PRÁCTICA 1: COMUNICACIÓN SERIE CON UN PC. DISEÑO DEL RECEPTOR

Alumnos: Pablo Menéndez Ruiz de Azúa y Carles Olucha Royo
Fecha de realización: 27/01/2020
Fecha de entrega: 10/02/2021
Asignatura: Sistemas digitales I
Profesor: Pedro Olmos González

Índice

Objetivos

El objetivo de esta práctica es el diseño de un receptor serie RS-232 para recibir datos de un PC y mostrarlos en binario en los LEDs de la tarjeta DE1.

Introducción

Los datos que se reciben desde el PC constan de 1 bit de arranque, 8 bits de datos, un bit de paridad (paridad impar) y un bit de stop. La velocidad de transmision será de 19200 baudios. Como en la transmisión serie RS-232 no se transmite el reloj, es necesario sincronizar el reloj del receptor en cada byte, para lo que se utiliza el bit de arranque. En la figura 1 se muestra un diagrama de tiempos de la transmisión, en el que se muestran los instantes en los que es deseable muestrear la línea de datos. Como se puede apreciar, para disminuir los efectos de la asincronía entre los relojes del emisor y el receptor, los datos han de muestrearse en la mitad del tiempo de transmisión de cada bit, tal como se muestra en la figura.

El funcionamiento del circuito será el siguiente: cuando se detecte el flanco de bajada de la señal de datos, lo cual indica el comienzo de la trama, se esperará el tiempo de medio bit y se verificará que la señal de datos sigue a cero. A continuación, se esperará el tiempo de un bit y se muestreará el estado de la señal de entrada, introduciendo su valor en un registro de desplazamiento. Este proceso se repetirá ocho veces para obtener en la salida paralelo del registro de desplazamiento el dato enviado en la trama. A continuación, se leerá el bit de paridad (después de esperar el tiempo de un bit) y por último el bit de stop. Si se produce un error de paridad se iluminará un LED para indicarlo. Por otro lado, si el bit de start no es cero o el bit de stop no es uno, se iluminará un LED para indicar que se ha producido un error de trama (framing error). Ambos LED permanecerán encendidos hasta que se reciba el siguiente bit de start.

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.

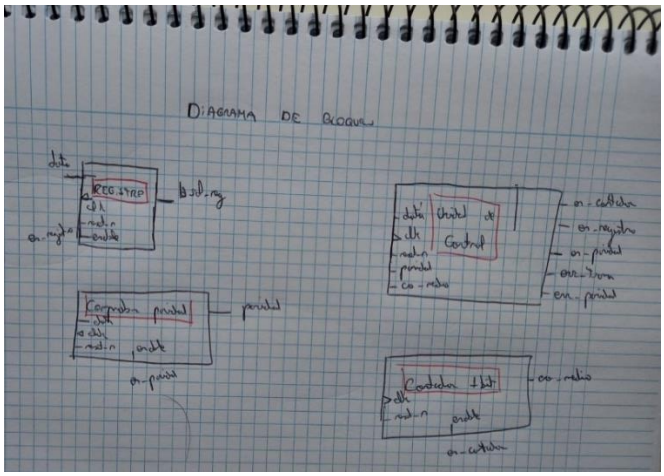


Figura 2 – Diagrama de bloques del circuito

En esta imagen, podemos ver los 4 bloques principales de nuestro receptor: el registro serie-paralelo, el comprobador de paridad, la unidad de control y el contador de un bit que nos va a permitir controlar los tiempos.

Componentes

Los componentes que hemos diseñado para esta práctica han sido los siguientes: Un registro serie-paralelo de 8 bits que utilizaremos para ir registrando el dato que nos pasa el transmisor, un comprobador de paridad para ver si hay errores de paridad, un contador de un bit para generar un pulso cuando pase el tiempo correspondiente a medio bit, un contador de módulo 9 para que se repita la acción de registrar y esperar un bit, la cual se tiene que repetir 8 veces, un detector de flanco de bajada para saber que se empiezan a transmitir los bits, un registro paralelo que nos permite copiar los bits que van saliendo y la unidad de control que va a ser la que controle en qué momento ocurre cada cosa.

