5. Diseño de una UART

El objetivo de esta práctica guiada es realizar un diseño de mediana complejidad. El diseño que se realizará establece una comunicación asíncrona entre la placa y un PC. El diseño de la UART servirá para repasar las máquinas de estados finitos y para aplicar conceptos vistos en las prácticas anteriores, como los divisores de frecuencia. Además, para comprobar la funcionalidad se hará uso de los bancos de pruebas (testbench) y la simulación en Modelsim. También se introducirá el diseño jerárquico.

En esta práctica se diseñará un transmisor/receptor serie asíncrono (*Universal Asynchronous Receiver-Transmitter*) que siga la norma RS232. Este módulo nos permitirá comunicar nuestra placa con el puerto serie del PC. Aunque en la norma se definen más señales, sólo son imprescindibles tres señales: la línea de transmisión de datos (`TxD`), la de recepción (`RxD`), y la línea de masa (`GND`), ver figura 5-1.

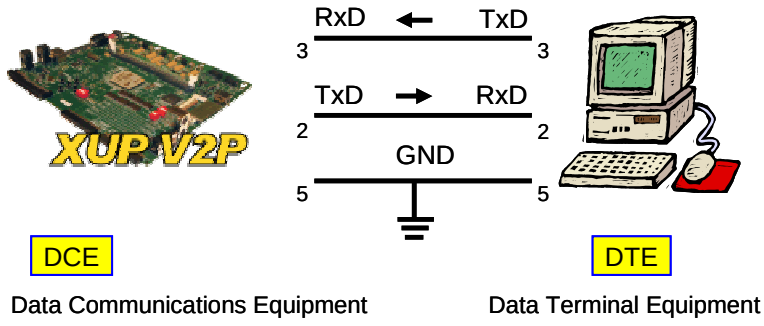

Figura 5-1: Esquema de la conexión RS232

En la norma se definen dos tipos de terminales: DTE y DCE, donde DTE es el equipo terminal de datos (el PC) y DCE es el de comunicación de datos, y que habitualmente es un MODEM y en nuestro caso es la placa. Para la conexión se usa un cable con conector db9 como el mostrado en la figura 5-2. Los pines de la FPGA que están conectados con el puerto RS232 se muestran en la figura 5-3. De ellos, sólo usaremos los dos primeros.

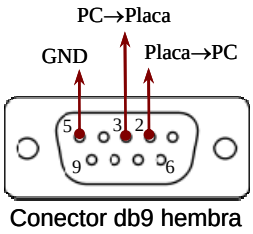

Figura 5-2: Conector db9 y pines utilizados

Conexión de la FPGA		
RS232_TX_DATA	Salida	AE7
RS232_RX_DATA	Entrada	AJ8
RS232_DSR_OUT	Salida	AD10
RS232_CTS_OUT	Salida	AE8
RS232_RTS_IN	Entrada	AK8

Figura 5-3: Conexiones de la FPGA con el puerto RS232 de la placa

Existen distintas velocidades de transmisión, que se definen en bits por segundo (bps) o baudios (921600, 460800, 230400, 115200, 57600, 38400, 19200, 9600, 4800, …). También se puede variar el número de bits del dato que se envía, así como el envío de un bit de paridad y el número de bits de fin.

La línea serie permanece a nivel alto ('1') mientras no se envían datos. Cuando el transmisor va a empezar la transmisión, lo hace enviando un bit de inicio que está a '0'. Posteriormente se envían consecutivamente los bits del dato empezando por el menos significativo. Después del último bit de dato se envía el bit de paridad en caso de que se haya especificado. Por último, se cierra la trama con uno o dos bits de fin con valor '1'. En la figura 5-4 se muestra el cronograma de un envío RS232 con 8 datos, un bit de paridad (par) y un bit de fin.

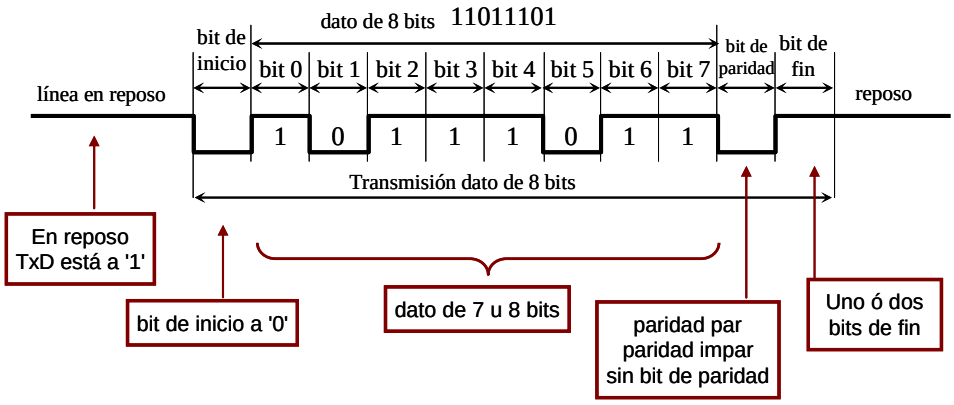

Figura 5-4: Trama de un envío en RS232 con 8 bits, bit de paridad y un bit de fin

La práctica consistirá en diseñar una UART compatible con RS232 que envíe 8 bits de datos, que pueda recibir y transmitir simultáneamente (*full-duplex*), sin bit de paridad y con un bit de fin.

Para ir paso a paso, primero se realizará sólo el transmisor. Posteriormente se realizará el receptor, para luego unirlos en un único diseño.

5.1 Diseño del transmisor

El transmisor de la UART tendrá el siguiente aspecto (figura 5-5):

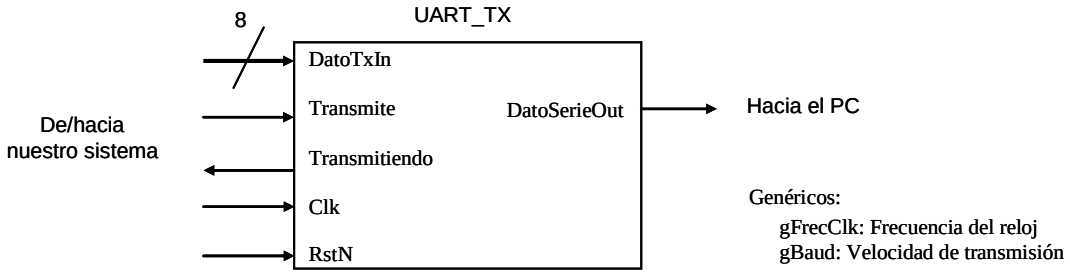

Figura 5-5: Entradas y salidas del transmisor

En la figura, los puertos de la izquierda son los que se relacionan con nuestro sistema (el que hayamos implementado en la FPGA) y el puerto de la derecha (`DatoSerieOut`) envía el dato serie al PC. Además habrá dos genéricos que harán que el circuito se pueda adaptar fácilmente a distintas frecuencias de reloj (y así poder implementar el diseño en otra placa) y que se puedan implementar otras velocidades de transmisión. Las especificaciones de los puertos se muestran en la tabla 5-1:

Señal	bits	I/O	Descripción
`RstN`	1	I	Señal de reset asíncrono
`Clk`	1	I	Señal de reloj de la placa, en principio de 100MHz, pero configurable por `gFrecClk`
`Transmite`	1	I	Señal del sistema que ordena al módulo la transmisión del dato que se encuentra en `DatoTxIn`. La orden será de un único ciclo de reloj
`DatoTxIn`	8	I	Dato que se quiere enviar, se proporciona de manera simultánea a cuando `Transmite='1'`.
`Transmitiendo`	1	O	Indica al sistema que el módulo está transmitiendo y por tanto no podrá atender a ninguna nueva orden de transmisión. Ignorará a `Transmite` cuando `Transmitiendo` esté a uno
`DatoSerieOut`	1	O	Trama que se envía al PC, sigue el formato RS232

Tabla 5-1: Características de los puertos del transmisor

La declaración de esta entidad se muestra a continuación:

```
entity UART_TX is
  generic (
    gFrecClk      : integer := 100000000;   --100MHz
    gBaud         : integer := 9600         --9600bps
  );
  port(
    RstN          : in std_logic;
    Clk           : in std_logic;
    Transmite     : in std_logic;
    DatoTxIn      : in std_logic_vector (7 downto 0);
    Transmitiendo : out std_logic;
    DatoSerieOut  : out std_logic
  );
end UART_TX;
```

Código 5-1: Entidad del transmisor

Con estas especificaciones tenemos información suficiente para hacer el transmisor. Existen muchas posibilidades de implementación, a continuación se propone una de ellas. Cada uno de los bloques mostrados en la figura 5-6 se puede implementar en uno o varios procesos. De todos modos, la figura 5-6 es una **versión inicial** que se hace antes de realizar el circuito. Mientras se diseña se irá variando esta versión ya que aparecen problemas y cuestiones que no hemos tenido en cuenta.

