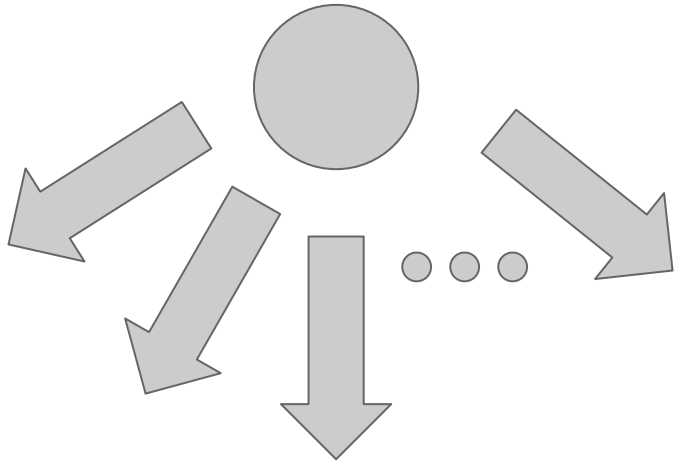
Ejemplo: mux2a1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux2a1 is
  port ( e1:  IN std_ulogic;
         e2:  IN std_ulogic;
         sel: IN std_ulogic;
         o1: OUT std_ulogic);
end mux2a1;

architecture mux2a1_arch of mux2a1 is
begin

  o1 <= e1 when sel= '0' else e2;

end mux2a1_arch;
```

Sentencias concurrentes

With… select:

```
with e select
  d <= not a when "01",
       b when "10",
       c when others;

with obj1 select
  obj2 <= expr1 when value1,
          expr2 when value2,
          <...>
           expr3 when others;
```



- No pueden usarse **dentro de un process**

Ejemplo: mux2a1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux2a1 is
  port ( e1:  IN std_ulogic;
         e2:  IN std_ulogic;
         sel: IN std_ulogic;
         o1: OUT std_ulogic);
end mux2a1;

architecture mux2a1_arch of mux2a1 is
begin
  with sel select
  o1 <=
    e1 when '0',
    e2 when others;
end mux2a1_arch;
```

Procesos

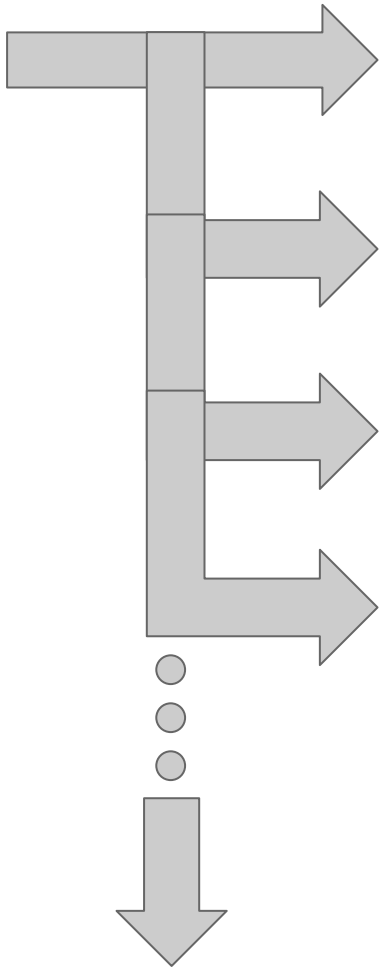
Procesos:

- Asignaciones: `b <= a;`
- Operaciones lógicas: `c <= a and (not b);`
- Operaciones aritméticas: `f <= d + e;`
- **if**… **elsif** … **else**
- **case** … **when**

Procesos

If… elsif… else:

```
if (rst_sync = '1') then
  p_cont <= (others=>'0');
elsif (enable = '1') then
  p_cont <= cont + 1;
else
  p_cont <= cont;
end if;
```

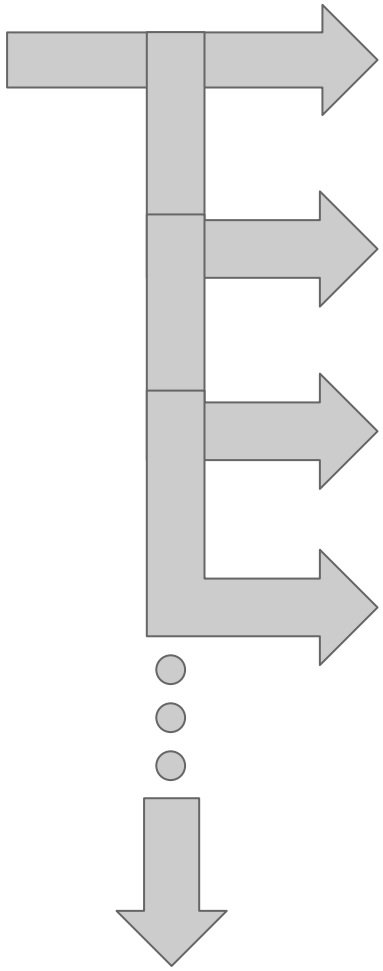


- No puede usarse **fuera de un process**

Procesos

If… elsif… else:

```
if (cond1) then
  <sentencias>
elsif (cond2) then
  <sentencias>
<... más elsif ...>
else
  <sentencias>
end if;
```



- No puede usarse **fuera de un process**

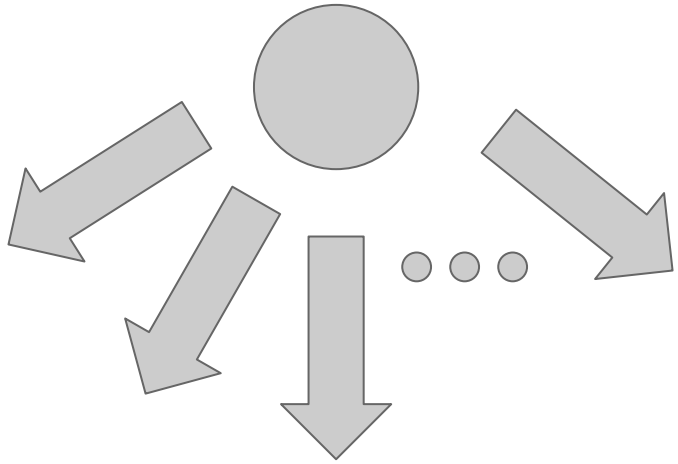
Ejemplo: mux2a1

```
architecture mux2a1_arch of mux2a1 is
begin

  process(e1,e2, sel)
  begin
    if (sel='0') then
      o1 <= e1;
    else
      o1<= e2;
    end if;
  end process;

end mux2a1_arch;
```