De todos estos componentes, únicamente vamos a explicar y a enseñar en el informe el código del registro serie-paralelo, el comprobador de paridad, 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 de desplazamiento de 8 bits para almacenar el dato recibido. 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 solo realice un desplazamiento mientras este activa una señal de control (enable).

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

entity Registro is
    port(
        clk     : in std_logic;
        enable  : in std_logic;
        reset_n : in std_logic;
        data    : in std_logic;
        bits    : out std_logic_vector(7 downto 0)
    );
end Registro;

architecture behavioral of Registro is
    signal registro : std_logic_vector(7 downto 0);
begin
    process(clk, reset_n)
    begin
        if reset_n= '0' then
            registro <= (others =>'0');
        else
            if rising_edge(clk) then
                if enable = '1' then
                    registro(7) <= data;
                    registro(6 downto 0) <= registro(7 downto 1);
                end if;
            end if;
        end if;
    end process;

    bits <= registro(7 downto 0);
end behavioral;
```

Figura 3 – Código VHDL del registro serie-paralelo

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.

```
63  p_clk: process
64  begin
65      clk <= '0';
66      wait for clk_per / 2;
67      clk <= '1';
68      wait for clk_per / 2;
69  end process;
70
71  p_rstn : process
72  begin
73      reset_n <= '0';
74      wait for 100 ns;
75      reset_n <= '1';
76      wait;
77  end process;
78
79
80  p_stim : PROCESS
81  -- optional sensitivity list
82  -- (        )
83  -- variable declarations
84  BEGIN
85      data <= '0';
86      enable <= '0';
87
88      wait until reset_n = '1';
89
90      wait for 5 ns;
91
92      for n in 0 to 3 loop
93
94          data <= '1';
95
96          wait for clk_per;
97
98          enable <= '1';
```

```
99
100         wait for clk_per;
101
102         enable <= '0';
103
104         wait for 80 ns;
105
106         data <= '0';
107
108         wait for clk_per;
109
110         enable <= '1';
111
112         wait for clk_per;
113
114         enable <= '0';
115
116         wait for 80 ns;
117     end loop;
118
119     assert bits = "01010101"
120     report "Error: salida del registro errónea"
121         severity failure;
122
123     assert false
124     report "Fin de la simulación"
125         severity failure;
126
127         -- code executes for every event on sensitivity list
128 END PROCESS p_stim;
129 END Registro_arch;
130
```

Figura 4 – Testebench del registro

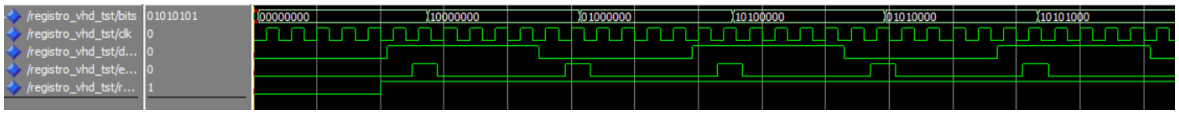


Figura 5 – Simulación del registro serie-paralelo

Detector de errores de paridad

En segundo lugar, es preciso diseñar un circuito secuencial para verificar que la transmisión ha sido correcta. En el caso de paridad impar, la suma de todos los bits, incluido el de paridad, ha de ser impar.

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

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

architecture behavioral of Comprobar_paridad is
    signal b : std_logic;
begin

    process(clk,reset_n)
    begin
        if reset_n = '0' then
            b <= '0';
         elsif back_to_0 = '1' then
                b <= '0';
        elsif rising_edge(clk) then
            if enable = '1' then
                b <= data xor b;
            end if;
        end if;
    end process;

    paridad <= b;
end behavioral;
```

Figura 6 – Código VHDL del detector 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, incluido el de paridad, la función devuelve 1 es impar y si devuelve 0 es par. Además, en el código hemos añadido una señal que actúa como reset pero cuando lo necesitemos, ya que al simular el componente final de la práctica tuvimos un pequeño problema con esto y era que la variable paridad no volvía a 0, por lo que ya el siguiente número nos daba un error de paridad

