PRÁCTICA 2: COMUNICACIÓN SERIE CON UN PC. DISEÑO DEL TRANSMISOR

Alumnos: Pablo Menéndez Ruiz de Azúa y Carles Olucha Royo
Fecha de realización: 10/02/2020
Fecha de entrega: 24/02/2021
Asignatura: Sistemas digitales II
Profesor: Pedro Olmos González

Índice

Objetivos

El objetivo de esta práctica es el diseño de un circuito de transmisión serie RS-232 para enviar datos desde la tarjeta de FPGA hacia un PC.

Introducción

Los datos que se transmiten al PC constan de 8 bits cada uno y un bit de paridad (paridad impar) y un bit de stop. La velocidad de transmisión será de 19200 baudios. En la figura 1 se muestra un diagrama de tiempos de la transmisión.

El funcionamiento del circuito será el siguiente: cuando se active el pulsador, se guardará en el registro el byte para realizar la conversión paralelo serie (lo opuesto a lo que hicimos en la práctica anterior). Después se generará el bit de paridad y después un bit para decir que la transmisión ha terminado. En conclusión, se enviará por este orden el bit de arranque, los 8 bits de datos, el de paridad y el de stop.

Figura 1 – Diagrama de tiempos de la señal transmitida

Diagrama de bloques

Antes de empezar a programar cada uno de los componentes necesarios para diseñar nuestro receptor, vamos a hacer un diagrama de bloques con cada componente del circuito.

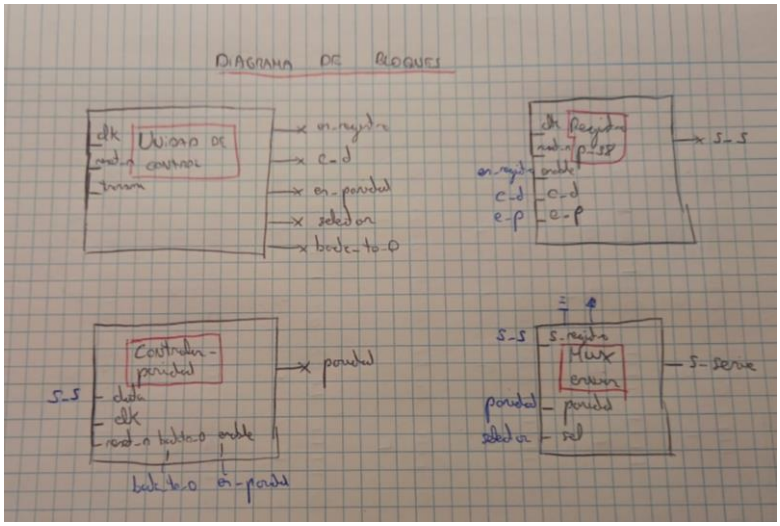

Figura 2 – Diagrama de bloques del circuito

En esta imagen, podemos ver los 4 bloques principales de nuestro receptor: el registro serie-paralelo, el controlador de paridad, la unidad de control y el multiplexor que nos va a servir para controlar la salida del circuito. El resto de los bloques como el contador de un bit se encuentran dentro de la unidad de control.

Componentes

Los componentes que hemos diseñado para esta práctica han sido los siguientes: Un contador de 8 bits para controlar el registro, un contador de 1 bit para que cuente el tiempo que dura cada bit, un controlador de paridad que genera el bit de paridad, un detector de flanco de bajada, un multiplexor para enviar un dato u otro dependiendo de en qué momento de la transmisión estemos, un registro paralelo-serie de 8 bits que usaremos para sacar un bit y finalmente la unidad de control que permita controlar todas las salidas y entradas.

De todos estos componentes, únicamente vamos a explicar y a enseñar en el informe el código del registro paralelo-serie, el controlador de paridad, el multiplexor, el contador de un bit y la unidad de control. Hemos elegido estos porque nos parecen los más importantes y necesarios para explicar. El resto de los componentes consideramos que no son muy complejos por lo que no vemos necesario enseñar y explicar su código. De todas formas, si quieren ver de todos los componentes utilizados para esta práctica y los testbenches utilizados para las simulaciones, se adjuntan junto a este documento.