Procesos

Case… when:

```
case state is
    when idle =>
      <sentencias>
    when count =>
      <sentencias>
    when header =>
      <sentencias>
    when others =>
      <sentencias>
end case;
```



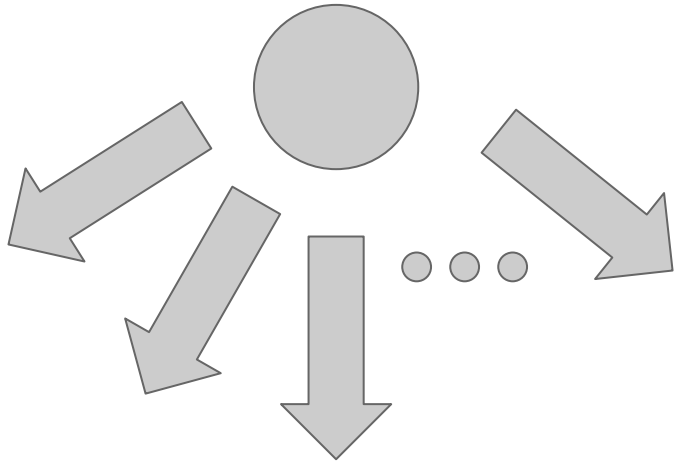
Muy utilizado para describir máquinas de estados

- No puede usarse **fuera de un process**

Procesos

Case… when:

```
case obj1 is
    when value1 =>
      <sentencias>
    when value2 =>
      <sentencias>
    when value3 =>
      <sentencias>
    when others =>
      <sentencias>
end case;
```



Muy utilizado para describir máquinas de estados

- No puede usarse **fuera de un process**

Ejemplo: mux2a1

```
architecture mux2a1_arch of mux2a1 is
begin

  process(e1,e2, sel)
  begin
    case sel is
      when '0' =>
        o1 <= e1;
      when others =>
        o1 <= e2;
    end case;
  end process;

end mux2a1_arch;
```

`when others` es obligatorio si no están cubiertos todos los casos

No olvidemos que al ser `std_ulogic` puede tomar 9 valores posibles y no sólo 2

Resumen

		Fuera de un `process`	**Dentro** de un `process`
Asignación	<=, :=	✓	✓
Operaciones lógicas	**and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**	✓	✓
Operaciones aritméticas	+, -, *, /	✓	✓
Decisión con prioridad	**when** ... **else**	✓	✗
	if ... **elsif** ... **else**	✗	✓
Decisión no solapante	**with** ... **select**	✓	✗
	case ... **when**	✗	✓

Generación de latches

- Los latches se generan cuando se deja una sentencia condicional incompleta

```
process(a,b)
 begin
   if a = '0' then
     f <= '0';
     if b = '1' then
       f <= '1';
     end if;
   end if;
 end process;
```

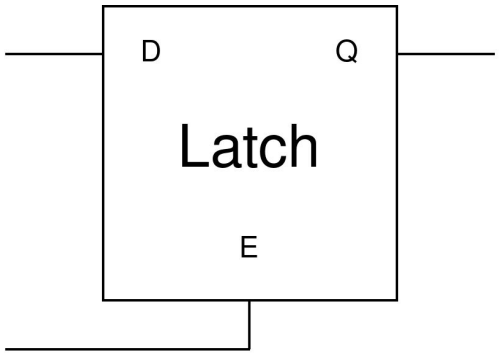
ab	f
00	0
01	1
10	?
11	?

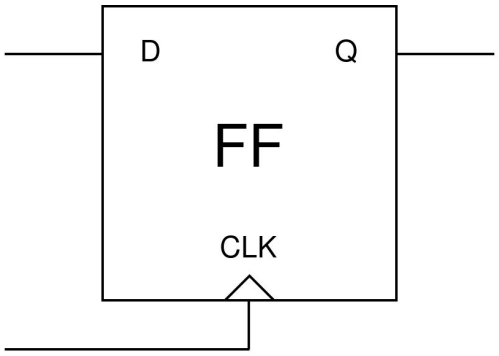
Sentencias condicionales incompletas

- No se ha definido qué ocurre si: `a = '1';`
- Al igual que si fuera software, se intenta mantener el valor anterior
- Pero como las puertas lógicas **no tienen memoria**, es necesario instanciar un elemento de memoria para mantener ese valor
- Y como en el proceso no hay señal de reloj, se genera un biestable activo por nivel (**latch**)

Latches vs flip-flops

- Latch = biestable activo por nivel
  - Normalmente no se quiere en diseño digital porque puede dar (y da) muchos problemas de temporización
- Flip-flop (FF) = biestable activo por flanco
  - Estos sí se usan muchísimo en diseño digital





Latch(es) = problemas

- Hay que tratar los warnings tipo *“latch inferred”*, *“found N-bit latch”* como **errores**
  - Simulation mismatch
  - No funciona luego al configurar la FPGA
  - Peor caso: funciona casi siempre en la FPGA pero cada X millones de ciclos de reloj falla algún dato suelto
    - Buena suerte depurando eso
- En algunos casos sí se usan
  - Sólo si tenéis muy claro por qué hay que usarlos!

¿Cómo arreglamos el latch?