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

```
46      BEGIN
47          data <= '0';
48          enable <= '0';
49
50          wait until reset_n = '1';
51
52          wait for 10 ns;
53
54          for n in 0 to 3 loop
55
56              data <= '1';
57              enable <= '1';
58              wait for clk_per;
59
60              enable <= '0';
61
62              wait for 100 ns;
63
64              data <= '0';
65              enable <= '1';
66
67              wait for clk_per;
68
69              enable <= '0';
70
71              wait for 100 ns;
72          end loop;
73
74          data <= '1';
75          enable <= '1';
76          wait for clk_per;
77
78          enable <= '0';
79
80          wait for 100 ns;
81
82          assert paridad = '0'
83          report "Error: salida de paridad errónea"
84              severity failure;
85
86          assert false
87          report "Fin de la simulación"
88              severity failure;
```

Figura 7 – Testbench del comprobador de paridad

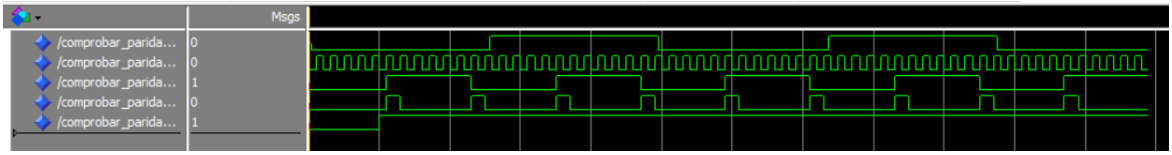


Figura 8 – Simulación detector de paridad

Como vemos, en este caso, le hemos pasado 9 bits de los cuales 5 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 nos llega 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 cuando se tenga que registrar un bit, que es en el centro de cada pulso que veíamos en la figura uno. Después de hacer los cálculos, llegamos a que un bit se manda cada 2604 pulsos de nuestro reloj de 50 Mhz. Por lo tanto, como siempre queremos registrar cuando esté en medio, vamos a hacer un contador que llegue como máximo hasta 2604 pero que saque una señal cada 1302 pulsos.

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

entity Contador1bit is
    port(
        reset_n : in std_logic;
        back_to_0 : in std_logic;
        clk     : in std_logic;
        en      : in std_logic;
        co_medio    : out std_logic
    );
end Contador1bit;

architecture behavioral of Contador1bit is
    signal contador : unsigned(11 downto 0);
begin
    process(reset_n,clk)
    begin
        if reset_n = '0' then
            contador <= (others => '0');
        elsif back_to_0 = '1' then
                contador <= (others => '0');
        else
            if rising_edge(clk) then
                if en='1' then
                    if contador= 2604 then
                        contador <= (others => '0');
                    else
                        contador <= contador + 1;
                    end if;
                end if;
            end if;
        end if;
    end process;
    co_medio <= '1' when contador = 1302 and en = '1' else '0';
end behavioral;
```

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

Como vemos, en este código también le hemos añadido la señal back_to_0, porque sino cuando había registrado el primer número se quedaba en 1302, y el siguiente bit de arranque ya no se miraba cuando pasara medio bit sino cuando pasaba un bit y ya se nos descuadraba todo el circuito. Al ser un elemento tan sencillo como un contador, este componente decidimos no simularlo.

Unidad de Control

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

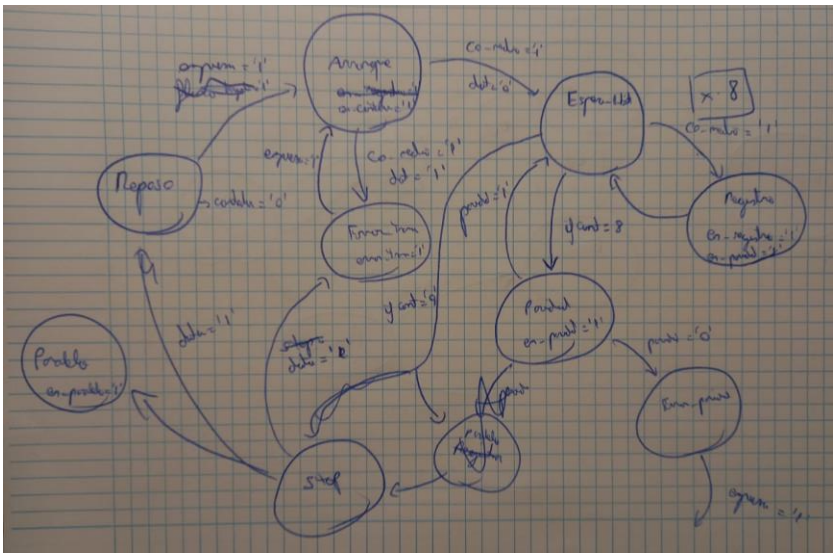


Figura 10 – Diagrama de estados de la unidad de control

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. Lo único que hemos añadido respecto al apartado de introducción es que, si al final no hay error de trama ni de paridad, se pase a un estado que llamamos Paralelo, que nos permite copiar los datos del registro en paralelo a la salida.

