DISEÑOS VHDL

PRESENTADO POR:

LUIS MATEO ORTEGA GOYES

PRESENTADO A:

CARLOS HERNAN TOBAR ARTEAGA

UNIVERSIDAD DEL CAUCA

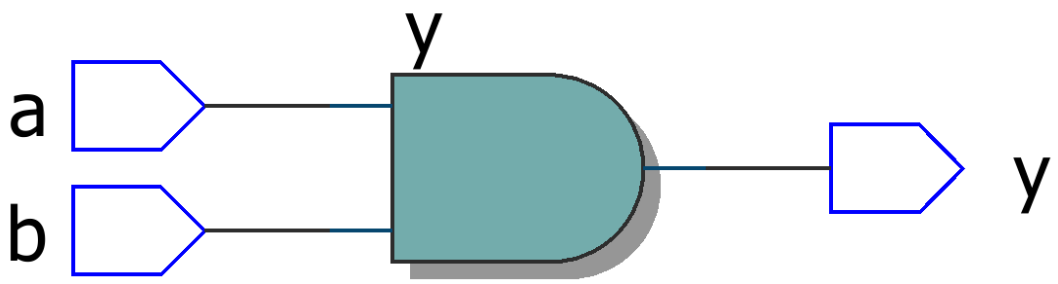
FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

TECNOLOGÍA EN TELEMÁTICA

POPAYÁN - SEPTIEMBRE - 2023

➢ **AND GATE**

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity and_gate is
5       Port ( a : in STD_LOGIC;
6              b : in STD_LOGIC;
7              y : out STD_LOGIC);
8   end and_gate;
9
10  architecture Behavioral of and_gate is
11  begin
12      y <= a and b;
13  end Behavioral;
14
```



TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_gate_tb is
end and_gate_tb;

architecture behavior of and_gate_tb is

    signal a, b : std_logic := '0';
    signal y : std_logic;

    component and_gate is
        Port ( a : in STD_LOGIC;
               b : in STD_LOGIC;
               y : out STD_LOGIC);
    end component;

begin

    uut: and_gate port map (
        a => a,
        b => b,
        y => y
    );

    stim_proc: process

    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        a <= '0'; b <= '0'; -- test 0 AND 0
        wait for 100 ns;
        a <= '1'; b <= '0'; -- test 1 AND 0
        wait for 100 ns;
        a <= '0'; b <= '1'; -- test 0 AND 1
        wait for 100 ns;
        a <= '1'; b <= '1'; -- test 1 AND 1
        wait for 100 ns;
        wait;

    end process;
end behavior;

```

EXPLICACIÓN AND:

- Código VHDL para la compuerta AND:

- entity and_gate is ... end and_gate;: Esta parte del código define la interfaz de la compuerta AND. Tiene dos entradas (a y b) y una salida (y).
- architecture Behavioral of and_gate is ... end Behavioral;: Esta parte del código describe el comportamiento de la compuerta AND. En este caso, la salida y es el resultado de la operación AND (a and b) de las entradas a y b.

- Código VHDL para el test bench:

- entity and_gate_tb is ... end and_gate_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.

- architecture behavior of and_gate_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.
- component and_gate is ... end component;: Aquí se declara la compuerta AND como un componente que se utilizará en el test bench.
- uut: and_gate port map ...;: Aquí se crea una instancia de la compuerta AND (denominada uut, que significa "unidad bajo prueba") y se conectan las señales del test bench a las entradas y salidas de la compuerta AND.
- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas a y b y las aplica a la compuerta AND.

➢ **AND16 GATE**

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity and16_gate is
5       Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
6              b : in STD_LOGIC_VECTOR (15 downto 0);
7              y : out STD_LOGIC_VECTOR (15 downto 0));
8   end and16_gate;
9
10  architecture Behavioral of and16_gate is
11  begin
12      y <= a and b;
13  end Behavioral;
14
```

TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and16_gate_tb is
end and16_gate_tb;

architecture behavior of and16_gate_tb is

    signal a, b : std_logic_vector (15 downto 0) := (others => '0');
    signal y : std_logic_vector (15 downto 0);

    component and16_gate is
        Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
               b : in STD_LOGIC_VECTOR (15 downto 0);
               y : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

begin

    uut: and16_gate port map (
        a => a,
        b => b,
        y => y
    );

    stim_proc: process

    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        a <= "0000000000000000"; b <= "0000000000000000"; -- test 0 AND 0
        wait for 100 ns;
        a <= "1111111111111111"; b <= "0000000000000000"; -- test 1 AND 0
        wait for 100 ns;
        a <= "0000000000000000"; b <= "1111111111111111"; -- test 0 AND 1
        wait for 100 ns;
        a <= "1111111111111111"; b <= "1111111111111111"; -- test 1 AND 1
        wait for 100 ns;
        wait;

    end process;
end behavior;
```

EXPLICACIÓN AND 16:

- Código VHDL para la compuerta AND de 16 bits:

- entity and16_gate is ... end and16_gate;: Esta parte del código define la interfaz de la compuerta AND de 16 bits. Tiene dos entradas (a y b) y una salida (y), todas de 16 bits.
- architecture Behavioral of and16_gate is ... end Behavioral;: Esta parte del código describe el comportamiento de la compuerta AND de 16 bits. En este caso, la salida y es el resultado de la operación AND (a and b) de las entradas a y b.

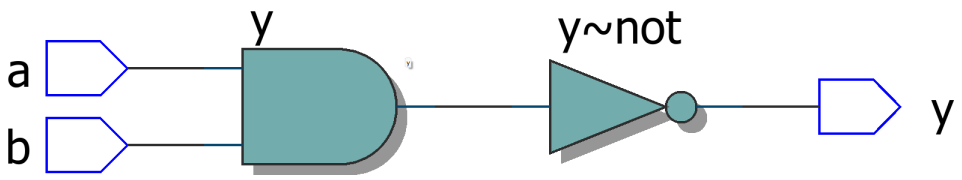
- Código VHDL para el test bench:

- entity and16_gate_tb is ... end and16_gate_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- architecture behavior of and16_gate_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.
- component and16_gate is ... end component;: Aquí se declara la compuerta AND de 16 bits como un componente que se utilizará en el test bench.

