PROYECTO VHDL

PRESENTADO POR:

BRANDON DAVID ORTEGA

PRESENTADO A:

CARLOS HERNAN TOBAR ARTEAGA

UNIVERSIDAD DEL CAUCA
FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES
DEPARTAMENTO DE TELECOMUNICACIONES
POPAYÁN-2023

PROYECTO 1:

And

Código VHDL para la compuerta AND:

- La entidad andgate define la interfaz de la compuerta AND, con dos entradas (a y b) y una salida (y).
- La arquitectura Behavioral 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:

- La entidad andgate_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- La arquitectura behavior describe el comportamiento del test bench.
- El componente andgate se declara como un componente que se utilizará en el test bench.
- 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.
- El proceso stim_proc 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.

Código en vhdl:

```
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;
```

TestBench:

```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;

```

And16

Código VHDL para la compuerta AND de 16 bits:

- La entidad and16_gate define la interfaz de la compuerta AND de 16 bits, con dos entradas (a y b) y una salida (y), todas de 16 bits.

- La arquitectura Behavioral 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:

- La entidad and16_gate_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
- La arquitectura behavior describe el comportamiento del test bench.
- El componente and16_gate se declara como un componente que se utilizará en el test bench.
- 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.
- El proceso stim_proc 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.

Código en vhdl:

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

entity 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 and16_gate;

architecture Behavioral of and16_gate is
begin
    y <= a and b;
end Behavioral;
```

TestBench:

```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;
```

Nand

- Código VHDL para la compuerta NAND:

  La entidad nand_gate define la interfaz de la compuerta NAND. Tiene dos entradas (a y b) y una salida (y).

La arquitectura Behavioral 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:

  La entidad nand_gate_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
  La arquitectura behavior describe el comportamiento del test bench.
  El componente nand_gate se declara como un componente que se utilizará en el test bench.
  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.
  El proceso stim_proc 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.

Código en vhdl:

```
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;
```

Testbench:

```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;
```

Or

- Código VHDL para la compuerta OR de 16 bits:

  La entidad or16_gate define la interfaz de la compuerta OR de 16 bits. Tiene dos entradas (a y b) y una salida (y), todas de 16 bits.

La arquitectura Behavioral 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:

  La entidad or16_gate_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
  La arquitectura behavior describe el comportamiento del test bench.
  El componente or16_gate se declara como un componente que se utilizará en el test bench.
  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.
  El proceso stim_proc 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.

Código vhdl:

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

TestBench:

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

entity OrGate_tb is
end entity;

architecture behavior of OrGate_tb is
    -- Component declaration
    component OrGate is
        Port (
            in1 : in STD_LOGIC;
            in2 : in STD_LOGIC;
            salida : out STD_LOGIC
        );
    end component;

    -- Signals declaration
    signal in1_tb, in2_tb, salida_tb : STD_LOGIC;

begin
    uut: OrGate port map (in1_tb, in2_tb, salida_tb);

    stim_proc: process
    begin
        in1_tb <= '0';
        in2_tb <= '0';
        wait for 10 ns;

        in1_tb <= '0';
        in2_tb <= '1';
        wait for 10 ns;

        in1_tb <= '1';
        in2_tb <= '0';
        wait for 10 ns;

        in1_tb <= '1';
        in2_tb <= '1';
        wait for 10 ns;
        wait;
    end process;
end behavior;
```

Not

Código vhdl para la compuerta Not:

- La entidad define variables de entrada y salida. La entrada sería un bit normal y la salida sería la entrada negada.

- La arquitectura describe el comportamiento de la compuerta Nor, En este caso simplemente se niega la variable de entrada y se la asigna a la variable de salida.

Código vhdl:

```
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;
```

TestBench:

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

entity NotGate_test is
end entity;

architecture arch_test of NotGate_test is

        component NotGate
                port(
                        x       :       in      std_logic;
                        f       :       out     std_logic
                );
        end component;

        signal x_test,  f_test  : std_logic := '0';

        begin

        dut1    : NotGate
                port map (
                        x => x_test,
                        f => f_test
                );