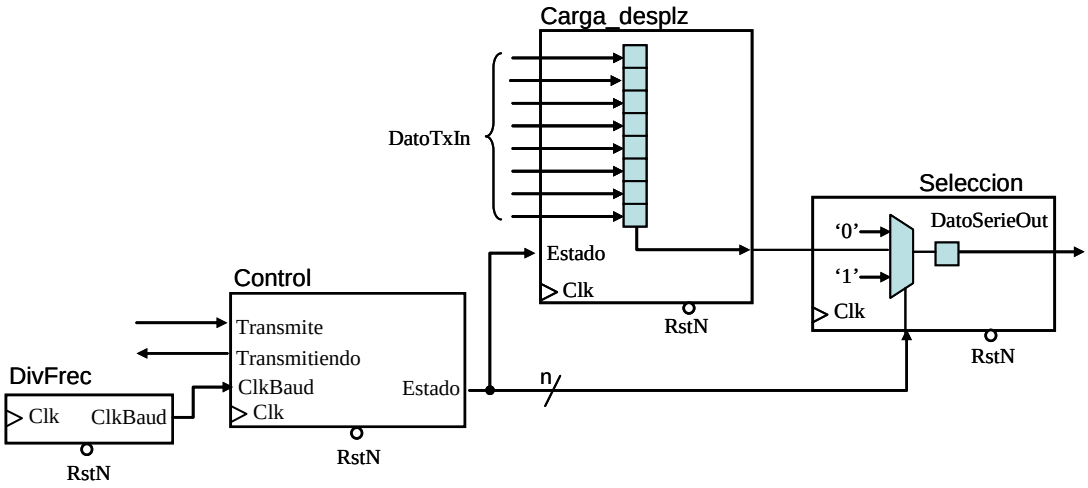

Figura 5-6: Diagrama de bloques preliminar del transmisor

El funcionamiento de los bloques de la figura 5-6 se ha visto en las prácticas anteriores:

- `DivFrec` es un divisor de frecuencia, similar al `Nuevo Contador` de la figura 3-1 (proceso `P_Conta` del código 3-3)
- `Control` es la máquina de estados finitos (práctica anterior).
- `Carga_desplz` es un registro de carga paralelo y salida serie
- `Seleccion` es un multiplexor que selecciona la señal de salida según el estado. Este multiplexor termina en un biestable ya que su salida (`DatoSerieOut`) es la salida del circuito, y por seguridad, conviene que estén registrada.

5.1.1 Divisor de frecuencia

Queremos realizar un divisor de frecuencia genérico, esto es, que si queremos que el divisor de frecuencia proporcione una frecuencia de salida diferente, no tengamos que rehacer el diseño, sino que sólo tengamos que modificar el valor de una constante. Este tipo de constantes se llaman "genéricos".

En nuestro diseño usaremos dos genéricos:

- `gFrecClk:` indica la frecuencia de reloj de la placa. Usando este genérico podremos implementar la UART en otra placa que tenga un reloj diferente. Por ejemplo, la podríamos usar en la placa *Pegasus* que se usó en la asignatura Electrónica Digital II, y que su reloj es de 50MHz (el reloj de nuestra placa es de 100MHZ)
- `gBaud:` indica la velocidad de transmisión de la UART. Si no usásemos un genérico tendríamos que diseñar una UART diferente para cada velocidad de transmisión. Diseñando con genéricos usaremos la misma UART para cualquier velocidad de transmisión, solamente necesitamos modificar el genérico y volver a implementar el diseño en la placa.

El diseño genérico es más complicado al principio y hace que se tarde más en diseñar. Sin embargo, a la larga hace ahorrar mucho tiempo porque no hace falta rediseñar. Además el diseño genérico suele hacer ser más seguro, ya que si tenemos que rediseñar un circuito podemos cometer errores.

En nuestro caso, el diseño genérico también nos puede hacer ahorrar tiempo de simulación, ya que para simular se pueden poner frecuencias mayores, y así no tener que esperar los largos tiempos de simulación.

Ya hemos visto que los genéricos se declaran en la entidad. Así, los genéricos de nuestra entidad `UART_TX` (figura 5-5) se declararan como se ve en el código 5-1. Como se ve en el código, los genéricos tienen un valor por defecto.

Antes de pensar en el diseño genérico, veamos cómo haríamos el diseño **con valores concretos**.

A partir del reloj de la placa de 100MHz (`Clk`) queremos proporcionar una señal con frecuencia de 9600Hz (`clkBaud`), que es la velocidad de transmisión escogida. Este reloj tendrá por tanto un periodo de 104,167 μs, y estará un sólo ciclo de reloj a uno el resto del tiempo a cero. La figura 5-7 representa los cronogramas de las señales.

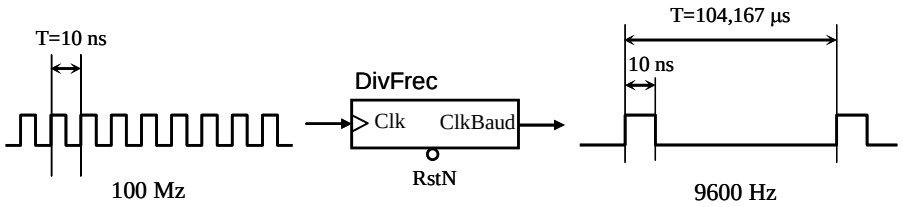

Figura 5-7: Divisor de frecuencia

Para diseñar el divisor de frecuencia se divide la frecuencia de entrada entre la frecuencia de salida (100MHz/9,6kHz = 10416,67→10417) y el número resultante nos dará el contador que se necesitará para obtener la frecuencia de salida. Haciéndolo de manera inversa, 10417 cuentas de 10ns (`Clk`) nos da un periodo de 104,17µs, que es una buena aproximación de la frecuencia que queremos: queremos 9600Hz y obtenemos 9599,69Hz.

Para la creación de este módulo habría que tener en cuenta lo explicado referente a los rangos numéricos del apartado 2.3, y en vez de crear un contador con rango de 0 a 10416 que no tiene ninguna correspondencia física. Lo haríamos con el rango que delimite el ancho del bus. Como $\log_2(10416) = 13{,}35$; necesitaríamos 14 biestables para el contador.

Aunque en este ejemplo no es tan importante contar un ciclo más o menos, ya que 10ns es despreciable frente a 104,167µs, en general hay que ser cuidadosos para hacer un contador que cuente el número de ciclos que queremos. Conviene dibujar el cronograma del circuito (figura 5-8) y posteriormente diseñarlo.

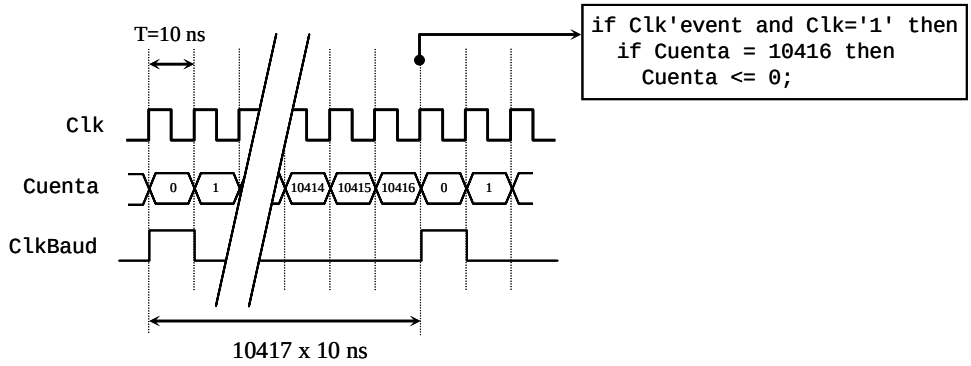

Figura 5-8: Cronograma del contador del divisor de frecuencia

A partir del cronograma podemos ver que cuando `Cuenta` valga 10416 es cuando debemos inicializar el contador. Como las transiciones ocurren durante el flanco activo de reloj (`Clk'event and Clk='1'`) podemos confundirnos con los valores que se evalúan para cada señal en el evento de reloj, ¿qué valor hemos de tomar, el anterior o el posterior a la transición?

En la figura 5-9 se muestra el cronograma ampliado, en él podemos apreciar que es a partir del flanco del reloj que las señales registradas (`Cuenta` y `ClkBaud`) cambian de valor, por tanto, se evalúan los valores de las señales anteriores al flanco de reloj

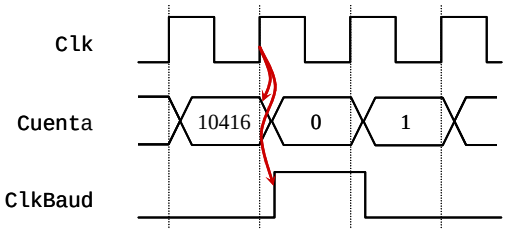

Figura 5-9: Detalle del cronograma en la transición a cero

Ahora que ya sabemos cómo hacer el diseño con valores concretos, veamos cómo se haría **genérico**:

En el caso de valores concretos, para calcular el fin de cuenta dividimos la frecuencia entre la velocidad de transmisión (100MHz/9,6kHz = 10416,67→10417). Por lo tanto, en vez de hacer que el fin de cuenta sea hasta 10417, usaremos una constante (`cFinCuenta`) que tenga el valor resultante de la división (código 5-2)

```
constant cFinCuenta : natural := gFrecClk/gBaud-1;
signal   Cuenta     : natural range 0 to cFinCuenta;
```

Código 5-2: Declaración de la constante de fin de cuenta y la señal que cuenta

Por otro lado, la señal `Cuenta`, tiene un rango de 0 hasta `cFinCuenta` para que tenga un tamaño acorde con la cuenta que tiene que realizar. Es poco probable que este tamaño sea potencia de dos, pero no nos importa mucho, pues nosotros vamos a controlar la cuenta y **nunca dejaremos que se desborde**.

Es importante resaltar que normalmente los sintetizadores no admiten operaciones de división (ni tampoco multiplicación). Sin embargo, cuando hacemos operaciones con operandos constantes, las operaciones no se sintetizan, sino que se obtienen los resultados numéricos en un proceso previo a la síntesis. Este proceso se llama elaboración. Por tanto, no hay ningún problema en usar este tipo de operaciones entre valores constantes.