Registro serie-paralelo

En primer lugar, es necesario realizar un registro paralelo-serie de desplazamiento de 8 bits para ir enviando los datos uno a uno. Ha de tenerse en cuenta que dicho registro ha de ser síncrono, es decir, su reloj ha de ser el mismo que el de los demás componentes de la FPGA (50 MHz). Por tanto, ha de diseñarse el registro de manera que solamente realice un desplazamiento mientras esté activa una señal de control (enable).

```
3   use ieee.numeric_std.all;
4
5   entity Registrops8 is
6       port(
7           clk : in std_logic;
8           reset_n : in std_logic;
9           enable : in std_logic;
10          c_d : in std_logic;
11          e_p : in std_logic_vector(7 downto 0);
12          s_s : out std_logic);
13  end Registrops8;
14
15  architecture behavioral of Registrops8 is
16      signal registro : std_logic_vector(7 downto 0);
17  begin
18      process(clk,reset_n,enable)
19      begin
20          if reset_n ='0' then
21              registro<=(others=>'0');
22          else
23              if rising_edge(clk) then
24                  if enable='1' then
25                      if c_d='0' then
26                          registro<=e_p;
27                      else
28                          registro(7)<='0';
29                          registro(6 downto 0)<=registro(7 downto 1);
30                      end if;
31                  end if;
32              end if;
33          end if;
34      end process;
35      s_s<=registro(0);
36  end behavioral;
```

Figura 3 – Código VHDL del registro paralelo-serie de 8 bits

Una vez que hemos diseñado en VHDL el registro, vamos a simularlo para comprobar que funciona correctamente. Para ello, nos creamos primero un testbench y lo simulamos obteniendo el siguiente resultado. Todo el código VHDL y los testbenches utilizados para realizar esta práctica vienen adjuntos junto a este documento.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Registrops8_vhd_tst IS
END Registrops8_vhd_tst;
ARCHITECTURE Registrops8_arch OF Registrops8_vhd_tst IS

    signal clk: std_logic:='0';
    signal reset_n : std_logic;
    signal enable    : std_logic;
    signal c_d       : std_logic;
    signal s_s   : std_logic;
    signal e_p       : std_logic_vector(7 downto 0);
    constant clk_per : time:= 10 ns;

    component Registrops8 is
        port(
            clk : in std_logic;
            reset_n : in std_logic;
            enable    : in std_logic;
            c_d         : in std_logic;
            s_s           : out std_logic;
            e_p        : in std_logic_vector(7 downto 0)
        );
    end component;

begin

    i1 : Registrops8
    port map(
        clk => clk,
        reset_n => reset_n,
        enable => enable,
        c_d => c_d,
        s_s => s_s,
        e_p => e_p
    );
```

```vhdl
begin

    i1 : Registrops8
    port map(
        clk => clk,
        reset_n => reset_n,
        enable => enable,
        c_d => c_d,
        s_s => s_s,
        e_p => e_p
    );

    p_clk: process
    begin
        clk <= '0';
        wait for clk_per / 2;
        clk <= '1';
        wait for clk_per / 2;
    end process;

    p_rstn : process
    begin
        reset_n <= '0';
        wait for 100 ns;
        reset_n <= '1';
        wait;
    end process;

    p_stim : process
    begin
        e_p <= "00000000";
        enable <= '0';
        c_d <= '0';
        wait until reset_n = '1';
        wait for 5 ns;
        e_p <= "10101010";
        enable <= '1';
        wait for clk_per;
        enable <= '0';
        c_d <= '1';
        wait for 100 ns;
        for n in 0 to 7 loop
            enable <= '1';
            wait for clk_per;
            assert s_s = '0';
            enable <= '0';
            wait for 100 ns;
        end loop;
        assert false
            report "Fin de la simulación"
            severity failure;

    end process p_stim;
end Registrops8_arch;
```

Figura 4 – Testbench del registro

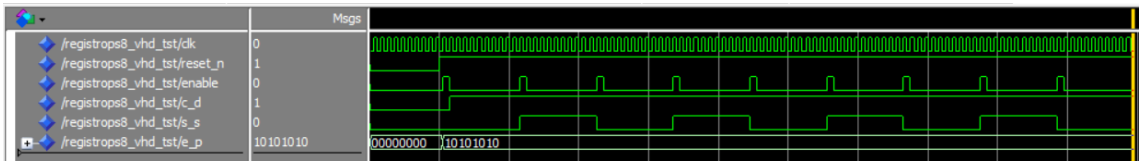

Figura 5 – Simulación del registro paralelo-serie

Generador del bit de paridad

En segundo lugar, es preciso diseñar un circuito secuencial para calcular el bit de paridad que se transmitirá después del byte de datos. En el caso de paridad impar, la suma de todos los bits, incluido el de paridad, ha de ser impar.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Controlar_paridad is
5      port(
6          data : in std_logic;
7          paridad : out std_logic;
8          clk     : in std_logic;
9          back_to_0 : in std_logic;
10         reset_n : in std_logic;
11         enable  : in std_logic
12     );
13 end Controlar_paridad;
14
15 architecture behavioral of Controlar_paridad is
16     signal b : std_logic;
17 begin
18
19     process(clk,reset_n)
20     begin
21         if reset_n = '0' then
22             b <= '0';
23         elsif rising_edge(clk) then
24             if enable = '1' then
25                 b <= data xor b;
26             elsif back_to_0 = '1' then
27                 b <= '0';
28             end if;
29         end if;
30     end process;
31
32     paridad <= not b;
33 end behavioral;
```