        Stimulus        : process
        begin
          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;

                report "Test successful"
                severity note;
                wait;

        end process;
end architecture;

```

Not16

- Codigo en vhdl de una variable de entrada negada de 16 bits.

Este código define una entidad llamada Not16 que tiene dos puertos: x y f. El puerto x es una entrada de 16 bits, y el puerto f es una salida de 16 bits. La arquitectura arch describe la

implementación de la compuerta NOT de 16 bits. En esta implementación, cada bit de la salida f es el resultado de aplicar la operación NOT al bit correspondiente de la entrada x. Por ejemplo, el primer bit de la salida f (f(0)) es igual a la negación del primer bit de la entrada x (x(0)). De manera similar, el segundo bit de la salida (f(1)) es igual a la negación del segundo bit de la entrada (x(1)), y así sucesivamente hasta el último bit.

Código vhdl:

```
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;
```

TestBench:

```
21          component Not16
22                  port(
23                          x       :       in      std_logic_vector(15 downto 0);
24                          f       :       out     std_logic_vector(15 downto 0)
25                  );
26          end component;
27
28          -- Signal declaration
29          signal x_test   : std_logic_vector(15 downto 0) := "0000000000000000";
30          signal f_test   : std_logic_vector(15 downto 0);
31
32          begin
33
34          -- DUT instantiation
35          dut1    : Not16
36                  port map (
37                          x => x_test,
38                          f => f_test
39                  );
40
41          -- Stimulus generation
42          Stimulus        : process
43          begin
44
45            report "Start of the test of NotGate"
46                  severity note;
47
48                  x_test <= "0000000000000000";
49                  wait for 1 ns;
50                  assert f_test = "1111111111111111"
51                    report "Failure para x = [0000000000000000]"
52                    severity failure;
53
54                  x_test <= "1111111111111111";
55                  wait for 1 ns;
56                  assert f_test = "0000000000000000"
57                    report "Failure para x = [1111111111111111]"
58                    severity failure;
59
60                  x_test <= "0100110000000011";
61                  wait for 1 ns;
62                  assert f_test = "1011001111111100"
63                    report "Failure para x = [0100110000000011]"
64                    severity failure;
65
66                  x_test <= "1111100001011110";
67                  wait for 1 ns;
68                  assert f_test = "0000011110100001"
69                    report "Failure para x = [1111100001011110]"
70                    severity failure;
71
72                  report "Test successful"
73                  severity note;
74                  wait;
75
```

Mux16

- Código del Mux16: Este código define un multiplexor de 16 bits que selecciona una de dos señales de entrada (in0 e in1) para enviarla a la salida (o), dependiendo de una señal de control (sel). Si sel es '0', la salida 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 cuatro 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.

Código vhdl:

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Mux16 is
5
6       Port ( sel : in STD_LOGIC;
7              in0 : in STD_LOGIC_VECTOR (15 downto 0);
8              in1 : in STD_LOGIC_VECTOR (15 downto 0);
9              o : out STD_LOGIC_VECTOR (15 downto 0));
10  end Mux16;
11
12  architecture Behavioral of Mux16 is
13  begin
14
15      o <= in0 when sel = '0' else in1;
16
17  end Behavioral;
```

TestBench:

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

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;
```

Dmux

- Código VHDL para el DMUX:

  La entidad dmux define la interfaz del DMUX. Tiene dos entradas (d y s) y dos salidas (y0 y y1).

  La arquitectura Behavioral 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:

  La entidad dmux_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.

  La arquitectura behavior describe el comportamiento del test bench.

  El componente dmux se declara como un componente que se utilizará en el test bench.

  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.

  El proceso stim_proc genera las señales de prueba. En este caso, genera todas las combinaciones posibles de las entradas d y s y las aplica al DMUX.

Código vhdl:

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

entity dmux is
    Port ( d : in STD_LOGIC;
           s : in STD_LOGIC;
           y0 : out STD_LOGIC;
           y1 : out STD_LOGIC);
end dmux;

architecture Behavioral of dmux is
begin
    y0 <= d and not s;
    y1 <= d and s;
end Behavioral;
```

TestBench:

```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;
```

Xor

- Código VHDL para la compuerta XOR:
  La entidad xor_gate define la interfaz de la compuerta XOR. Tiene dos entradas (a y b) y una salida (y).
  La arquitectura Behavioral 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:
  La entidad xor_gate_tb define la interfaz del test bench. No tiene entradas ni salidas porque es un entorno de prueba.
  La arquitectura behavior describe el comportamiento del test bench.
  El componente xor_gate se declara como un componente que se utilizará en el test bench.
  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.
  El proceso stim_proc 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.

