PROYECTOS PRIMER CORTE-CIRCUITOS DIGITALES 2

OSCAR JAVIER MACÍAS PIZO

Trabajo presentado a:

Ing. Carlos Hernán Tobar Arteaga

UNIVERSIDAD DEL CAUCA

FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

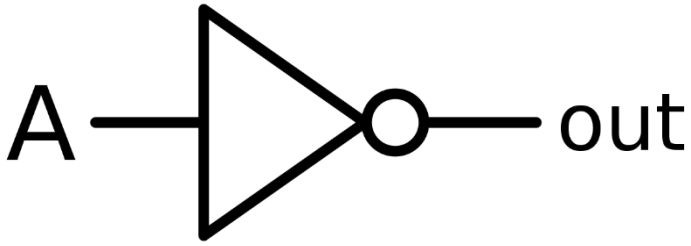
POPAYAN

2023

RESUMEN PROYECTO 1

Para este proyecto se planteó realizar diferentes chips en lenguaje VHDL que permitiera representar algunas compuertas lógicas que pueden ser útiles para el desarrollo de la asignatura en genera. Por lo tanto, en este documento se mostrará el trabajo realizado para cada compuerta evidenciando el código necesario para representar las compuertas lógicas ya mencionadas con su respectivo archivo de pruebas también hecho en lenguaje VHDL

- **NOT GATE**
  Este tipo de compuerta se representa de la siguiente manera



Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

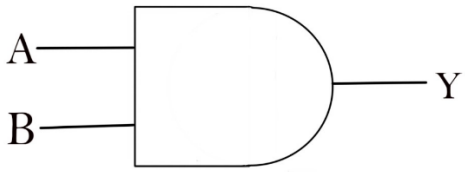
```
-- Architecture (Implementation)
architecture arch of NotGate is
begin
    F <= x nand x;
end architecture;
```

Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Start of the test of NotGate"
 severity note;

 x_test <= '0';
 wait for 1 ns;
 assert f_test = '1'
   report "Falla para x = 0"
   severity failure;

 x_test <= '1';
 wait for 10 ns;
 assert f_test = '0'
   report "Falla para x = 1"
   severity failure;
```

Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **AND GATE**
  Este tipo de compuerta se representa de la siguiente manera



Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

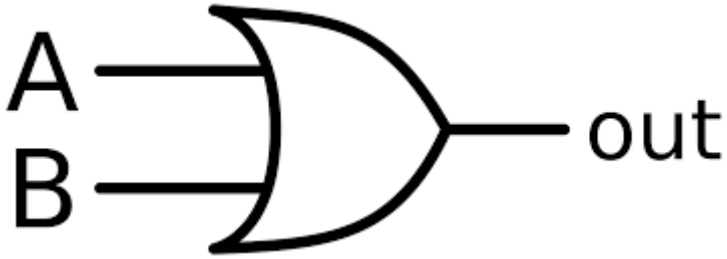
```
architecture arch of AndGate is
    signal x1 : std_logic;

begin
    x1 <= A nand B;
    X <= not x1;
```

Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test de AndGate"
       severity note;

       A_test <= '0';
       B_test <= '0';
       wait for 10 ns;
       assert X_test = '0'
              report "Falla para {A,B,X) = {0,0,0}"
              severity failure;

       A_test <= '0';
       B_test <= '1';
       wait for 10 ns;
       assert X_test = '0'
              report "Falla para {A,B,X) = {0,0,1}"
              severity failure;
```

Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **OR GATE**
  Este tipo de compuerta se representa de la siguiente manera



Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
architecture arch of OrGate is
signal x1, x2 : std_logic;
begin
      x1 <= not A;
      x2 <= not B;
      X <= x1 nand x2;
```

Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test de Orgate"
      severity note;

      A_test <= '0';
      B_test <= '0';
      wait for 1 ns;
      assert X_test = '1'
        report "Falla en {A,B}={0,0}"
        severity failure;

      A_test <= '0';
      B_test <= '1';
      wait for 1 ns;
      assert X_test = '0'
        report "Falla en {A,B}={0,1}"
        severity failure;

      A_test <= '1';
      B_test <= '0';
      wait for 10 ns;
      assert X_test = '0'
        report "Falla en {A,B}={1,0}"
        severity failure;
```

Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **XOR GATE**
  Este tipo de compuerta se representa de la siguiente manera

Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

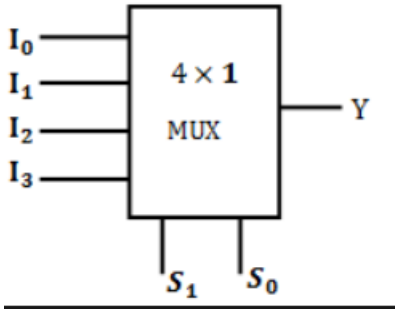
```
architecture arch of XorGate is
    signal x1, x2, x3, x4 : std_logic;
begin
    x1 <= not A;
    x2 <= not B;
    x3 <= x1 and B;
    x4 <= x2 and A;
    X  <= x3 or x4;
end architecture;
```

Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
begin
 report "Inicio de test para XorGate"
  severity note;

  A_test <= '0';
  B_test <= '0';
  wait for 1 ns;
  assert X_test = '0'
    report "Falla para A = 0  and B = 0"
    severity failure;

  A_test <= '0';
  B_test <= '1';
  wait for 1 ns;
  assert X_test = '1'
    report "Falla en A = 0 and B = 1"
    severity failure;

  A_test <= '1';
  B_test <= '0';
  wait for 1 ns;
  assert X_test = '1'
    report "Falla en A = 1 and B = 0"
    severity failure;
```

Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **MUX**
  El multiplexor se representa de la siguiente forma gráficamente



  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo del multiplexor

```
architecture arch of MuxGate is
    signal x1, x2 : std_logic;

begin
        x1 <= ( not s ) and A;
        x2 <= s and B;
        X <= x1 or x2;
end architecture;
```

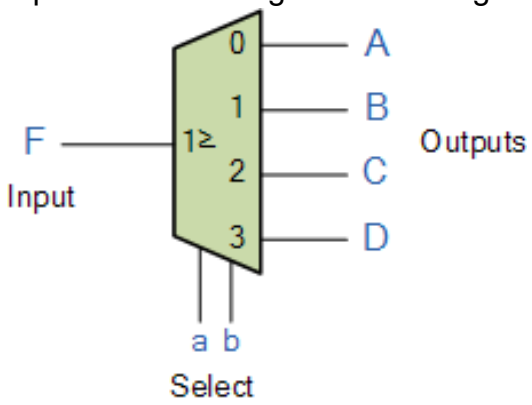
  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test para Mux"
severity note;

A_test   <= '0';
B_test   <= '1';
s_test <= '0';
wait for 1 ns;
assert X_test = '0'
  report "Falla en A=0, B=1 y s=0"
  severity failure;

A_test   <= '0';
B_test   <= '0';
s_test <= '0';
wait for 1 ns;
assert X_test = '0'
  report "Falla en A=0, B=0 y s=0"
  severity failure;

A_test   <= '1';
B_test   <= '0';
s_test <= '0';
wait for 1 ns;
assert X_test = '1'
  report "Falla en A=1, B=0 y s=0"
  severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **MUX**

El demultiplexor se representa de la siguiente forma gráficamente



Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo del multiplexor

```
architecture arch of DMuxGate is

   begin
        X <= (not s) and A;
        Y <= A and s;
end architecture;
```

Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test de DMuxGate"
      severity note;

      A_test <= '0';
      s_test <= '0';
      wait for 1 ns;
      assert X_test = '0';
      assert Y_test = '0';
        report "Falla para A=0 y s=0"
        severity failure;

      A_test <= '0';
      s_test <= '1';
      wait for 1 ns;
      assert X_test = '0';
      assert Y_test = '0';
        report "Falla para A=0 y s=1"
        severity failure;
```

Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **NOT16**
  Esta compuerta es representada como la misma compuerta not ya expuesta pero ahora se trabajará con una cantidad de 16 bits, buscando el mismo objetivo que se mostró anteriormente

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
-- Architecture (Implementation)
architecture arch of Not16 is
begin
    X(0) <= not A(0);
    X(1) <= not A(1);
    X(2) <= not A(2);
    X(3) <= not A(3);
    X(4) <= not A(4);
    X(5) <= not A(5);
    X(6) <= not A(6);
    X(7) <= not A(7);
    X(8) <= not A(8);
    X(9) <= not A(9);
    X(10) <= not A(10);
    X(11) <= not A(11);
    X(12) <= not A(12);
    X(13) <= not A(13);
    X(14) <= not A(14);
    X(15) <= not A(15);
end architecture;
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test para NotGate"
 severity note;

 A_test <= "0000000000000000";
 wait for 1 ns;
 assert X_test = "1111111111111111"
   report "Failure para A = [0000000000000000]"
   severity failure;

 A_test <= "1111111111111111";
 wait for 1 ns;
 assert X_test = "0000000000000000"
   report "Failure para A = [1111111111111111]"
   severity failure;

 A_test <= "0100110000000011";
 wait for 1 ns;
 assert X_test = "1011001111111100"
   report "Failure para A = [0100110000000011]"
   severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **AND16**

  Esta compuerta es representada como la misma compuerta AND ya expuesta pero ahora se trabajará con una cantidad de 16 bits, buscando el mismo objetivo que se mostró anteriormente

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
begin
    x1(0) <= not A(0);
    x2(0) <= not B(0);
    X(0) <= x1(0) nor x2(0);

    x1(1) <= not A(1);
    x2(1) <= not B(1);
    X(1) <= x1(1) nor x2(1);

    x1(2) <= not A(2);
    x2(2) <= not B(2);
    X(2) <= x1(2) nor x2(2);

    x1(3) <= not A(3);
    x2(3) <= not B(3);
    X(3) <= x1(3) nor x2(3);

    x1(4) <= not A(4);
    x2(4) <= not B(4);
    X(4) <= x1(4) nor x2(4);

    x1(5) <= not A(5);
    x2(5) <= not B(5);
    X(5) <= x1(5) nor x2(5);

    x1(6) <= not A(6);
    x2(6) <= not B(6);
    X(6) <= x1(6) nor x2(6);

    x1(7) <= not A(7);
    x2(7) <= not B(7);
    X(7) <= x1(7) nor x2(7);
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
report "Inicio del test de pruebas para And16"
 severity note;

 A_test <= "0000000000000000";
 B_test <= "0000000000000000";
 wait for 1 ns;
 assert X_test = "0000000000000000"
   report "Falla en A=0000000000000000 y B=0000000000000000"
   severity failure;

 A_test <= "0000000000000000";
 B_test <= "1111111111111111";
 wait for 1 ns;
 assert X_test = "0000000000000000"
   report "Falla en A=0000000000000000 y B=1111111111111111"
   severity failure;

 A_test <= "1111111111111111";
 B_test <= "1111111111111111";
 wait for 1 ns;
 assert X_test = "1111111111111111"
   report "Falla en A=1111111111111111 y B=1111111111111111"
   severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **OR16**

  Esta compuerta es representada como la misma compuerta OR ya expuesta pero ahora se trabajará con una cantidad de 16 bits, buscando el mismo objetivo que se mostró anteriormente