```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;
        data    : in std_logic;
        paridad : in std_logic;
        co_medio  : in std_logic;
        en_contador : out std_logic;
        en_registro : out std_logic;
        en_paridad  : out std_logic;
        err_trama : out std_logic;
        err_paridad : out std_logic;
        en_paralelo: out std_logic;
        contar      : out std_logic_vector(3 downto 0);
        pack_to_0   : out std_logic
        -- esta variable la utilizo solo para la simulaciÃ³n
    );
end Unidad_control;

architecture behavioral of Unidad_control is

    signal empieza,en, aux: std_logic;
    signal cuenta : std_logic_vector(3 downto 0);

    type t_estados is (Reposo,Arranque,Esperabit,Registrar,Par,ErrorParidad,ErrorTrama,Parar,Paralelo); --
    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;
    component Contador
        port(
            reset_n : in std_logic;
            aux     : in std_logic;
            clk     : in std_logic;
            en      : in std_logic;
            salida  : out std_logic_vector(3 downto 0)
        );
    end component;


begin
    i1_Flanco: DetectorFlancoBajada
    port map(
        e   => data,
        reset_n => reset_n,
        clk => clk,
        s   => empieza
    );

    i1_Contador : Contador
    port map(
        reset_n => reset_n,
        aux     => aux,
        clk     => clk,
        en      => en,
        salida  => cuenta
    );


    VarEstado: process(clk,reset_n)
    begin
      if reset_n = '0' then
        estado_act <= Reposo;
      elsif rising_edge(clk) then
        estado_act <= estado_sig;
      end if;
    end process VarEstado;
```

Inicialmente, hemos llamado a dos 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 empiezan a mandarnos datos. El otro es un contador de módulo 9 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 y pase a comprobar la paridad y cuando llegue a 9 pase a comprobar el bit de stop.

```
TransicionEstados : process (estado_act, estado_sig,empieza,co_medio,data,paridad)
begin
    estado_sig <= estado_act;
    case estado_act is
        when Reposo =>
            if empieza = '1' then
                estado_sig <= Arranque;
            end if;
        when Arranque =>
            if co_medio = '1' then
                if data = '0' then
                    estado_sig <= Espera1bit;
                elsif data = '1' then
                    estado_sig <= ErrorTrama;
                end if;
            end if;
        when Espera1bit =>
            if co_medio = '1' then
                if cuenta = "1000" then
                    estado_sig <= Par;
                elsif cuenta = "1001" then
                    estado_sig <= Parar;
                else
                    estado_sig <= Registrar;
                end if;
            end if;
        when Registrar =>
            estado_sig <= Espera1bit;
        when Par =>
            estado_sig <= Espera1bit;
        when Parar =>
            if data = '0' then
                estado_sig <= ErrorTrama;
            elsif paridad = '0' then
                estado_sig <= ErrorParidad;
            else
                estado_sig <= Paralelo;
            end if;
        when Paralelo =>
            estado_sig <= Reposo;
        when ErrorTrama =>
            if empieza = '1' then
                estado_sig <= Arranque;
            end if;
        when ErrorParidad =>
            if empieza = '1' then
                estado_sig <= Arranque;
            end if;
        when others =>
            estado_sig <= Reposo;
    end case;
end process TransicionEstados;
```

Figura 11 – 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 la señal de empieza, que es la que sale del flanco de bajada, se pone a uno, pasamos al estado de Arranque. Una vez en este estado, esperamos medio bit y si al terminar ese tiempo data no es 0 pasamos a ErrorTrama y si, por el contrario, es 0, pasamos a Espera1bit. Después de esperar un bit, vamos a registrar el siguiente número que entre y volvemos a esperar un bit. Este proceso se repite 8 veces. Cuando termine este proceso, comprobamos la paridad: si es 1 pasamos a comprobar el bit de stop y si es 0, vamos a ErrorParidad. Si cuando comprobamos el bit de stop, data es 0 vamos a ErrorTrama y si es 1, vamos a registrar la salida en paralelo. Después de registrar la salida, volvemos al reposo.