Tal es el caso que incluso podemos usar funciones complejas como la de redondeo mostrada en el código 5-3, pues no se van a sintetizar. Esta función no es necesaria para nuestro caso, ya que un ciclo de reloj de más o de menos es poco comparado con las frecuencias de envío, pero se incluye para introducir la existencia de funciones en VHDL.

```
function div_redondea (A,B: integer)
  return integer is
    variable division : integer;
    variable resto    : integer;
begin
  division := A/B;
  resto    := A rem B;
  if (resto > (B/2)) then
    division := division + 1;
  end if;
  return (division);
end;

constant FinCuenta : integer
    := div_redondea(FrecClk,FrecBaud);
```

Código 5-3: Función para el redondeo

5.1.2 Circuito de control

El circuito de control es el encargado dirigir al resto de bloques. Si el circuito no se encuentra enviando ningún dato, cuando se recibe la indicación de transmitir (`Transmite='1'`) el circuito empieza una secuencia de envío, esta secuencia de envío determinará la máquina de estados. La secuencia de estados es la siguiente (ver figura 5-10):

- `eInit`: este estado es el inicial, el sistema está en reposo esperando la orden de transmitir, cuando la señal `Transmite='1'` se pasará a enviar el bit de inicio: `eBitInit`. En este momento se dará la orden de cargar el dato (`DatoTxIn`) en el registro de desplazamiento. En este momento también debemos sincronizar el contador del divisor de frecuencia, para ello hay dos opciones, o inicializar el contador o hacer que en el estado inicial no cuente, y al cambiar de estado se habilite el contador. Haremos esta última opción (esto no lo habíamos contemplado en la figura 5-6).
- `eBitInit`: en este estado se está enviando el bit de inicio. Se saldrá de este estado al recibir la señal `ClkBaud`, que nos dirá que debemos de pasar a enviar los bits de dato. Por tanto, el estado siguiente será `eBitsDato`.

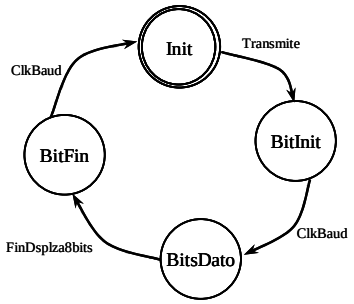

Figura 5-10: Diagrama de transición de estados

- `eBitsDato`: este estado se encarga de enviar los 8 bits de dato. Una alternativa a esta opción sería poner un estado para cada bit. Sin embargo, usaremos un estado para los 8 bits y, mediante un contador, se llevará la cuenta del número de bits que se han enviado. Cuando se hayan enviado los 8 bits (`FinDsplza8bits`) cambiaremos de estado, y pasaremos a enviar el bit de fin: estado `eBitFin` (si quisiésemos implementar el bit de paridad, iría ahora)
- `eBitFin`: este estado envía el bit de fin, al llegar un `ClkBaud` pasaríamos al estado inicial.

El diagrama de la figura 5-10 es claro e indica las señales que hacen cambiar de estado. Sin embargo, también es útil conocer el valor del las salidas, y qué sucede cuando hay varias entradas activas que hacen cambiar de estado. En nuestro caso, por ser una máquina de estados que no tiene bifurcaciones no es necesario. Pero normalmente no es así, y se ponen los valores de todas las entradas incluyendo también los valores de las salidas. En la figura 5-11 se muestra este diagrama de transición de estados. En cada flecha se indican los valores que provocan esa transición y tras una barra inclinada se indican los valores de las salidas. Las X indican que da igual el valor de la entrada correspondiente. Las salidas no pueden ser X.

En nuestro caso vemos que es una máquina de Mealy porque las salidas no dependen sólo del estado, sino también de las entradas. Nota también el valor de `Dsplza` cuando se está en `BitsDato`.

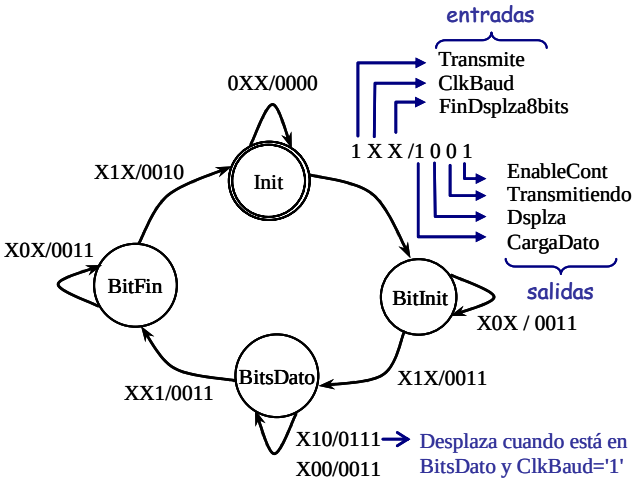

Figura 5-11: Diagrama de transición de estados incluyendo salidas

El diagrama de la figura 5-11 es sólo una propuesta, existen otras posibilidades de implementar el circuito. A partir de la definición de estados y señales que hemos hecho, surgen modificaciones al diagrama de bloques de la figura 5-6.

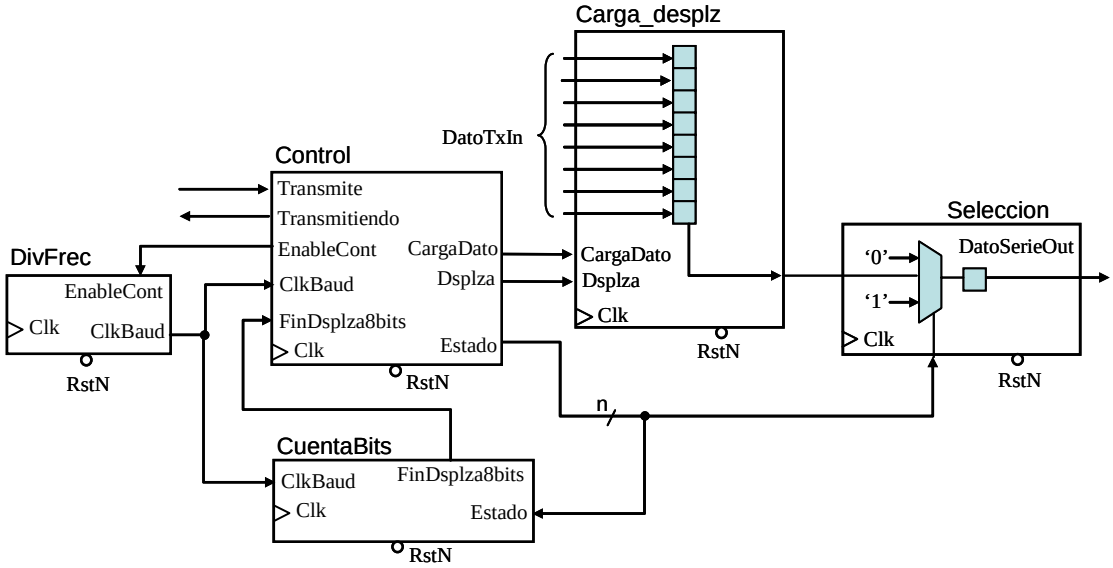

Figura 5-12: Diagrama de bloques del transmisor

En la figura 5-12 aparecen las nuevas señales que se definieron en la figura 5-11 y un nuevo bloque `CuentaBits` que se encarga de llevar la cuenta de los bits que se han transmitido. Este bloque será quien indique al bloque de control que se han enviado todos los bits.

A continuación se muestra el código de la máquina de estados, a la izquierda el proceso que establece la transición de los estados (código 5-4) y a la derecha, el proceso que obtiene las salidas (código 5-5).

```
P_Control_FSM: Process (RstN, Clk)
begin
  if RstN = '0' then
    Estado <= eInit;
  elsif Clk'event and Clk='1' then
    case Estado is
      when eInit =>
        if Transmite = '1' then
          Estado <= eBitInit;
        end if;
      when eBitInit =>
        if ClkBaud = '1' then
          Estado <= eBitsDato;
        end if;
      when eBitsDato =>
        if FinDsplza8bits = '1' then
          Estado <= eBitFin;
        end if;
      when eBitFin =>
        if ClkBaud = '1' then
          Estado <= eInit;
        end if;
    end case;
  end if;
end process;
```

Código 5-4: Proceso de transición de estados

```
POut: Process (Estado, Transmite, ClkBaud)
begin
  Dsplza        <= '0';
  CargaDato     <= '0';
  EnableCont    <= '1';
  Transmitiendo <= '1';
  case Estado is
    when eInit =>
      EnableCont    <= '0';
      Transmitiendo <= '0';
      if Transmite = '1' then
        CargaDato  <= '1';
        EnableCont <= '1';
      end if;
    when eBitInit =>  null;
    when eBitsDato =>
      if ClkBaud = '1' then
        Dsplza <= '1';
      end if;
    when eBitFin =>
      if ClkBaud = '1' then
        EnableCont <= '0';
      end if;
  end case;
end process;
```

Código 5-5: Proceso de las salidas

Por último, se muestran también el código VHDL del proceso que controla el número de bits que se envían con la sentencia que asigna el valor a la señal `FinDsplza8bits` (código 5-6). Esta señal establece el fin del estado `eBitDato`. A la derecha se incluye el proceso del divisor de frecuencia controlado por la señal `EnableCont` (código 5-7).

```
P_CuentaBits: Process (RstN, Clk)
begin
  if RstN = '0' then
    CuentaBits     <= 0;
  elsif Clk'event and Clk='1' then
    if Estado = eBitsDato then
      if ClkBaud = '1' then
        if CuentaBits = 7 then
          CuentaBits <= 0;
        else
          CuentaBits <= CuentaBits + 1;
        end if;
      end if;
    else
      CuentaBits <= 0;
    end if;
  end if;
end process;
FinDsplza8bits <= '1' when CuentaBits=7 and
                  ClkBaud = '1' else '0';
```

Código 5-6: Proceso del control del número de bits

```
P_DivFrec: Process (RstN, Clk)
begin
  if RstN = '0' then
    Cuenta   <=  0;
    ClkBaud  <= '0';
  elsif Clk'event and Clk='1' then
    if EnableCont = '1' then
      if Cuenta = cFinCuenta then
        Cuenta  <= 0;
        ClkBaud <= '1';
      else
        Cuenta  <= Cuenta + 1;
        ClkBaud <= '0';
      end if;
    else  --  inhabilitado
      Cuenta  <= 0;
      ClkBaud <= '0';
    end if;
  end if;
end process;
```

Código 5-7: Proceso del divisor de frecuencia

Este circuito se ha realizado de manera modular, separando cada funcionalidad en distintos procesos. La funcionalidad de los códigos 5-4, 5-5 y 5-6 se podían haber implementado en un único proceso.

Para terminar este circuito sólo quedaría realizar el proceso del registro de desplazamiento y el multiplexor de la salida. Una vez que lo hayas terminado, pasaremos a la simulación del circuito.