- uut: and16_gate port map ...;: Aquí se crea una instancia de la compuerta AND de 16 bits (denominada uut, que significa “unidad bajo prueba”) y se conectan las señales del test bench a las entradas y salidas de la compuerta AND de 16 bits.
- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas a y b y las aplica a la compuerta AND de 16 bits.

➢ **NAND GATE**

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity nand_gate is
5
6       Port ( a : in STD_LOGIC;
7              b : in STD_LOGIC;
8              y : out STD_LOGIC);
9
10  end nand_gate;
11
12  architecture Behavioral of nand_gate is
13
14  begin
15      y <= not (a and b);
16
17  end Behavioral;
18
```



TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nand_gate_tb is
end nand_gate_tb;

architecture behavior of nand_gate_tb is

    signal a, b : std_logic := '0';
    signal y : std_logic;

    component nand_gate is
        Port ( a : in STD_LOGIC;
               b : in STD_LOGIC;
               y : out STD_LOGIC);
    end component;

begin
    uut: nand_gate port map (
        a => a,
        b => b,
        y => y
    );

    stim_proc: process

    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        a <= '0'; b <= '0'; -- test 0 NAND 0
        wait for 100 ns;
        a <= '1'; b <= '0'; -- test 1 NAND 0
        wait for 100 ns;
        a <= '0'; b <= '1'; -- test 0 NAND 1
        wait for 100 ns;
        a <= '1'; b <= '1'; -- test 1 NAND 1
        wait for 100 ns;
        wait;

    end process;

end behavior;
```

EXPLICACIÓN NAND:

- Código VHDL para la compuerta NAND:

- entity nand_gate is ... end nand_gate;: Esta parte del código define la interfaz de la compuerta NAND. Tiene dos entradas (a y b) y una salida (y).
- architecture Behavioral of nand_gate is ... end Behavioral;: Esta parte del código describe el comportamiento de la compuerta NAND. En este caso, la salida y es el resultado de la operación NAND (not (a and b)) de las entradas a y b.

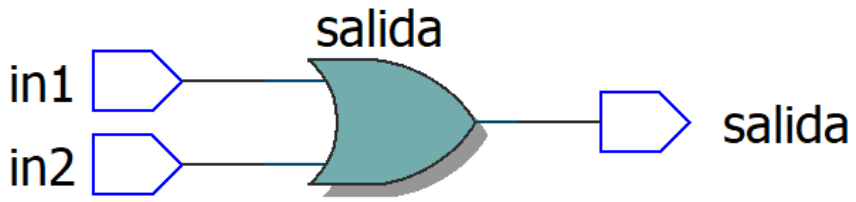
- Código VHDL para el test bench:

- entity nand_gate_tb is ... end nand_gate_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- architecture behavior of nand_gate_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.

- component nand_gate is ... end component;: Aquí se declara la compuerta NAND como un componente que se utilizará en el test bench.
- uut: nand_gate port map ...;: Aquí se crea una instancia de la compuerta NAND (denominada uut, que significa "unidad bajo prueba") y se conectan las señales del test bench a las entradas y salidas de la compuerta NAND.
- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas a y b y las aplica a la compuerta NAND.

➢ **OR GATE**

```
1   --Librerias
2
3   library IEEE;
4
5   use IEEE.STD_LOGIC_1164.ALL;
6
7   --entidad
8
9   entity OrGate is
10
11      Port (in1 : in STD_LOGIC;
12            in2 : in STD_LOGIC;
13            salida : out STD_LOGIC);
14
15  end entity;
16
17  --arquitectura
18
19  architecture behavioral of OrGate is
20
21     begin
22
23         salida <= in1 or in2;
24
25  end architecture;
```



➢ **OR16 GATE**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or16_gate is

    Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
           b : in STD_LOGIC_VECTOR (15 downto 0);
           y : out STD_LOGIC_VECTOR (15 downto 0));

end or16_gate;

architecture Behavioral of or16_gate is

begin
    y <= a or b;

end Behavioral;

```

TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or16_gate_tb is
end or16_gate_tb;

architecture behavior of or16_gate_tb is

    signal a, b : std_logic_vector (15 downto 0) := (others => '0');
    signal y : std_logic_vector (15 downto 0);

    component or16_gate is

        Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
               b : in STD_LOGIC_VECTOR (15 downto 0);
               y : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

begin

    uut: or16_gate port map (
        a => a,
        b => b,
        y => y
    );

    stim_proc: process

    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        a <= "0000000000000000"; b <= "0000000000000000"; -- test 0 OR 0
        wait for 100 ns;
        a <= "1111111111111111"; b <= "0000000000000000"; -- test 1 OR 0
        wait for 100 ns;
        a <= "0000000000000000"; b <= "1111111111111111"; -- test 0 OR 1
        wait for 100 ns;
        a <= "1111111111111111"; b <= "1111111111111111"; -- test 1 OR 1
        wait for 100 ns;
        wait;

    end process;
end behavior;

```

EXPLICACIÓN OR 16:

- Código VHDL para la compuerta OR de 16 bits:

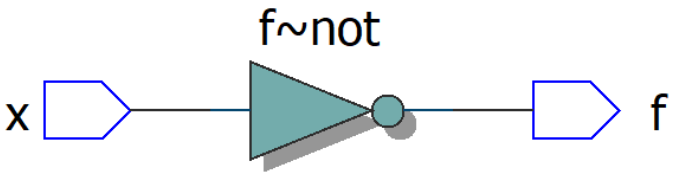
- entity or16_gate is ... end or16_gate;: Esta parte del código define la interfaz de la compuerta OR de 16 bits. Tiene dos entradas (a y b) y una salida (y), todas de 16 bits.
- architecture Behavioral of or16_gate is ... end Behavioral;: Esta parte del código describe el comportamiento de la compuerta OR de 16 bits. En este caso, la salida y es el resultado de la operación OR (a or b) de las entradas a y b.