Asegurándonos de que se cubren todos los casos posibles en la lógica:

```
process(a,b)
 begin
   if a = '0' then
     f <= '0';
     if b = '1' then
       f <= '1';
     end if;
   else
     f <= '0';
   end if;
 end process;
```

ab	f
00	0
01	1
10	0
11	0

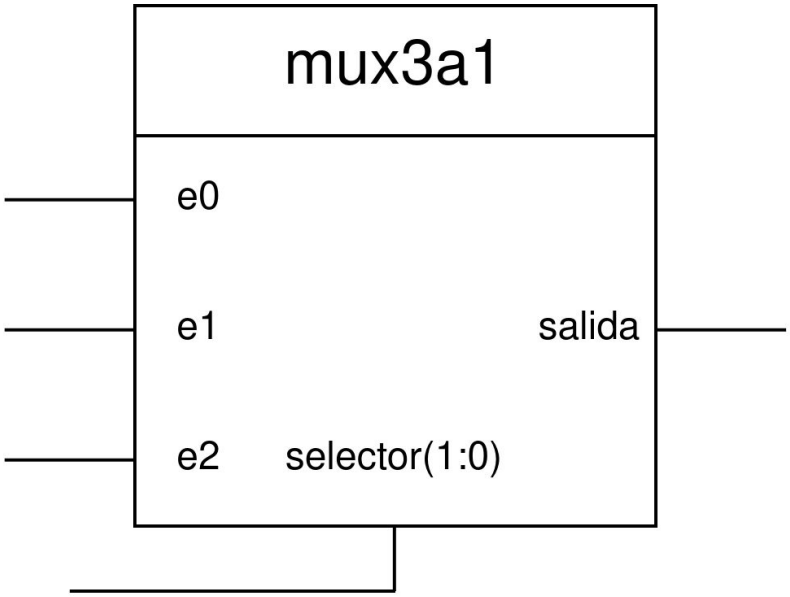
```
process(a,b)
 begin
   f <= '0';
   if (a = '0') and (b = '1') then
     f <= '1';
   end if;
 end process;
```

El uso de un valor por defecto suele hacer el código más simple y legible

Ejemplo: mux3a1

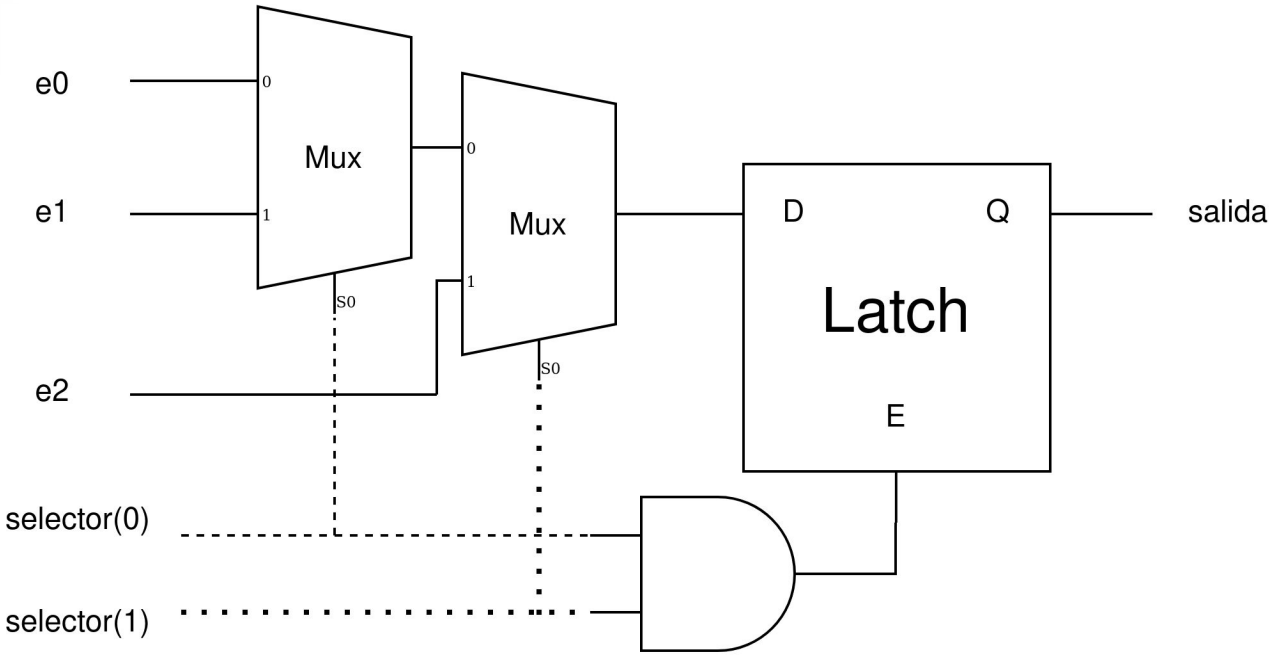
```
process(a,b)
 begin
   if (selector = "00") then
     salida <= e0;
   elsif (selector = "01") then
     salida <= e1;
   elsif (selector = "10") then
     salida <= e2;
   end if;
 end process;
```

¿Tiene este código algún problema?

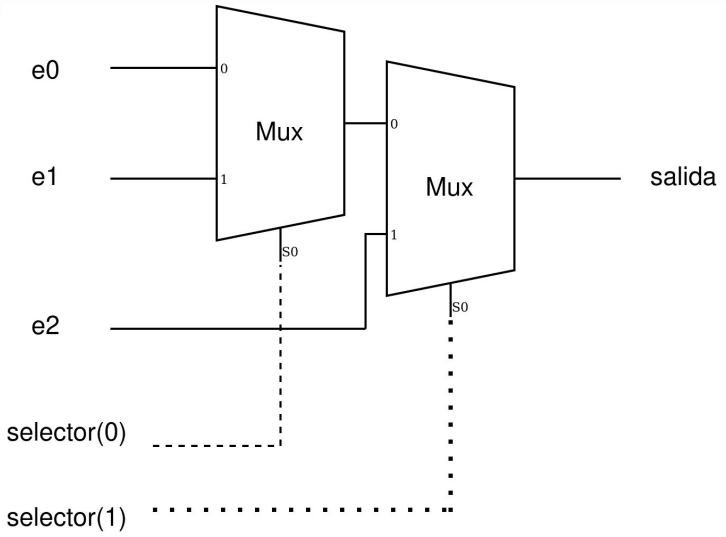


Ejemplo: mux3a1

- Problema: no se define qué ocurre cuando `selector = "11"`
- Resultado: el sintetizador infiere un latch para ese caso
- El circuito con el latch:
  - Ocupa más área
  - No realiza la función lógica que esperamos
  - ¡Es sensible a pulsos (glitches)!