Figura 6 – Código VHDL del generador de paridad

Para detectar la paridad lo único que vamos a hacer es pasar el bit de entrada por una xor que lo compara con un acumulado de xor de los bits anteriores. Por tanto, si al final después de haber pasado por todos los bits de datos, el resultado de la xor es 0, quiere decir que hay un número par de unos por lo que tenemos que el bit de paridad será un 1 para que el número de unos sea impar. Si el resultado de la xor es un 1, el bit de paridad será 0.

Además, en el código hemos añadido una señal que actúa como reset pero cuando lo necesitemos, ya que después de terminar de transmitir un dato necesitamos que la señal b, en la que se va acumulando el resultado de la xor, vuelva a 0.

Este componente también lo vamos a simular y al simularlo, obtenemos lo siguiente.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Controlar_paridad_vhd_tst IS
END Controlar_paridad_vhd_tst;
ARCHITECTURE Controlar_paridad_arch OF Controlar_paridad_vhd_tst IS

    signal clk: std_logic:='0';
    signal reset_n : std_logic;
    signal enable    : std_logic;
    signal back_to_0    : std_logic;
    signal data       : std_logic;
    signal paridad   : std_logic;
    constant clk_per : time:= 10 ns;

    component Controlar_paridad is
        port(
            clk : in std_logic;
            reset_n : in std_logic;
            enable    : in std_logic;
            back_to_0   : in std_logic;
            data        : in std_logic;
            paridad     : out std_logic
        );
    end component;

begin

    i1 : Controlar_paridad
    port map(
        clk => clk,
        reset_n => reset_n,
        enable => enable,
        back_to_0 => back_to_0,
        data => data,
        paridad => paridad
    );

    p_clk: process
    begin
        clk <= '0';
        wait for clk_per / 2;
        clk <= '1';
        wait for clk_per / 2;
    end process;

    p_rstn : process
    begin
        reset_n <= '0';
        wait for 100 ns;
        reset_n <= '1';
        wait;
    end process;

    p_stim : process
    begin
        data <= '0';
        back_to_0 <= '0';
        enable <= '0';
        wait until reset_n = '1';
        wait for 5 ns;
        for n in 0 to 3 loop
            enable <= '1';
            data <= '1';
            wait for clk_per;
            enable <= '0';
            wait for 100 ns;
            enable <= '1';
            data <= '0';
            wait for clk_per;
            enable <= '0';
            wait for 100 ns;
        end loop;
        assert paridad = '1'
            report "La paridad tiene un valor erróneo"
            severity failure;
        wait for 100 ns;
        back_to_0 <= '1';
        wait for 100 ns;
        assert paridad = '0'
            report "La paridad no vuelve a 0"
            severity failure;
        wait for 100 ns;
        assert false
            report "Fin"
            severity failure;
    end process p_stim;
end Controlar_paridad_arch;
```

Figura 7 – Testbench del generador de paridad

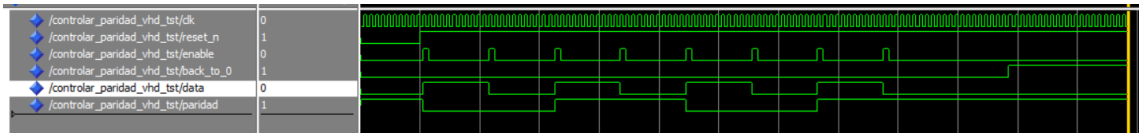

Figura 8 – Simulación del generador de paridad

Como vemos, en este caso, le hemos pasado 8 bits de los cuales 4 eran unos. Por lo tanto, nos tenía que devolver al final un uno el bit de paridad como ha ocurrido.

Contador de 1 bit

Como la señal llega al PC a una velocidad de 19200 baudios debemos sincronizar esta señal con nuestro reloj de 50 MHz. Para ello, debemos crearnos un contador que devuelva un pulso cada vez que pase un tiempo proporcional al ancho de un bit. Después de hacer los cálculos, llegamos a que un bit se manda cada 2604 pulsos de nuestro reloj de 50 Mhz. Por lo tanto, diseñaremos un contador que genere un pulso (enable) cada 2604 pulsos del reloj, y cada una de estas señales de enable se mandan al registro, para que saque el bit siguiente.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity Contador1bit is
6       port(
7           reset_n : in std_logic;
8           clk     : in std_logic;
9           en      : in std_logic;
10          co      : out std_logic;
11          back_to_0 : in std_logic
12      );
13  end Contador1bit;
14
15  architecture behavioral of Contador1bit is
16      signal contador : unsigned(11 downto 0);
17
18  begin
19
20      process(reset_n,clk)
21      begin
22          if reset_n = '0' then
23              contador <= (others => '0');
24          else
25              if rising_edge(clk) then
26                  if en='1' then
27                      if contador= 2604 then
28                          contador <= (others => '0');
29                      else
30                          contador <= contador + 1;
31                      end if;
32                  elsif back_to_0 = '1' then
33                      contador <= (others => '0');
34                  end if;
35              end if;
36          end if;
37      end process;
38
39      co <= '1' when contador = 2604 and en = '1' else '0';
40
41  end behavioral;
```