- Código VHDL para el test bench:

- entity or16_gate_tb is ... end or16_gate_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- architecture behavior of or16_gate_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.
- component or16_gate is ... end component;: Aquí se declara la compuerta OR de 16 bits como un componente que se utilizará en el test bench.
- uut: or16_gate port map ...;: Aquí se crea una instancia de la compuerta OR de 16 bits (denominada uut, que significa "unidad bajo prueba") y se conectan las señales del test bench a las entradas y salidas de la compuerta OR de 16 bits.
- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas a y b y las aplica a la compuerta OR de 16 bits.

➢ **NOT GATE**

```
1   -- Not gate:
2   -- f = not x
3
4
5   -- Library and packages
6   library  IEEE;
7   use IEEE.std_logic_1164.all;
8
9   -- Entity (Interface)
10  entity NotGate is
11
12     port(
13        x  :  in    std_logic;
14        f  :  out   std_logic);
15
16  end entity;
17
18  -- Architecture (Implementation)
19  architecture arch of NotGate is
20
21     begin
22
23        F <= x nand x;
24
25  end architecture;
```

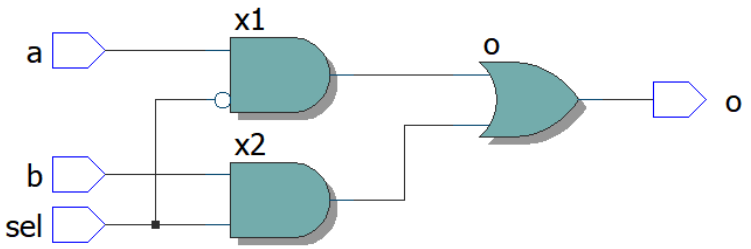


➢ **NOT16**

```
1   -- Not16 gate:
2   -- for i=0..15, f[i] = not x[i]
3
4
5   -- Library and packages
6   library  IEEE;
7   use IEEE.std_logic_1164.all;
8
9   -- Entity (Interface)
10  entity Not16 is
11
12     port(
13        x  :  in    std_logic_vector(15 downto 0);
14        f  :  out   std_logic_vector(15 downto 0));
15
16  end entity;
17
18  -- Architecture (Implementation)
19  architecture arch of Not16 is
20
21  begin
22
23     f(0) <= not x(0);
24     f(1) <= not x(1);
25     f(2) <= not x(2);
26     f(3) <= not x(3);
27     f(4) <= not x(4);
28     f(5) <= not x(5);
29     f(6) <= not x(6);
30     f(7) <= not x(7);
31     f(8) <= not x(8);
32     f(9) <= not x(9);
33     f(10) <= not x(10);
34     f(11) <= not x(11);
35     f(12) <= not x(12);
36     f(13) <= not x(13);
37     f(14) <= not x(14);
38     f(15) <= not x(15);
39
40  end architecture;
```

➢ **MUX**

```vhdl
-- Mux gate:
-- o = a, if sel = 0
-- b, in other case


-- Library and packages
library  IEEE;
use IEEE.std_logic_1164.all;

-- Entity (Interface)
entity Mux is

   port(
      a   :  in    std_logic;
      b   :  in    std_logic;
      sel:  in    std_logic;
      o   :  out   std_logic);

end entity;

-- Architecture (Implementation)
architecture arch of Mux is

signal x1, x2 : std_logic;

begin

   x1 <= ( not sel ) and a;
   x2 <= sel and b;
   o <= x1 or x2;

end architecture;

```



➢ **MUX 16**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux16 is

    Port ( sel : in STD_LOGIC;
           in0 : in STD_LOGIC_VECTOR (15 downto 0);
           in1 : in STD_LOGIC_VECTOR (15 downto 0);
           o : out STD_LOGIC_VECTOR (15 downto 0));
end Mux16;

architecture Behavioral of Mux16 is
begin

    o <= in0 when sel = '0' else in1;

end Behavioral;

```

TEST BENCH:

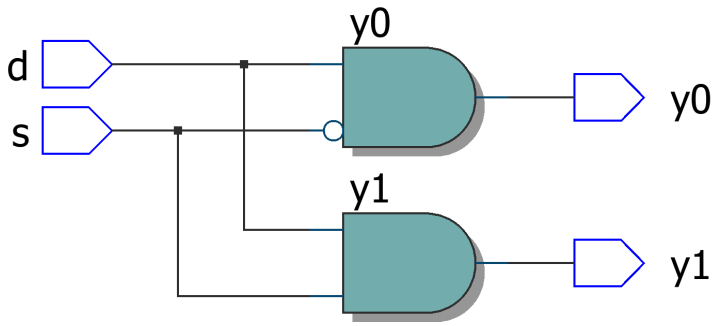
```vhdl
entity Mux16_tb is
end Mux16_tb;

architecture Behavioral of Mux16_tb is
    signal sel : STD_LOGIC := '0';
    signal in0, in1, o : STD_LOGIC_VECTOR (15 downto 0);
begin
    uut: entity work.Mux16
        port map (sel => sel, in0 => in0, in1 => in1, o => o);

    stimulus : process
    begin
        -- Test case 1
        in0 <= "0000000000000000";
        in1 <= "1111111111111111";
        sel <= '0';
        wait for 10 ns;