Ejemplo: mux3a1



Solución: asegurarse de que definimos **todos** los casos

```
process(a,b)
 begin
   if (selector = "00") then
     salida <= e0;
   elsif (selector = "01") then
     salida <= e1;
   elsif (selector = "10") then
     salida <= e2;
   else
     salida <= e2;
   end if;
 end process;
```

simplificar

```
process(a,b)
 begin
   if (selector = "00") then
     salida <= e0;
   elsif (selector = "01") then
     salida <= e1;
   else
     salida <= e2;
   end if;
 end process;
```

Conclusiones

- Los circuitos combinaciones deben estar completamente especificados
- Se genera un latch siempre que no se especifica el valor de la salida para todos los posibles valores de las entradas
- Un latch es un circuito muy sensible a pulsos y temporización (timing)
  - Sólo debe ser usado en casos concretos

Ejemplo de diseño combinacional

Contador de bits activos

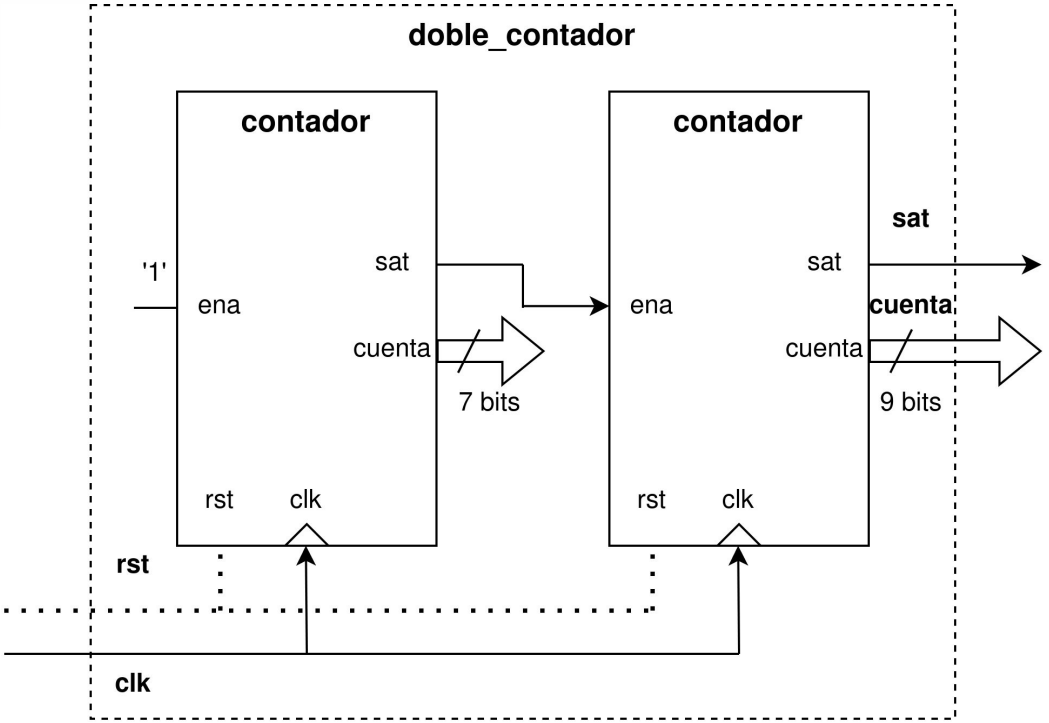
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity bitcounter is
  port (
    input  : in  std_ulogic_vector(2 downto 0);
    output : out unsigned(1 downto 0)
  );
end bitcounter;

architecture bitcounter_arch of bitcounter is
begin

  process(input)
  begin
    case input is
      when "000" =>
        output <= to_unsigned(0,2);
      when "001" =>
        output <= to_unsigned(1,2);
      when "010" =>
        output <= to_unsigned(1,2);
      when "011" =>
        output <= to_unsigned(2,2);
      when "100" =>
        output <= to_unsigned(1,2);
      when "101" =>
        output <= to_unsigned(2,2);
      when "110" =>
        output <= to_unsigned(2,2);
      when "111" =>
        output <= to_unsigned(3,2);
      when others =>
        output <= to_unsigned(0,2);
    end case;
  end process;