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
begin
   x1(0) <= A(0) nand A(0);
   x2(0) <= B(0) nand B(0);
   x1(1) <= A(1) nand A(1);
   x2(1) <= B(1) nand B(1);
   x1(2) <= A(2) nand A(2);
   x2(2) <= B(2) nand B(2);
   x1(3) <= A(3) nand A(3);
   x2(3) <= B(3) nand B(3);
   x1(4) <= A(4) nand A(4);
   x2(4) <= B(4) nand B(4);
   x1(5) <= A(5) nand A(5);
   x2(5) <= B(5) nand B(5);
   x1(6) <= A(6) nand A(6);
   x2(6) <= B(6) nand B(6);
   x1(7) <= A(7) nand A(7);
   x2(7) <= B(7) nand B(7);
   x1(8) <= A(8) nand A(8);
   x2(8) <= B(8) nand B(8);
   x1(9) <= A(9) nand A(9);
   x2(9) <= B(9) nand B(9);
   x1(10) <= A(10) nand A(10);
   x2(10) <= B(10) nand B(10);
   x1(11) <= A(11) nand A(11);
   x2(11) <= B(11) nand B(11);
   x1(12) <= A(12) nand A(12);
   x2(12) <= B(12) nand B(12);
   x1(13) <= A(13) nand A(13);
   x2(13) <= B(13) nand B(13);
   x1(14) <= A(14) nand A(14);
   x2(14) <= B(14) nand B(14);
   x1(15) <= A(15) nand A(15);
   x2(15) <= B(15) nand B(15);
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo de la compuerta

```
begin
    report "Inicio del test para Or16"
        severity note;

        A_test <= "0000000000000000";
        B_test <= "0000000000000000";
        wait for 1 ns;
        assert X_test = "0000000000000000"
              report "Falla en A=0000000000000000 y B=0000000000000000"
              severity failure;

        A_test <= "0000000000000000";
        B_test <= "1111111111111111";
        wait for 1 ns;
        assert X_test = "1111111111111111"
              report "Falla en A=0000000000000000 y B=1111111111111111"
              severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **MUX16**

  Este tipo de multiplexor cumple la misma función que el ya expuesto pero ahora se trabajará con una cantidad de 16 bits, buscando el mismo objetivo que se mostró anteriormente

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo del multplexor

```
begin
   mux15: MuxGate
      Port map(
         A => A(15),
         B => B(15),
         s => s,
         X => X(15)
      );
   mux14: MuxGate
      Port map(
         A => A(14),
         B => B(14),
         s => s,
         X => X(14)
      );
   mux13: MuxGate
      Port map(
         A => A(13),
         B => B(13),
         s => s,
         X => X(13)
      );
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
report "Inicio del test para Mux16"
 severity note;

 A_test <= "0000000000000000";
 B_test <= "1111111111111111";
 s_test <= '0';
 wait for 1 ns;
 assert X_test = "0000000000000000"
   report "Falla en A=0000000000000000, B=1111111111111111, s=0"
   severity failure;

 A_test <= "0000000000000000";
 B_test <= "1111111111111111";
 s_test <= '1';
 wait for 1 ns;
 assert X_test = "1111111111111111"
   report "Falla en A=0000000000000000, B=1111111111111111, s=1"
   severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **OR8WAY**

  Este tipo de compuerta cumple la misma función que el ya expuesto, pero ahora se trabajará con una cantidad de ocho entradas donde se realizará la operación OR entre ellas mismas hasta llegar a una única salida de respuesta teniendo en cuenta las entradas ya mencionadas

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
architecture arch of Or8Way is
    signal x1, x2, x3, x4, x5, x6 : std_logic := '0';

begin
    x1 <= (not A) nand (not B);
    x2 <= (not C) nand (not D);
    x3 <= (not E) nand (not F);
    x4 <= (not G) nand (not H);
    x5 <= (not x1) nand (not x2);
    x6 <= (not x3) nand (not x4);
    X <= (not x5) nand (not x6);
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
report "Inicio de test para Orgate"
severity note;

    A_test <= '0';
    B_test <= '0';
    C_test <= '0';
    D_test <= '0';
    E_test <= '0';
    F_test <= '0';
    G_test <= '0';
    H_test <= '0';
    wait for 1 ns;
    assert X_test = '0'
        report "Failure en a=0, b=0, c=0, d=0, e=0, f=0, g=0 y h=0"
        severity failure;