```
134     Salidas : process (estado_act)
135     begin
136
137         aux         <= '0';
138           back_to_0     <= '0';
139         en          <= '0';
140         en_contador   <= '0';
141         err_trama   <= '0';
142         en_registro   <= '0';
143         en_paridad    <= '0';
144         err_paridad <= '0';
145         en_paralelo <= '0';
146
147         case estado_act is
148             when Reposo =>
149                 back_to_0 <= '1';
150             when Arranque =>
151                 aux <= '1';
152                 en_contador <= '1';
153             when Espera1bit =>
154                 en_contador <= '1';
155             when Registrar =>
156                 en <= '1';
157                 en_registro <= '1';
158                 en_paridad <= '1';
159                 en_contador <= '1';
160             when Par =>
161                 en <= '1';
162                 en_paridad <= '1';
163                 en_contador <= '1';
164             when Parar =>
165                 null;
166             when Paralelo =>
167                 en_paralelo <= '1';
168             when ErrorTrama =>
169                 err_trama <= '1';
170             when ErrorParidad =>
171                 err_paridad <= '1';
172             when others =>
173                 null;
174         end case;
175     end process Salidas;
176
177     contar <= cuenta;
178
179 end behavioral;
```

Figura 12 – Salidas en de la Máquina de estados

En lo que se refiere a las salidas, ocurre lo siguiente. Inicialmente, colocamos todos nuestros componentes en 0. Cuando estamos en Arranque, habilitamos la cuenta del contador de un bit. Esta cuenta va a seguir habilitada cuando espera un bit, cuando está registrando y cuando comprueba la paridad. Cuando estamos en el estado de Registro habilitamos el registro para que se vayan registrando los bits. En Registro y cuando comprobamos la paridad, también tendremos activas las señales del contador de 9 bits y la que habilita el componente que comprueba la paridad. Por último, si estamos en cualquiera de los dos estados de error, se activará la correspondiente señal de error y cuando estemos en paralelo, se habilitará el registro paralelo.

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.

```
73      p_stim: process
74      begin
75          data <= '1';
76          paridad <= '0';
77          co_medio <= '0';
78
79          wait until reset_n = '1';
80
81          wait for 5 ns;
82
83          -- debería empezar el programa
84
85          data <= '0';
86
87          wait for 60 ns;
88
89          assert en_contador = '1'
90             report "El contador no se enciende"
91              severity failure;
92          wait for 40 ns;
93          co_medio <= '1';
94
95
96          wait for clk_per;
97          co_medio <= '0';
98          assert en_contador = '1'
99              report "El contador se para después al pasar a Espera1bit"
100             severity failure;
101
102         wait for 80 ns;
103
104         for n in 0 to 3 loop
105             data <= '1';
106
107             wait for 100 ns;
108
109             co_medio <= '1';
110
111             wait for clk_per;
112
113             co_medio <= '0';
114
115             wait for 80 ns;
116
117             data <= '0';
```