Código vhdl:

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

entity xor_gate is

    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           y : out STD_LOGIC);

end xor_gate;

architecture Behavioral of xor_gate is

begin

    y <= a xor b;

end Behavioral;
```

TestBench:

```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;
```

Or8

- Código del Or8Way:
  El código define un componente llamado Or8Way que tiene ocho entradas de 1 bit (in0 a in7) y una salida de 1 bit (out). La operación que realiza este componente es una operación OR bit a bit en las ocho entradas. 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 out <= in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7;.

- Código del Test Bench:
  El código se utiliza para probar el funcionamiento del Or8Way. Primero, se definen 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.

Código Vhdl:

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

entity Or8Way is
    Port ( in0, in1, in2, in3, in4, in5, in6, in7 : in STD_LOGIC;
           outt : out STD_LOGIC);
end Or8Way;

architecture Behavioral of Or8Way is
begin
    outt <= in0 or in1 or in2 or in3 or in4 or in5 or in6 or in7;
end Behavioral;
```

TestBench:

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

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;
```

Mux8_16bits

- Código del Mux8Way16:
  El 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):
  El código se utiliza para probar el funcionamiento del Mux8Way16. Primero, se definen 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.

Código vhdl:

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Mux8Way16 is
5
6       Port ( sel : in STD_LOGIC_VECTOR (2 downto 0);
7              in0, in1, in2, in3, in4, in5, in6, in7 : in STD_LOGIC_VECTOR (15 downto 0);
8              outt : out STD_LOGIC_VECTOR (15 downto 0));
9
10  end Mux8Way16;
11
12  architecture Behavioral of Mux8Way16 is
13  begin
14
15      with sel select
16          outt <= in0 when "000",
17                  in1 when "001",
18                  in2 when "010",
19                  in3 when "011",
20                  in4 when "100",
21                  in5 when "101",
22                  in6 when "110",
23                  in7 when others;
24
25  end Behavioral;
```

TestBench:

```vhdl
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;

end Behavioral;
```

Mux4_16

- un multiplexor de 16 bits con cuatro entradas. El multiplexor Mux4_16 tiene cuatro entradas: a0, a1, a2 y a3, y dos señales de selección sel0 y sel1. La señal de selección determina qué entrada se dirige a la salida. Si sel0 y sel1 son ambos '0', la entrada a0 se dirige a la salida. Si sel0 es '0' y sel1 es '1', la entrada a1 se dirige a la salida. Si sel0 es '1' y sel1 es '0', la entrada a2 se dirige a la salida. Finalmente, si ambos son '1', la entrada a3 se dirige a la salida 1.
- testbench para el multiplexor de 16 bits con cuatro entradas Mux4_16. Un testbench es un módulo en VHDL que se utiliza para probar y verificar el comportamiento de un

diseño 1. En este caso, el testbench Mux4_16_tb se encarga de proporcionar los estímulos necesarios para probar el multiplexor Mux4_16

Código Vhdl:

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Mux4_16 is
5           port(
6                   a0, a1, a2, a3 : in std_logic_vector(15 downto 0);
7                   sel0, sel1 : in std_logic;
8                   y : out std_logic_vector(15 downto 0)
9           );
10  end entity;
11
12  architecture arch of Mux4_16 is
13          signal x1 : std_logic_vector(15 downto 0);
14
15  begin
16
17     x1 <= a0 when (sel0='0' and sel1='0') else
18           a1 when (sel0='0' and sel1='1') else
19                          a2 when (sel0='1' and sel1='0') else
20                          a3 when (sel0='1' and sel1='1');
21          y<=x1;
22  end architecture;
```

TestBench:

```vhdl
entity Mux4_16_tb is
end entity;

architecture Behavioral of Mux4_16_tb is
    -- Component declaration for the DUT (Design Under Test)
    component Mux4_16 is
        port (
            a0, a1, a2, a3 : in std_logic_vector(15 downto 0);
            sel0, sel1 : in std_logic;
            y : out std_logic_vector(15 downto 0)
        );
    end component;