    A_test <= '1';
    B_test <= '0';
    C_test <= '0';
    D_test <= '0';
    E_test <= '0';
    F_test <= '0';
    G_test <= '0';
    H_test <= '0';
    wait for 1 ns;
    assert X_test = '1'
        report "Failure en a=1, b=0, c=0, d=0, e=0, f=0, g=0 y h=0"
        severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **MUX4WAY16**
  Este tipo de multiplexor cumple la misma función que el ya expuesto, pero ahora se trabajará con una cantidad de cuatro entradas donde se apicararán operaciones lógicas que permitan dar con una salida de 16 bits

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
architecture archo of Mux4Way16 is

    signal x1, x2, x3, x4 : std_logic;

begin

    x1 <= (not s1) and (not s2) and A;
    x2 <= s1 and (not s2) and B;
    x3 <= (not s1) and s2 and C;
    x4 <= s1 and s2 and D;
    X <= x1 or x2 or x3 or x4;
```

```
begin
   mux15: Mux4way16
      Port map(
         A => A(15),
         B => B(15),
         C => D(15),
         D => D(15),
         s1 => s1,
         s2 => s2,
         X => X(15)
      );
   mux14: Mux4way16
      Port map(
         A => A(14),
         B => B(14),
         C => C(14),
         D => D(14),
         s1 => s1,
         s2 => s2,
         X => X(14)
      );
   mux13: Mux4way16
      Port map(
         A => A(13),
         B => B(13),
         C => C(13),
         D => D(13),
         s1 => s1,
         s2 => s2,
         X => X(13)
      );
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
report "Inicio del test para Mux4Way16"
severity note;

A_test <= "0000000000000000";
B_test <= "0000000000000000";
C_test <= "0000000000000000";
D_test <= "0000000000000000";
s1_test <= '0';
s2_test <= '0';
wait for 1 ns;
assert X_test = "0000000000000000"
      report "Falla para A=0000000000000000, B=0000000000000000, C=0000000000000000, D=0000000000000000, s1=0 y s2=0"
      severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **MUX8WAY16**

  Este tipo de multiplexor cumple la misma función que el ya expuesto, pero ahora se trabajará con una cantidad de ocho entradas donde se apicararán operaciones lógicas que permitan dar con una salida de 16 bits

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
Mux8Way15: Mux8Way
Port map(

A => A(15),
B => B(15),
C => C(15),
D => D(15),
E => E(15),
F => F(15),
G => G(15),
H => H(15),
s1 => s1,
s2 => s2,
s3 => s3,
X => X(15)
);

Mux8Way14 : Mux8Way
Port map(

A => A(14),
B => B(14),
C => C(14),
D => D(14),
E => E(14),
F => F(14),
G => G(14),
H => H(14),
s1 => s1,
s2 => s2,
s3 => s3,
X => X(14)
);
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
A_test <= "0000000000000000";
B_test <= "0000000000000000";
C_test <= "0000000000000000";
D_test <= "0000000000000000";
E_test <= "0000000000000000";
F_test <= "0000000000000000";
G_test <= "0000000000000000";
H_test <= "0000000000000000";
s1_test <= '0';
s2_test <= '0';
s3_test <= '0';
wait for 1 ns;
assert X_test = "0000000000000000"
  report "Falla en  A=0000000000000000, B=0000000000000000, C=0000000000000000, D=0000000000000000, E=0000000000000000, F=0000000000000000, G=00000
  severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **DMUX4WAY**

  Este tipo de demultiplexor cumple la misma función que el ya expuesto, pero ahora se trabajará con una cantidad de cuatro entradas donde se apicararán operaciones lógicas que permitan dar con una única salida