end bitcounter_arch;
```

Interconexión de componentes y entidades

- Entidades son literalmente `entity` VHDL que reutilizamos
- Componentes (`component`) pueden ser entidades VHDL o bien estar descritos en otro lenguaje (Verilog), ser IP cores de los cuales no tenemos el código fuente, o incluso ser primitivas de la FPGA
- Estos elementos suelen conectarse entre sí utilizando `signals`

Instancia de componentes

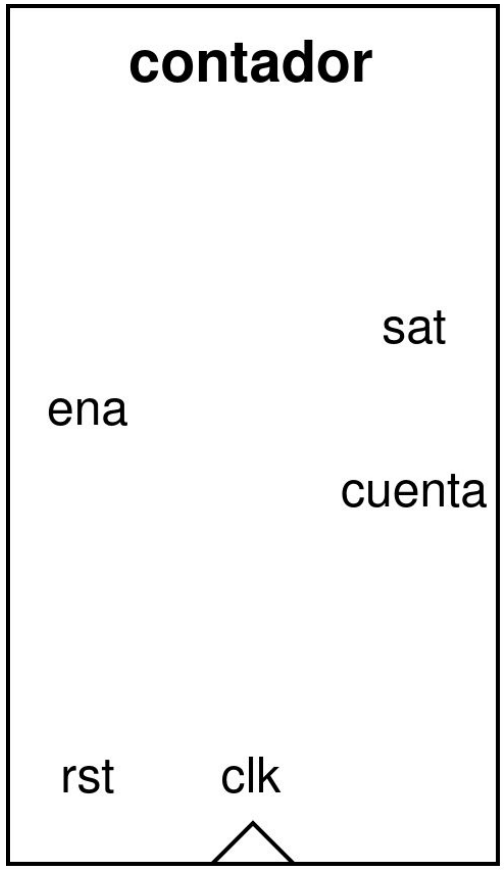
Supongamos que queremos hacer este diseño



Instancia de componentes

Previamente debemos haber diseñado el nivel de jerarquía inferior (`contador`)

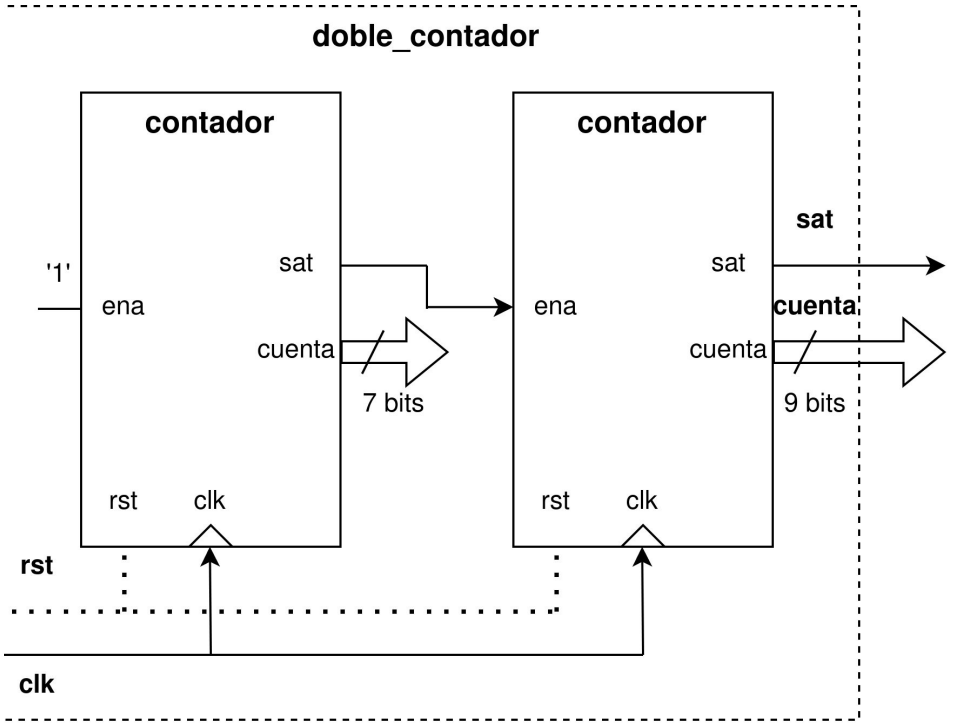
```
entity contador is
  generic (N: integer := 10);
  port (
    clk    : in  std_ulogic;
    rst    : in  std_ulogic;
    ena    : in  std_ulogic;
    cuenta : out unsigned(N-1 downto 0);
    sat    : out std_ulogic
);
end contador;

architecture contador_arch of contador is
begin
. . . . .
end contador;
```



Instancia de componentes

Creamos la entidad de nivel superior (`doble_contador`)

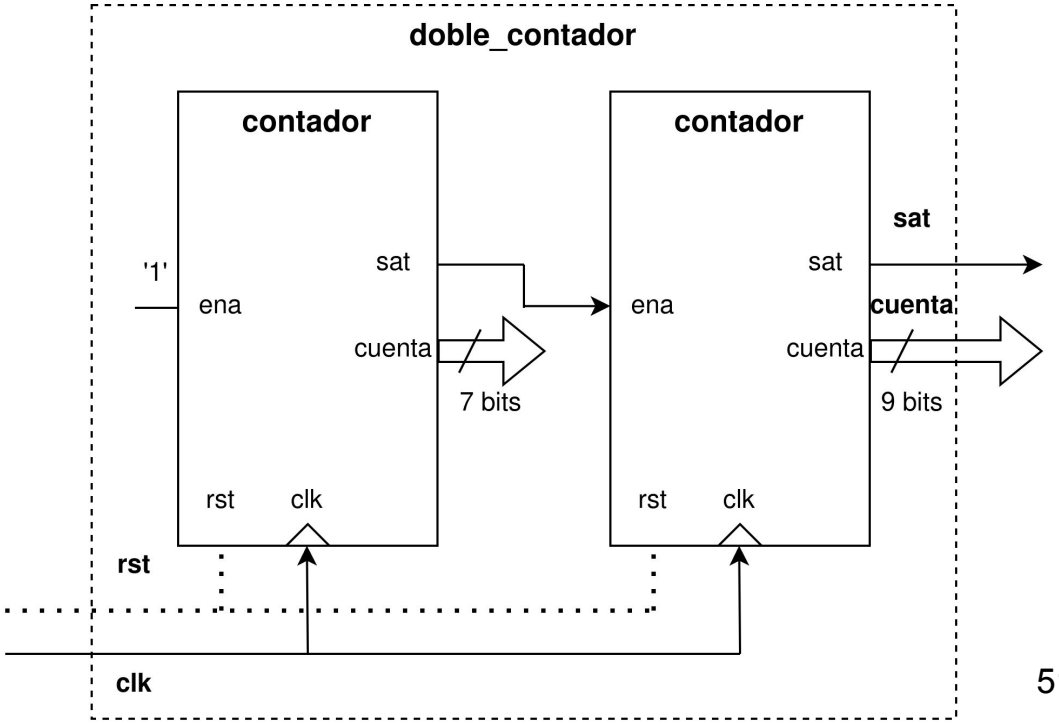
```
entity doble_contador is
  port (
    clk    : in  std_ulogic;
    rst    : in  std_ulogic;
    cuenta : out unsigned(8 downto 0);
    sat    : out std_ulogic);
end doble_contador;
```



Instancia de componentes

Antes del `begin` de la arquitectura, declaramos las señales necesarias (cables que nos salen a exterior, técnicamente: objetos que no son ports)

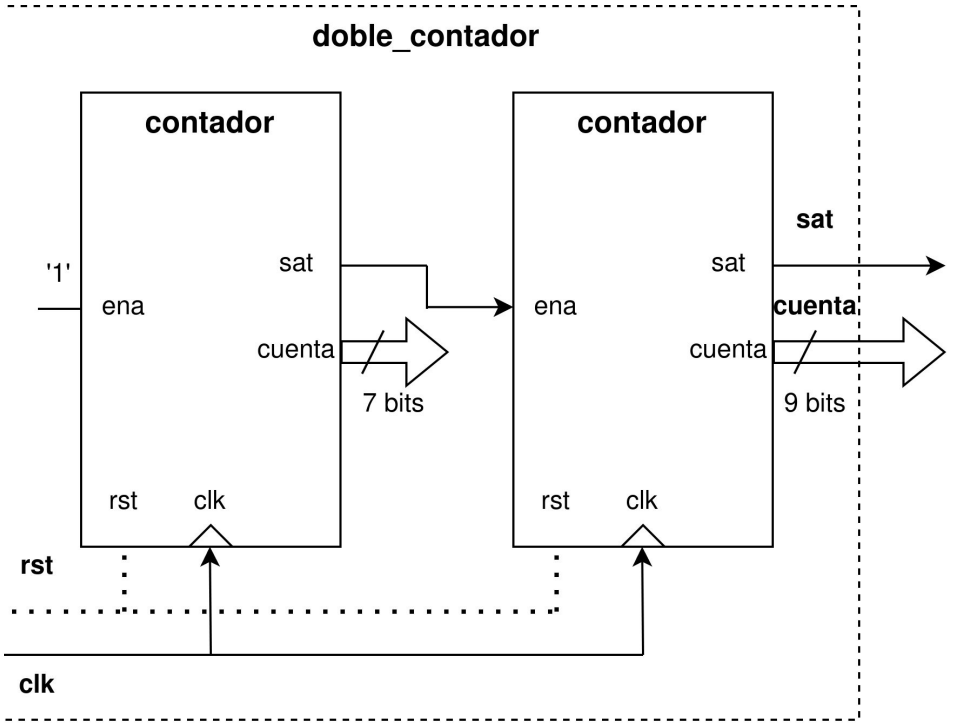
```
architecture doble_contador_arch of doble_contador is
  signal s_enable, s_uno: std_ulogic;
...
```



Instancia de componentes

Antes del **begin** de la arquitectura, declaramos `contador` como **component**