    -- Signal declarations for the testbench
    signal a0, a1, a2, a3 : std_logic_vector(15 downto 0);
    signal sel0, sel1 : std_logic;
    signal y : std_logic_vector(15 downto 0);

begin
    -- Instantiate the DUT
    dut: Mux4_16 port map (
        a0 => a0,
        a1 => a1,
        a2 => a2,
        a3 => a3,
        sel0 => sel0,
        sel1 => sel1,
        y => y
    );

    stimulus: process
    begin
    a0 <= "0000000000000000";
    a1 <= "1111111111111111";
    a2 <= "0101010101010101";
    a3 <= "1010101010101010";
    sel0 <= '0';
    sel1 <= '0';
    wait for 10 ns;

    a0 <= "1111000011110000";
    a1 <= "0000111100001111";
    a2 <= "0101010101010101";
    a3 <= "1010101010101010";
    sel0 <= '0';
    sel1 <= '1';
    wait for 10 ns;

        wait;
    end process;

```

Dmux4

- Código del DMux4:
  Este código define un componente llamado DMux4 que tiene una entrada de 1 bit (in0) y cuatro salidas de 1 bit (out0 a out3). El propósito de este componente es dirigir la entrada a una de las cuatro salidas según una señal de selección de 2 bits (sel). Esto

se logra mediante las líneas out0 <= in0 when sel = “00” else ‘0’; y similares. Dependiendo del valor de sel, la entrada se dirige a una de las salidas correspondientes. 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 DMux4. En primer lugar, se definen unas señales que se conectan al DMux4 en la línea uut: entity work.DMux4 port map (sel => sel, in0 => in0, out0 => out0, out1 => out1, out2 => out2, out3 => out3);. A continuación, en el proceso stimulus, se cambian los valores 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.

Código vhdl:

```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;
```

TestBench:

```vhdl
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;
```

Code view is read-only.

Dmux8

- 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.

Código vhdl:

```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;
```

Testbench:

```vhdl
library IEEE;
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;
```

PROYECTO 2:

Half Adder

- En este caso, el medio sumador tiene dos entradas (a y b) y dos salidas (sum y carry_out). La suma se calcula utilizando la operación XOR (xor) entre a y b, mientras que el acarreo se calcula utilizando la operación AND (and) entre a y b1.

- Se comienza definiendo una entidad llamada halfadder_tb que no tiene puertos porque es un banco de pruebas.Luego, se define una arquitectura para halfadder_tb. Dentro de

esta arquitectura, se declara un componente llamado halfadder que tiene dos entradas (a y b) y dos salidas (sum y carry_out). Se declaran cuatro señales (a_tb, b_tb, sum_tb, carry_out_tb) que se usarán para probar el half-adder. Se instancia el componente halfadder (denotado como uut para “unidad bajo prueba”) y se mapean las señales a los puertos correspondientes del half-adder.Finalmente, se define un proceso llamado stimulus que genera diferentes combinaciones de entradas para a y b y espera 10 ns después de cada cambio. Esto simula el comportamiento del half-adder para todas las posibles combinaciones de entradas.

Código vhdl:

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

entity halfadder is
  port (
    a, b : in std_logic;
    sum, carry_out : out std_logic
  );
end halfadder;

architecture dataflow of halfadder is
begin
  sum <= a xor b;
  carry_out <= a and b;
end dataflow;
```

Testbench:

```vhdl
end halfadder_tb;

architecture testbench of halfadder_tb is
  component halfadder is
    port (
      a, b : in std_logic;
      sum, carry_out : out std_logic
    );
  end component;

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

  signal sum : std_logic;
  signal carry_out : std_logic;

begin

  uut: halfadder port map (
    a => a,
    b => b,
    sum => sum,
    carry_out => carry_out
  );
  stim_proc: process
  begin

    a <= '0';
    b <= '0';

    wait for 10 ns;
    assert sum = '0' and carry_out = '0'
      report "Error: Unexpected output values"
      severity error;
    a <= '1';
    b <= '0';
    wait for 10 ns;

    assert sum = '1' and carry_out = '0'
      report "Error: Unexpected output values"
      severity error;

    a <= '0';
    b <= '1';
    wait for 10 ns;

    assert sum = '1' and carry_out = '0'
      report "Error: Unexpected output values"
      severity error;
    a <= '0';
    b <= '0';
    wait for 10 ns;

    assert sum = '0' and carry_out = '0'
      report "Error: Unexpected output values"
      severity error;
    wait;
  end process;
end testbench;
```

FullAdder

- Primero, se define una entidad llamada fulladder con tres entradas (a, b y cin) y dos salidas (sum y carry_out).