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
end component;
begin
     DMux1: DMuxGate
     Port map(
          A => A,
          s => s1,
          X => x1,
          Y => x2
     );
     DMux2: DMuxGate
     Port map(
          A => x1,
          s => s2,
          X => X,
          Y => Y
     );
     DMux3: DMuxGate
     Port map(
          A => x2,
          s => s2,
          X => Z,
          Y => W
     );
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
report "Inicio del test para DMux4Way"
severity note;

A_test   <= '0';
s1_test  <= '0';
s2_test   <= '0';
wait for 1 ns;
assert X_test = '0' and Y_test = '0' and Z_test = '0' and W_test = '0'
    report "Falla en A=0, s1=0 y s2=0"
    severity failure;

A_test   <= '0';
s1_test  <= '0';
s2_test   <= '1';
wait for 1 ns;
assert X_test = '0' and Y_test = '0' and Z_test = '0' and W_test = '0'
    report "Falla en A=0, s1=0 y s2=1"
    severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

- **DMUX8WAY**

  Este tipo de demultiplexor cumple la misma función que el ya expuesto, pero ahora se trabajará con una cantidad de ocho entradas donde se apicararán operaciones lógicas que permitan dar con una única salida

  Mediante la arquitectura del chip en cuestión se plantea su respectiva arquitectura que realiza las operaciones necesarias para cumplir con el objetivo de la compuerta

```
begin

   DMux1 : DMuxGate

      Port map(
         A => A,
         s => s1,
         X => y1,
         Y => y2
      );

    DMux2 : DMuxGate

      Port map(
         A => y1,
         s => s2,
         X => y3,
         Y => y4

      );

      DMux3 : DMuxGate
      Port map(
         A => y2,
         s => s2,
         X => y5,
         Y => y6

      );
```

  Ahora se realiza un código de pruebas que permita evidenciar que efectivamente se está cumpliendo el objetivo del multiplexor

```
A_test <= '0';
s1_test <= '0';
s2_test <= '0';
s3_test <= '0';
wait for 1 ns;
assert X1_test = '0' and X2_test = '0' and X3_test = '0' and X4_test = '0' and X5_test = '0' and X6_test = '0' and X7_test = '0' and X8_test = '0'
  report "Falla en A=0, s1=0, s2=0 y s3=0"
  severity failure;

A_test <= '0';
s1_test <= '0';
s2_test <= '0';
s3_test <= '1';
wait for 1 ns;
assert X1_test = '0' and X2_test = '0' and X3_test = '0' and X4_test = '0' and X5_test = '0' and X6_test = '0' and X7_test = '0' and X8_test = '0'
  report "Falla en A=0, s1=0, s2=0 y s3=1"
  severity failure;
```

  Es de aclarar que este tipo de pruebas se pueden comprobar en el aplicativo ModelSim que mostrará cuando efectivamente el test se completó de manera exitosa

RESUMEN PROYECTO 2

- **HALF ADDER**

  Un half adder, también conocido como "semisumador", es aquel circuito lógico utilizado en electrónica digital y en diseño de circuitos digitales para sumar dos bits individuales. Su función principal es realizar la suma de dos números binarios de un solo bit y generar dos salidas: una suma y un acarreo

  La arquitectura del circuito demuestra cuales son las operaciones necesarias para cumplir lo ya mencionado, teniendo en cuenta las entradas y sus respectivas salidas (suma y acarreo)

```
entity HalfAdder is
    port(
        A   :  in    std_logic;
        B   :  in    std_logic;
        sum : out std_logic;
        carry   :  out   std_logic

    );
end entity;

architecture HalfAdder_arch of HalfAdder is
    begin
          sum <= A xor B;
          carry <= A and B;
```