5.2 Simulación del transmisor

El transmisor que hemos realizado tiene cierta complejidad, y a diferencia de los circuitos que hemos realizado hasta ahora, no es fácil de comprobar si lo hemos diseñado bien. En los anteriores circuitos disponíamos de los LED para ver el comportamiento del circuito, ahora no tenemos los LED, y por tanto es casi imprescindible simular el circuito antes de implementarlo en la placa. En el apartado 2.4 vimos los pasos para simular un circuito. Ahora vamos a seguir los mismos pasos, pero realizaremos un banco de pruebas de mayor complejidad.

La realización de un buen banco de prueba es una tarea muy importante en el diseño. Muchas veces el banco de pruebas y el diseño lo realizan personas diferentes para evitar que el diseñador realice el banco de pruebas en base a lo que ha diseñado. Un buen banco de pruebas debe comprobar que el diseño hace lo que debe hacer y no hace lo que no debe hacer.

A veces la complejidad del banco de pruebas es muy grande y se crean modelos de componentes para la simulación. La descripción de estos modelos puede ser muy diferente a los modelos que se sintetizan, ya que para simulación se acepta todo el conjunto del VHDL, mientras que para síntesis sólo se acepta un conjunto restringido.

Para empezar el banco de pruebas, creamos una nueva fuente de tipo **VHDL TestBench** y la llamamos `tb_uart_tx.vhd`. La entidad del banco de pruebas tendrá los mismos genéricos que la del transmisor (código 5-8).

```
entity TB_UART_TX IS
  generic (
    gFrecClk      : integer := 100000000;   --100MHz
    gBaud         : integer := 9600         --9600bps
  );
end TB_UART_TX;
```

Código 5-8: Entidad del banco de pruebas

Como estamos haciendo un diseño genérico, y la frecuencia del reloj es genérica, sería conveniente crear el proceso del reloj con genéricos. Para ello, definimos unas constantes de tiempo que indican el periodo del reloj y el periodo de los bits de envío (y también sus mitades).

```
-- Lo ponemos en nanosegundos (10**9) para que sea un numero entero
  constant cPeriodoClk       : time := ((10**9)/gFrecClk) * 1 ns;
  constant cMitadPeriodoClk  : time := cPeriodoClk/2;
  constant cFinCuenta        : natural := gFrecClk/gBaud -1;
  constant cPeriodoBaud      : time := cFinCuenta * cPeriodoClk;
  constant cMitadPeriodoBaud : time := cPeriodoBaud/2;
```

Código 5-9: Constantes temporales para la simulación

Usando las constantes del periodo de reloj (código 5-9) creamos el proceso que genera la señal de reloj:

```
PClk:Process
begin
  Clk <= '1';
  wait for cMitadPeriodoClk;
  Clk <= '0';
  wait for cMitadPeriodoClk;
end process;
```

Código 5-10: Proceso que genera la señal de reloj con constantes temporales

Ahora, de manera similar al proceso del código 2-5, creamos el proceso que simula la señal de reset.

Antes de crear el proceso de los estímulos definiremos un vector constante donde se pondrán los datos a enviar. Para crear este vector de `std_logic_vector` tendremos que definir un tipo `(vector_datos)` que es un vector de vectores de tipo `std_logic_vector(7 downto 0)`. El número de elementos del tipo `vector_datos` puede ser cualquiera (`natural`) y se especificará para cada señal de ese tipo. En nuestro caso, la constante `DatosTest` tiene 4 elementos, que por ser constante y por tanto, tener un valor asignado, no hace falta especificar que el rango es `(0 to 3)`.

```
  type vector_datos is array (natural range <>) of std_logic_vector(7 downto 0);
  constant DatosTest : vector_datos := ("10001101","01010101","11001010","00101101");
```

A continuación se muestra un proceso que genera los estímulos para las señales `Transmite` y `DatoTxIn`. En ocasiones el proceso se sincroniza con la señal de reloj (`Clk`), e interactúa con el transmisor, observando cuando está libre (`Transmitiendo='0'`). En este proceso se realizan tres envíos, aunque también se realizan envíos a los que el transmisor no debería responder pues aún se encuentra enviando el último dato. Esto es importante a la hora de crear los bancos de pruebas, debemos probar que hace lo que queremos, pero también, que cuando hay un comportamiento imprevisto del resto de componentes, sigue haciendo lo que debe

```
Estimulos : Process
begin
  Transmite <= '0';
  DatoTxIn <= (others => '0');       -- Es lo mismo que todo a cero: "0000000"
  wait until RstN'event and RstN='1';  -- Esperamos a que el reset se restablezca a '1'
  wait until Clk'event and Clk='1';
  ---------------- PRIMER ENVIO (envio 0) ---------------------------------
  Transmite <= '1';
  DatoTxIn <= DatosTest(0);
  wait until Clk'event and Clk='1';  -- Bajamos transmite en el siguiente flanco
  Transmite <= '0';
  DatoTxIn <= (others => '0');
  wait until Transmitiendo = '0';    -- Esperamos a que deje de transmitir
  ---------------- SEGUNDO ENVIO (envio 1) ---------------------------------
  Transmite <= '1';
```

```
  DatoTxIn <= DatosTest(1);
  wait until Clk'event and Clk='1';
  Transmite <= '1';                  -- A ver que pasa si dejamos transmite=1
  DatoTxIn <= (others => '0');       -- y cambiamos el dato
  wait for 100 ns;
  Transmite <= '0';
  DatoTxIn <= (others => '0');
  wait until Transmitiendo = '0';    -- Esperamos a que deje de transmitir
  DatoTxIn <= NOT DatosTest(1);      -- Ponemos otro dato, pero transmite=0
  Transmite <= '0';
  wait until Clk'event and Clk='1';   -- Esperamos a que deje de transmitir
  ---------------- TERCER ENVIO (envio 2) ----------------------------------
  Transmite <= '1';
  DatoTxIn <= DatosTest(2);
  wait until Clk'event and Clk='1';  -- Esperamos a que deje de transmitir
  Transmite <= '0';
  DatoTxIn <= NOT DatosTest(2);      -- Ponemos otro numero
  wait for 100 ns;
  Transmite <= '1';                  -- Ponemos otro transmite
  DatoTxIn <= NOT DatosTest(2);      -- que no lo deberia transmitir
  wait for 100 ns;
  Transmite <= '0';
  DatoTxIn <= (others => '0');   -- Es lo mismo que todo a cero
  wait for 30 ns;
  ---------------- CUARTO ENVIO (envio 3) ----------------------------------
  wait until Transmitiendo = '0';    -- Esperamos a que deje de transmitir
  wait for cMitadPeriodoBaud;
  wait for cMitadPeriodoBaud/2;      -- Esperamos 3/4 partes de un periodo Baud
  Transmite <= '1';
  DatoTxIn <= DatosTest(3);
  wait until Clk'event and Clk='1';  -- Esperamos a que deje de transmitir
  Transmite <= '0';
  DatoTxIn <= (others => '0');
  FinEnvio <= '1';               -- Indicamos que hemos acabado de enviar
  wait;                          -- Fin del proceso, no vuelve al principio
end process;
```

Código 5-11: Proceso generador de estímulos

Una vez que has generado estos procesos, guarda el fichero y simúlalo, comprobando que los datos se envían correctamente. Corrige el código del transmisor en caso de que encuentres algún error.

Como has realizado el diseño con genéricos, el simulador toma los valores de los genéricos de la entidad de más alto nivel. Si se quiere simular con valores diferentes se puede indicar en la ventana que aparece al pinchar en **Simulate→ Start Simulation**, seleccionando la pestaña **Others**. Se pincha en **Add...** . Ver la figura 5-14.

Posteriormente aparecerá una nueva ventana para que entremos el genérico y su nuevo valor. Ponemos el nuevo valor de `FrecBaud` a 115200, indicando que sobrescriba los valores que tenga ya indicados (*Override Instance-specific Values*). Figura 5-13.

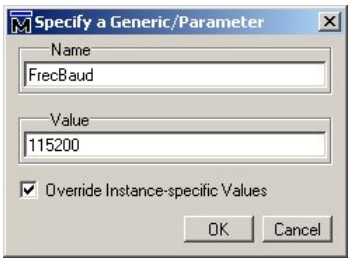

Figura 5-13: Especificación del valor de los genéricos

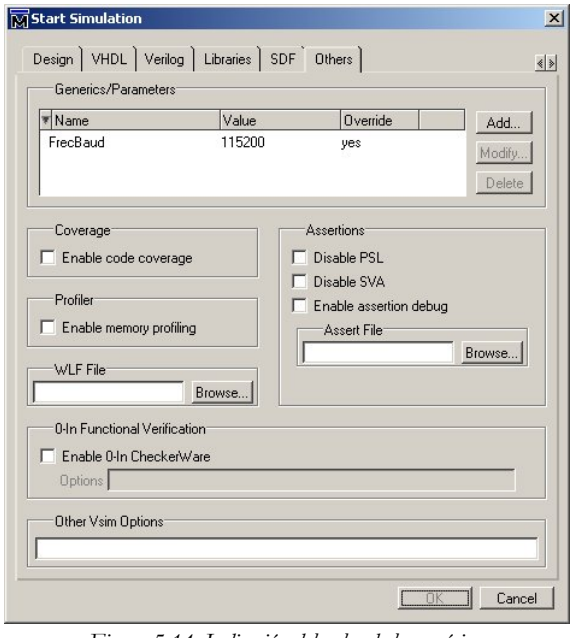

Figura 5-14: Indicación del valor de los genéricos

Le damos a aceptar y empezamos la simulación como ya sabemos hacer.

5.2.1 Modelo del receptor

Acabamos de simular el circuito y es posible que nos haya servido para descubrir algún error. Sin embargo, ¿podríamos asegurar que el circuito funcionará bien en todas las situaciones posibles? Esto es una pregunta difícil de responder, aunque cuanto más sofisticado sea el banco de pruebas, más probabilidades tendremos de obtener un circuito sin errores.

