

# Introducción a HDLs (VHDL)

---

Primer Cuatrimestre 2023

Diseño de Sistemas Digitales con FPGA  
DC - UBA

Conceptos generales

VHDL - Generalidades

Descripción de circuitos combinacionales

- Repaso de circuitos combinacionales

- Generación paramétrica 1: `generic`

- Escribiendo código combinacional secuencialmente: `process`

Descripción de circuitos secuenciales

- Repaso de circuitos secuenciales

- Escribiendo código secuencial: `process`

Más generalidades

Ejercicios

# Conceptos generales

---

Veremos que podemos tomar al menos tres perspectivas complementarias a la hora de construir hardware:

Veremos que podemos tomar al menos tres perspectivas complementarias a la hora de construir hardware:

- La perspectiva **comportamental**.

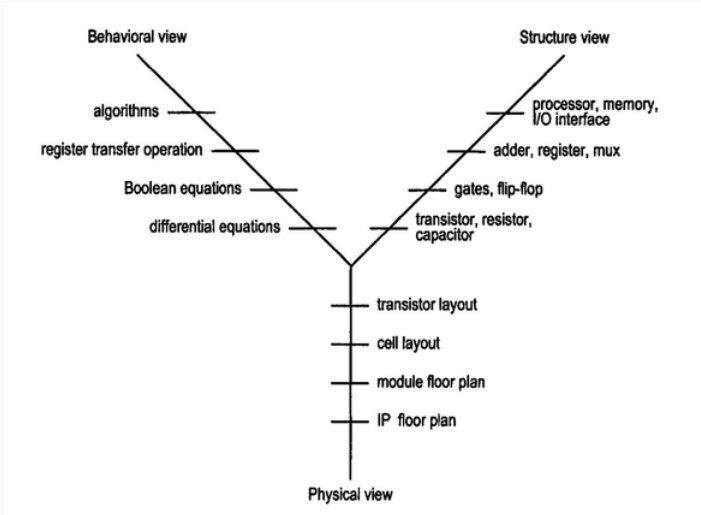
Veremos que podemos tomar al menos tres perspectivas complementarias a la hora de construir hardware:

- La perspectiva **comportamental**.
- La perspectiva **estructural**.

Veremos que podemos tomar al menos tres perspectivas complementarias a la hora de construir hardware:

- La perspectiva **comportamental**.
- La perspectiva **estructural**.
- La perspectiva **física**

# Tres perspectivas para construir hardware





¿Cómo podemos describir un mux con una salida de alta impedancia en **VHDL**?

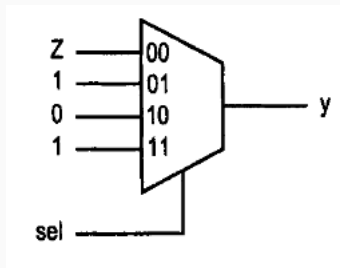
Ejemplo:

```
with sel select  
y <= 'Z' when "00",  
    '1' when "01"|"11",  
    '0' when others;
```

Ejemplo:

```
with sel select  
y <= 'Z' when "00",  
    '1' when "01"|"11",  
    '0' when others;
```

Síntesis:



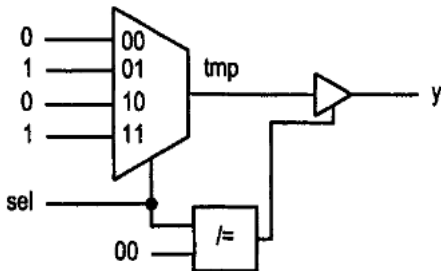
Ejemplo:

```
with sel select  
tmp <= '1' when "01"|"11",  
      '0' when others;  
y <= tmp when sel /="00" else  
  'Z';
```

Ejemplo:

```
with sel select  
tmp <= '1' when "01"|"11",  
      '0' when others;  
y <= tmp when sel /= "00" else  
  'Z';
```

Síntesis:



Los lenguajes de especificación de hardware (**HDL**) dan una descripción a nivel de transferencia de registros de un circuito (**RTL**).