Figura 9 – Código VHDL del contador de 1 bit (2604 pulsos de reloj)

Multiplexor

El multiplexor diseñado en esta práctica tiene como función controlar qué salida sale en cada momento. Es decir, tiene que controlar si sale el bit de start, los bits de la salida del registro, el bit de paridad o el de stop. Esta salida la controla a través de una señal llamada selector.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Mux_enviar is
5       port(
6           s_registro : in std_logic;
7           paridad    : in std_logic;
8           sel        : in std_logic_vector(1 downto 0);
9           s_serie    : out std_logic
10      );
11  end Mux_enviar;
12
13  architecture behavioral of Mux_enviar is
14  begin
15
16      with sel select
17          s_serie <=
18          '0' when "00",
19          s_registro when "01",
20          paridad when "10",
21          '1' when "11",
22          '1' when others;
23
24  end behavioral;
25
```

Figura 10 – Código VHDL del multiplexor

Como se puede observar en el código si el selector es 00, la salida es el bit de start; si es 01, la salida es la del registro; si es 10 es el bit de paridad y si es 11 es el bit de stop.

Unidad de Control

La unidad de control nos sirve para controlar tanto el registro como el generador del bit de paridad, como el multiplexor. Antes de empezar a diseñar la unidad de control en vhdl tenemos que hacernos el diagrama de estados, que es el siguiente.

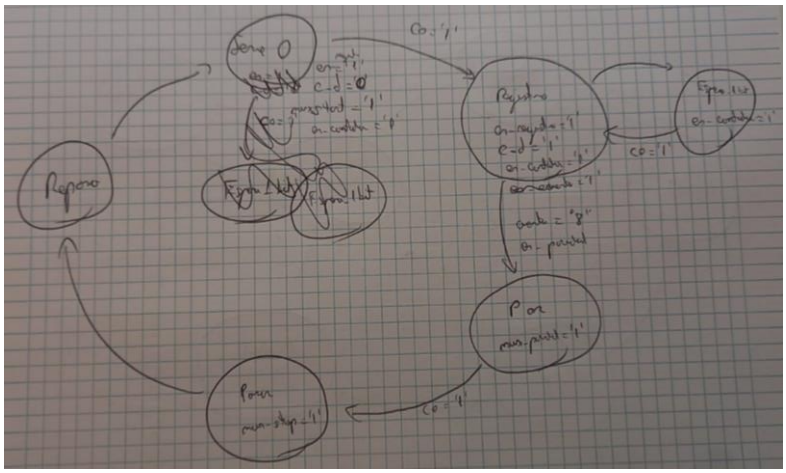

Figura 11 – Diagrama de estados de la unidad de control

Sabemos que en la imagen no se ve muy claro el diagrama, pero una vez que veamos en el código cómo cambian de un estado a otro y cómo van cambiando las salidas, creemos que quedará más claro.

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

entity Unidad_control is
    port(
        clk : in std_logic;
        reset_n : in std_logic;
        transmi : in std_logic;
        en_registro : out std_logic;
        c_d         : out std_logic;
        en_paridad  : out std_logic;
        selector    : out std_logic_vector(1 downto 0);
        back_to_0   : out std_logic
    );
end Unidad_control;

architecture behavioral of Unidad_control is

    signal empieza,en_conta8, en_contalbit, colbitsenyal, reinit : std_logic;
    signal cuenta : std_logic_vector(3 downto 0);

    type t_estados is (Reposo,Serie0, Registro, Esperalbit, Paridad, Parar);
    signal estado_act, estado_sig : t_estados;

    component DetectorFlancoBajada
    port(
      e       : in std_logic;
      reset_n     : in std_logic;
      clk         : in std_logic;
      s       : out std_logic
    );
    end component;
```

```vhdl
component Contador
    port(
        reset_n : in std_logic;
        clk     : in std_logic;
        en      : in std_logic;
        back_to_0: in std_logic;
        salida  : out std_logic_vector(3 downto 0)
    );
end component;

component Contador1bit
  port(
    reset_n : in std_logic;
    clk     : in std_logic;
    en      : in std_logic;
    back_to_0     : in std_logic;
  co        : out std_logic
    -- co_medio    : out std_logic
);
end component;
```