Al simular el circuito anterior tuvimos que comprobar su funcionamiento de manera visual a través de las formas de onda resultantes. Esta manera de comprobar el circuito no es muy segura, y podemos equivocarnos, y además requiere que sepamos cómo debe funcionar el circuito. Como alternativa podemos crear un modelo del receptor RS232 para simulación. Este modelo comprobará si los bits recibidos se corresponden con los enviados. Es decir, el propio banco de pruebas se auto-comprobará.

El modelo está descrito en el siguiente código:

```
Receptor : Process       -- Este proceso se encarga de recibir y comprobar
  variable  numbit : natural; -- declaracion de variable
BEGIN
  -- Esperamos al comienzo del envio
  wait until DatoSerieOut'event and DatoSerieOut = '0';
  -- empieza a enviar, nos situamos a la mitad del ciclo para evitar errores.
  wait for cMitadPeriodoBaud;
  -- Ahora debemos estar a la mitad del bit de inicio
  -- Por tanto DatoSerieOut debe seguir siendo = 0 lo comprobamos con ASSERT
  assert DatoSerieOut = '0'  -- Si no se cumple da el aviso
    report "Fallo en el bit de inicio"
    severity ERROR;  -- niveles de severidad: NOTE,WARNING,ERROR,FAILURE
  numbit := 0;
  for i in 0 to 7 loop
    wait for cPeriodoBaud;
    assert DatoSerieOut = DatosTest(numenvio)(numbit)
      report "Fallo en un bit de datos" severity ERROR;
    numbit := numbit+1;
  end loop;
  wait for cPeriodoBaud; -- bit de fin
  assert DatoSerieOut = '1'
    report "Fallo en el bit de fin" severity ERROR;
  numenvio <= numenvio + 1;
  if FinEnvio = '1' then  -- si ya no hay mas envios paramos la simulacion
    wait for cPeriodoBaud;
    FinSimulacion <= '1';
    wait;
  end if;
end process;
```

Código 5-12: Modelo del receptor

El modelo del receptor se basa en los tiempos de envío:

- `wait until DatoSerieOut'event and DatoSerieOut = '0'`: Con esta sentencia se espera a que baje `DatoSerieOut`, esto significará que comienza el envío del transmisor, y está enviando el bit de inicio.
- `wait for cMitadPeriodoBaud`: Espera la mitad del periodo de envío del bit para ponerse en la mitad del envío
- `wait for cPeriodoBaud`: Va de bit en bit, en medio del envío.

En la figura 5-15 se muestra el cronograma de la seña `DatoSerieOut`, y cómo las sentencias `wait` se sitúan adecuadamente para recibir cada uno de los bits.

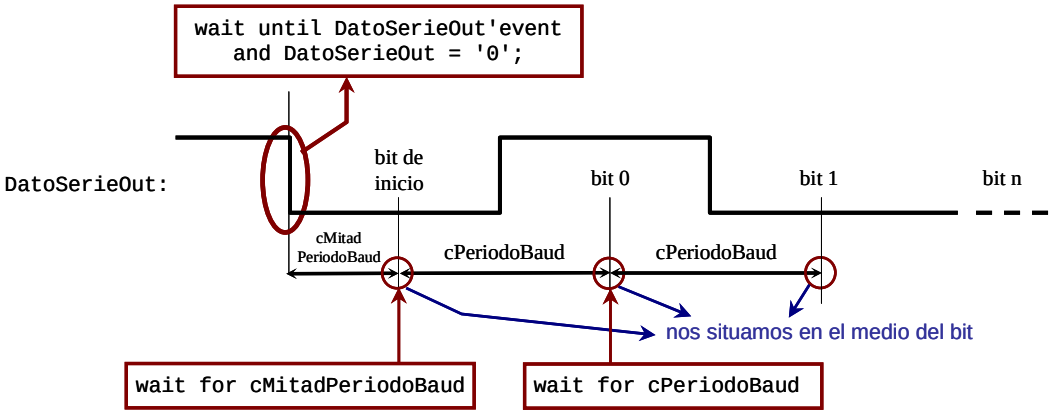

Figura 5-15: Cronograma de la señal `DatoSerieOut`

Hay varias cosas interesantes del código 5-12:

- Se usa la variable `numbit`. En VHDL se pueden usar variables, que a diferencia de las señales, su valor se actualiza en el instante en que se asigna. Nota que la sentencia de asignación de las variables es con `":="` en vez de `"<="`. El uso de variables se explicará más adelante.
- El modelo del receptor no es sintetizable, es un modelo para simulación. Y por eso no refleja el hardware que le corresponde.

- La capacidad de autocomprobación del modelo de receptor reside en el uso de las sentencia `assert`. Esta sentencia permite la comprobación de una condición y en caso de que **no** se cumpla se comunica un mensaje (`report`), la gravedad del incumplimiento de la condición se determina por la severidad asignada (`severity`). Esta severidad puede ser `NOTE`, `WARNING`, `ERROR` y `FAILURE`; siendo `NOTE` la menos grave y `FAILURE` la de mayor gravedad. Los simuladores comunican estos mensajes y pueden configurarse para que detengan la simulación en caso de que superen una severidad determinada.

Para no preocuparnos del cuánto tiempo debemos de correr la simulación, podemos hacer que la simulación se auto-detenga cuando se haya terminado. Para eso se han creado las señales `FinEnvio` (código 5-11) y `FinSimulacion` (código 5-12). La señal `FinEnvio` indica cuándo se le da al transmisor la última orden de envío. La señal `FinSimulacion` indica que el receptor acaba de terminar de recibir el último envío. Por tanto, cuando `FinSimulacion` sea '1' ya no hace falta seguir simulando. Para parar la simulación los procesos deben terminar en un `wait`. Esto se hace en el modelo del receptor (código 5-12) cuando `FinEnvio='1'`. Y también se debe poner un `wait` en el proceso del reloj:

```
PClk:Process
begin
  Clk <= '1';
  wait for 5 ns; -- 100 MHz: 5ns + 5ns
  Clk <= '0';
  wait for 5 ns;
  if FinSimulacion = '1' then
    wait;        -- se para la generacion de reloj
  end if;
end process;
```

Código 5-13: Proceso del reloj con parada

Ahora, en vez de simular por tiempo determinado, simularemos todo mediante el comando **Simulate→Run→Run -All.** (también hay un icono para este comando). Y si lo hemos hecho bien, se deberá parar automáticamente.

Al terminar la simulación podemos ver las formas de onda y si hemos tenido algún aviso a causa de los `ASSERT`. En la siguiente figura se muestran varios mensajes de aviso de este tipo. Pinchando en el mensaje, el cursor de la ventana de las formas de ondas (**Wave**) se sitúa en el tiempo en que ha ocurrido esa condición (figura 5-16).

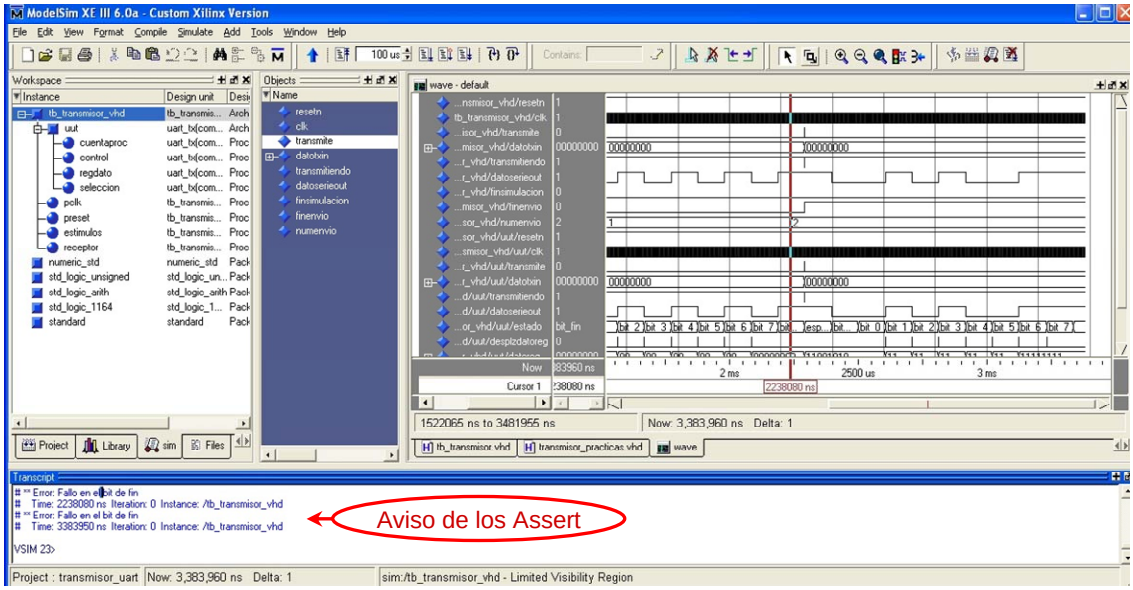

Figura 5-16: Indicación de los `Assert`

Comprueba que no haya indicaciones de los `Assert`, en caso de que haya comprueba el diseño del transmisor y realiza las modificaciones necesarias. En este caso, revisa también por qué no te diste cuenta del error durante la primera simulación. En el siguiente apartado implementaremos el transmisor en la FPGA.

5.3 Implementación en la FPGA: diseño estructural

Que el circuito sea funcionalmente correcto no implica que ya no vayamos a tener ningún problema en su implementación. Puede ocurrir que tengamos restricciones temporales, de área o consumo que nos obliguen a realizar modificaciones. Además, a veces ocurre que los resultados de simulación no se corresponden con los de síntesis, ya que, entre otros motivos, por ser el conjunto de síntesis más reducido que el de simulación, la síntesis no siempre traduce fielmente el modelo VHDL.

Ahora tenemos nuestro transmisor que en teoría funciona bien. Sin embargo sería inútil implementarlo en la FPGA sin nada más, pues el transmisor necesita una orden para transmitir. Así que tenemos que hacer un circuito sencillo que nos de algo que transmitir. Para ello emplearemos los pulsadores, pues ya sabemos que funcionan, y haremos que al pulsar se envíe un byte por el transmisor al PC.