Los lenguajes de especificación de hardware (**HDL**) dan una descripción a nivel de transferencia de registros de un circuito (**RTL**).

No es, por lo tanto, una descripción directa de la **implementación en términos de la lógica discreta a usar**.

Los lenguajes de especificación de hardware (**HDL**) dan una descripción a nivel de transferencia de registros de un circuito (**RTL**).

No es, por lo tanto, una descripción directa de la **implementación en términos de la lógica discreta a usar**.

La implementación del circuito va a realizarse a través de un proceso conocido como **síntesis**.



El proceso de síntesis va a *realizar* el diseño dado en la descripción de **HDL** sobre una librería de celdas específicas de la tecnología sobre la cuál vamos a construirlo.

El proceso de síntesis va a *realizar* el diseño dado en la descripción de **HDL** sobre una librería de celdas específicas de la tecnología sobre la cuál vamos a construirlo.

Básicamente es una proyección desde los bloques funcionales, dados en a nivel de transferencia de registros a las celdas de la tecnología.

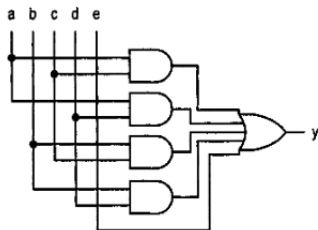
El problema de realizar un diseño es computacionalmente complejo. Esto lleva a que las soluciones dadas por la síntesis:

El problema de realizar un diseño es computacionalmente complejo. Esto lleva a que las soluciones dadas por la síntesis:

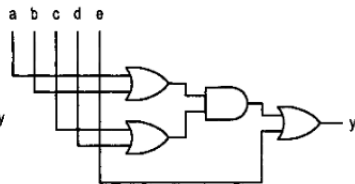
- No sean óptimas.

El problema de realizar un diseño es computacionalmente complejo. Esto lleva a que las soluciones dadas por la síntesis:

- No sean óptimas.
- O no sean posibles aún cuando en la simulación podamos ejecutar nuestros diseños.








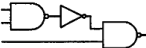

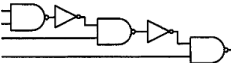
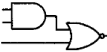
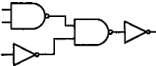

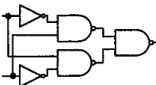
(a) Two-level implementation



(b) Multilevel implementation

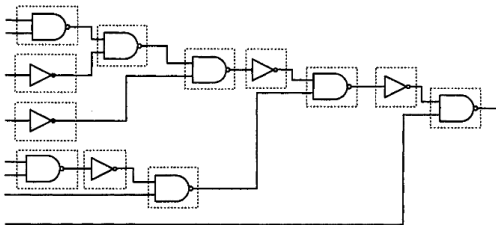
Veamos cómo se ve una librería de celdas.

# Ejemplo de una librería

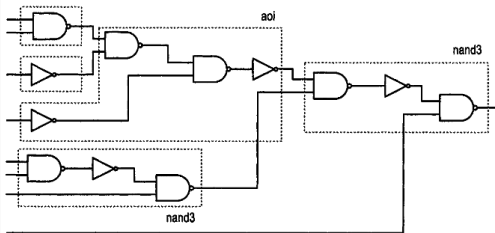
cell name (cost)	symbol	nand-not representation
not (2)		
nand2 (3)		
nand3 (4)		
nand4 (5)		
aoi (4)		
xor (4)		



Ahora veamos dos posibles proyecciones de un mismo diseño sobre las celdas de esa librería.