Inicialmente, hemos llamado a tres componentes que vamos a necesitar en la unidad lógica. Uno de ellos es el detector de flanco de bajada, que nos será útil para saber cuándo empezar a enviar datos. El otro es un contador de módulo 8 que nos va a servir para ir controlando los cambios de un estado a otro. Este contador nos va a servir específicamente para que cuando llegue a 8 deje de registrar. Además hay que tener en cuenta el Contador1bit que nos ayudará a controlar el tiempo que dura la trasmisión de cada bit.

```vhdl
TransicionEstados : process (estado_act, estado_sig,empieza,colbitsenyal)
begin
    estado_sig <= estado_act;
    case estado_act is
        when Reposo =>
            if empieza = '1' then
                estado_sig <= Serie0;
            end if;
        when Serie0 =>
            if colbitsenyal = '1' then
                estado_sig <= Registro;
            end if;
        when Registro =>
            if cuenta = "1000" then
                estado_sig <= Paridad;
            else
                estado_sig <= Esperalbit;
            end if;

        when Esperalbit =>
        if colbitsenyal = '1' then
            estado_sig<=Registro;
        end if;
        when Paridad =>
        if colbitsenyal = '1' then
            estado_sig<=Parar;
        end if;
        when Parar =>
            if colbitsenyal = '1' then
                estado_sig<=Reposo;
            end if;
        when others =>
            estado_sig <= Reposo;
    end case;
end process TransicionEstados;
```

Figura 12 – Transición de estados de la máquina de estados

En la imagen de arriba vemos como cambian los estados de uno a otro y vamos a explicar esta transición. En primer lugar, si pulsamos el botón que inicializa la trasmisión pasamos a un estado Serie0. Cuando pasa un bit, ya se ha transmitido el bit de start, por lo que pasamos a transmitir el byte de datos y para ello usamos dos estados Registro y Espera1bit. Una vez se hayan registrado y transmitido los 8 bits de datos se transmite el de paridad a través del estado Paridad. Por último, enviamos el bit de stop, después de que pase el tiempo equivalente a un bit transmitido.

```
130
131     Salidas : process (estado_act)
132     begin
133
134         en_contalbit   <= '0';
135         c_d            <= '0';
136         en_registro      <= '0';
137         en_paridad       <= '0';
138       en_conta8      <= '0';
139         selector       <= "11";
140         reinit       <= '0';
141
142         case estado_act is
143             when Reposo =>
144                 null;
145             when Serie0 =>
146                 reinit <= '1';
147                 en_registro <= '1';
148                 c_d <= '0';
149                 selector <= "00";
150                 en_contalbit <= '1';
151             when Registro =>
152                 en_registro <= '1';
153                 c_d <= '1';
154                 selector <= "01";
155             en_conta8<='1';
156             en_paridad<='1';
157             when Esperalbit =>
158                 en_contalbit <= '1';
159                 selector <= "01";
160             when Paridad =>
161                 en_contalbit<='1';
162             selector <= "10";
163             when Parar =>
164                 en_contalbit<='1';
165                 selector <= "11";
166             when others =>
167                 null;
168         end case;
169     end process Salidas;
170
171     back_to_0 <= reinit;
172
173
174 end behavioral;
```

Figura 13 – Salidas en de la Máquina de estados

En lo que se refiere a las salidas, ocurre lo siguiente. Inicialmente, colocamos todos nuestras señales a 0. Cuando estamos transmitiendo el bit de start, ponemos todos nuestros componentes a 0 con reinit (back_to_0), activamos el enable del registro, pero para cargar el número que el estamos pasando, colocamos el selector a 00 para que transmita el bit de start y activamos el enable del contador de 1 bit. Cuando registramos, activamos únicamente la señal de enable del registro para ir desplazando los datos y el selector lo ponemos en “01” para que transmita lo que sale del registro. Además, habilitamos el contador de módulo 8 y el generador de paridad porque calcula la paridad de forma secuencial. En el resto de estados restante, lo único que vamos cambiando es el selector para que vaya cambiando la salida del circuito.