```
component contador is
  generic (N: integer := 10);
  port (
    clk     : in  std_ulogic;
    rst     : in  std_ulogic;
    ena     : in  std_ulogic;
    cuenta  : out unsigned(N-1 downto 0);
    sat     : out std_ulogic
);
end component;
```

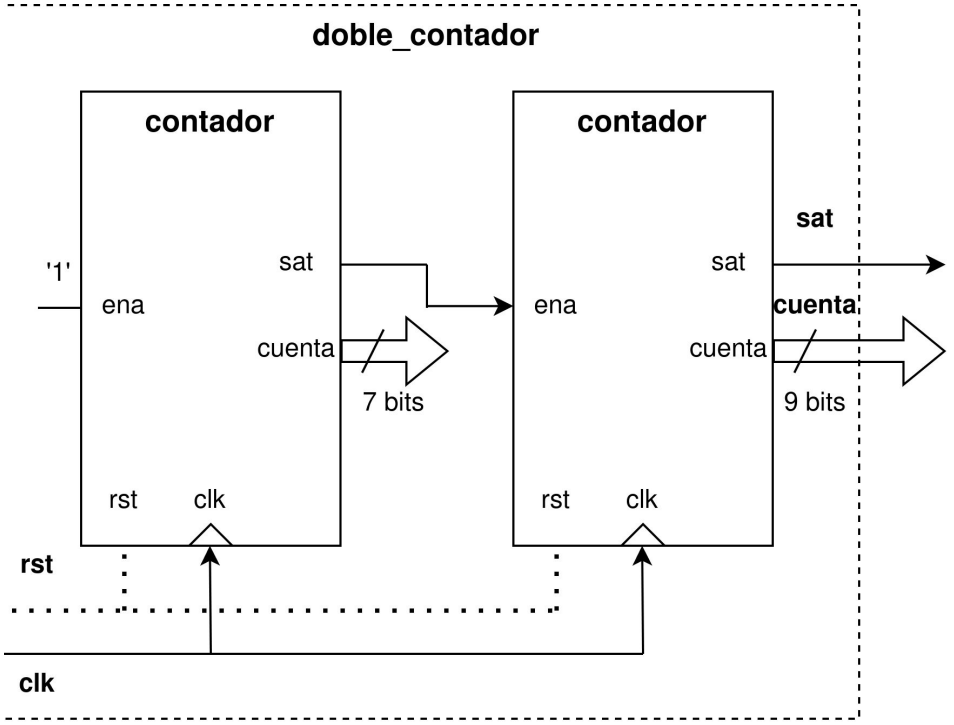


Las secciones **generic** y **port** son copia del **entity**

Instancia de componentes

Después del **begin** de la arquitectura, realizamos la instancia del primer `contador`

```
begin

  s_uno<='1';