(a) Initial mapping



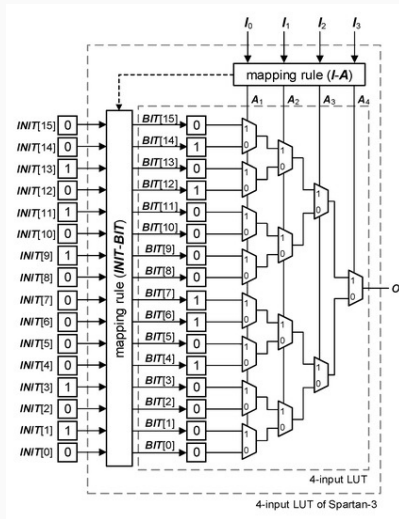
(b) Better mapping

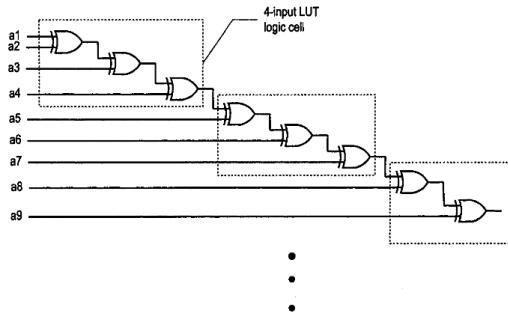
Las **FPGA** permiten (re)configurar lógica discreta en un soporte de hardware de forma que realice nuestros diseños dados en HDL.

Las **FPGA** permiten (re)configurar lógica discreta en un soporte de hardware de forma que realice nuestros diseños dados en HDL.

El flujo de prototipado es el mismo que presentamos hasta ahora, y las celdas utilizadas en su librería suelen consistir de **LUTs** (look up tables).

# LUT de 4 entradas





(a) 4-input LUT mapping of an odd-parity circuit

El proceso de síntesis es computacionalmente complejo y depende entre otras cosas de:

El proceso de síntesis es computacionalmente complejo y depende entre otras cosas de:

- La complejidad del diseño inicial.



El proceso de síntesis es computacionalmente complejo y depende entre otras cosas de:

- La complejidad del diseño inicial.
- La librería de celdas sobre la cual queremos realizar el diseño.

El proceso de síntesis es computacionalmente complejo y depende entre otras cosas de:

- La complejidad del diseño inicial.
- La librería de celdas sobre la cual queremos realizar el diseño.
- La madurez y eficiencia de los algoritmos involucrados en el flujo de síntesis.

Debido a esto resulta vital poder escribir especificaciones que faciliten la síntesis para conseguir una solución **primero realizable, pero también eficiente.**

# **VHDL - Generalidades**

---

La declaración de la entidad define la interfaz de nuestro componente...

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

entity alu is
port(
    a_i : in std_logic_vector(3 downto 0);
    b_i : in std_logic_vector(3 downto 0);
    op_i: in std_logic_vector(3 downto 0);
    s_o: out std_logic_vector(3 downto 0);
    z_o: out std_logic;
    c_o: out std_logic;
    v_o: out std_logic;
    n_o: out std_logic
);
end entity;
```

...y la arquitectura el comportamiento y la forma de implementarlo.

architecture logic of alu is

*-- Declaración de señales, variables, constantes, (y componentes)*

signal a\_int: signed(4 downto 0);

signal b\_int: signed(4 downto 0);

signal s\_int: signed(4 downto 0);

begin

*-- Aquí comienza el cuerpo de la arquitectura*

*-- Se extienden un bit para capturar el C*

a\_int <= signed('0' & a\_i); *-- Concatenación: &*

b\_int <= signed('0' & b\_i);

*-- continúa...*

```
-- ...de filmina anterior
-- Selección (Multiplexado de señales)

-- Operaciones
with op_i select
s_int <= a_int + b_int when "00",
        a_int - b_int when "01",
        signed(a_int and b_int) when "10",
        signed(a_int or b_int) when others;

-- Cero
z_o <= '1' when s_int(3 downto 0) = to_signed(0,4) else
      '0';

-- continúa...
```

```
-- El carry es simplemente el MSb
c_o <= s_int(4);

-- El overflow depende de la operación, por ello la xor con op_i
-- también se puede implementar con un mux (mejor...)
v_o <= ((a_i(3) xnor b_i(3)) xor op_i(0)) and
      (a_i(3) xor s_int(3));