Una vez que hemos terminado nuestro código y hemos comprobado que no teníamos errores compilando, vamos a simular la unidad de control para ver si nuestro código es funcional. A la hora de simular, le he metido una variable al circuito para que fuera llevando la cuenta del contador de módulo 9 para que pudiera ver si había algún error. Como se puede observar la simulación funciona correctamente, ya que realiza el trabajo para el que ha sido diseñada.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Unidad_control_vhd_tst IS
END Unidad_control_vhd_tst;
ARCHITECTURE Unidad_control_arch OF Unidad_control_vhd_tst IS
    signal clk : std_logic:='0';
    signal reset_n :std_logic;
    signal transmi :  std_logic;
    signal en_registro : std_logic;
    signal c_d         : std_logic;
    signal en_paridad  : std_logic;
    signal selector    : std_logic_vector(1 downto 0);
    signal back_to_0   : std_logic;
    constant clk_per : time:= 10 ns;

    component Unidad_control is
        port(
        clk : in std_logic;
        reset_n : in std_logic;
        transmi : in std_logic;
        en_registro : out std_logic;
        c_d         : out std_logic;
        en_paridad  : out std_logic;
        selector    : out std_logic_vector(1 downto 0);
        back_to_0   : out std_logic
        );
    end component;
begin
    i1 : Unidad_control
    port map(
        clk => clk,
        reset_n => reset_n,
        transmi => transmi,
        en_registro => en_registro,
        c_d         => c_d,
        en_paridad  => en_paridad,
        selector    => selector,
        back_to_0   => back_to_0
    );

    p_clk: process
    begin
        clk <= '0';
        wait for clk_per / 2;
        clk <= '1';
        wait for clk_per / 2;
    end process;

    p_rstn : process
    begin
        reset_n <= '0';
        wait for 100 ns;
        reset_n <= '1';
        wait;
    end process;

    p_stim : process
    begin
        transmi <= '1';
        wait until reset_n = '1';
        wait for 5 ns;
        transmi <= '0';
        wait for 20 ns;
       -- assert selector = "00"
        --    report "No detecta el flanco de bajada"
        --    severity failure;
        wait for 26020 ns;
        for i in 0 to 4 loop
            wait for 26040 ns;
            wait for 26040 ns;
        end loop;

