DISEÑOS VHDL - PROYECTO 2

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

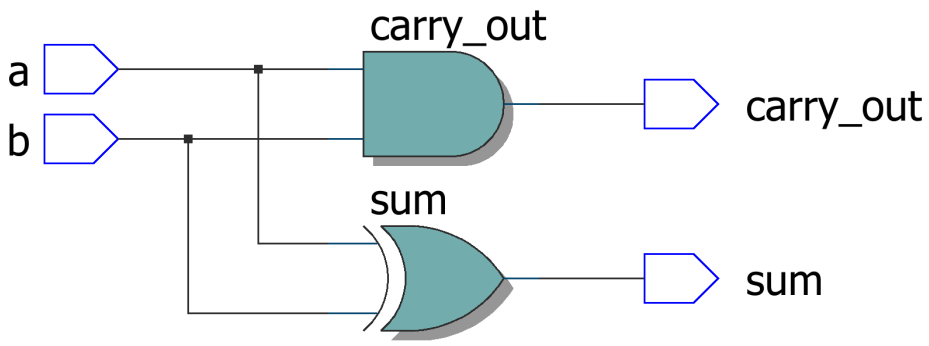
❖ **HalfAdder**

- Specifications:

```
/**
* HalfAdder
* Computes the sum of two bits.
*/

CHIP HalfAdder {
   IN a, b;    // 1-bit inputs
   OUT sum,    // Right bit of a + b
       carry;  // Left bit of a + b

   PARTS:
   // Put you code here:
          Xor(a=a, b=b, out=sum);
          And(a=a, b=b, out=carry);}
/**
```




```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity halfadder is
5     port (
6       a, b : in std_logic;
7       sum, carry_out : out std_logic
8     );
9   end halfadder;
10
11  architecture dataflow of halfadder is
12  begin
13    sum <= a xor b;
14    carry_out <= a and b;
15  end dataflow;
```

TEST BENCH:

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity halfadder_tb is
5   end halfadder_tb;
6
7   architecture tb_arch of halfadder_tb is
8     component halfadder
9       port (
10        a, b : in std_logic;
11        sum, carry_out : out std_logic
12      );
13    end component;
14
15    signal a_tb, b_tb, sum_tb, carry_out_tb : std_logic;
16  begin
17    uut: halfadder port map (a => a_tb, b => b_tb, sum => sum_tb, carry_out => carry_out_tb);
18
19    stimulus: process
20    begin
21      a_tb <= '0';
22      b_tb <= '0';
23      wait for 10 ns;
24
25      a_tb <= '0';
26      b_tb <= '1';
27      wait for 10 ns;
28
29      a_tb <= '1';
30      b_tb <= '0';
31      wait for 10 ns;
32
33      a_tb <= '1';
34      b_tb <= '1';
35      wait for 10 ns;
36
37      wait;
38    end process stimulus;
39  end tb_arch;
```

EXPLICACIÓN HALF ADDER:

- Primero, se define 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.

❖ **FullAdder**

- Specifications:

```
/**
 * FullAdder
 * Computes the sum of three bits.
```

```
*/

CHIP FullAdder {
   IN a, b, c;  // 1-bit inputs
   OUT sum,     // Right bit of a + b + c
       carry;   // Left bit of a + b + c

   PARTS:
   // Put you code here:
         HalfAdder(a=a, b=b, sum=sum1, carry=carry1);
         HalfAdder(a=sum1, b=c, sum=sum, carry=carry2);
         Or(a=carry1, b=carry2, out=carry);
}
/**
```

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity fulladder is
5     port (
6       a, b, cin : in std_logic;
7       sum, carry_out : out std_logic
8     );
9   end fulladder;
10
11  architecture dataflow of fulladder is
12    component halfadder
13      port (
14        a, b : in std_logic;
15        sum, carry_out : out std_logic
16      );
17    end component;
18
19    signal s1, c1, c2 : std_logic;
20
21  begin
22    ha1: halfadder port map (a => a, b => b, sum => s1, carry_out => c1);
23    ha2: halfadder port map (a => s1, b => cin, sum => sum, carry_out => c2);
24    carry_out <= c1 or c2;
25
26  end dataflow;
27
```

EXPLICACIÓN FULL ADDER:

- 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:

- Primero, 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.