-- El flag N simplemente es el MSb de la salida
n_o <= s_int(3);

s_o <= std_logic_vector(s_int(3 downto 0));

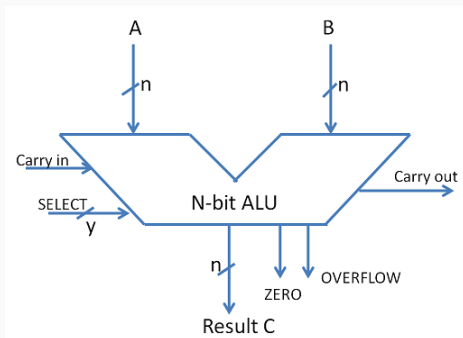
end architecture;
```



# Descripción de circuitos combinacionales

---

Son circuitos (sistemas) cuya salida sólo depende de las entradas  
Por ejemplo: nuestra **ALU** sencilla:



Veamos este ejemplo para aprender VHDL...

Empecemos por parametrizar el tamaño de palabra de la ALU..

Permite pasar información a una entidad.  
No puede ser modificado dentro de la arquitectura (constante?).

```
-- ...  
entity alu is  
generic(N:natural := 4);  ---> Generic!  
port(  
    a_i : in std_logic_vector(N-1 downto 0);  
    b_i : in std_logic_vector(N-1 downto 0);  
    op_i: in std_logic_vector(1 downto 0);  
    s_o: out std_logic_vector(N-1 downto 0);  
    z_o: out std_logic;  
    c_o: out std_logic;  
    v_o: out std_logic;  
    n_o: out std_logic  
);  
end entity;  
  
-- ...
```

```
-- ...  
  
architecture logic of alu is  
  
    -- Declaración de señales, variables, constantes, (y componentes)  
    signal a_int: signed(N downto 0);  
    signal b_int: signed(N downto 0);  
    signal s_int: signed(N downto 0);  
  
begin  
    -- ...
```

```
-- ...

-- Cero
z_o <= '1' when s_int(N-1 downto 0) = to_signed(0,N) else
      '0';

-- El carry es simplemente el MSb
c_o <= s_int(N);

-- El overflow depende de la operación, por ello la xor con op_i
-- también se puede implementar con un mux (mejor...)
v_o <= ((a_i(N-1) xnor b_i(N-1)) xor op_i(0)) and
      (a_i(N-1) xor s_int(N-1));

-- El flag N simplemente es el MSb de la salida
n_o <= s_int(N-1);

s_o <= std_logic_vector(s_int(N-1 downto 0));

end architecture;
```

Es una declaración concurrente en si misma.  
Permite hacer declaraciones secuenciales dentro de ella.

```
U_OV_CALC: process(a_i, b_i, s_int, op_i) -- (...) -> Lista de sensibilidad
    variable ov : std_logic;
begin
    ov:='0';
    if op_i="00" or op_i="01" then
        if a_i(N-1)=b_i(N-1) and a_i(N-1)/=s_int(N-1) then
            ov := '1';
        end if;
    end if;
    v_o <= ov;
end process;
```

Notar que el comportamiento es el mismo que el código concurrente...  
¿Generará el mismo hardware...?

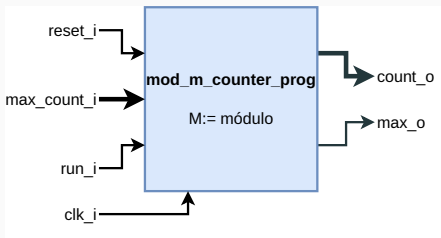
# Descripción de circuitos secuenciales

---

Son circuitos (sistemas) cuya salida depende de las entradas y del estado actual:

## Tienen memoria

Por ejemplo: un contador





Declaremos la entidad...

