

Práctica 2 - Aspectos avanzados del lenguaje VHDL

Sistemas Digitales - FIUBA

Abril 26, 2021

Herramientas

EDApplayground: <https://www.edaplayground.com/>

GTKWave, GHDL: <https://www.youtube.com/watch?v=H2GyAIYwZbw>

Modelsim: https://www.mentor.com/company/higher_ed/modelsim-student-edition

Nota: En todos los ejercicios se deberá implementar una entidad de simulación y simular la resolución del mismo.

Ayuda: Los atributos predefinidos de VHDL para arreglos son:

`'left, 'right, 'high, 'low, 'range, 'reverse_range, 'length`

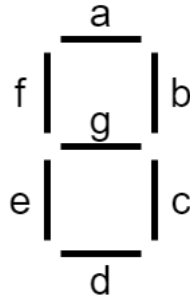
Ejercicio 1 - Crear un paquete de utilidades al cual se le vayan agregando todas las funciones, procedimientos y declaración de tipos, constantes, etc., que se creen en los puntos siguientes. Implementar un *procedure* que corra la simulación por un tiempo t .

Ejercicio 2 - Declarar una constante M sólo visible dentro del paquete, y otra J visible para todos los usuarios del paquete.

Ejercicio 3 - Implementar una compuerta AND genérica como una función.

Ejercicio 4 - Implementar una función que rote una señal de tipo bit (MSB ... LSB \rightarrow LSB ... MSB).

Ejercicio 5 - Implementar una función que realice la conversión hexadecimal a 7 segmentos de una señal de 4 bits.



Ejercicio 6 - Implementar una función que convierta un número binario (de N bits) a entero sin signo, utilizando atributos (utilizar los predefinidos para arreglos). El encabezado de la función es el siguiente:

```
function bit2natural (bin: in bit_vector) return natural;
```

Ejercicio 7 - Implementar una función que convierta un número entero sin signo a un binario (de N bits), utilizando atributos (utilizar los predefinidos para arreglos). El encabezado de la función es el siguiente:

```
function natural2bit(x: natural, N: natural) return out bit_vector;
```

Ejercicio 8 - Dibujar el diagrama de tiempos de las siguientes asignaciones de señales, incluyendo los retardos δ , e indicando las transacciones.

a)

```
entity exercise is
end exercise;

architecture behavioral of exercise is
    signal a, b, c: bit := '0';
begin
    a <= '1';
    b <= not a;
    c <= not b;
end behavioral;
```

b)

```
entity exercise is
end exercise;

architecture behavioral of exercise is
    signal x: std_logic := 'z';
begin
    process
    begin
        x <= transport '0' after 5 ns;
        x <= '0' after 3 ns;
        x <= '1' after 11 ns;
        wait;
    end process;
end behavioral;
```

Si la asignación de la primera señal fuera inercial, habría algún cambio en el diagrama de tiempos? - Si la asignación de la tercera señal fuera por transporte, habría algún cambio en el diagrama de tiempos? En caso de que la respuesta sea afirmativa redibujar dicho diagrama

c)

```
entity exercise is
end exercise;

architecture behavioral of exercise is
    signal x: std_logic := 'z';
begin
    process
    begin
        x <= transport '1' after 5 ns;
        x <= transport '0' after 8 ns;
        x <= transport '1' after 6 ns;
        wait;
    end process;
end behavioral;
```

d)

```

entity exercise is
end exercise;

architecture behavioral of exercise is
    signal x: std_logic := 'z';
begin
    process
    begin
        x <= '1' after 5 ns, '0' after 10 ns, '1' after 20 ns;
        x <= '0' after 12 ns, '1' after 16 ns, '0' after 25 ns;
        wait;
    end process;
end behavioral;

```

e)

```

entity exercise is
end exercise;

architecture behavioral of exercise is
    signal x: std_logic := 'z';
begin
    process
    begin
        x <= '1' after 5 ns, '0' after 10 ns, '1' after 20 ns;
        x <= '0' after 12 ns, '1' after 16 ns, '0' after 25 ns;
        x <= '1' after 20 ns;
        wait;
    end process;
end behavioral;