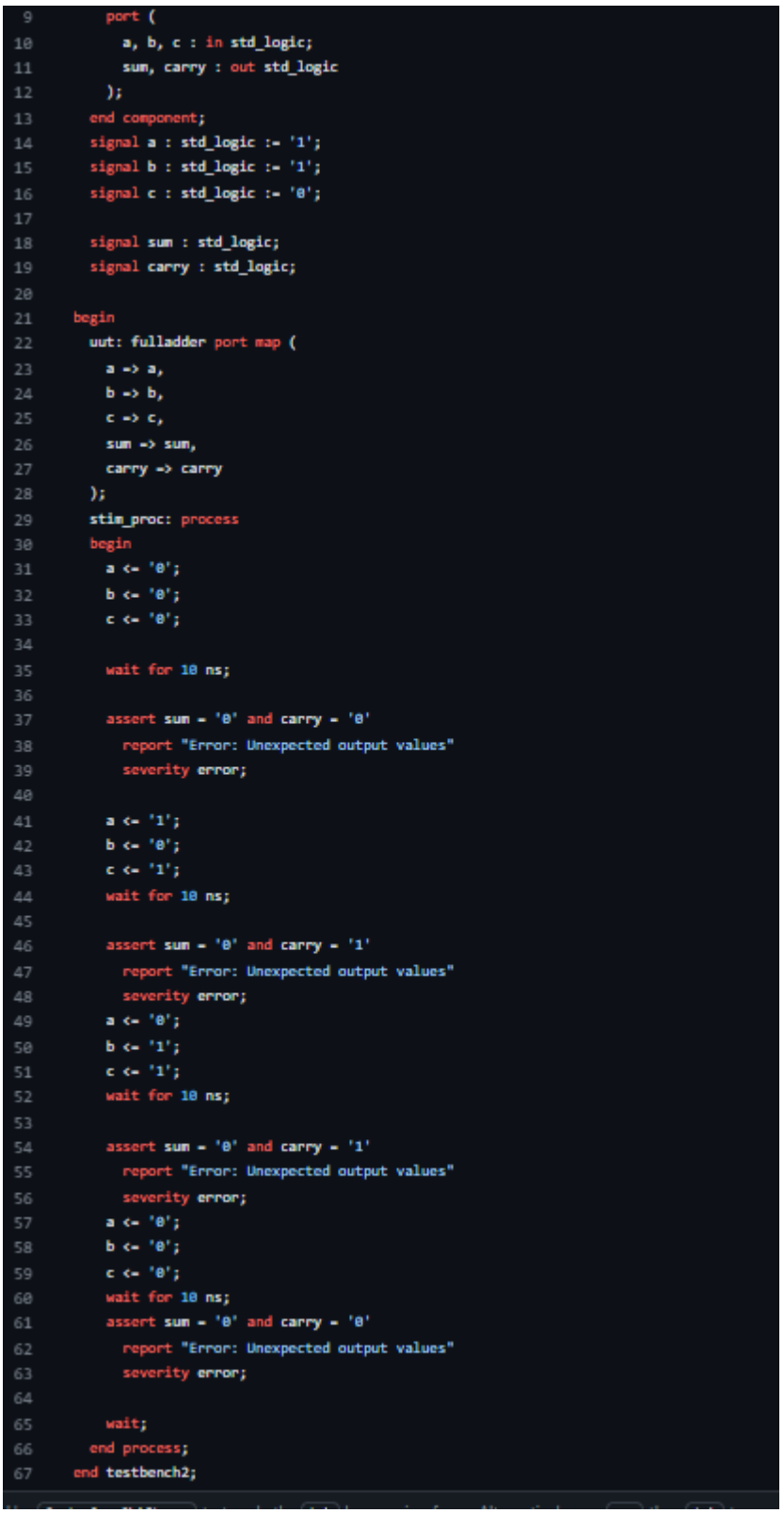
Luego, se define una arquitectura para fulladder. Dentro de esta arquitectura, se declara un componente llamado halfadder que tiene dos entradas (a y b) y dos salidas (sum y carry_out).
Se declaran tres señales (s1, c1, c2) que se usarán para conectar los half-adders.
Se instancian dos half-adders. El primer half-adder suma las entradas a y b. El segundo half-adder suma la salida s1 del primer half-adder y la entrada cin.
La salida sum del full-adder es la salida sum del segundo half-adder.La salida carry_out del full-adder es la salida OR de las salidas carry_out de ambos half-adders.