```
-- ...  
use IEEE.math_real.all;  
  
entity mod_m_counter_prog is  
    generic(M : natural      -- Modulo  
            );  
    port(clk_i      : in  std_logic;  
         reset_i    : in  std_logic;  
         run_i      : in  std_logic;  
         max_count_i : in  std_logic_vector (natural(ceil(log2(real(M))))-1  
                                             downto 0);  
         count_o     : out std_logic_vector (natural(ceil(log2(real(M))))-1  
                                             downto 0);  
         max_o       : out std_logic  
    );  
end entity;
```

Ahora la arquitectura...

```
-- ...  
architecture mod_m_counter_prog_arch of mod_m_counter_prog is  
    constant NUM_BITS : natural := natural(ceil(log2(real(M))));  
    signal r_reg      : unsigned(NUM_BITS-1 downto 0);  
    signal r_next     : unsigned(NUM_BITS-1 downto 0);  
begin  
    NXT_STATE_PROC: process(clk_i, reset_i)  
    begin  
        if rising_edge(clk_i) then  
            if (reset_i = '1') then  
                r_reg <= (others => '0');  
            elsif run_i = '1' then  
                r_reg <= r_next;  
            end if;  
        end if;  
    end process;  
  
    r_next <= (others => '0') when r_reg = unsigned(max_count_i) else  
        r_reg + 1;  
  
-- ...
```

¿Y las salidas?

```
-- ...  
    max_o <= '1' when r_reg = unsigned(max_count_i) and (run_i = '1') else  
            '0';  
  
    count_o <= std_logic_vector(r_reg);  
  
end architecture;
```

## Más generalidades

---

**Motivación:** implementar un contador decimal de 2 dígitos:  
La declaración de esta entidad sería:

```
entity digit_counter is
    generic(B : natural := 10      -- Base
    );
    port(clk_i      : in  std_logic;
         reset_i    : in  std_logic;
         run_i      : in  std_logic;
         digit1_o    : out std_logic_vector (natural(ceil(log2(real(B))))-1
                                             downto 0);
         digit2_o    : out std_logic_vector (natural(ceil(log2(real(B))))-1
                                             downto 0);
         max_o      : out std_logic
    );
end entity;
-- ...
```

Usemos el `mod_m_counter_prog...`

```
-- ...  
architecture structural of digit_counter is  
  
    constant max_count : std_logic_vector(  
        natural(ceil(log2(real(B))))-1 downto 0)  
    := std_logic_vector(to_unsigned(B-1, natural(ceil(log2(real(B))))));  
  
    signal count2 : std_logic;  
  
-- ...  
  
begin  
  
-- ...
```

## Conectando dos contadores en cascada (1)...

```
-- ...  
  
CONT1: entity work.mod_m_counter_prog  
  generic map(M => B      -- Modulo  
             )  
  port map (clk_i      => clk_i,  
            reset_i    => reset_i,  
            run_i       => run_i,  
            max_count_i => max_count,  
            count_o     => digit1_o,  
            max_o       => count2  
            );  
  
-- ...
```

## Conectando dos contadores en cascada (2) ...

```
-- ...  
CONT2: entity work.mod_m_counter_prog  
    generic map(M => B           -- Modulo  
                )  
    port map (clk_i      => clk_i,  
              reset_i    => reset_i,  
              run_i      => count2,  
              max_count_i => max_count,  
              count_o     => digit2_o,  
              max_o       => max2  
              );  
  
-- ...  
end architecture;
```



## Conectando dos contadores en cascada (2) ...

```
-- ...  
CONT2: entity work.mod_m_counter_prog  
    generic map(M => B          -- Modulo  
               )  
    port map (clk_i      => clk_i,  
              reset_i    => reset_i,  
              run_i       => count2,  
              max_count_i => max_count,  
              count_o     => digit2_o,  
              max_o       => max2  
            );  
  
-- ...  
end architecture;
```

**¿Y si queremos hacer un contador  
de N dígitos en lugar de 2?!**

## Conectando dos contadores en cascada (2) ...