```

f)

```

entity exercise is
end exercise;

architecture behavioral of exercise is
    constant N : integer := 4;
    constant TA: bit_vector(0 to N-1) := "1010";
    constant TB: bit_vector(0 to N-1) := "1001";

```

```

    signal a,b,c: bit := '0';

begin

    c <= a xor b after 10 ns;

    process
    begin
        for i in 0 to N-1 loop
            a <= TA(i);
            b <= TB(i);
            wait on c for 20 ns;
        end loop;
        wait;
    end process;
end behavioral;

```

g)

```

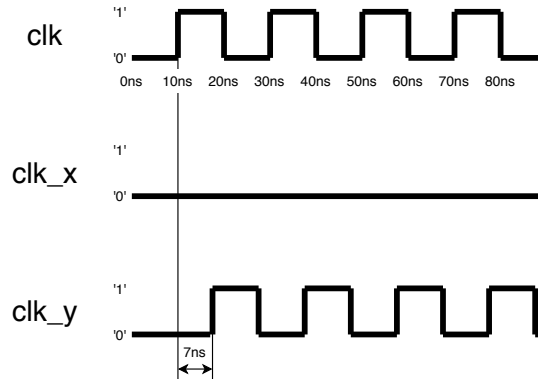
entity exercise is
end exercise;

architecture behavioral of exercise is
    signal a: bit := '0';
    signal b: bit := '1';
begin
    process
    begin
        wait for 10 ns;
        a <= '1';
        wait on 'b';
        a <= '0';
    end process;

    process
    begin
        wait until a = '1';
        wait for 20 ns;
        b <= '0';
    end process;
end behavioral;

```

Ejercicio 9 - Crear una señal clock de 20 ns de período. Luego, mediante dos asignaciones tal como se observa en la figura.



Ejercicio 10 - Determinar si el siguiente código es correcto. En caso de no serlo, indicar el tipo de error y encontrar la manera de corregirlo, usando siempre la misma sentencia case.

```
signal address : std_logic_vector(14 downto 0);
...
case address is
  when "0000100000000000" to "0000100011111111" => ...
...
```

Ejercicio 11 -. Implementar un registro de desplazamiento (shift register) genérico utilizando la sentencia generate mediante la instanciación de registros.

Ejercicio 12 - Implementar en VHDL un full-adder y luego convertirlo en un sumador genérico de N bits utilizando la sentencia generate. Los operandos deben estar definidos de la siguiente manera:

```
a: in std_logic_vector(N-1 downto 0);
b: in std_logic_vector(N-1 downto 0);
```

Luego de haber corroborado el funcionamiento del sumador modificar la definición de los operandos de la siguiente manera:

```
a: in std_logic_vector(0 to N-1);  
b: in std_logic_vector(0 to N-1);
```

El resultado obtenido será el mismo que en el caso anterior? Explicar.

Ejercicio 13 - Implementar en VHDL una memoria ROM.

Ejercicio 14 - Implementar en VHDL una memoria RAM *single port* asincrónica. Realizar la síntesis en FPGA.

Ejercicio 15 - Implementar en VHDL una memoria RAM *single port* sincrónica. Realizar la síntesis en FPGA.

Ejercicio 16 - Implementar en VHDL una memoria RAM *dual port* asincrónica. Realizar la síntesis en FPGA.

Ejercicio 17 - Implementar en VHDL una memoria RAM *dual port* sincrónica. Realizar la síntesis en FPGA.

Ejercicio 18 - Definir el tipo de dato `logic` que tome los valores 'x', '0' y '1'.

Ejercicio 19 - Definir el tipo de datos `logic_vector` como arreglo lineal del tipo de datos `logic`.

Ejercicio 20 - Implementar el tipo de datos `logic_matrix` como arreglo bidimensional del tipo de datos `logic`.

Ejercicio 21 - Implementar un flip-flop D genérico de N bits usando el tipo de datos `logic` definido anteriormente. Es sintetizable?

Ejercicio 22 - Definir el tipo de datos `logic_wired_or` tal que se asocie a una función de resolución que resuelva la asignación de múltiples señales de tipo `logic_wired_or` por medio de una OR cableada (*wired-or*).