        -- Test case 2
        sel <= '1';
        wait for 10 ns;
    end process;
end Behavioral;

```

EXPLICACIÓN MUX 16:

- Código del Mux16: Este código define un multiplexor de 16 bits. Un multiplexor es un dispositivo que toma dos o más señales de entrada y, dependiendo de una señal de control (en este caso sel), selecciona una de ellas para enviarla a la salida. En este caso, si sel es '0', la salida (o) será igual a in0. Si sel es '1', la salida será igual a in1.

- Código del Test Bench: Este código se utiliza para probar el funcionamiento del Mux16. Se definen unas señales (in0, in1, o y sel) que se conectan al Mux16 (uut: entity work.Mux16 port map). Luego, en el proceso stimulus, se cambian los valores de las señales de entrada y se observa cómo cambia la salida. En el primer caso de prueba, sel es '0', por lo que la salida debería ser igual a in0. En el segundo caso de prueba, sel es '1', por lo que la salida debería ser igual a in1.

➢ **DMUX**




```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity dmux is
5       Port ( d : in STD_LOGIC;
6              s : in STD_LOGIC;
7              y0 : out STD_LOGIC;
8              y1 : out STD_LOGIC);
9   end dmux;
10
11  architecture Behavioral of dmux is
12  begin
13      y0 <= d and not s;
14      y1 <= d and s;
15  end Behavioral;
16
```

TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dmux_tb is
end dmux_tb;

architecture behavior of dmux_tb is
    signal d, s : std_logic := '0';
    signal y0, y1 : std_logic;
    component dmux is
        Port ( d : in STD_LOGIC;
               s : in STD_LOGIC;
               y0 : out STD_LOGIC;
               y1 : out STD_LOGIC);
    end component;
begin
    uut: dmux port map (
        d => d,
        s => s,
        y0 => y0,
        y1 => y1
    );
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        d <= '0'; s <= '0'; -- test 0 DMUX 0
        wait for 100 ns;
        d <= '1'; s <= '0'; -- test 1 DMUX 0
        wait for 100 ns;
        d <= '0'; s <= '1'; -- test 0 DMUX 1
        wait for 100 ns;
        d <= '1'; s <= '1'; -- test 1 DMUX 1
        wait for 100 ns;
        wait;
    end process;
end behavior;
```

EXPLICACIÓN DMUX:

- Código VHDL para el DMUX:

- entity dmux is ... end dmux;: Esta parte del código define la interfaz del DMUX. Tiene dos entradas (d y s) y dos salidas (y0 y y1).
- architecture Behavioral of dmux is ... end Behavioral;: Esta parte del código describe el comportamiento del DMUX. En este caso, las salidas y0 y y1 son el resultado de las operaciones AND y NOT con las entradas d y s.

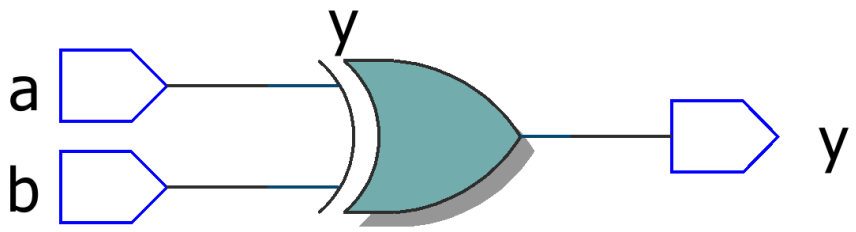
- Código VHDL para el test bench:

- entity dmux_tb is ... end dmux_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- architecture behavior of dmux_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.
- component dmux is ... end component;: Aquí se declara el DMUX como un componente que se utilizará en el test bench.

- uut: dmux port map ...;: Aquí se crea una instancia del DMUX (denominada uut, que significa "unidad bajo prueba") y se conectan las señales del test bench a las entradas y salidas del DMUX.
- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas d y s y las aplica al DMUX.

➢ **XOR**

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity xor_gate is
5
6       Port ( a : in STD_LOGIC;
7              b : in STD_LOGIC;
8              y : out STD_LOGIC);
9
10  end xor_gate;
11
12  architecture Behavioral of xor_gate is
13
14  begin
15
16      y <= a xor b;
17
18  end Behavioral;
19
```



TEST BENCH:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity xor_gate_tb is
end xor_gate_tb;

architecture behavior of xor_gate_tb is

    signal a, b : std_logic := '0';
    signal y : std_logic;

    component xor_gate is
        Port ( a : in STD_LOGIC;
               b : in STD_LOGIC;
               y : out STD_LOGIC);

    end component;

begin

--unidad bajo prueba
    uut: xor_gate port map (
        a => a,
        b => b,
        y => y
    );

    stim_proc: process

    begin

        -- hold reset state for 100 ns.
        wait for 100 ns;
        a <= '0'; b <= '0'; -- test 0 XOR 0
        wait for 100 ns;
        a <= '1'; b <= '0'; -- test 1 XOR 0
        wait for 100 ns;
        a <= '0'; b <= '1'; -- test 0 XOR 1
        wait for 100 ns;
        a <= '1'; b <= '1'; -- test 1 XOR 1
        wait for 100 ns;
        wait;

    end process;

end behavior;
```

❖ **Explicación XOR:**

- Código VHDL para la compuerta XOR:

- entity xor_gate is ... end xor_gate;: Esta parte del código define la interfaz de la compuerta XOR. Tiene dos entradas (a y b) y una salida (y).

- architecture Behavioral of xor_gate is ... end Behavioral;: Esta parte del código describe el comportamiento de la compuerta XOR. En este caso, la salida y es el resultado de la operación XOR (xor) de las entradas a y b.

- Código VHDL para el test bench:

- entity xor_gate_tb is ... end xor_gate_tb;: Esta parte del código define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.