```
119             wait for 100 ns;
120
121             co_medio <= '1';
122
123             wait for clk_per;
124
125             co_medio <= '0';
126
127             wait for 80 ns;
128         end loop;
129
130         -- paridad
131         data <= '1';
132         wait for 100 ns;
133         co_medio <= '1';
134         wait for clk_per;
135         co_medio <= '0';
136         wait for 180 ns;
137         co_medio <= '1';
138         wait for clk_per;
139         co_medio <= '0';
140
141         wait for 100 ns;
142
143         assert false
144             report "Fin de la simulación"
145             severity failure;
146
147     end process p_stim;
148 end Unidad_control_arch;
```

Figura 13 – Testbench de la unidad lógica

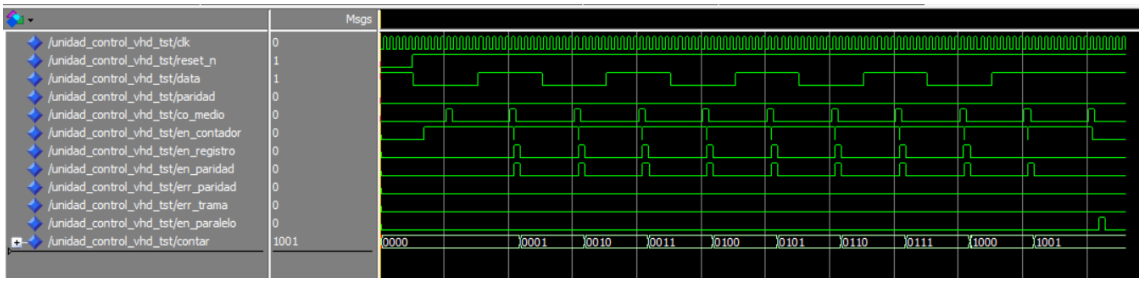


Figura 14 – Simulación de la Unidad Lógica

Circuito completo

Por último, debemos juntar todos los componentes en un componente más grande al que llamaremos práctica1. Lo único que vamos a hacer en este componente es instanciar el resto de los componentes que hemos nombrado en el apartado anterior.

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

entity Practica1 is
    port(
        clk : in std_logic;
        reset_n : in std_logic;
        data    : in std_logic;
        err_paridad : out std_logic;
        err_trama   : out std_logic;
        --sal_registro : out std_logic_vector(7 downto 0);
        --contar      : out std_logic_vector(3 downto 0);
        bits        : out std_logic_vector(7 downto 0)
        --para        : out std_logic
    );
end Practica1;

architecture structural of Practica1 is

    signal paridad, en_paridad, en_registro,co_medio,en_contador,back_to_0: std_logic;
    signal en_paralelo: std_logic;
    signal sal_regi : std_logic_vector(7 downto 0);

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

    component Registro
    port(
        clk     : in std_logic;
        enable  : in std_logic;
        reset_n : in std_logic;
        data    : in std_logic;
        bits    : out std_logic_vector(7 downto 0)
    );
    end component;
```

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

    component Unidad_Control
    port(
        clk : in std_logic;
        reset_n : in std_logic;
        data    : in std_logic;
        paridad : in std_logic;
        co_medio    : in std_logic;
        en_contador : out std_logic;
        en_registro : out std_logic;
        en_paridad  : out std_logic;
        err_trama : out std_logic;
        err_paridad : out std_logic;
        en_paralelo : out std_logic;
        contar      : out std_logic_vector(3 downto 0);
          back_to_0   : out std_logic
    );
    end component;

    component Registro_paralelo
    port(
        clk     : in std_logic;
      enable  : in std_logic;
        reset_n : in std_logic;
        entrada : in std_logic_vector(7 downto 0);
        salida  : out std_logic_vector(7 downto 0)
    );
    end component;
```

```
83  begin
84
85      i1_Comprobarparidad: Comprobar_paridad
86      port map(
87          data => data,
88          paridad => paridad,
89          clk     => clk,
90          reset_n => reset_n,
91          enable  => en_paridad,
92            back_to_0 => back_to_0
93      );
94
95      i1_Registro : Registro
96      port map(
97          clk     => clk,
98          enable  => en_registro,
99          reset_n => reset_n,
100         data    => data,
101         bits    => sal_regi
102     );
103
104     i1_Contador : Contador1bit
105     port map(
106         clk => clk,
107           back_to_0 => back_to_0,
108         reset_n => reset_n,
109         en => en_contador,
110         co_medio => co_medio
111     );
112
113     i1_UnidadControl: Unidad_Control
114     port map(
115         clk => clk,
116         reset_n => reset_n,
117         data    => data,
118         paridad => paridad,
119         co_medio => co_medio,
120         en_contador => en_contador,
121         en_registro => en_registro,
122         en_paridad  => en_paridad,
123         err_trama => err_trama,
124         err_paridad => err_paridad,
125         en_paralelo => en_paralelo,
126         contar => open,
127           back_to_0 => back_to_0
128     );
```

```
130     i1_RegistroPara: Registro_paralelo
131     port map(
132         clk     => clk,
133         enable  => en_paralelo,
134         reset_n => reset_n,
135         entrada => sal_regi,
136         salida  => bits
137     );
138
139     --sal_registro <= sal_regi;
140     --para <= en_paralelo;
141 end structural;
```

Figura 15 – 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.