  cont1: contador
    generic map  (N=>7)
    port map (
      clk    => clk,
      rst    => rst,
      ena    => s_uno,
      cuenta => open,
      sat    => s_enable
    );
```



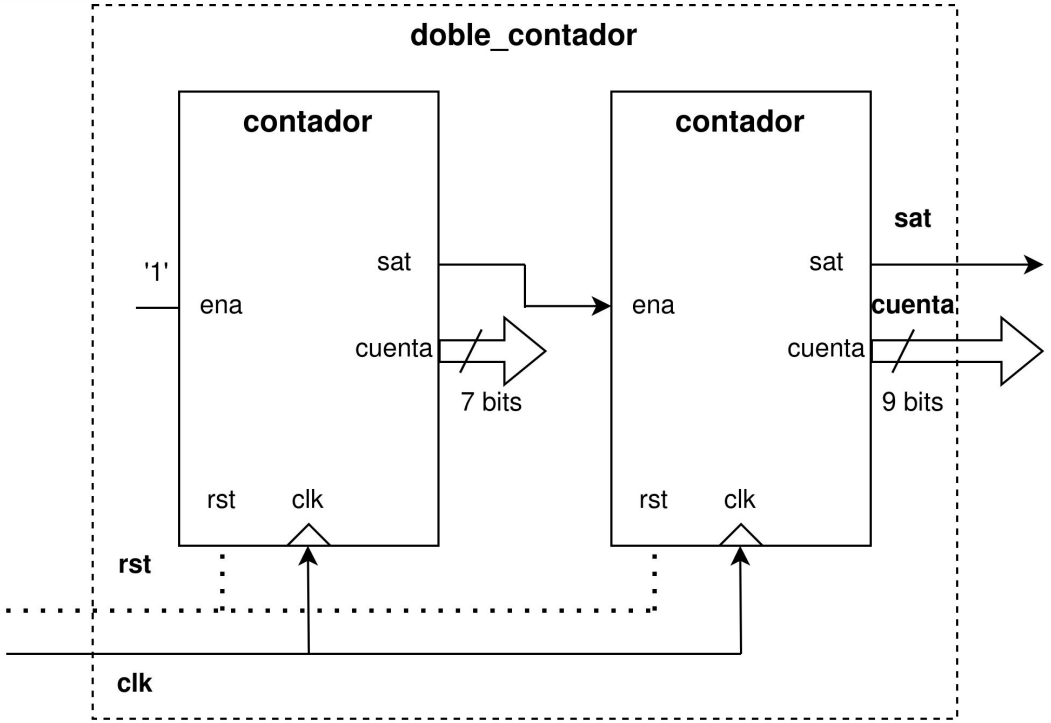
Sintaxis: `port_del_component => objeto_del_top`

También hemos puesto `s_uno` a nivel alto usando una sentencia concurrente

Instancia de componentes

Después del **begin** de la arquitectura, realizamos la instancia del segundo `contador`

```
cont2: contador
  generic map  (N=>9)
  port map (
    clk    => clk,
    rst    => rst,
    ena    => s_enable,
    cuenta => cuenta,
    sat    => sat
  );
```



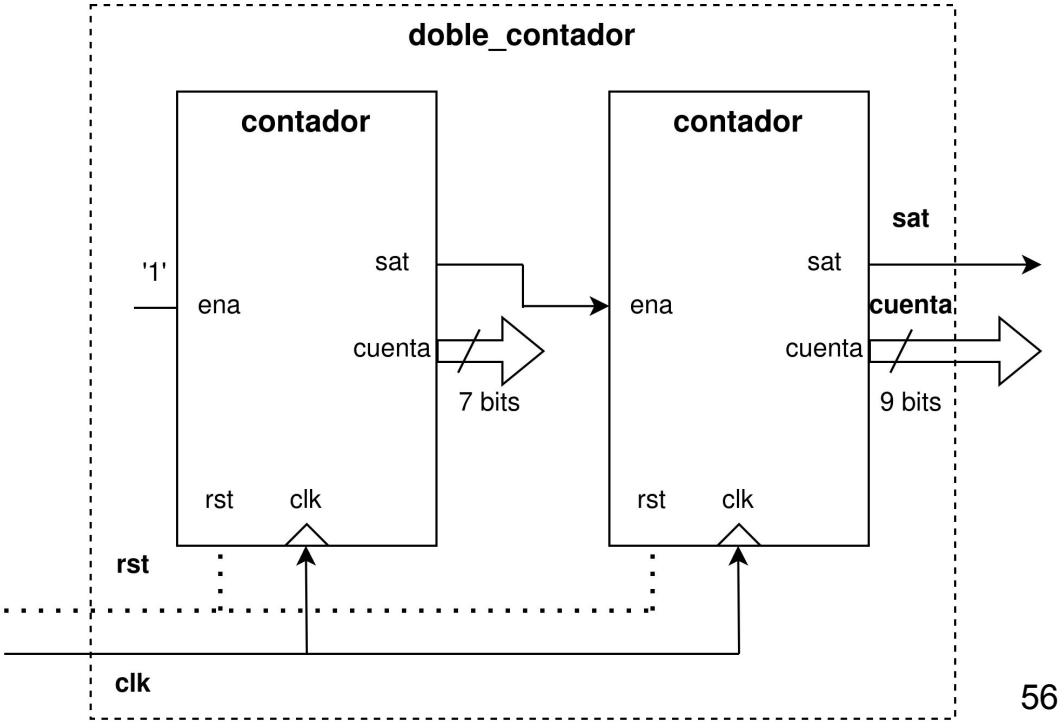
Instancia de entidades

- Podemos instanciar entidades (`entity`) directamente, sin necesidad de declararlas como `component`
- La única restricción es que realmente deben ser entidades **descritas en VHDL**
- Para instanciar módulos Verilog o IP cores que **no sean VHDL**, siempre será necesario declararlos como **componentes**

Instancia de entidades

- **No es necesario** declararlas como `component`, simplemente se instancian directamente
- **Opción recomendada** para entidades VHDL: reducimos **duplicidad de código** y evitamos tener que propagar cambios al `component`