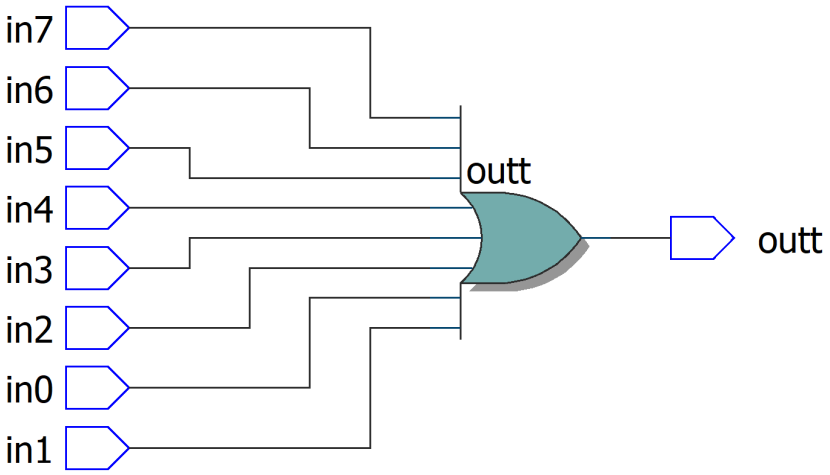
- architecture behavior of xor_gate_tb is ... end behavior;: Esta parte del código describe el comportamiento del test bench.

- component xor_gate is ... end component;: Aquí se declara la compuerta XOR como un componente que se utilizará en el test bench.

- uut: xor_gate port map ...;: Aquí se crea una instancia de la compuerta XOR (denominada uut, que significa "unidad bajo prueba") y se conectan las señales del test bench a las entradas y salidas de la compuerta XOR.

- stim_proc: process ... end process;: Este es el proceso que genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas a y b, y las aplica a la compuerta XOR.

➢ **Or8Way**




```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Or8Way is
5       Port ( in0, in1, in2, in3, in4, in5, in6, in7 : in STD_LOGIC;
6              outt : out STD_LOGIC);
7   end Or8Way;
8
9   architecture Behavioral of Or8Way is
10  begin
11      outt <= in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7;
12  end Behavioral;
13
```

TEST BENCH:

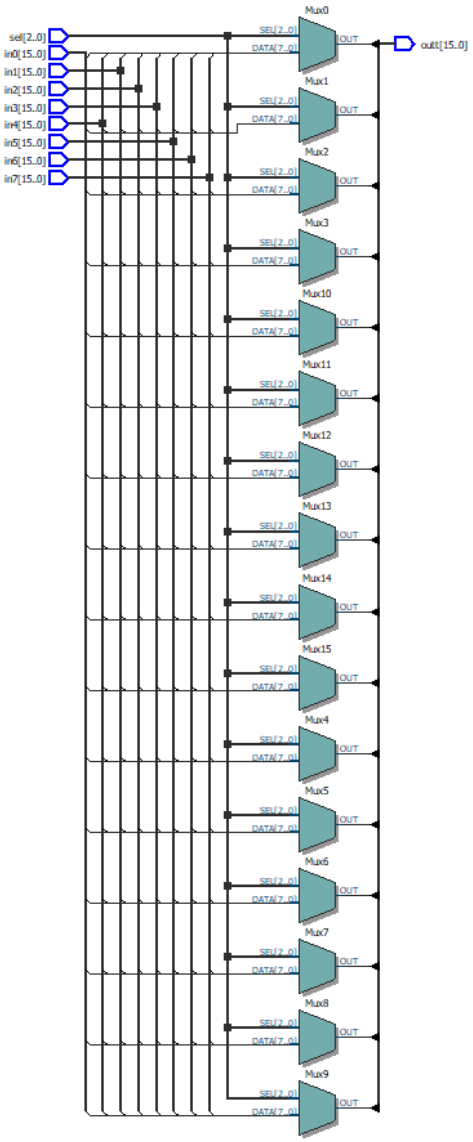
```
entity Or8Way_tb is
end Or8Way_tb;

architecture Behavioral of Or8Way_tb is

    signal in0, in1, in2, in3, in4, in5, in6, in7, outt : STD_LOGIC;

begin

    uut: entity work.Or8Way

        port map (in0 => in0, in1 => in1, in2 => in2, in3 => in3, in4 => in4, in5 => in5, in6 => in6, in7 => in7, outt => outt);

    stimulus : process
    begin
        -- Test case 1
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 2
        in0 <= '1'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 3
        in0 <= '0'; in1 <= '1'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 4
        in0 <= '0'; in1 <= '0'; in2 <= '1'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 5
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '1'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 6
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '1'; in5 <= '0'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 7
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '1'; in6 <= '0'; in7 <= '0';
        wait for 10 ns;

        -- Test case 8
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '1'; in7 <= '0';
        wait for 10 ns;

        -- Test case 9
        in0 <= '0'; in1 <= '0'; in2 <= '0'; in3 <= '0'; in4 <= '0'; in5 <= '0'; in6 <= '0'; in7 <= '1';
        wait for 10 ns;

    end process;
end Behavioral;
```

EXPLICACIÓN OR 8 WAY:

- Código del Or8Way: Este código define un componente llamado Or8Way que tiene ocho entradas de 1 bit (in0 a in7) y una salida de 1 bit (ouTt). La operación que realiza este componente es una operación OR bit a bit en las ocho entradas. Esto significa que si al menos una de las entradas es ‘1’, la salida será ‘1’. Si todas las entradas son ‘0’, la salida será ‘0’. Esta operación se realiza en la línea outt <= in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7;.

- Código del Test Bench: Este código se utiliza para probar el funcionamiento del Or8Way. Primero, se definen unas señales que se conectan al Or8Way en la línea uut: entity work.Or8Way port map (in0 => in0, in1 => in1, in2 => in2, in3 => in3, in4 => in4, in5 => in5, in6 => in6, in7 => in7, out => out);. Luego, en el proceso stimulus, se cambian los valores de las señales de entrada y se observa cómo cambia la salida. En cada caso de prueba, solo una de las entradas es ‘1’, por lo que la salida debería ser ‘1’. En el primer caso de prueba, todas las entradas son ‘0’, por lo que la salida debería ser ‘0’. Los casos de prueba se ejecutan uno tras otro, con un retardo de 10 ns entre ellos.