```
62      p_stim : process
63
64      begin
65          data <= '1';
66
67          wait until reset_n = '1';
68
69          wait for 5 ns;
70
71          --bits de start
72
73          data <= '0';
74
75          wait for 26040 ns;
76
77          --bits de entrada
78
79          for n in 0 to 1 loop
80              data <= '0';
81
82              wait for 26040 ns;
83
84              data <= '0';
85
86              wait for 26040 ns;
87          end loop;
88
89             data <= '1';
90
91              wait for 26040 ns;
92
93              data <= '1';
94
95              wait for 26040 ns;
96
97              data <= '0';
98
99              wait for 26040 ns;
100
101             data <= '0';
102
103             wait for 26040 ns;
104
105         -- bit de paridad
106
107         data <= '1';
```

```
109         wait for 26040 ns;
110
111         --  bit de stop
112         data <= '1';
113
114         wait for 50000 ns;
115
116         assert bits="00110000"
117             report "El número obtenido no es el deseado"
118             severity failure;
119
120          data <= '0';
121
122         wait for 26040 ns;
123
124         --bits de entrada
125
126
127             data <= '1';
128
129             wait for 26040 ns;
130
131             data <= '0';
132
133             wait for 26040 ns;
134
135             data <= '0';
136
137             wait for 26040 ns;
138
139             data <= '0';
140
141             wait for 26040 ns;
142
143          data <= '1';
144
145             wait for 26040 ns;
146
147             data <= '1';
148
149             wait for 26040 ns;
150
151             data <= '0';
152
153             wait for 26040 ns;
```

```
155             data <= '0';
156
157             wait for 26040 ns;
158
159         -- bit de paridad
160
161         data <= '0';
162
163         wait for 26040 ns;
164
165         --  bit de stop
166         data <= '1';
167
168         wait for 50000 ns;
169
170         assert bits="00110001"
171             report "El número obtenido no es el deseado"
172             severity failure;
173
174          data <= '0';
175
176         wait for 26040 ns;
177
178         --bits de entrada
179
180         for n in 0 to 3 loop
181             data <= '1';
182
183             wait for 26040 ns;
184
185             data <= '0';
186
187             wait for 26040 ns;
188         end loop;
189
190         -- bit de paridad
191
192         data <= '1';
193
194         wait for 26040 ns;
195
196         --  bit de stop
197         data <= '1';
198
199         wait for 50000 ns;
200
```