- El banco de pruebas (test bench) para el full-adder funciona de la siguiente manera:
  Se define una entidad llamada fulladder_tb que no tiene puertos porque es un banco de pruebas.
  Luego, se define una arquitectura para fulladder_tb. Dentro de esta arquitectura, se declara un componente llamado fulladder que tiene tres entradas (a, b, cin) y dos salidas (sum, carry_out).
  Se declaran cinco señales (a_tb, b_tb, cin_tb, sum_tb, carry_out_tb) que se usarán para probar el full-adder.
  Se instancia el componente fulladder (denotado como uut para "unidad bajo prueba") y se mapean las señales a los puertos correspondientes del full-adder.
  Finalmente, se define un proceso llamado stimulus que genera diferentes combinaciones de entradas para a, b y cin y espera 10 ns después de cada cambio. Esto simula el comportamiento del full-adder para todas las posibles combinaciones de entradas.

Código vhdl:

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

entity fulladder is
  port (
    a, b, c : in std_logic;
    sum, carry : out std_logic
  );
end fulladder;

architecture arch of fulladder is

  component halfadder
    port (
      a, b : in std_logic;
      sum, carry_out : out std_logic
    );
  end component;

  signal s1, c1, c2 : std_logic;

begin
  ha1: halfadder port map (a => a, b => b, sum => s1, carry_out=> c1);
  ha2: halfadder port map (a => s1, b => c, sum => sum, carry_out => c2);
  carry <= c1 or c2;

end arch;
```

TestBench:



```
    port (
      a, b, c : in std_logic;
      sum, carry : out std_logic
    );
  end component;
  signal a : std_logic := '1';
  signal b : std_logic := '1';
  signal c : std_logic := '0';

  signal sum : std_logic;
  signal carry : std_logic;

begin
  uut: fulladder port map (
    a => a,
    b => b,
    c => c,
    sum => sum,
    carry => carry
  );
  stim_proc: process
  begin
    a <= '0';
    b <= '0';
    c <= '0';

    wait for 10 ns;

    assert sum = '0' and carry = '0'
      report "Error: Unexpected output values"
      severity error;

    a <= '1';
    b <= '0';
    c <= '1';
    wait for 10 ns;

    assert sum = '0' and carry = '1'
      report "Error: Unexpected output values"
      severity error;
    a <= '0';
    b <= '1';
    c <= '1';
    wait for 10 ns;

    assert sum = '0' and carry = '1'
      report "Error: Unexpected output values"
      severity error;
    a <= '0';
    b <= '0';
    c <= '0';
    wait for 10 ns;
    assert sum = '0' and carry = '0'
      report "Error: Unexpected output values"
      severity error;