  Igualmente se realiza un código de test de pruebas que permita comprobar que si se está cumpliendo la respectiva función del semisumador en diferentes casos

```
report "Start of the test of HalfAdder"
severity note;

A_test   <= '0';
B_test   <= '0';
wait for 1 ns;
assert sum_test = '0' and carry_test = '0'
  report "Falla para A=0 y B=0"
  severity failure;

A_test   <= '0';
B_test   <= '1';
wait for 1 ns;
assert sum_test = '1' and carry_test = '0'
  report "Falla para A=0 y B=1"
  severity failure;
```

- **FULL ADDER**

  Este sumador completo tiene como principal objetivo realizar la suma de tres bits de entrada, representados por dos bits individuales como en el caso anterior y asimismo, un bit de acarreo. Igualmente, como en el caso anterior se generan únicamente dos salidas: la suma y su respectivo acarreo

  En el código se puedes observar como la arquitectura del sistema representa con x1 el acarreo adicional que se estará sumando a los dos bits adicionales

```
architecture arch of FullAdder is

    signal x1, x2, x3 : std_logic;

    component HalfAdder is
        Port(
            A : in std_logic;
            B : in std_logic;
            sum : out std_logic;
            carry : out std_logic
        );
    end component;
begin
    HalfAdder1 : HalfAdder
        Port map(
            A => A,
            B => B,
            sum => x1,
            carry => x2
        );
    HalfAdder2 : HalfAdder
        Port map(
            A => x1,
            B => C,
            sum => sum,
            carry => x3
        );
    carry <= x2 or x3;
```

  Asimismo, se comprueba el buen funcionamiento del sumador completo haciendo un test de pruebas aplicando diferentes casos y que efectivamente se identifique su veracidad

```
report "Inicio de test para FullAdder"
severity note;

A_test   <= '0';
B_test   <= '0';
C_test   <= '0';
wait for 1 ns;
assert sum_test = '0' and carry_test = '0'
  report "Falla para A=0, B=0 y C=0"
  severity failure;

A_test   <= '0';
B_test   <= '0';
C_test   <= '1';
wait for 1 ns;
assert sum_test = '1' and carry_test = '0'
  report "Falla para A=0, B=0 y C=1"
  severity failure;
```

- **ADD16**

  Este tipo de chip cumple un objetivo muy parecido al ya mencionado en los dos casos anteriores. La única particularidad de este caso es que sirve para realizar la suma de números con 16 bits de longitud cada uno. En el código es posible evidenciarlo de mejor manera

```
begin
FullAdder15: FullAdder
   Port map(
      A => A(15),
      B => B(15),
      C => x1(15),
      sum => sum(15),
      carry => carry
      );

FullAdder14: FullAdder
   Port map(
      A => A(14),
      B => B(14),
      C => x1(14),
      sum => sum(14),
      carry => x1(15)
      );

FullAdder13: FullAdder
   Port map(
      A => A(13),
      B => B(13),
      C => x1(13),
      sum => sum(13),
      carry => x1(14)
      );
```

  Se comprueba asimismo la veracidad de los resultados realizando un test de pruebas que permita mostrar como se tomaría la suma de los respectivos términos de 16 de bits obteniendo como salidas su respectiva suma y acarreo

```
report "Inicio del test para Add16"
severity note;

A_test   <= "0000000000000000";
B_test   <= "0000000000000000";
C_test   <= '0';
wait for 1 ns;
assert (sum_test = "0000000000000000") and (carry_test = '0')
  report "Falla para A=0000000000000000, B=0000000000000000 y C=0"
  severity failure;

A_test   <= "1111111111111111";
B_test   <= "1111111111111111";
C_test   <= '0';
wait for 1 ns;
assert (sum_test = "0000000000000000") and (carry_test = '1')
  report "Falla para A=1111111111111111, B=1111111111111111 y C=0"
  severity failure;

report "Test exitoso"
severity note;
```