```
201         assert bits="01010101"
202             report "El número obtenido no es el deseado"
203             severity failure;
204
205          data <= '0';
206
207         wait for 26040 ns;
208
209         --bits de entrada
210
211         for n in 0 to 2 loop
212             data <= '1';
213
214             wait for 26040 ns;
215
216             data <= '0';
217
218             wait for 26040 ns;
219         end loop;
220          data <= '1';
221
222             wait for 26040 ns;
223
224             data <= '1';
225
226             wait for 26040 ns;
227
228         -- bit de paridad
229
230         data <= '0';
231
232         wait for 26040 ns;
233
234         --  bit de stop
235         data <= '1';
236
237         wait for 50000 ns;
238
239         assert bits="11010101"
240             report "El número obtenido no es el deseado"
241             severity failure;
242
243
244         assert false
245         report "Fin de la simulación"
246             severity failure;
247     end process p_stim;
248 end Practica1_arch;
```

Figura 16 – Testbench de la práctica

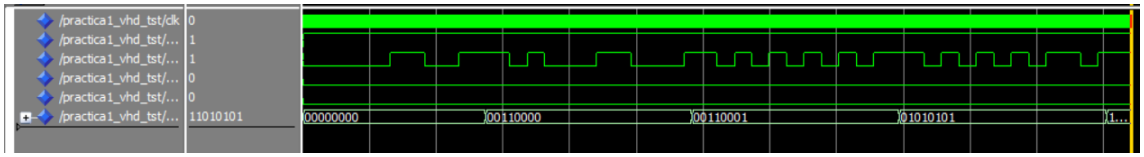


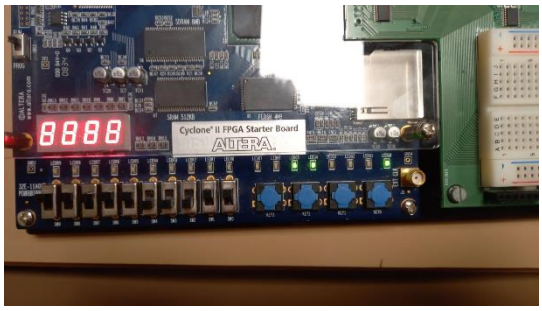
Figura 17 – Simulación de la práctica entera

Como vemos le hemos pasado 4 números seguidos con paridad impar para ver si funciona correctamente nuestro código y efectivamente ha devuelto lo que esperábamos.

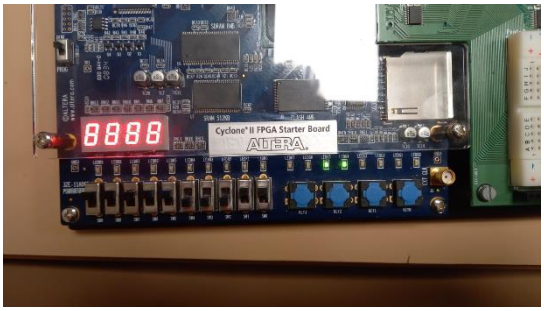
Implantación física

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:



1--> 49 -> 00110001 (binario) -> 31 (hexadecimal)

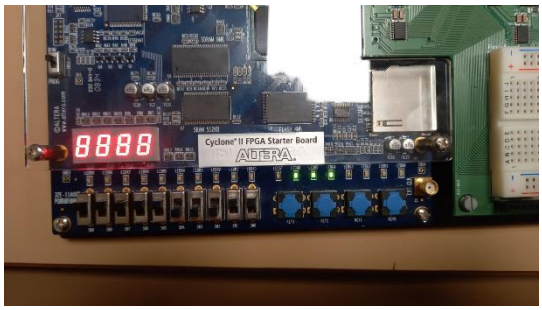


0--> 48 -> 00110000 (binario) -> 30 (hexadecimal)

c-->99 -> 01100011 (binario) -> 63 (hexadecimal)

m--> 109 -> 01101101 (binario) -> 6D (hexadecimal)

o--> 111 -> 01101111 (binario) -> 6F (hexadecimal)



p--> 112 -> 01110000 -> 70 (hexadecimal)

Conclusiones y análisis de los resultados

Al inicio de esta práctica teníamos tres objetivos principales. El primero era comprender el funcionamiento del transmisor y su unidad de control. Al principio, nos fue muy difícil comprender la práctica, qué era lo que teníamos que hacer, pero tras la explicación del profesor y mientras íbamos haciendo el código, fuimos comprendiendo mucho mejor cómo funcionaba un transmisor RS-232. Después de haber realizado la práctica y, principalmente, haber diseñado en VHDL el transmisor RS-232, hemos llegado a comprender cómo funciona completamente.

Nuestro segundo objetivo era diseñar, simular e implantar el transmisor RS-232 en FPGA. Este objetivo también lo hemos cumplido ya que como hemos visto hemos diseñado en VHDL un transmisor desde 0, componente a componente. Después de haber escrito el código, hemos conseguido simularla e implantarla en la placa FPGA con éxito.

Una vez realizada la práctica, debemos decir que realmente nos ha costado llegar a simular e implantar con éxito el componente. Hemos cometido bastantes errores, pero lo bueno es que gracias a la simulación hemos podido identificarlos para después corregirlos. Por ejemplo, la primera vez que lo implantamos en la placa, íbamos confiados porque la simulación nos había salido bien. Sin embargo, únicamente habíamos realizado la simulación del componente pasándole un número y no varios. Por lo tanto, al meterlo en la placa, el primer número se transmitía bien, pero si seguías transmitiendo, aparecían errores de paridad, de trama y se iluminaban bits que no se tenían que iluminar. Después de realizar una simulación más completa descubrimos que había que reinicializar nuestro comprobador de paridad y también nuestro contador de un bit. Una vez que lo corregimos, lo volvimos a implantar y en este caso ya funcionaba correctamente en la placa.