➢ **Mux8Way16**




```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux8Way16 is

    Port ( sel : in STD_LOGIC_VECTOR (2 downto 0);
           in0, in1, in2, in3, in4, in5, in6, in7 : in STD_LOGIC_VECTOR (15 downto 0);
           outt : out STD_LOGIC_VECTOR (15 downto 0));

end Mux8Way16;

architecture Behavioral of Mux8Way16 is
begin

    with sel select
        outt <= in0 when "000",
                in1 when "001",
                in2 when "010",
                in3 when "011",
                in4 when "100",
                in5 when "101",
                in6 when "110",
                in7 when others;

end Behavioral;
```

TEST BENCH:

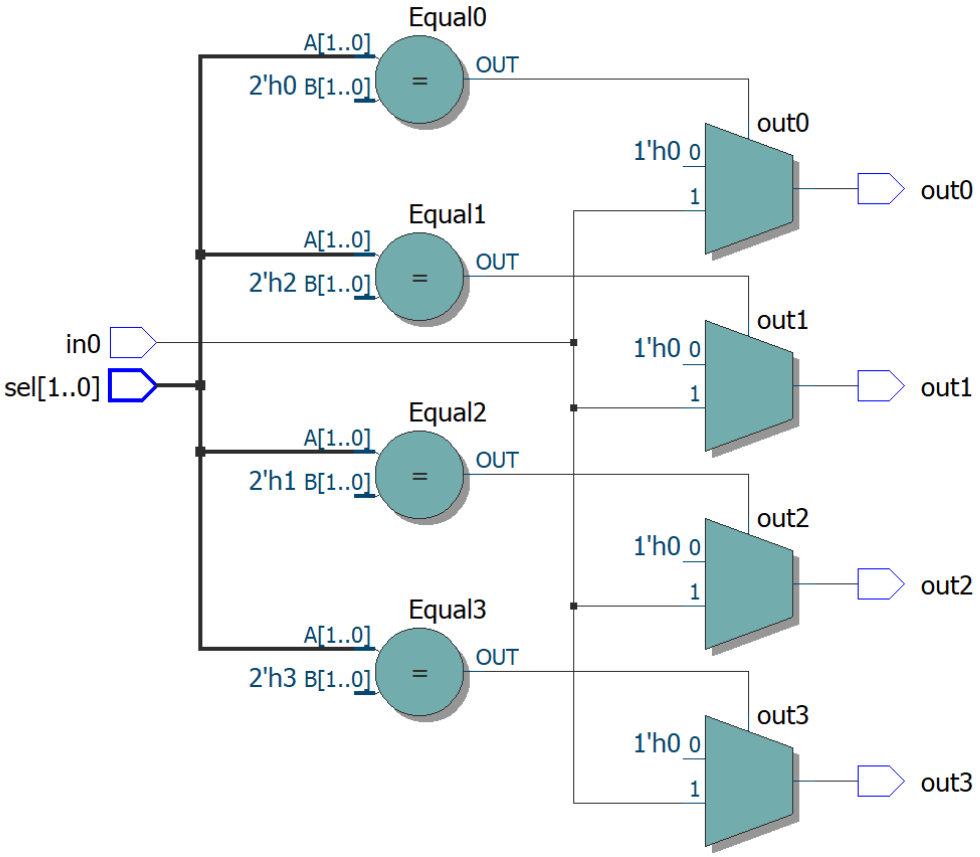
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux8Way16_tb is
end Mux8Way16_tb;

architecture Behavioral of Mux8Way16_tb is

    signal sel : STD_LOGIC_VECTOR (2 downto 0);
    signal in0, in1, in2, in3, in4, in5, in6, in7, outt : STD_LOGIC_VECTOR (15 downto 0);

begin

    uut: entity work.Mux8Way16
        port map (sel => sel, in0 => in0, in1 => in1, in2 => in2, in3 => in3, in4 => in4, in5 => in5, in6 => in6, in7 => in7, outt => outt);

    stimulus : process
    begin

        -- Test case 1
        in0 <= "0000000000000000"; in1 <= "1111111111111111"; in2 <= "0000000000000000"; in3 <= "1111111111111111";
        in4 <= "0000000000000000"; in5 <= "1111111111111111"; in6 <= "0000000000000000"; in7 <= "1111111111111111";
        sel <= "000";
        wait for 10 ns;

        -- Test case 2
        sel <= "001";
        wait for 10 ns;

        -- Test case 3
        sel <= "010";
        wait for 10 ns;

        -- Test case 4
        sel <= "011";
        wait for 10 ns;

        -- Test case 5
        sel <= "100";
        wait for 10 ns;

        -- Test case 6
        sel <= "101";
        wait for 10 ns;

        -- Test case 7
        sel <= "110";
        wait for 10 ns;