❖ **Add16 - 16-bit Adder**

- Specifications:

```
/**
 * Add16
 * Adds two 16-bit values.
 * The most significant carry bit is ignored.
 */

CHIP Add16 {
   IN a[16], b[16];
   OUT out[16];

   PARTS:
  // Put you code here:
        FullAdder(a=a[0], b=b[0], c=false, sum=out[0], carry=carry1);
        FullAdder(a=a[1], b=b[1], c=carry1, sum=out[1], carry=carry2);
        FullAdder(a=a[2], b=b[2], c=carry2, sum=out[2], carry=carry3);
        FullAdder(a=a[3], b=b[3], c=carry3, sum=out[3], carry=carry4);
        FullAdder(a=a[4], b=b[4], c=carry4, sum=out[4], carry=carry5);
        FullAdder(a=a[5], b=b[5], c=carry5, sum=out[5], carry=carry6);
        FullAdder(a=a[6], b=b[6], c=carry6, sum=out[6], carry=carry7);
        FullAdder(a=a[7], b=b[7], c=carry7, sum=out[7], carry=carry8);
        FullAdder(a=a[8], b=b[8], c=carry8, sum=out[8], carry=carry9);
        FullAdder(a=a[9], b=b[9], c=carry9, sum=out[9], carry=carry10);
        FullAdder(a=a[10], b=b[10], c=carry10, sum=out[10], carry=carry11);
        FullAdder(a=a[11], b=b[11], c=carry11, sum=out[11], carry=carry12);
        FullAdder(a=a[12], b=b[12], c=carry12, sum=out[12], carry=carry13);
        FullAdder(a=a[13], b=b[13], c=carry13, sum=out[13], carry=carry14);
        FullAdder(a=a[14], b=b[14], c=carry14, sum=out[14], carry=carry15);
        FullAdder(a=a[15], b=b[15], c=carry15, sum=out[15], carry=carry);
}
```

/**

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity add16 is
5    port (
6      a, b : in std_logic_vector(15 downto 0);
7      out : out std_logic_vector(15 downto 0)
8    );
9  end add16;
10
11 architecture dataflow of add16 is
12   component fulladder
13     port (
14       a, b, cin : in std_logic;
15       sum, carry_out : out std_logic
16     );
17   end component;
18
19   signal carry : std_logic_vector(16 downto 0);
20
21 begin
22   carry(0) <= '0';
23
24   gen : for i in 0 to 15 generate
25     fa : fulladder port map (a => a(i), b => b(i), cin => carry(i), sum => out(i), carry_out => carry(i+1));
26   end generate gen;
27
28 end dataflow;
29
```

TEST BENCH:

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

entity add16_tb is
end add16_tb;

architecture tb_arch of add16_tb is
  component add16
    port (
      a, b : in std_logic_vector(15 downto 0);
      out : out std_logic_vector(15 downto 0)
    );
  end component;

  signal a_tb, b_tb, out_tb : std_logic_vector(15 downto 0);
begin
  uut: add16 port map (a => a_tb, b => b_tb, out => out_tb);

  stimulus: process
  begin
    a_tb <= "0000000000000000";
    b_tb <= "0000000000000000";
    wait for 10 ns;

    a_tb <= "0000000000000001";
    b_tb <= "0000000000000001";
    wait for 10 ns;

    a_tb <= "1111111111111111";
    b_tb <= "0000000000000000";
    wait for 10 ns;

    a_tb <= "0000000000000000";
    b_tb <= "1111111111111111";
    wait for 10 ns;

    a_tb <= "1111111111111111";
    b_tb <= "1111111111111111";
    wait for 10 ns;

    wait;
  end process stimulus;
end tb_arch;
```

EXPLICACIÓN ADD 16:

- Primero, se define una entidad llamada add16 con dos entradas de 16 bits (a y b) y una salida de 16 bits (out).
- Luego, se define una arquitectura para add16. Dentro de esta arquitectura, se declara un componente llamado fulladder que tiene 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’ porque 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:

- Primero, se define una entidad llamada add16_tb que no tiene puertos porque es un banco de pruebas.
- Luego, se define una arquitectura para add16_tb. Dentro de esta arquitectura, se declara un componente llamado add16 que tiene 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 se usarán para probar el Add16.
- Se instancia el componente add16 (denotado como uut para "unidad bajo prueba") y se mapean las señales a los puertos correspondientes del Add16.
- 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 Add16 para todas las posibles combinaciones de entradas.

❖ **Inc16 - 16-bit incrementer**

- Specifications:

```
/**
 * Inc16
 * 16-bit incrementer:
 * out = in + 1 (arithmetic addition)
 */

CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
   // Put you code here:
            HalfAdder(a=in[0], b=true, sum=out[0], carry=carry1);
            HalfAdder(a=in[1], b=carry1, sum=out[1], carry=carry2);
            HalfAdder(a=in[2], b=carry2, sum=out[2], carry=carry3);
            HalfAdder(a=in[3], b=carry3, sum=out[3], carry=carry4);
            HalfAdder(a=in[4], b=carry4, sum=out[4], carry=carry5);
            HalfAdder(a=in[5], b=carry5, sum=out[5], carry=carry6);
            HalfAdder(a=in[6], b=carry6, sum=out[6], carry=carry7);
            HalfAdder(a=in[7], b=carry7, sum=out[7], carry=carry8);
            HalfAdder(a=in[8], b=carry8, sum=out[8], carry=carry9);
            HalfAdder(a=in[9], b=carry9, sum=out[9], carry=carry10);
            HalfAdder(a=in[10], b=carry10, sum=out[10], carry=carry11);
            HalfAdder(a=in[11], b=carry11, sum=out[11], carry=carry12);
            HalfAdder(a=in[12], b=carry12, sum=out[12], carry=carry13);
            HalfAdder(a=in[13], b=carry13, sum=out[13], carry=carry14);
            HalfAdder(a=in[14], b=carry14, sum=out[14], carry=carry15);
            HalfAdder(a=in[15], b=carry15, sum=out[15], carry=false);
```

}

/**

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

entity inc16 is
  port (
    in : in std_logic_vector(15 downto 0);
    out : out std_logic_vector(15 downto 0)
  );
end inc16;

architecture dataflow of inc16 is
  component halfadder
    port (
      a, b : in std_logic;
      sum, carry_out : out std_logic
    );
  end component;

  signal carry : std_logic_vector(16 downto 0);

begin
  carry(0) <= '1';

  gen : for i in 0 to 15 generate
    ha : halfadder port map (a => in(i), b => carry(i), sum => out(i), carry_out => carry(i+1));
  end generate gen;

end dataflow;

```

TEST BENCH:

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity inc16_tb is
5   end inc16_tb;
6
7   architecture tb_arch of inc16_tb is
8     component inc16
9       port (
10        in : in std_logic_vector(15 downto 0);
11        out : out std_logic_vector(15 downto 0)
12      );
13    end component;
14
15    signal in_tb, out_tb : std_logic_vector(15 downto 0);
16  begin
17    uut: inc16 port map (in => in_tb, out => out_tb);
18
19    stimulus: process
20    begin
21      in_tb <= "0000000000000000";
22      wait for 10 ns;
23
24      in_tb <= "0000000000000001";
25      wait for 10 ns;
26
27      in_tb <= "1111111111111110";
28      wait for 10 ns;
29
30      in_tb <= "1111111111111111";
31      wait for 10 ns;
32
33      wait;
34    end process stimulus;
35  end tb_arch;
36  s
```

EXPLICACIÓN INCREMENTER 16:

- Primero, se define una entidad llamada inc16 con una entrada de 16 bits (in) y una salida de 16 bits (out).
- Luego, se define una arquitectura para inc16. 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 declara un vector de señales carry de 17 bits para almacenar los acarreos de cada HalfAdder.
- Se inicializa el primer bit de carry a ‘1’ porque se está incrementando el número de entrada en 1.
- Luego, se genera una serie de HalfAdders utilizando una estructura de bucle for generate. Cada HalfAdder suma un bit de in y el acarreo del HalfAdder 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 Inc16 funciona de la siguiente manera:

- Primero, se define una entidad llamada inc16_tb que no tiene puertos porque es un banco de pruebas.