Ejercicio 23 - Sobrecargar todos los operadores lógicos (and, or, not, xor, etc) para el tipo de datos `logic`.

Ejercicio 24 - Para el tipo de datos `logic_vector` definir los siguientes operadores aritméticos:

- Unsigned "+": toma dos tipos operandos A, B de tipo `logic_vector` de tamaño arbitrario no necesariamente iguales y devuelve un tipo

de dato `logic_vector` de tamaño $\max\{A'length, B'length\} + 1$ tal que represente la suma sin signo de A y B.

- Signed "+": toma dos tipos operandos A, B de tipo `logic_vector` de tamaño arbitrario no necesariamente iguales y devuelve un tipo de dato `logic_vector` de tamaño $\max\{A'length, B'length\} + 1$ tal que represente la suma con signo de A y B.
- Unsigned "*": toma dos tipos operandos A, B de tipo `logic_vector` de tamaño arbitrario no necesariamente iguales y devuelve un tipo de dato `logic_vector` de tamaño $A'length + B'length$ tal que represente el producto sin signo de A y B.
- Signed "*": toma dos tipos operandos A, B de tipo `logic_vector` de tamaño arbitrario no necesariamente iguales y devuelve un tipo de dato `logic_vector` de tamaño $A'length + B'length$ tal que represente el producto con signo de A y B.

Existe alguna diferencia entre la suma con signo y sin signo? Y entre el producto con signo y sin signo?

Ejercicio 25 - Implemente los siguientes circuitos con el tipo de dato `logic_vector` y la definición de los operadores del ejercicio anterior:

- Un sumador sin signo.
- Un sumador con signo.
- Un multiplicador sin signo.
- Un multiplicador con signo.

Realice la síntesis en FPGA.


```

package my_pkg is

    constant J : bit_vector(3 downto 0) := "1011";
    function return_M return bit_vector;

    procedure run_sim(t: time);

    type logic is ('x', '0', '1');
    type logic_vector is array (natural range <>) of logic;
    type logic_matrix is array (natural range <>, natural range <>) of logic;

    function func_and(x: bit_vector) return bit;
    function func_rot(x: bit_vector) return bit_vector;

    function "not"    (x: logic) return logic;
    function "and"    (x: logic; y: logic) return logic;
    function "nand"   (x: logic; y: logic) return logic;
    function "or"     (x: logic; y: logic) return logic;
    function "nor"    (x: logic; y: logic) return logic;
    function "xor"    (x: logic; y: logic) return logic;
    function "xnor"   (x: logic; y: logic) return logic;

    function or_resolv_logic (x: logic_vector) return logic;

    subtype logic_wired_or is or_resolv_logic logic;

    subtype ulogic_vector is logic_vector;
    subtype slogic_vector is logic_vector;

    function "+"      (x: ulogic_vector; y: ulogic_vector) return ulogic_vector;
    -- function "+"    (x: slogic_vector; y: slogic_vector) return slogic_vector;
    -- function "*"    (x: ulogic_vector; y: ulogic_vector) return ulogic_vector;
    -- function "*"    (x: slogic_vector; y: slogic_vector) return slogic_vector;

end my_pkg;

-- #####

package body my_pkg is

    -----

    constant M : bit_vector(3 downto 0) := "0110";

```

```

function return_M return bit_vector is
begin
    return M;
end return_M;

-----

procedure run_sim(t: time) is
begin
    wait for t; -- run the simulation for this duration
    assert false
        report "Simulation finished."
        severity failure;
end run_sim;

-----

function func_and(x: bit_vector) return bit is

    variable partial : bit := '1';

begin

    for i in x'range loop
        partial := partial and x(i);
    end loop;

    return partial;

end func_and;

-----

function func_rot(x: bit_vector) return bit_vector is

    variable partial : bit_vector(x'range);

begin

    for i in x'range loop
        partial(x'length-1-i) := x(i);
    end loop;

    return partial;

end func_rot;

```

```

function "not"  (x: logic) return logic is
begin

    case x is
        when '0' => return '1';
        when '1' => return '0';
        when 'x' => return 'x';
    end case;

end "not";