        -- Test case 8
        sel <= "111";
        wait for 10 ns;

    end process;
```

EXPLICACIÓN MUX 8 WAY 16:

- Código del Mux8Way16: Este código define un componente llamado Mux8Way16 que tiene ocho entradas de 16 bits (in0 a in7) y una salida de 16 bits (outt). La operación que realiza este componente es seleccionar una de las ocho entradas basándose en una señal de selección de 3 bits (sel). Esto se realiza en el bloque with sel select. Dependiendo del valor de sel, la salida será igual a una de las ocho entradas. Por ejemplo, si sel es “000”, la salida será igual a in0. Si sel es “001”, la salida será igual a in1, y así sucesivamente. Si sel es cualquier otro valor (en este caso, solo puede ser “111”), la salida será igual a in7.

- Código del banco de pruebas (Test Bench): Este código se utiliza para probar el funcionamiento del Mux8Way16. Primero, se definen unas señales que se conectan al Mux8Way16 en la línea uut: entity work.Mux8Way16 port map (sel => sel, in0 => in0, in1 => in1, in2 => in2, in3 => in3, in4 => in4, in5 => in5, in6 => in6, in7 => in7, outt => outt);. Luego, en el proceso stimulus, se cambian los valores de las señales de entrada y la señal de selección, y se observa cómo cambia la salida. En cada caso de prueba, sel toma un valor diferente, por lo que la salida debería ser igual a la entrada correspondiente. Los casos de prueba se ejecutan uno tras otro, con un retardo de 10 ns entre ellos.

➢ **DMux4Way**




```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DMux4Way is

    Port ( sel : in STD_LOGIC_VECTOR (1 downto 0);
           in0 : in STD_LOGIC;
           out0, out1, out2, out3 : out STD_LOGIC);

end DMux4Way;

architecture Behavioral of DMux4Way is

begin

    out0 <= in0 when sel = "00" else '0';
    out1 <= in0 when sel = "01" else '0';
    out2 <= in0 when sel = "10" else '0';
    out3 <= in0 when sel = "11" else '0';

end Behavioral;
```

TEST BENCH:

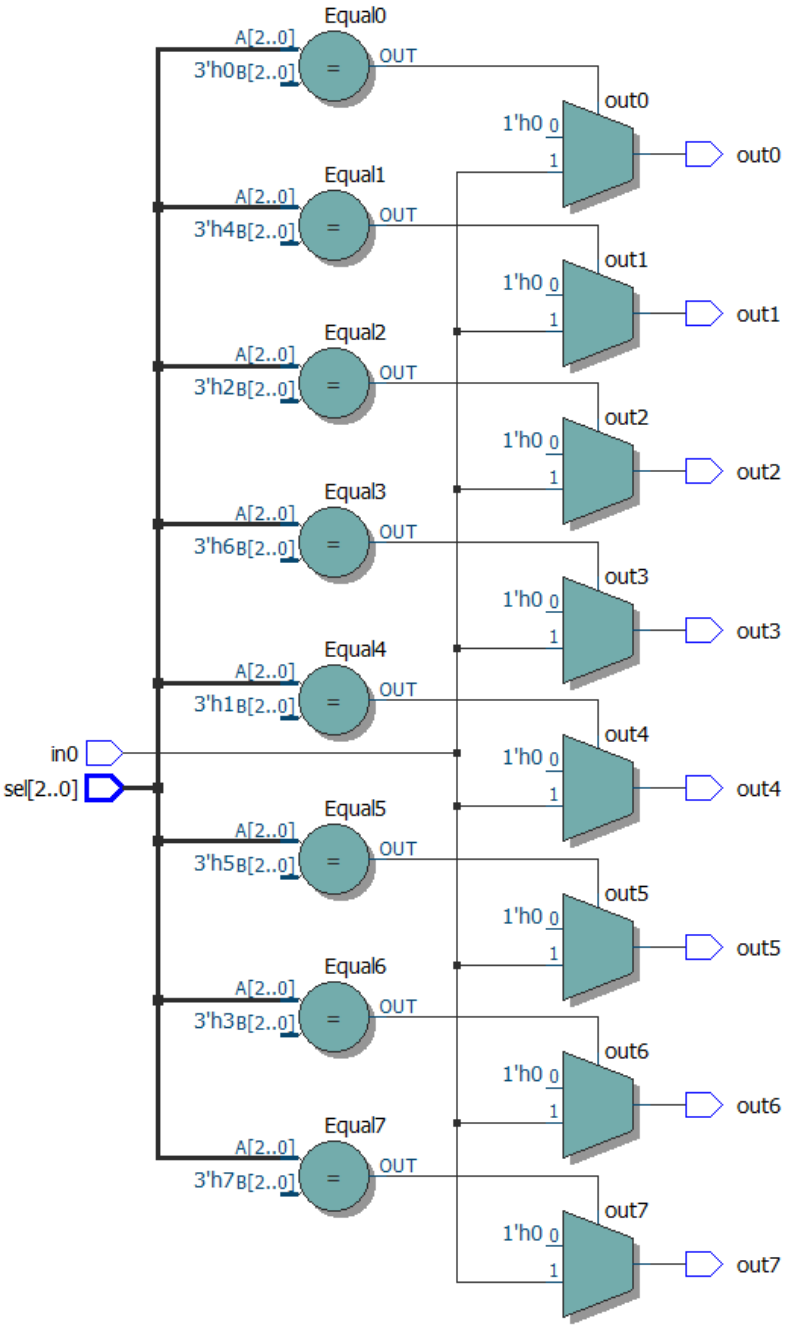
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DMux4Way_tb is
end DMux4Way_tb;

architecture Behavioral of DMux4Way_tb is

    signal sel : STD_LOGIC_VECTOR (1 downto 0);
    signal in0, out0, out1, out2, out3 : STD_LOGIC;

begin

    uut: entity work.DMux4Way

        port map (sel => sel, in0 => in0, out0 => out0, out1 => out1, out2 => out2, out3 => out3);

    stimulus : process

    begin

        -- Test case 1
        in0 <= '1'; sel <= "00";
        wait for 10 ns;

        -- Test case 2
        sel <= "01";
        wait for 10 ns;

        -- Test case 3
        sel <= "10";
        wait for 10 ns;

        -- Test case 4
        sel <= "11";
        wait for 10 ns;

    end process;

end Behavioral;
```

EXPLICACIÓN DMUX 4 WAY:

- Código del DMux4Way: Este código define un componente llamado DMux4Way que tiene una entrada de 1 bit (in0) y cuatro salidas de 1 bit (out0 a out3). La operación que realiza este componente es canalizar la entrada a una de las cuatro salidas basándose en una señal de selección de 2 bits (sel). Esto se realiza en las líneas out0 <= in0 when sel = "00" else '0'; y similares. Dependiendo del valor de sel, la entrada será canalizada a una de las salidas. Por ejemplo, si sel es “00”, out0 será igual a in0 y las demás salidas serán ‘0’. Si sel es “01”, out1 será igual a in0 y las demás salidas serán ‘0’, y así sucesivamente.