- Luego, se define una arquitectura para inc16_tb. Dentro de esta arquitectura, se declara un componente llamado inc16 que tiene una entrada de 16 bits (in) y una salida de 16 bits (out).
- Se declaran dos vectores de señales (in_tb, out_tb) de 16 bits que se usarán para probar el Inc16.
- Se instancia el componente inc16 (denotado como uut para "unidad bajo prueba") y se mapean las señales a los puertos correspondientes del Inc16.
- Finalmente, se define un proceso llamado stimulus que genera diferentes combinaciones de entradas para in y espera 10 ns después de cada cambio. Esto simula el comportamiento del Inc16 para todas las posibles combinaciones de entradas.

❖ **ALU- Arithmetic Logic Unit**

- Specifications:

```
/**
 * The ALU (Arithmetic Logic Unit).
 * Computes one of the following functions:
 * x+y, x-y, y-x, 0, 1, -1, x, y, -x, -y, !x, !y,
 * x+1, y+1, x-1, y-1, x&y, x|y on two 16-bit inputs,
 * according to 6 input bits denoted zx,nx,zy,ny,f,no.
 * In addition, the ALU computes two 1-bit outputs:
 * if the ALU output == 0, zr is set to 1; otherwise zr is set to 0;
 * if the ALU output < 0, ng is set to 1; otherwise ng is set to 0.
 */

// Implementation: the ALU logic manipulates the x and y inputs
// and operates on the resulting values, as follows:
// if (zx == 1) set x = 0        // 16-bit constant
// if (nx == 1) set x = !x       // bitwise not
// if (zy == 1) set y = 0        // 16-bit constant
// if (ny == 1) set y = !y       // bitwise not
// if (f == 1)  set out = x + y  // integer 2's complement addition
// if (f == 0)  set out = x & y  // bitwise and
// if (no == 1) set out = !out   // bitwise not
// if (out == 0) set zr = 1
// if (out < 0) set ng = 1

CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute out = x + y (if 1) or x & y (if 0)
        no; // negate the out output?

    OUT
```

```
    out[16], // 16-bit output
    zr, // 1 if (out == 0), 0 otherwise
    ng; // 1 if (out < 0),  0 otherwise

 PARTS:
// Put you code here:
        Not16(in=x, out=notX);
        Inc16(in=notX, out=comp2X);
        Add16(a=x, b=comp2X, out=zeroX);
        Mux16(a=x, b=zeroX, sel=zx, out=x1);
        Not16(in=x1, out=notX1);
        Mux16(a=x1, b=notX1, sel=nx, out=x2);
        Not16(in=y, out=notY);
        Inc16(in=notY, out=comp2Y);
        Add16(a=y, b=comp2Y, out=zeroY);
        Mux16(a=y, b=zeroY, sel=zy, out=y1);
        Not16(in=y1, out=notY1);
        Mux16(a=y1, b=notY1, sel=ny, out=y2);
        Add16(a=x2, b=y2, out=sumaX2Y2);
        And16(a=x2, b=y2, out=andX2Y2);
        Mux16(a=andX2Y2, b=sumaX2Y2, sel=f, out=out1);
        Not16(in=out1, out=notOut1);
        Mux16(a=out1, b=notOut1, sel=no, out=out);

}
```