```

```

function "and"  (x: logic; y: logic) return logic is
    variable aux : logic_vector(1 downto 0);
begin

    aux := x & y;

    case aux is
        when "00" => return '0';
        when "01" => return '0';
        when "0x" => return '0';
        when "10" => return '0';
        when "11" => return '1';
        when "1x" => return 'x';
        when "x0" => return '0';
        when "x1" => return 'x';
        when "xx" => return 'x';
    end case;

end "and";

```

```

function "nand" (x: logic; y: logic) return logic is
begin

    return not(x and y);

end "nand";

```

```

function "or"    (x: logic; y: logic) return logic is
    variable aux : logic_vector(1 downto 0);
begin

    aux := x & y;

    case aux is
        when "00" => return '0';
        when "01" => return '1';
        when "0x" => return 'x';
        when "10" => return '1';
        when "11" => return '1';
        when "1x" => return '1';
        when "x0" => return 'x';
        when "x1" => return '1';
        when "xx" => return 'x';
    end case;

end "or";

```

```

function "nor"   (x: logic; y: logic) return logic is
begin

    return not(x or y);

end "nor";

```

```

function "xor"   (x: logic; y: logic) return logic is
begin

    return ((not x) and y) or (x and (not y));

end "xor";

```

```

function "xnor"  (x: logic; y: logic) return logic is
begin

    return not(x xor y);

end "xnor";

```

```

-----
function or_resolv_logic (x: logic_vector) return logic is
    variable aux : logic := '0';
begin
    if x'length=1 then
        return x(0);
    else
        for i in x'range loop
            if x(i)='1' then
                aux := '1';
            end if;
        end loop;
        if aux='1' then
            return '1';
        end if;

        for i in x'range loop
            if x(i)='x' then
                aux := 'x';
            end if;
        end loop;
        if aux='x' then
            return 'x';
        end if;

        return '0';

    end if;
end or_resolv_logic;

```

```

-----
-- Ayuda para los operadores ariméticos
-----

```

```

-----
-- Suma sin signo
-----

```

```

--
-- A 4 bits, 1111 max(A) = 15d
-- B 3 bits, 111 max(B) = 7d
--

```

```

-- max(A+B) = 15d+7d = 22d => 5 bits
--
-- Entonces se necesitan (max(A'length,B'length) + 1) bits
-----

-----
-- Suma con signo (complemento a 2)
-----
--
-- A 4 bits, 0111 max(A) = +7d
-- A 4 bits, 1000 min(A) = -8d
--
-- B 3 bits, 011 max(B) = +3d
-- B 3 bits, 100 min(B) = -4d
--
-- max(A)+max(B) = (+7d) + (+3d) = +10d      => 4 bits + 1 bit de signo = 5 bits
-- min(A)+min(B) = (-8d) + (-3d) = -11d      => 4 bits + 1 bit de signo = 5 bits
--
-- Entonces se necesitan (max(A'length,B'length) + 1) bits
-----

-----
-- Multiplicación sin signo
-----
--
-- A 4 bits, 1111 max(A) = 15d
-- B 3 bits, 111 max(B) = 7d
--
-- max(A*B) = 15d*7d = 105d => 7 bits
--
-- Entonces se necesitan (A'length + B'length) bits
-----

-----
-- Multiplicación con signo (complemento a 2)
-----
--
-- A : 4 bits, 0111 max(A) = +7d
-- A : 4 bits, 1000 min(A) = -8d
-- B : 3 bits, 011 max(B) = +3d
-- B : 3 bits, 100 min(B) = -4d
--
-- max(A)*max(B) = (+7d)*(+3d) = +21d      => 5 bits + 1bit de signo = 6 bits