- Código del Test Bench: Este código se utiliza para probar el funcionamiento del DMux4Way. Primero, se definen unas señales que se conectan al DMux4Way en la línea uut: entity work.DMux4Way port map (sel => sel, in0 => in0, out0 => out0, out1 => out1, out2 => out2, out3 => out3);. Luego, en el proceso stimulus, se cambia el valor de la señal de entrada y la señal de selección, y se observa cómo cambian las salidas. En cada caso de prueba, sel toma un valor diferente, por lo que la salida correspondiente debería ser igual a in0 y las demás deberían ser ‘0’. Los casos de prueba se ejecutan uno tras otro, con un retardo de 10 ns entre ellos.

➢ **DMux8Way**


Equal0
A[2..0]
3'h0
B[2..0]
=
OUT
Equal1
A[2..0]
3'h4
B[2..0]
=
OUT
Equal2
A[2..0]
3'h2
B[2..0]
=
OUT
Equal3
A[2..0]
3'h6
B[2..0]
=
OUT
Equal4
A[2..0]
3'h1
B[2..0]
=
OUT
Equal5
A[2..0]
3'h5
B[2..0]
=
OUT
Equal6
A[2..0]
3'h3
B[2..0]
=
OUT
Equal7
A[2..0]
3'h7
B[2..0]
=
OUT
in0
sel[2..0]
out0
1'h0
0
1
out0
out1
1'h0
0
1
out1
out2
1'h0
0
1
out2
out3
1'h0
0
1
out3
out4
1'h0
0
1
out4
out5
1'h0
0
1
out5
out6
1'h0
0
1
out6
out7
1'h0
0
1
out7

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DMux8Way is

    Port ( sel : in STD_LOGIC_VECTOR (2 downto 0);
           in0 : in STD_LOGIC;
           out0, out1, out2, out3, out4, out5, out6, out7 : out STD_LOGIC);

end DMux8Way;

architecture Behavioral of DMux8Way is

begin
    out0 <= in0 when sel = "000" else '0';
    out1 <= in0 when sel = "001" else '0';
    out2 <= in0 when sel = "010" else '0';
    out3 <= in0 when sel = "011" else '0';
    out4 <= in0 when sel = "100" else '0';
    out5 <= in0 when sel = "101" else '0';
    out6 <= in0 when sel = "110" else '0';
    out7 <= in0 when sel = "111" else '0';

end Behavioral;
```

TEST BENCH:

```vhdl
use IEEE.STD_LOGIC_1164.ALL;

entity DMux8Way_tb is

end DMux8Way_tb;

architecture Behavioral of DMux8Way_tb is

    signal sel : STD_LOGIC_VECTOR (2 downto 0);
    signal in0, out0, out1, out2, out3, out4, out5, out6, out7 : STD_LOGIC;

begin

    uut: entity work.DMux8Way

        port map (sel => sel, in0 => in0, out0 => out0, out1 => out1, out2 => out2, out3 => out3, out4 => out4, out5 => out5, out6 => out6, out7 => out7);

    stimulus : process
    begin
        -- Test case 1
        in0 <= '1'; sel <= "000";
        wait for 10 ns;

        -- Test case 2
        sel <= "001";
        wait for 10 ns;

        -- Test case 3
        sel <= "010";
        wait for 10 ns;

        -- Test case 4
        sel <= "011";
        wait for 10 ns;

        -- Test case 5
        sel <= "100";
        wait for 10 ns;

        -- Test case 6
        sel <= "101";
        wait for 10 ns;

        -- Test case 7
        sel <= "110";
        wait for 10 ns;

        -- Test case 8
        sel <= "111";
        wait for 10 ns;

    end process;
end Behavioral;
```

EXPLICACIÓN DMux 8 Way:

- Código del DMux8Way: Este código define un componente llamado DMux8Way que tiene una entrada de 1 bit (in0) y ocho salidas de 1 bit (out0 a out7). La operación que realiza este componente es canalizar la entrada a una de las ocho salidas basándose en una señal de selección de 3 bits (sel). Esto se realiza en las líneas out0 <= in0 when sel = "000" else '0'; y similares. Dependiendo del valor de sel, la entrada será canalizada a una de las salidas. Por ejemplo, si sel es “000”, out0 será igual a in0 y las demás salidas serán ‘0’. Si sel es “001”,

out1 será igual a in0 y las demás salidas serán ‘0’, y así sucesivamente. Si sel es “010”, out2 será igual a in0 y las demás salidas serán ‘0’, y así sucesivamente hasta sel igual a “111”, donde out7 será igual a in0 y las demás salidas serán ‘0’.

- Código del Test Bench: Este código se utiliza para probar el funcionamiento del DMux8Way. Primero, se definen unas señales que se conectan al DMux8Way en la línea uut: entity work.DMux8Way port map (sel => sel, in0 => in0, out0 => out0, out1 => out1, out2 => out2, out3 => out3, out4 => out4, out5 => out5, out6 => out6, out7 => out7);. Luego, en el proceso stimulus, se cambia el valor de la señal de entrada y la señal de selección, y se observa cómo cambian las salidas. En cada caso de prueba, sel toma un valor diferente, por lo que la salida correspondiente debería ser igual a in0 y las demás deberían ser ‘0’. Los casos de prueba se ejecutan uno tras otro, con un retardo de 10 ns entre ellos.