- **INC16**
  Este tipo de chip es bastante fácil de entender, ya que como su mismo nombre lo dice es un incrementador, pero con la particularidad de que se encargar de 1 al numero binario especifico de entrada de 16 bits. Por lo tanto, se aplica el mismo procedimiento para que las salidas, la suma y el acarreo quede en términos de 16 bits

```
sum(15) <= A(15) xor B(15);
sum(14) <= A(14) xor B(14);
sum(13) <= A(13) xor B(13);
sum(12) <= A(12) xor B(12);
sum(11) <= A(11) xor B(11);
sum(10) <= A(10) xor B(10);
sum(9)  <= A(9) xor B(9);
sum(8)  <= A(8) xor B(8);
sum(7)  <= A(7) xor B(7);
sum(6)  <= A(6) xor B(6);
sum(5)  <= A(5) xor B(5);
sum(4)  <= A(4) xor B(4);
sum(3)  <= A(3) xor B(3);
sum(2)  <= A(2) xor B(2);
sum(1)  <= A(1) xor B(1);
sum(0)  <= A(0) xor B(0);

carry(15) <= A(15) and B(15);
carry(14) <= A(14) and B(14);
carry(13) <= A(13) and B(13);
carry(12) <= A(12) and B(12);
carry(11) <= A(11) and B(11);
carry(10) <= A(10) and B(10);
carry(9)  <= A(9) and B(9);
carry(8)  <= A(8) and B(8);
carry(7)  <= A(7) and B(7);
carry(6)  <= A(6) and B(6);
carry(5)  <= A(5) and B(5);
carry(4)  <= A(4) and B(4);
carry(3)  <= A(3) and B(3);
carry(2)  <= A(2) and B(2);
carry(1)  <= A(1) and B(1);
carry(0)  <= A(0) and B(0);
```

  Y se realiza el respectivo test de pruebas para dar con los resultados esperados

```
A_test <= "0000000000000000";
B_test <= "0000000000000000";
wait for 1 ns;
assert (sum_test = "0000000000000000") and (carry_test = "0000000000000000")
  report "Falla para A=0000000000000000 y B=0000000000000000"
  severity failure;

A_test <= "0000000000000000";
B_test <= "1111111111111111";
wait for 1 ns;
assert (sum_test = "1111111111111111") and (carry_test = "0000000000000000")
  report "Falla para A=0000000000000000 y B=1111111111111111"
  severity failure;
```

- **ALU**
  Este chip tiene como principal objetivo ejecutar diferentes operaciones aritméticas como suma, resta, multiplicación o división y asimismo, operaciones lógicas como AND, OR, NOT, XOR. Estas operaciones lógicas son las que se han utilizado en ambos proyectos, por lo tanto estas serán las aplicadas a nuestra ALU como se muestra a continuación.

```
architecture arch of ALU is
    signal x1, x2, x3, x4 : std_logic := '0';
    signal y1, y2, y3, y4 : std_logic;

    begin
    x1 <= (not A) nand (not B);
    x2 <= (A and(not B)) or ((not A) and B);
    x3 <= (not A) nor (not B);
    x4 <= not A;
    y1 <= (not s1) and (not s2) and x1;
    y2 <= s1 and (not s2) and x2;
    y3 <= (not s1) and (not s2) and x3;
    y4 <= s1 and s2 and x4;

    X <= x1 or x2 or x3 or x4;
```

Asimismo, se comprueba con algunos ejemplos en el respectivo test para corroborar los resultados

```
report "Inicio para test de una ALU"
     severity note;

     x1_test <= '0';
     x2_test <= '0';
     x3_test <= '0';
     x4_test <= '0';
     s1_test <= '0';
     s2_test <= '0';
     wait for 1 ns;
     assert X_test ='0'
            report"Falla para x1=0, x2=0, x3=0, x4=0, s1=0 y s2=0"
            severity failure;

     x1_test <= '0';
     x2_test <= '0';
     x3_test <= '0';
     x4_test <= '1';
     s1_test <= '0';
     s2_test <= '0';
     wait for 1 ns;
     assert X_test ='0'
            report"Falla para x1=0, x2=0, x3=0, x4=1, s1=0 y s2=0"
            severity failure;
```