        assert false
            report "Fin de la simulación"
            severity failure;
    end process p_stim;
end Unidad_control_arch;
```

Figura 14 – Testbench de la unidad de control

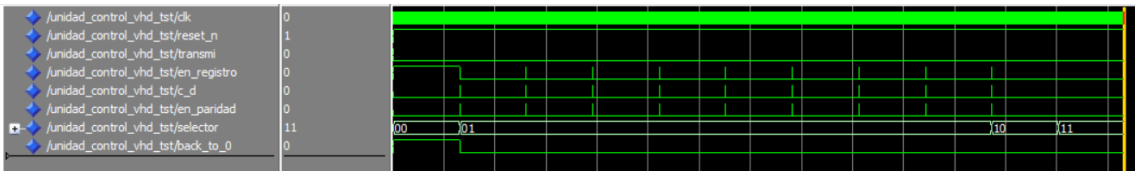

Figura 15 – Simulación de la Unidad de control

Circuito completo

Por último, debemos juntar todos los componentes en un componente más grande al que llamaremos pract2. Lo único que vamos a hacer en este componente es instanciar el resto de los componentes que hemos nombrado en el apartado anterior.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity pract2 is
6       port(
7       clk : in std_logic;
8       e_p : in std_logic_vector(7 downto 0);
9       reset : in std_logic;
10      s_s : out std_logic;
11      transmi : in std_logic
12      );
13  end pract2;
14
15  architecture behavioral of pract2 is
16      signal en_registro_senyal : std_logic;
17      signal c_d_senyal: std_logic;
18      signal s_registro_senyal: std_logic;
19      signal paridad_senyal: std_logic;
20      signal back_to_0_senyal: std_logic;
21      signal en_paridad_senyal: std_logic;
22      signal selector_senyal : std_logic_vector(1 downto 0);
23      --signal s_ssenyal_final : std_logic;
24
25      component registrops8
26          port(
27          clk : in std_logic;
28          reset_n : in std_logic;
29          enable : in std_logic;
30          c_d : in std_logic;
31          e_p : in std_logic_vector(7 downto 0);
32          s_s : out std_logic
33      );
34      end component;
35
36      component Controlar_paridad
37          port(
38          data : in std_logic;
39              paridad : out std_logic;
40              clk     : in std_logic;
41              back_to_0 : in std_logic;
42              reset_n : in std_logic;
43              enable  : in std_logic
44          );
45      end component;
46
```

```
47      component Mux_enviar
48          port(
49              s_registro : in std_logic;
50              paridad    : in std_logic;
51              sel        : in std_logic_vector(1 downto 0);
52              s_serie    : out std_logic
53      );
54      end component;
55
56      component Unidad_control
57          port(
58          clk : in std_logic;
59          reset_n : in std_logic;
60          transmi : in std_logic;
61          en_registro : out std_logic;
62          c_d         : out std_logic;
63          en_paridad  : out std_logic;
64          selector    : out std_logic_vector(1 downto 0);
65          back_to_0   : out std_logic
66      );
67      end component;
68
69  begin
70
71      i_registrops8 : registrops8
72      port map(
73          clk =>clk,
74          reset_n => reset,
75          enable => en_registro_senyal,
76          c_d =>c_d_senyal,
77          e_p =>e_p,
78          s_s =>s_registro_senyal);
79
80      i_Controlar_paridad : Controlar_paridad
81      port map(
82          data =>s_registro_senyal,
83          paridad =>paridad_senyal,
84          clk     =>clk,
85          back_to_0 =>back_to_0_senyal,
86          reset_n => reset,
87          enable  =>en_paridad_senyal
88          );
89
```

```
90      i_Mux_enviar : Mux_enviar
91      port map(
92          s_registro =>s_registro_senyal,
93          paridad    =>paridad_senyal,
94          sel        =>selector_senyal,
95          s_serie    =>s_s);
96
97      i_Unidad_control : Unidad_control
98      port map(
99          clk =>clk,
100         reset_n =>reset,
101         transmi =>transmi,
102         en_registro =>en_registro_senyal,
103         c_d         =>c_d_senyal,
104         en_paridad  =>en_paridad_senyal,
105         selector    =>selector_senyal,
106         back_to_0   =>back_to_0_senyal
107     );
108
109 end behavioral;
```

Figura 16 – Código VHDL de la práctica1

Una vez que hemos juntado todos los componentes en el elemento práctica 1, vamos a simular todo el componente para ver si realmente funciona nuestro código.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY pract2_vhd_tst IS
END pract2_vhd_tst;
ARCHITECTURE pract2_arch OF pract2_vhd_tst IS
    signal clk : std_logic :='0';
    signal reset : std_logic;
    signal transmi   : std_logic;
    signal e_p     : std_logic_vector(7 downto 0);
    signal s_s       : std_logic;
    constant clk_per :time:= 10 ns;

    component pract2 is
        port(
            clk : in std_logic;
            e_p : in std_logic_vector(7 downto 0);
            reset : in std_logic;
            s_s : out std_logic;
            transmi : in std_logic
            );
    end component;

begin

    i1 : pract2
    port map(
        clk => clk,
        e_p => e_p,
        reset => reset,
        s_s   => s_s,
        transmi => transmi
    );

    p_clk: process
    begin
        clk <= '0';
        wait for clk_per / 2;
        clk <= '1';
        wait for clk_per / 2;
    end process;

    p_rstn : process
    begin
        reset <= '0';
        wait for 100 ns;
        reset <= '1';
        wait;
    end process;

    p_stim : process
    begin
        e_p <= "01010111";
        transmi <= '1';

        wait until reset = '1';
        wait for 5 ns;
        transmi <= '0';
        wait for 26050 ns;
        for i in 0 to 4 loop
            wait for 26040 ns;
            wait for 26040 ns;
        end loop;