```
cont2: entity work.contador
  generic map  (N=>9)
  port map (
    clk    => clk,
    rst    => rst,
    ena    => s_enable,
    cuenta => cuenta,
    sat    => sat
  );
```



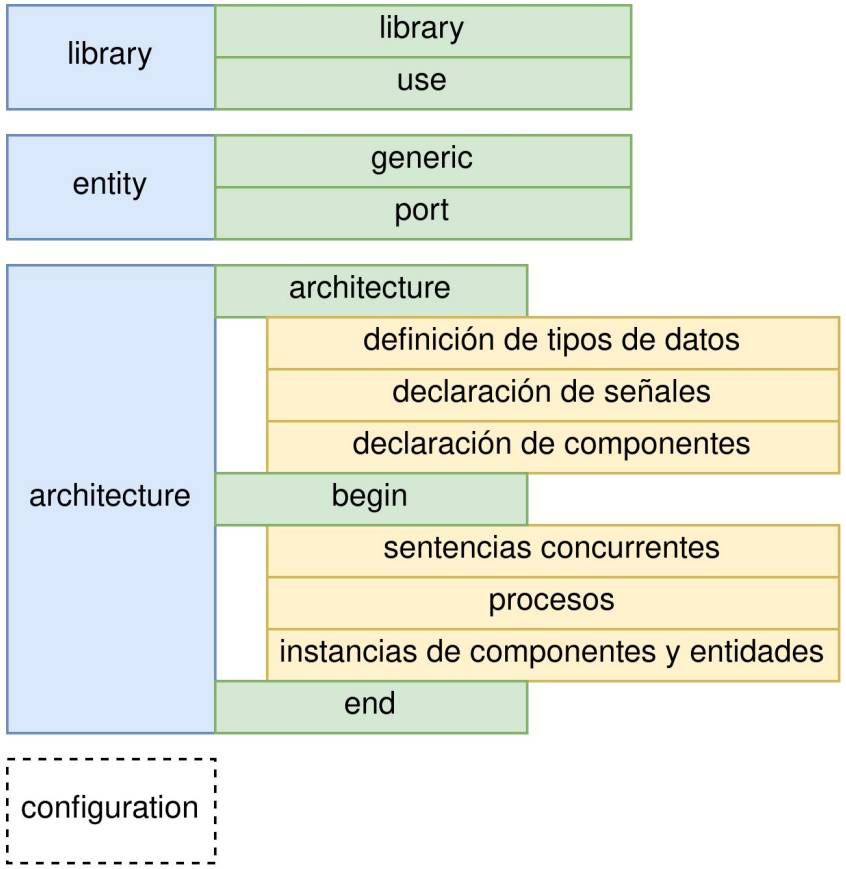
Tarea

Describir un **multiplexor 4 a 1** de las siguientes formas:

1. Con una única sentencia concurrente utilizando operaciones lógicas (`and`, `not`, `or`, etc…)
2. Utilizando `if … elsif … else` (en un `process`)
3. Utilizando `case … when` (en un `process`)
4. Utilizando `when … else` (fuera de un `process`)
5. Utilizando `with … select` (fuera de un `process`)
6. Instanciando 3 multiplexores 2 a 1

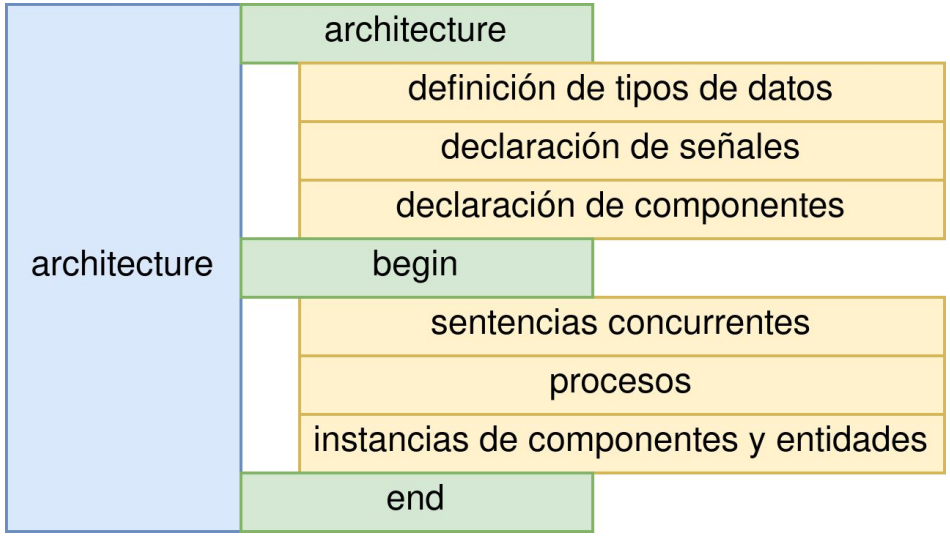
Resumen (I)

- Library
  - Inclusión de librerías y paquetes
- Entity
  - Descripción de caja negra
- Architecture
  - Descripción de la funcionalidad
- Configuration
  - Normalmente no se utiliza



Resumen (II)

- Antes del begin
  - Definimos tipos de datos si los necesitamos
  - Declaramos los signals necesarios para conectar los elementos del diseño
  - Declaramos los componentes que necesitemos instanciar
- Después del begin
  - Pocas sentencias concurrentes
  - Principalmente usamos procesos para describir la funcionalidad
  - Instanciamos componentes y entidades (reutilización de código)



Conclusiones

- La mayor parte del esfuerzo al describir VHDL se invierte en describir la funcionalidad, es decir lo que ocurre tras el `begin` del `architecture`
- Absolutamente todo lo que hay en un architecture VHDL funciona en paralelo, de manera concurrente
- El `process` es un mecanismo que nos permite escribir código secuencial (entendible) que luego será convertido por el sintetizador en hardware (paralelo)
- Para tomar decisiones en VHDL, principalmente se utiliza `if` … `elsif` … `else` y `case` … `when`, aunque es bueno saber que `when` … `else` y `with` … `select` existen por si los encontramos en códigos de terceros

Conclusiones

- La misma funcionalidad puede describirse de distintas maneras, usando distintas sentencias para la toma de decisiones
- Es muy importante especificar siempre los procesos combinaciones para evitar latches
- Se pueden reutilizar entidades completas, o componentes/IP de terceros realizando instancias de `entity` y `component`

Bibliografía

- Brian Mealy, Fabrizio Tappero, *Free Range VHDL*. Free Range Factory, 2018
- *The VHDL Golden Reference Guide.* Doulos, 1995
- Ricardo Jasinski, *Effective Coding with VHDL: principles and best practice.* The MIT Press, 2016

Resultados de aprendizaje

- Entender el concepto de concurrencia, tanto en general como en el contexto de una `architecture` VHDL
- Conocer qué es un process, qué nos aporta como diseñadores, y cómo funciona
- Entender el mecanismo de lista de sensibilidad de un process
- Conocer las sentencias más utilizadas para describir funcionalidad en un diseño VHDL
- Saber en qué contextos pueden utilizarse dichas sentencias y en cuáles no

Resultados de aprendizaje

- Saber cuándo se generan latches en VHDL, qué problemas producen y cómo arreglar el código para eliminarlos
- Conocer cómo se realizan instancias de componentes y entidades
- Saber determinar cuándo se debe reutilizar un módulo instanciándolo como `component`, y cuándo debe instanciarse como `entity`