```

```
-- min(A)*min(B) = (-8d)*(-4d) = +32d    => 6 bits + 1bit de signo = 7 bits  <----- Ma
-- max(A)*min(B) = (+7d)*(-4d) = -28d    => 5 bits + 1bit de signo = 6 bits
-- min(A)*max(B) = (-8d)*(+3d) = -24d    => 5 bits + 1bit de signo = 6 bits
```

```
-- Entonces se necesitan (A'length + B'length) bits
-----
```

```
-----
function "+" (x: ulogic_vector; y: ulogic_vector) return ulogic_vector is

    variable max_len      : natural;
    variable partial_cy   : logic;
    variable partial_sum  : logic_vector(x'length + y'length downto 0) := (others => '0');
    variable aux_x        : logic;
    variable aux_y        : logic;

begin

    if x'length > y'length then
        max_len := x'length;
    else
        max_len := y'length;
    end if;

    partial_cy := '0';

    for i in 0 to max_len-1 loop
        if i < x'length then
            aux_x := x(i);
        else
            aux_x := '0';
        end if;

        if i < y'length then
            aux_y := y(i);
        else
            aux_y := '0';
        end if;

        partial_sum(i) := partial_cy xor (aux_x xor aux_y);
        partial_cy      := (aux_x and aux_y) or (aux_y and partial_cy) or (aux_x and
        partial_cy);

    end loop;
```

```

        partial_sum(max_len) := partial_cy;

        return partial_sum(max_len downto 0);

    end "+";

-----
--      function "+"      (x: slogic_vector; y: slogic_vector) return slogic_vector is
--
--  end "+";
-----
--      function "*"      (x: ulogic_vector; y: ulogic_vector) return ulogic_vector is
--
--  end "*";
-----
--      function "*"      (x: slogic_vector; y: slogic_vector) return slogic_vector is
--
--  end "*";

end my_pkg;

use work.my_pkg.all;

entity tb is
end tb;

architecture behavioral of tb is

    -- Ej 1
    constant SIM_TIME_NS : time := 200 ns;

    -- Ej 2
    signal xa : bit_vector(3 downto 0) := J;
    signal xb : bit_vector(3 downto 0) := return_M;

    -- Ej 3
    signal x1 : bit_vector(2 downto 0) := "011";
    signal x2 : bit_vector(2 downto 0) := "111";

    signal and_x1 : bit;
    signal and_x2 : bit;

```



```

-- Ej 4
signal x3      : bit_vector(2 downto 0) := "011";
signal rot_x3  : bit_vector(2 downto 0);

-- Ej 9
signal clock   : bit := '0';
signal clock_x : bit;
signal clock_y : bit;

-- Ej 22
signal x7 : logic_wired_or := '1';
signal x8 : logic_wired_or := '0';
signal x9 : logic_wired_or := 'x';
signal x10 : logic_wired_or := '0';
signal x11 : logic_wired_or;
signal x12 : logic_wired_or;
signal x13 : logic_wired_or;

-- Ej 24.a
signal x4 : ulogic_vector(3 downto 0) := "1011"; -- unsigned 11
signal x5 : ulogic_vector(2 downto 0) := "111";  -- unsigned 7
signal x6 : ulogic_vector(4 downto 0);

begin

-- Ej 1
sim: process
begin
    run_sim(SIM_TIME_NS);
end process;

-- Ej 3
and_x1 <= func_and(x1);
and_x2 <= func_and(x2);

-- Ej 4
rot_x3 <= func_rot(x3);

-- Ej 9
clock   <= not clock after 10 ns;
clock_x <= inertial clock after 12 ns;
clock_y <= transport clock after 7 ns;

```

```

-- Ej 22
x11 <= x7; -- x11 de be valer '1'
x11 <= x8;
x11 <= x9;

x12 <= x8; -- x12 de be valer 'x'
x12 <= x9;

x13 <= x8; -- x13 de be valer '0'
x13 <= x10;

-- Ej 24.a
x6 <= x4 + x5; -- unsigned 18

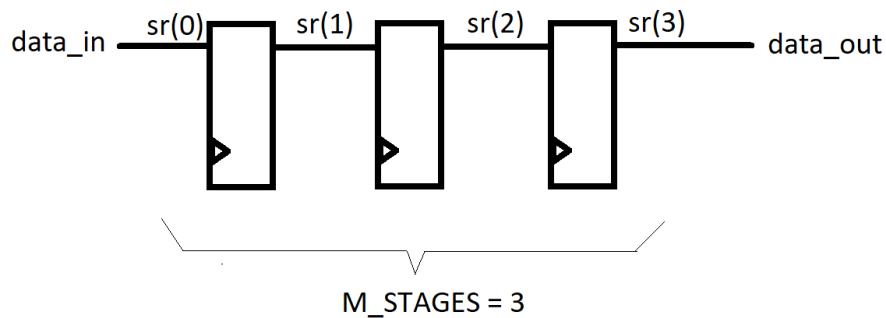
end behavioral;

```

Ejercicio 11

Implementar un registro de desplazamiento (shift register) genérico utilizando la sentencia generate mediante la instanciación de registros.