```
-- ...  
CONT2: entity work.mod_m_counter_prog  
    generic map(M => B      -- Modulo  
                )  
    port map (clk_i      => clk_i,  
              reset_i    => reset_i,  
              run_i       => count2,  
              max_count_i => max_count,  
              count_o     => digit2_o,  
              max_o       => max2  
              );  
  
-- ...  
end architecture;
```

**¿Y si queremos hacer un contador  
de N dígitos en lugar de 2?!**

Lo vemos la semana que viene....

VHDL es mucho más extenso que sólo el conjunto sintetizable:

```
-- ...  
entity mod_m_counter_tb IS  
end mod_m_counter_tb;  
  
architecture behavior of mod_m_counter_tb is  
    constant M : natural := 6;  
  
    --Inputs  
    signal clk_i   : std_logic := '1';  
    signal reset_i : std_logic := '1';  
    signal run_i   : std_logic := '0';  
  
    --Outputs  
    signal count_o : std_logic_vector(ceil2power(M)-1 downto 0);  
    signal max_o   : std_logic;  
  
    -- Clock period definitions  
    constant clk_period : time := 1 us;  
    -- ...
```

VHDL es mucho más extenso que sólo el conjunto sintetizable:

```
-- ...  
begin  
    -- Clock process definitions  
    clk_process : process  
    begin  
        clk_i <= '0';  
        wait for clk_period/2;  
        clk_i <= '1';  
        wait for clk_period/2;  
    end process;  
  
    reset_i <= '1', '0' after 5 us;  
    run_i    <= '0', '1' after 20 us, '0' after 30 us;  
-- ...
```

Finalmente instanciamos el componente a probar:

```
-- ...

-- Instantiate the Unit Under Test (UUT)
 uut : mod_m_counter_prog
generic map(M => B      -- Modulo
          )
port map (clk_i      => clk_i,
         reset_i     => reset_i,
         run_i       => run_i,
         max_count_i => max_count,
         count_o     => count_o,
         max_o       => max_o
        );
end architecture;
```

# Ejercicios

---

Podemos controlar la intensidad de un led controlando el tiempo de encendido del mismo.

Se pide diseñar un sistema que genere una señal de encendido del led controlada por una entrada de  $N$  bits interpretada como un unsigned llamada `duty_cycle`. Es decir, cuando la entrada vale 0 el led deberá estar apagado, cuando vale  $2^N - 1$  debe estar encendido, y en los valores intermedios el encendido debe ser proporcional al valor. Un esquema para  $N = 3$  sería:

	tiempo ---->
<code>duty_cycle: 0</code>	-> out: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
<code>duty_cycle: 7</code>	-> out: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
<code>duty_cycle: 3</code>	-> out: 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 ...
	T segs

Se nos pide también controlar la frecuencia de repetición del patrón, con otra entrada de  $M$  bits llamada *freq* de forma que tarde  $T$  segundos cuando la misma valga 0,  $2T$  cuando valga 1,  $3T$  cuando valga 2, etc. Es decir:

```
duty_cycle: 3      tiempo ---->
freq: 0    -> out: 1 1 1 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 1 1 ...
freq: 1    -> out: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 ...
freq: 2    -> out: 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 ...
                |      T segs      |
                |      2T segs      |
                |      3T segs      |
```



### Guidelines:

- Definir la interfaz de la entidad (incluir clocks y resets).
- Pensar primero el problema para `duty_cycle`.
- Debuggearlo con las herramientas (GHDL, GtkWave :) o PlanAhead :(.).
- Extender el problema considerando `freq`.
- Debuggearlo con las herramientas (GHDL, GtkWave :) o PlanAhead :(.).
- Opcional (por hoy): probarlo en las placas de desarrollo :)

## Compilación, simulación con ghdl:

```
$ ghdl -a <archivos_del_diseño_en_orden> # Analiza
$ ghdl -e <top_entity> # Elabora
$ ghdl -r <top_entity> --vcd=<nombre_archivo_waveform>.vcd # Corre
# La simulación se puede interrumpir con Ctrl+C
```

## Visualización con GtkWave:

```
$ gtkwave <nombre_archivo_waveform>.vcd # Carga las formas de onda
```