        assert false
            report "Fin de la simulación"
            severity failure;
    end process p_stim;
end pract2_arch;
```

Figura 17 – Testbench de la práctica

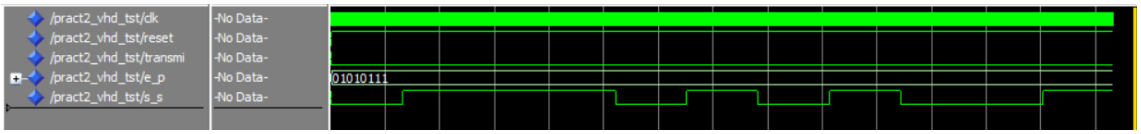

Figura 18 – Simulación de la práctica entera

Implantación física

Para realizar la implantación física y la comprobación mediante el código ASCII, hemos necesitado conectar la placa mediante el cable USB y mediante el COM1 usando el programa hyperterm.

Hemos obtenido los siguientes resultados:

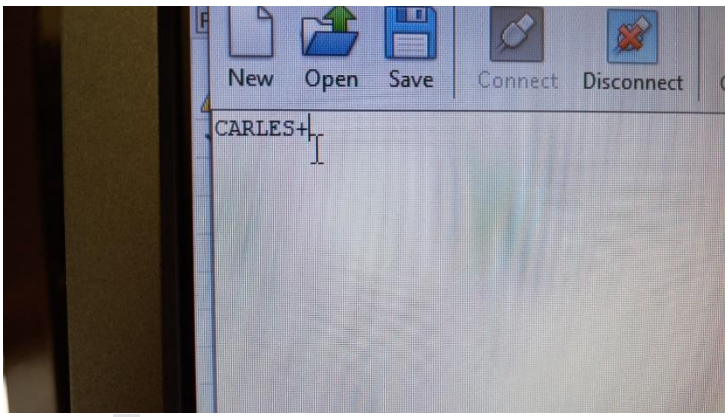

Figura 19 – Volcar en la placa y escribir el nombre de Carles
67 | 65 | 82 | 76 | 69 | 83

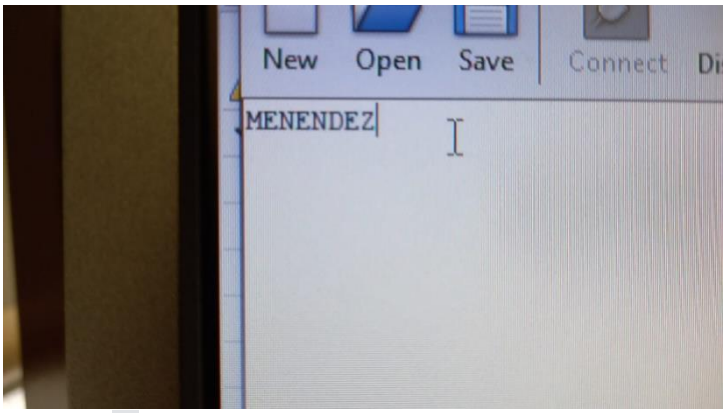

Figura 20 – Volcar en la placa y escribir el nombre Menéndez
77 | 69 | 78 | 69 | 78 | 68 | 69 | 90

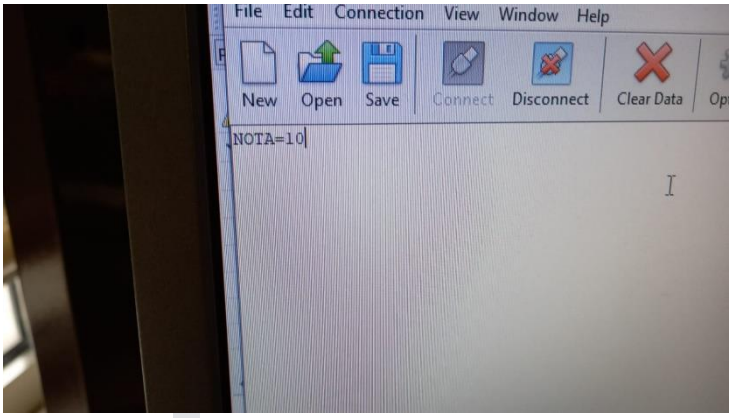

Figura 21 – Volcar en la placa y escribir NOTA=10
78 | 79 | 84 | 65 | 61 | 49 | 48

Conclusiones y análisis de los resultados

Al inicio de esta práctica teníamos un objetivo principal que era diseñar un transmisor serie RS-232 para enviar datos desde la placa FPGA hasta el ordenador. Después de haberlo diseñado utilizando código VHDL, haberlo simulado con éxito, haberlo implantado en la placa y haber visto cómo los caracteres que enviábamos realmente se transmitían, podemos decir que hemos cumplido con nuestro objetivo.

Esta práctica, después de haber realizado, la del receptor RS-232, nos ha resultado más sencilla ya que desde el principio entendíamos muy bien que era lo que teníamos que hacer. De todas formas, sí que hemos cometido algunos errores que gracias a las simulaciones hemos podido detectar.