Diseño



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

entity regist is
  generic (
    N_BITS : natural := 8
  );
  port(
    rst      : in  std_logic;
    clk      : in  std_logic;
    enable   : in  std_logic;
    data_in  : in  std_logic_vector(N_BITS-1 downto 0);
    data_out : out std_logic_vector(N_BITS-1 downto 0)
  );
end regist;

architecture behavioral of regist is

begin
  process(clk, rst)
  begin
    if rst='1' then
      data_out <= (others=>'0');
    elsif clk = '1' and clk'event then
```

```

        if enable = '1' then
            data_out <= data_in;
        end if;
    end if;
end process;
end behavioral;

entity shift_reg is
    generic (
        N_BITS    : natural := 8;
        M_STAGES : natural := 10
    );
    port(
        rst       : in  std_logic;
        clk       : in  std_logic;
        enable    : in  std_logic;
        data_in   : in  std_logic_vector(N_BITS-1 downto 0);
        data_out  : out std_logic_vector(N_BITS-1 downto 0)
    );
end shift_reg;

architecture behavioral of shift_reg is
    type std_logic_matrix is array(natural range <>) of std_logic_vector;
    signal sr : std_logic_matrix(M_STAGES-1 downto 0)(N_BITS downto 0);
begin

    regs: for i in 0 to M_STAGES-1 generate
        reg_n: entity work.regist
            generic map (
                N_BITS => N_BITS
            )
            port map (
                rst       => rst,
                clk       => clk,
                enable    => enable,
                data_in   => sr(i),
                data_out  => sr(i+1)
            );
    end generate;

    sr(0) <= data_in;
    data_out <= sr(M_STAGES);
end

```

```
end behavioral;
```

Testbench

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

entity tb_shift_reg is
end tb_shift_reg;

architecture behavioral of tb_shift_reg is

    constant SIM_TIME_NS : time := 200 ns;

    constant TB_N_BITS    : natural := 4;
    constant TB_M_STAGES  : natural := 3;

    signal tb_clk          : std_logic := '0';
    signal tb_rst          : std_logic;
    signal tb_value        : std_logic_vector(TB_N_BITS-1 downto 0);
    signal tb_load         : std_logic;
    signal tb_enable       : std_logic;
    signal tb_data_in      : std_logic_vector(TB_N_BITS-1 downto 0);
    signal tb_data_out     : std_logic_vector(TB_N_BITS-1 downto 0);

begin

    tb_rst      <= '0', '1' after 7 ns, '0' after 50 ns;
    tb_clk      <= not tb_clk after 5 ns;
    tb_value    <= "0110";
    tb_enable   <= '1', '0' after 83 ns, '1' after 87 ns;
    tb_load     <= '0', '1' after 133 ns, '0' after 147 ns;
    tb_data_in  <= "0010", "0001" after 72 ns, "1010" after 82 ns, "1111" after 92 ns, "0000" after 102 ns;

    stop_simulation : process
    begin
        wait for SIM_TIME_NS; --run the simulation for this duration
        assert false
            report "Simulation finished."
            severity failure;
    end process;

    I1: entity work.shift_reg(behavioral)
    generic map(
        N_BITS => TB_N_BITS,
```

```

        M_STAGES => TB_M_STAGES
    )
    port map(
        rst      => tb_rst,
        clk      => tb_clk,
        enable   => tb_enable,
        load     => tb_load,
        value    => tb_value,
        data_in  => tb_data_in,
        data_out => tb_data_out
    );

end behavioral;

```

Ejercicio 17

Implementar en VHDL una memoria RAM *dual port* síncrona. Realizar la síntesis en FPGA.