Este ejemplo nos servirá para introducir el diseño estructural.

Volvemos al proyecto en el ISE, allí tendremos el componente transmisor y el banco de pruebas. En la subventana de fuentes (**Sources**) seleccionamos la fuentes para síntesis e implementación (**Sources for: Synthesis/Implementation)**.

Ahora crearemos un diseño estructural, donde tengamos el transmisor y el interfaz con los pulsadores. Nuestro diseño final tendrá este aspecto (figura 5-17):

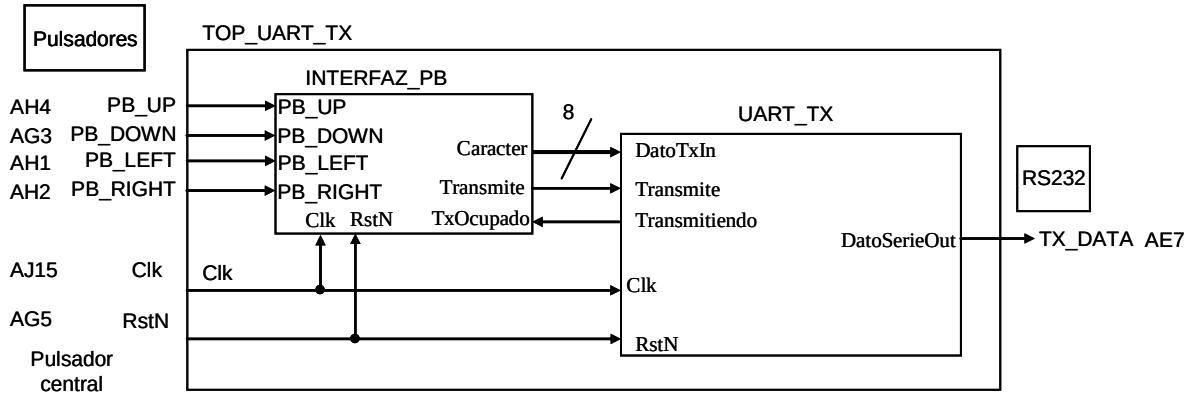

Figura 5-17: Esquema estructural del circuito que vamos a implementar

Cada uno de los bloques de la figura 5-17 representa una entidad VHDL y lo crearemos en un fichero distinto:

- El módulo `UART_TX` es el transmisor que ya hemos creado.
- El bloque `INTERFAZ_PB` es un módulo que dará la orden de transmitir cuando apretemos un pulsador
- El bloque `TOP_UART_TX` es un bloque estructural, que sólo establece las conexiones entre los módulos que contiene.

Creamos el módulo de interfaz con los pulsadores: Creamos una nueva fuente (**New Source**) de tipo **VHDL Module**, cuyos puertos son los mostrados en la 5-17. Todos serán de tipo `std_logic`, menos `Caracter` que será `std_logic_vector(7 downto 0)`. Siguiendo los pasos explicados en la primera práctica el ISE nos proporcionará la entidad VHDL del circuito y nos ponemos a realizar la arquitectura.

Como los pulsadores son elementos totalmente asíncronos, el interfaz de los pulsadores será síncrono para lograr un mejor acoplamiento con el transmisor, reduciendo la posibilidad de transmitir valores indeterminados (metaestabilidad). Para cada uno de los pulsadores haremos un detector de flanco (código 4-1).

El proceso principal enviará un carácter diferente según el botón pulsado. Este carácter se transmitirá si el transmisor no está ocupado (`TxOcupado='0'`).

```
P_Interfaz:Process(RstN, Clk)
begin
  if RstN = '0' then
    Caracter <= (others=>'0');
    Transmite <= '0';
  elsif Clk'event and Clk='1' then
    Transmite <= '0';
    Caracter <= x"00";     -- ascci: Caracter nulo ;
    if TxOcupado = '0' then  -- No esta transmitiendo el transmisor
      if PulsoLeft = '1' then
        Transmite <= '1';
        Caracter <= x"68";   -- ascci: h ; la x es hexadecimal
      elsif PulsoUp = '1' then
        Transmite <= '1';
        Caracter <= x"6F";   -- ascci: o ;
      elsif PulsRight = '1' then
        Transmite <= '1';
        Caracter <= x"6C";   -- ascci: l ;
      elsif PulsoDown = '1' then
        Transmite <= '1';
        Caracter <= x"61";   -- ascci: a ;
      end if;
    end if;
  end if;
end process;
```

Código 5-14: Proceso principal del interfaz

Incluimos este proceso en la arquitectura y comprobamos su sintaxis en el ISE (**Check Syntax**). Nota cómo el módulo `interfaz_pulsadores` tiene menos opciones en la subventana **Processes**. Mientras que si seleccionamos la `UART_TX` tiene todas las opciones de implementación. Esto se debe a que `UART_TX` figura como el módulo de más alto nivel y que está indicado por unos cuadraditos a su izquierda. Para poner un módulo como el de más alto nivel se pincha con el botón

derecho encima de él y se selecciona **Set as Top Module**. De todos modos, ninguno de estos dos módulos va a ser el superior en la jerarquía, sino el que vamos a crear a continuación.

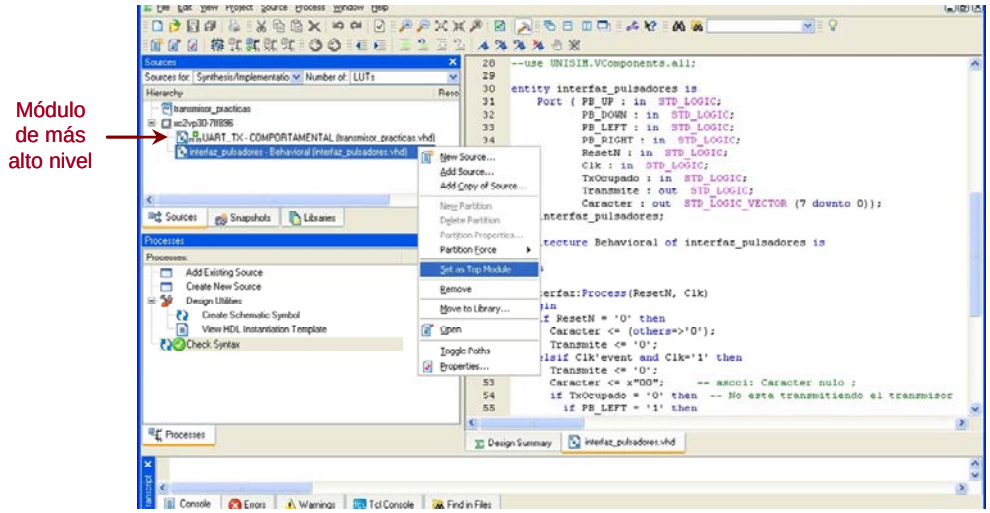

Figura 5-18: Indicación del módulo de más alto nivel

Ahora creamos el módulo que engloba a los anteriores, lo llamaremos `top_transmisor` y la arquitectura será de tipo `Estructural`. Los puertos serán los mostrados en la figura 5-17, todos ellos de tipo `std_logic`. Una vez que tenemos la entidad VHDL creada, hay que declarar y referenciar (*instanciar*) los componentes, y establecer las conexiones entre los puertos y componentes

```
architecture Estructural of TOP_UART_TX is
  component UART_TX
  generic (
    gFrecClk      : integer := 100000000;   --100MHz
    gBaud         : integer := 9600
  );
    Port(
      RstN          : in std_logic;
      Clk           : in std_logic;
      Transmite     : in std_logic;
      DatoTxIn      : in std_logic_vector(7 downto 0);
      Transmitiendo : out std_logic;
      DatoSerieOut  : out std_logic
    );
  end component;
  component INTERFAZ_PB
    Port (
      RstN          : in  std_logic;
      Clk           : in  std_logic;
      PB_UP         : in  std_logic;
      PB_DOWN       : in  std_logic;
      PB_LEFT       : in  std_logic;
      PB_RIGHT      : in  std_logic;
      TxOcupado     : in  std_logic;
      Transmite     : out  std_logic;
      Caracter      : out  std_logic_vector (7 downto 0)
    );
  end component;
  signal Transmitiendo   : std_logic;
  signal Transmite       : std_logic;
  signal DatoTxIn        : std_logic_vector (7 downto 0);
begin
  TX: UART_TX
    Generic Map (
      gFrecClk      => gFrecClk,
      gBaud         => gBaud
    )
    Port Map (
      RstN          => RstN,
      Clk           => Clk,
      Transmite     => Transmite,
      DatoTxIn      => DatoTxIn,
      Transmitiendo => Transmitiendo,
      DatoSerieOut  => TX_DATA
    );
  INTERFAZ: INTERFAZ_PB
```

```
    Port Map (
      RstN        => RstN,
      Clk         => Clk,
      PB_UP       => PB_UP,
      PB_DOWN     => PB_DOWN,
      PB_LEFT     => PB_LEFT,
      PB_RIGHT    => PB_RIGHT,
      TxOcupado   => Transmitiendo,
      Transmite   => Transmite,
      Caracter    => DatoTxIn
    );
end Estructural;
```

Código 5-15: Arquitectura estructural del módulo de mayor jerarquía

Una vez creado el diseño estructural, le indicamos al ISE que compruebe la sintaxis, y si esta es correcta, lo más seguro es que detecte que es el diseño de mayor jerarquía, y lo señale como tal. Si no fuese así, se lo indicamos nosotros.

Ahora sintetizamos el diseño (estando seleccionada la unidad de mayor jerarquía) y vemos las indicaciones que nos da la síntesis y si hay algún error y observamos las advertencias (*warnings*) por si hay alguna que sea importante.

A continuación creamos el fichero de restricciones, que indica la conexión de los pines y la frecuencia del reloj.