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

entity ALU is
  port (
    x, y : in std_logic_vector(15 downto 0);
    zx, nx, zy, ny, f, no : in std_logic;
    out : out std_logic_vector(15 downto 0);
    zr, ng : out std_logic
  );
end ALU;

architecture dataflow of ALU is
  signal x1, x2, y1, y2, out1, out2 : std_logic_vector(15 downto 0);
  signal zero, neg : std_logic_vector(15 downto 0);
begin
  -- if (zx == 1) set x = 0
  x1 <= (others => '0') when zx = '1' else x;

  -- if (nx == 1) set x = !x
  x2 <= not x1 when nx = '1' else x1;

  -- if (zy == 1) set y = 0
  y1 <= (others => '0') when zy = '1' else y;

  -- if (ny == 1) set y = !y
  y2 <= not y1 when ny = '1' else y1;

  -- if (f == 1)  set out = x + y
  -- if (f == 0)  set out = x & y
  out1 <= x2 + y2 when f = '1' else x2 and y2;

  -- if (no == 1) set out = !out
  out2 <= not out1 when no = '1' else out1;

  -- output
  out <= out2;

  -- if (out == 0) set zr = 1
  zr <= '1' when out2 = zero else '0';

  -- if (out < 0) set ng = 1
  ng <= out2(15);
end dataflow;

```

TEST BENCH:

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

entity ALU_tb is
end ALU_tb;

architecture tb_arch of ALU_tb is
  component ALU
    port (
      x, y : in std_logic_vector(15 downto 0);
      zx, nx, zy, ny, f, no : in std_logic;
      out : out std_logic_vector(15 downto 0);
      zr, ng : out std_logic
    );
  end component;

  signal x_tb, y_tb, out_tb : std_logic_vector(15 downto 0);
  signal zx_tb, nx_tb, zy_tb, ny_tb, f_tb, no_tb, zr_tb, ng_tb : std_logic;
begin
  uut: ALU port map (x => x_tb, y => y_tb, zx => zx_tb, nx => nx_tb, zy => zy_tb, ny => ny_tb, f => f_tb, no => no_tb, out => out_tb, zr => zr_tb, ng => ng_tb);

  stimulus: process
  begin
    x_tb <= "0000000000000000";
    y_tb <= "0000000000000000";
    zx_tb <= '0';
    nx_tb <= '0';
    zy_tb <= '0';
    ny_tb <= '0';
    f_tb <= '0';
    no_tb <= '0';
    wait for 10 ns;

    -- Aquí puedes agregar más combinaciones de entradas para probar la ALU.

    wait;
  end process stimulus;
end tb_arch;
```

EXPLICACIÓN ALU:

- Primero, se define una entidad llamada ALU con dos entradas de 16 bits (x y y), seis señales de control (zx, nx, zy, ny, f, no) y dos salidas (out, zr, ng).
- Luego, se define una arquitectura para ALU. 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 varias señales intermedias (x1, x2, y1, y2, out1, out2) que se usarán para almacenar los resultados intermedios.
- Se implementan varias operaciones condicionales en función de las señales de control. Por ejemplo, si zx es ‘1’, entonces x se pone a cero. Si nx es ‘1’, entonces x se niega. Y así sucesivamente para zy, ny, f y no.
- Finalmente, se calculan las salidas zr y ng en función de la salida out.
-

El banco de pruebas (test bench) para la ALU funciona de la siguiente manera:

- Primero, se define una entidad llamada ALU_tb que no tiene puertos porque es un banco de pruebas.
- Luego, se define una arquitectura para ALU_tb. Dentro de esta arquitectura, se declara un componente llamado ALU que tiene las mismas entradas y salidas que la entidad ALU.
- Se declaran varias señales (x_tb, y_tb, out_tb, zx_tb, nx_tb, zy_tb, ny_tb, f_tb, no_tb, zr_tb, ng_tb) que se usarán para probar la ALU.
- Se instancia el componente ALU (denotado como uut para “unidad bajo prueba”) y se mapean las señales a los puertos correspondientes de la ALU.
- Finalmente, se define un proceso llamado stimulus que genera diferentes combinaciones de entradas para x, y y las señales de control, y espera 10 ns después de cada cambio. Esto simula el comportamiento de la ALU para todas las posibles combinaciones de entradas.