Diseño

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

entity dpram is
  generic (
    BYTES_WIDTH : natural := 1; -- Ancho de palabra de la memoria medido en b
    ADDR_BITS   : natural := 8  -- Cantidad de bits de address (tamaño de la
  );
  port (
    rst           : in std_logic;
    clk           : in std_logic;
    data_wr       : in std_logic_vector(BYTES_WIDTH*8-1 downto 0);
    addr_wr : in std_logic_vector(ADDR_BITS-1 downto 0);
    ena_wr  : in std_logic;
    addr_rd : in std_logic_vector(ADDR_BITS-1 downto 0);
    data_rd  : out std_logic_vector(BYTES_WIDTH*8-1 downto 0)
  );
end dpram;

architecture rtl of dpram is

  -- Array para la memoria
  subtype t_word is std_logic_vector(BYTES_WIDTH*8-1 downto 0);
  type t_memory is array(2**ADDR_BITS-1 downto 0) of t_word;

  signal ram : t_memory;

  -- Address casting
  signal rd_pointer : integer range 0 to 2**ADDR_BITS-1;
  signal wr_pointer : integer range 0 to 2**ADDR_BITS-1;

begin

  -- Address casting
  rd_pointer <= to_integer(unsigned(addr_rd));
```



```

wr_pointer <= to_integer(unsigned(addr_wr));

-- Write
process(clk)
begin
    if clk='1' and clk'event then
        if ena_wr='1' then
            ram(wr_pointer) <= data_wr;
        end if;
    end if;
end process;

-- Read
process(clk)
begin
    if clk='1' and clk'event then
        data_rd <= ram(rd_pointer);
    end if;
end process;

end rtl;

```

Testbench

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

entity tb is
end tb;

architecture behavioral of tb is

    constant SIM_TIME_NS : time      := 1000 ns;
    constant BYTES_WIDTH : natural    := 1;
    constant ADDR_BITS   : natural    := 8;

    signal tb_rst         : std_logic := '0';
    signal tb_clk         : std_logic := '0';
    signal tb_data_wr     : std_logic_vector(BYTES_WIDTH*8-1 downto 0);
    signal tb_addr_wr     : std_logic_vector(ADDR_BITS-1 downto 0);
    signal tb_ena_wr      : std_logic;
    signal tb_data_rd     : std_logic_vector(BYTES_WIDTH*8-1 downto 0);
    signal tb_addr_rd     : std_logic_vector(ADDR_BITS-1 downto 0);

    signal wr_addr_count  : unsigned(ADDR_BITS-1 downto 0) := "00000010";
    signal rd_addr_count  : unsigned(ADDR_BITS-1 downto 0) := "00000000";
    signal wr_data_count  : unsigned(ADDR_BITS-1 downto 0) := "10110010";

begin

    tb_rst    <= '0', '1' after 1 ns, '0' after 2 ns;
    tb_clk    <= not tb_clk after 5 ns;

    tb_ena_wr <= '1';

    tb_addr_wr <= std_logic_vector(wr_addr_count);
    tb_addr_rd <= std_logic_vector(rd_addr_count);
    tb_data_wr <= std_logic_vector(wr_data_count);

    process(tb_clk)
    begin
        if tb_clk='0' and tb_clk'event then
            wr_addr_count <= wr_addr_count + 1;
            rd_addr_count <= rd_addr_count + 1;
        end if;
    end process;

end architecture;
```

```

        wr_data_count <= wr_data_count + 1;
    end if;
end process;

stop_simulation : process
begin
    wait for SIM_TIME_NS; --run the simulation for this duration
    assert false
        report "Simulation finished."
        severity failure;
end process;

DUT: entity work.dpram(rtl)
generic map(
    BYTES_WIDTH => BYTES_WIDTH,
    ADDR_BITS   => ADDR_BITS
)
port map(
    rst           => tb_rst,
    clk           => tb_clk,
    data_wr       => tb_data_wr,
    addr_wr => tb_addr_wr,
    ena_wr  => tb_ena_wr,
    data_rd       => tb_data_rd,
    addr_rd => tb_addr_rd
);

end behavioral;

```