    wait;
  end process;
end testbench2;
```

Add16

- En primer lugar, se establece una entidad llamada add16 con dos entradas de 16 bits (a y b) y una salida de 16 bits (out).

- A continuación, se define una arquitectura para add16. Dentro de esta arquitectura, se declara un componente denominado fulladder que cuenta con tres entradas (a, b y cin) y dos salidas (sum y carry_out). Se declara un vector de señales carry de 17 bits para almacenar los acarreos de cada FullAdder. Se inicializa el primer bit de carry a ‘0’ debido a que no hay acarreo de entrada para el primer bit. Luego, se genera una serie de FullAdders utilizando una estructura de bucle for generate. Cada FullAdder suma un bit de a y b junto con el acarreo del FullAdder anterior. El resultado se almacena en el bit correspondiente de out y el acarreo se almacena en el siguiente bit de carry.

- El banco de pruebas (test bench) para el Add16 funciona de la siguiente manera:
  En primer lugar, se define una entidad llamada add16_tb que no tiene puertos debido a que es un banco de pruebas.A continuación, se define una arquitectura para add16_tb. Dentro de esta arquitectura, se declara un componente denominado add16 que cuenta con dos entradas de 16 bits (a y b) y una salida de 16 bits (out). Se declaran tres vectores de señales (a_tb, b_tb, out_tb) de 16 bits que serán utilizados para probar el Add16. Se instancia el componente add16 (denominado como uut para “unidad bajo prueba”) y se mapean las señales a los puertos correspondientes del Add16. Por último, se define un proceso denominado stimulus que genera diferentes combinaciones de entradas para a y b y espera 10 ns después de cada cambio. Esto simula el comportamiento del Add16 para todas las posibles combinaciones de entradas .

Còdigo vhdl:

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

entity adder_16bit is
    port ( a, b : in std_logic_vector(15 downto 0);
           sum1 : out std_logic_vector(15 downto 0));
end adder_16bit;

architecture arch of adder_16bit is
    component fullAdder is
        port ( a, b, c: in std_logic;
               sum, carry : out std_logic);
    end component;

    signal  carry1, carry2, carry3, carry4, carry5, carry6, carry7,carry8, carry9, carry10, carry11, carry12, carry13, carry14, carry15,carry0: std_logic;

begin

    Full0: fullAdder port map(a=>a(0), b=>b(0), c=>'0', sum=>sum1(0), carry=>carry1);
         Full1: fullAdder port map(a=>a(1), b=>b(1), c=>carry1, sum=>sum1(1), carry=>carry2);
         Full2: fullAdder port map(a=>a(2), b=>b(2), c=>carry2, sum=>sum1(2), carry=>carry3);
         Full3: fullAdder port map(a=>a(3), b=>b(3), c=>carry3, sum=>sum1(3), carry=>carry4);
         Full4: fullAdder port map(a=>a(4), b=>b(4), c=>carry4, sum=>sum1(4), carry=>carry5);
         Full5: fullAdder port map(a=>a(5), b=>b(5), c=>carry5, sum=>sum1(5), carry=>carry6);
         Full6: fullAdder port map(a=>a(6), b=>b(6), c=>carry6, sum=>sum1(6), carry=>carry7);
         Full7: fullAdder port map(a=>a(7), b=>b(7), c=>carry7, sum=>sum1(7), carry=>carry8);
         Full8: fullAdder port map(a=>a(8), b=>b(8), c=>carry8, sum=>sum1(8), carry=>carry9);
         Full9: fullAdder port map(a=>a(9), b=>b(9), c=>carry9, sum=>sum1(9), carry=>carry10);
         Full10: fullAdder port map(a=>a(10), b=>b(10), c=>carry10, sum=>sum1(10), carry=>carry11);
         Full11: fullAdder port map(a=>a(11), b=>b(11), c=>carry11, sum=>sum1(11), carry=>carry12);
         Full12: fullAdder port map(a=>a(12), b=>b(12), c=>carry12, sum=>sum1(12), carry=>carry13);
    Full13: fullAdder port map(a=>a(13), b=>b(13), c=>carry13, sum=>sum1(13), carry=>carry14);
         Full14: fullAdder port map(a=>a(14), b=>b(14), c=>carry14, sum=>sum1(14), carry=>carry15);
         Full15:        fullAdder port map(a=>a(15), b=>b(15), c=>carry15, sum=>sum1(15), carry=>carry0);


end arch;
```

TestBench:

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

entity adder_16bit_tb is
end adder_16bit_tb;

architecture testbench of adder_16bit_tb is

    component adder_16bit is
        port (
            a, b : in std_logic_vector(15 downto 0);
            sum1 : out std_logic_vector(15 downto 0)
        );
    end component;

    signal a_t : std_logic_vector(15 downto 0) := (others => '0');
    signal b_t : std_logic_vector(15 downto 0) := (others => '0');
    signal sum1_t : std_logic_vector(15 downto 0);

begin

    dut: adder_16bit port map (
        a => a_t,
        b => b_t,
        sum1 => sum1_t
    );

    stim_proc: process
    begin

        a_t <= "0000000000000000";
        b_t <= "0000000000000000";
        wait for 10 ns;

        assert sum1_t = "0000000000000000"
            report "Error: Unexpected sum value"
            severity error;

                a_t <= "0000000000000001";
        b_t <= "0000000000000000";
        wait for 10 ns;

        assert sum1_t = "0000000000000001"
            report "Error: Unexpected sum value"
            severity error;

                a_t <= "0000000000000000";
        b_t <= "0000000000000001";
```

```
            wait for 10 ns;

            assert sum1_t = "1111111111111111"
                report "Error: Unexpected sum value"
                severity error;

            a_t <= "0000000000000000";
            b_t <= "1111111111111111";
            wait for 10 ns;

            assert sum1_t = "1111111111111111"
                report "Error: Unexpected sum value"
                severity error;

                    a_t <= "0000000000000001";
            b_t <= "0000000000000010";
            wait for 10 ns;

            assert sum1_t = "0000000000000011"
                report "Error: Unexpected sum value"
                severity error;

            a_t <= "0000000000000000";
            b_t <= "0000000000000000";
            wait for 10 ns;

            assert sum1_t = "0000000000000000"
                report "Error: Unexpected sum value"
                severity error;

            wait;
        end process;

    end testbench;
```