Finalmente ya estamos listos para llevar el diseño a la FPGA: encendemos la FPGA, conectamos el puerto serie al ordenador y a la placa, y programamos la FPGA con el *iMPACT* como ya lo hemos hecho otras veces.

Una vez que está programada (LED DONE está encendido), vamos a ver si funciona nuestro diseño. Para ello abrimos un *hiperterminal*, Windows suele tener uno en **Inicio→ Todos los programas→ Accesorios→ Comunicaciones→ HyperTerminal**. También hay otros de libre distribución como el *TeraTerm* que quizás te gusten más.

Usaremos el *HyperTerminal* de Microsoft, una vez que se sabe utilizar uno no es difícil usar otro.

Al abrirlo nos aparecerá una ventana para establecer una nueva conexión (figura 5-19), nos pide nombre de la conexión y un icono para ella (esto del nombre y del icono no siempre lo piden en otros *hiperterminales*). Si no nos lo pidiese pinchamos en **Archivo→Nueva Conexión**. Posteriormente (figura 5-20) aparecerá una ventana que dice *Conectar a*, le indicamos **Conectar usando COM1** (u otro COM según tu ordenador).

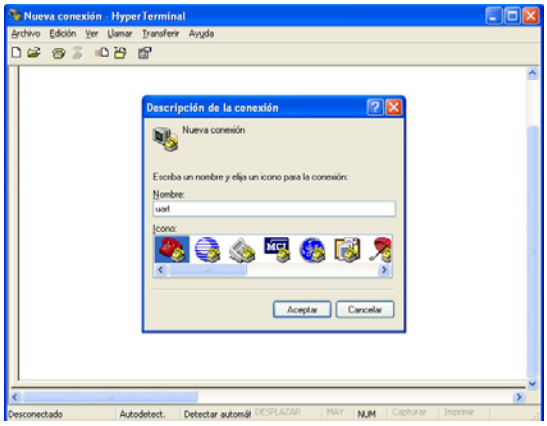

Figura 5-19: Nueva conexión del hiperterminal

Figura 5-20: Conexión al puerto serie

Y por último nos saldrán las propiedades de la comunicación con el puerto serie (figura 5-21). Después del diseño que hemos hecho, ya estaremos familiarizados con el significado de estas propiedades. Las rellenamos según las características de nuestro diseño: bits por segundo, paridad, ...

Después de darle a aceptar ya tendremos la conexión, y probamos a presionar uno de los cuatro botones (no el del medio que es el reset y no hará nada visible).

Comprueba que en el *hiperterminal* aparecen los caracteres que has enviado. Si te salen, ¡enhorabuena! Si no, comprueba que las características de la conexión que has establecido coinciden con las de tu circuito, y que tienes los pines correctamente configurados. A veces sucede que el *hiperterminal* no funciona a altas velocidades de transmisión. En estos casos, el error aparece al configurar el *hiperterminal*, sin siquiera conectar la placa al PC por el puerto serie. Si esto sucede, prueba a poner una velocidad baja, por ejemplo 9600.

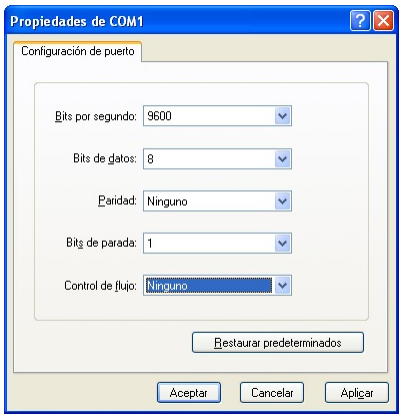

Figura 5-21: Características de la conexión

5.4 Diseño del receptor

A estas alturas ya estamos familiarizados con el protocolo de transmisión RS232. Hemos diseñado, simulado e implementado el transmisor, y además hemos realizado un modelo de un receptor para simulación (código 5-12). Por tanto, con los conocimientos adquiridos no nos debería ser muy difícil diseñar un receptor sintetizable.

La entidad que queremos diseñar tiene el siguiente aspecto:

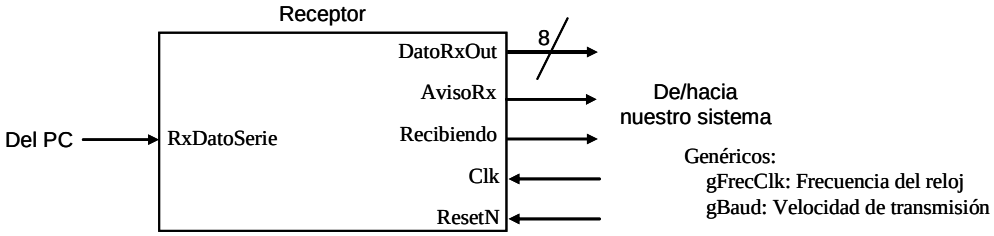

Figura 5-22: Entradas y salidas del receptor

De la figura 5-22, el puerto de la izquierda (`DatoSerieIn`) es el que se encarga de recibir el dato del PC. Los puertos de la derecha se relación con el sistema de la FPGA. Igual que en el transmisor (figura 5-5) se definen los genéricos `gFrecClk` y `gBaud`. Las especificaciones de los puertos se muestran en la tabla 5-2.

Señal	bits	I/O	Descripción
`RstN`	1	I	Señal de reset asíncrono. Activo a nivel bajo, esto es, un '0' reseteará el módulo.
`Clk`	1	I	Señal de reloj de la placa, en principio de 100MHz, pero configurable por `gFrecClk`
`RxDatoSerie`	1	I	Trama que se recibe del PC, sigue el formato RS232
`DatoRxOut`	8	O	El dato que se ha recibido. Este dato es sólo será válido desde que `AvisoRx` valga '1' y mientras `Recibiendo` sea '0'
`AvisoRx`	1	O	Aviso de que se ha recibido un nuevo dato y que está disponible en `DatoRxOut`. El aviso se dará poniendo la señal a '1' durante un único ciclo de reloj
`Recibiendo`	1	O	Cuando vale '1' indica al sistema que el módulo se encuentra recibiendo una trama y por tanto el valor del dato `DatoRxOut` no es válido

Tabla 5-2: Características de los puertos del transmisor

Para facilitar la comprensión del funcionamiento del receptor, en la figura 5-23 se muestra el cronograma de las señales. A partir de la figura se observa que el dato recibido sólo será válido a partir del aviso de la señal `AvisoRx` y hasta que la señal `Recibiendo` no se ponga a '1'. Desde que la señal `Recibiendo` esté a '1', el dato no será válido. Nunca podrán estar simultáneamente a '1' las señales `Recibiendo` y `AvisoRx`.

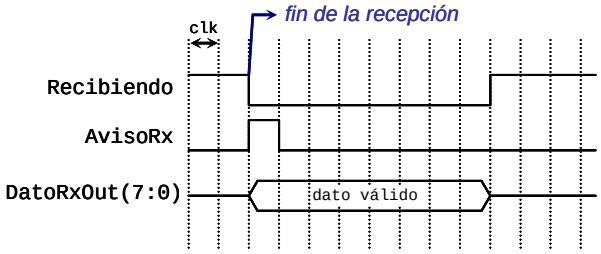

Figura 5-23 : Cronograma de las salidas del receptor

El diseño del receptor se puede realizar de manera parecida al del transmisor. Una primera implementación del diseño se muestra en la figura 5-24

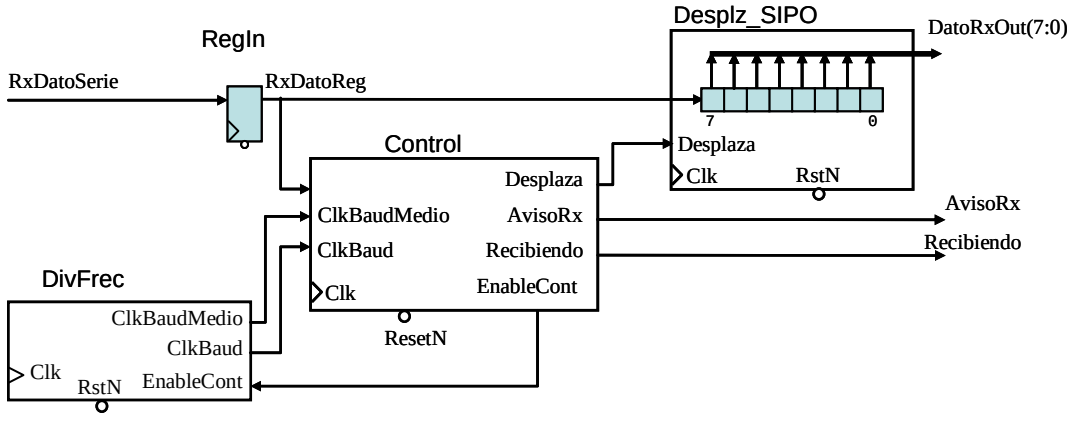

Figura 5-24 : Diagrama de bloques preliminar del receptor

A continuación se describirán brevemente cada uno de los bloques

5.4.1 Registro de desplazamiento

El registro de desplazamiento (`Desplz_SIPO`) es un registro al que se le van cargando los datos en serie y los devuelve en paralelo (*Serial In – Parallel Out*: SIPO). Es el registro opuesto que teníamos en el transmisor, que cargaba en paralelo y devolvía en serie (PISO). Como el primer bit que se recibe es el bit 0, si la carga serie se hace por el bit más significativo del registro y se hacen desplazar los bits hacia la derecha [bit n→ bit (n-1)], en el último desplazamiento, el bit 0 recibido estará en el bit 0 del registro, y estarán todos los bits ordenados. La carga y el desplazamiento se realizan bajo la orden de la señal `Desplaza` que parte del bloque `control`

5.4.2 Registro de entrada

La entrada de la comunicación serie `RxDatoSerie` se registra en `RxDatoReg` (módulo `RegIn` de la figura 5-24). Como `RxDatoSerie` es una señal asíncrona es conveniente registrarla, y para evitar meta-estabilidad se puede incluso registrar dos veces poniendo dos registros en cascada (recuerda la figura 0-11)

5.4.3 Divisor de frecuencia

Para el receptor, el divisor de frecuencia debe sincronizarse con la trama que se recibe porque el receptor no dirige la transmisión. El divisor de frecuencia se mantiene inactivo y a cero hasta que la entrada `EnableCont` se pone a '1'. El divisor de frecuencia saca dos señales: `ClkBaud` y `ClkBaudMedio`. `ClkBaud` marca la transición del estado, y `ClkBaudMedio` marca el punto medio de cada bit de la transmisión. En este punto medio es donde se evalúa el valor del bit recibido, evitando evaluar el bit cerca de las transiciones donde se puede tomar el valor del bit contiguo.

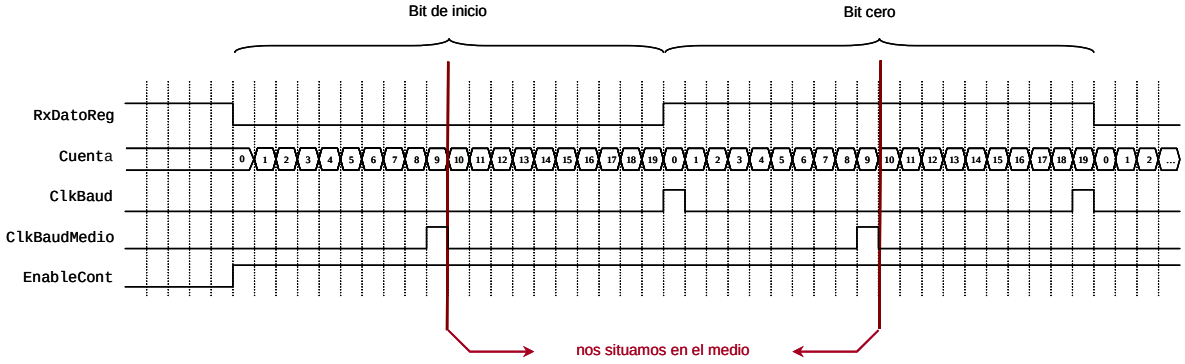

Figura 5-25 : Cronograma del divisor de frecuencia

En la figura 5-25 se muestra el cronograma del divisor de frecuencia con una cuenta de ejemplo de 20.

Se podría pensar en realizar un mismo divisor de frecuencia para el transmisor y el receptor. Para ello, el divisor de frecuencia no podría estar bloqueado. Y se podría generar una señal de una frecuencia tres veces mayor que `ClkBaud`, y sincronizarse con el segundo ciclo que llegue. Sin embargo, la FPGA que usamos es suficientemente grande para no tener que complicarnos por ahorrarnos un poco de área. Para FPGA más pequeñas sí que podría ser interesante.

5.4.4 Bloque de control

El bloque de control es una máquina de estados similar al del transmisor. El cambio de estado también viene provocado por `ClkBaud`. La orden de desplazamiento (`Desplaza`) se dará cuando se está en el los estados de recepción de dato (`BIT_DATO`) y llega la señal `ClkBaudMedio`

5.5 Banco de pruebas del receptor

Para comprobar la funcionalidad del receptor se puede realizar un banco de pruebas específico para él, de manera similar a como se hizo módulo transmisor.

También se pueden probar el transmisor y emisor juntos, de modo que utilizando el proceso generador de estímulos del código 5-12 se envíen al transmisor que ya tenemos, y éste se conecte al receptor.

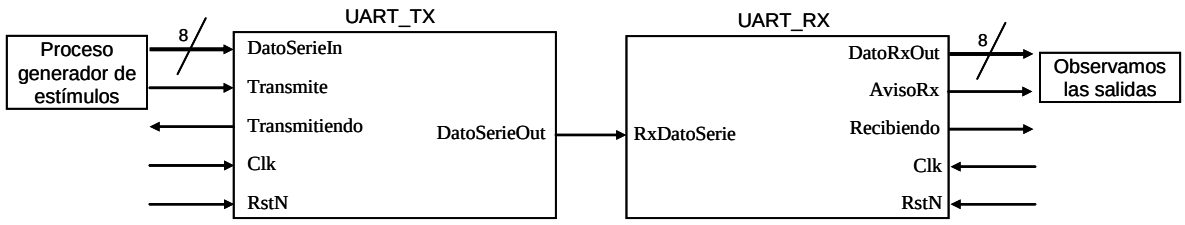

Figura 5-26 : Diagrama de bloques del banco de pruebas

5.6 Implementación de la UART completa

Uniendo en un diseño estructural el transmisor y receptor tendremos la UART completa. Éstos se pueden conectar de manera que lo recibido por el receptor se emita por el transmisor, consiguiéndose el eco en el *hiperterminal*. La conexión sería la siguiente

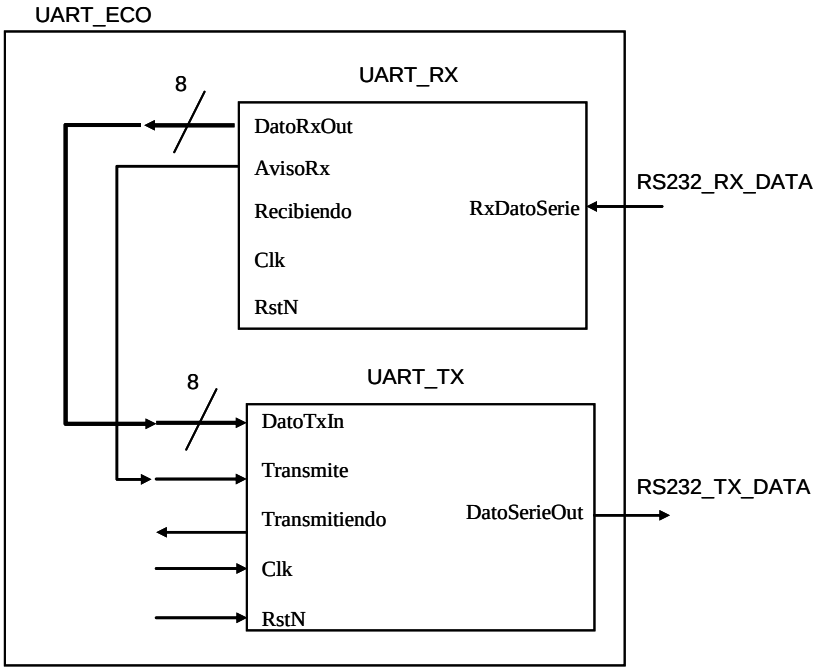

Figura 5-27: Conexiones del transmisor y receptor para generar un eco de la trama recibida

Si implementas el diseño y ves que no obtienes el eco, o que obtienes caracteres distintos a los que has enviado, revisa la configuración del *hiperterminal*. Si todo está correctamente configurado, deberás simular el circuito `UART_ECO`. Para ello puedes realizar un banco de pruebas como el mostrado en la figura 5-28.

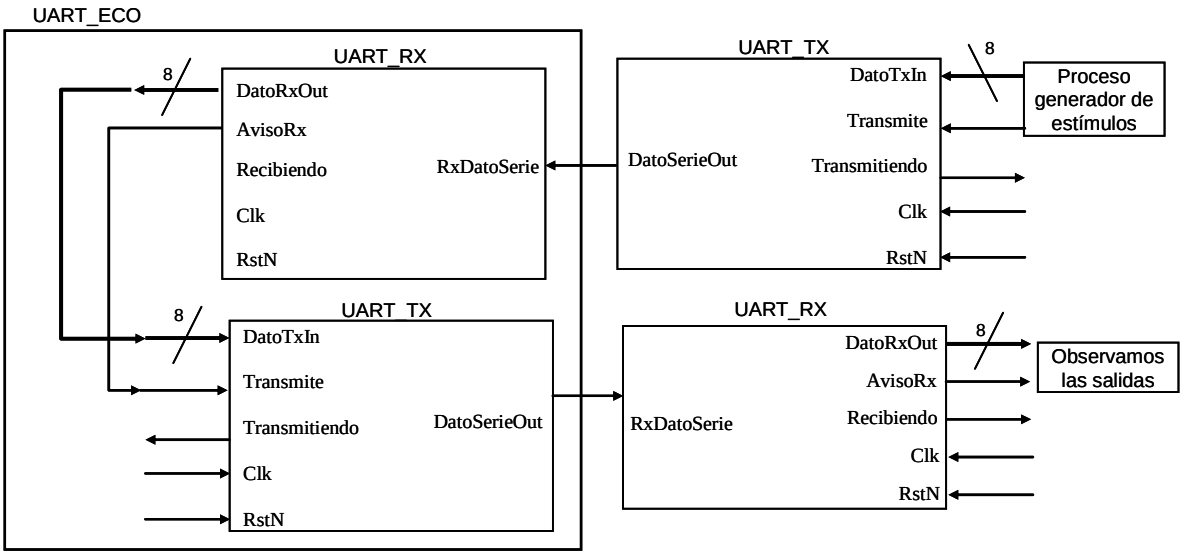

Figura 5-28: Banco de pruebas para la UART con eco

Según la figura 5-28, el banco de pruebas tendrá 3 componentes: `UART_ECO`, `UART_TX` y `UART_RX`. Dentro de `UART_ECO` están también el emisor y receptor, aunque desde el banco de pruebas no se ve. El proceso generador de estímulos puede ser el mismo proceso que hemos empleado en la simulación del transmisor (apartado 5.2, código 5-11).

A partir de ahora podremos usar los módulos `UART_TX` y `UART_RX` para cualquier diseño en el que necesitemos comunicarnos con el PC u otro dispositivo. Así que es conveniente que guardes las versiones de los modelos VHDL que funcionen, con ello ahorrarás mucho tiempo de diseño cuando las vuelvas a necesitar. Esto se llama **reutilización** y es algo muy utilizado. Incluso hay compañías que se dedican vender bloques VHDL (ej.: www.design-reuse.com/). También hay bloques de libre distribución (ej.: www.opencores.org/). Comúnmente estos bloques se denominan *IP*, que es el acrónimo de *Intellectual property* (propiedad intelectual).