Eduardo Pérez Rosas ***02 de Diciembre del 2021***
A01637346 Evidencia 1 ***Sistemas Digitales***

Introducción

El estándar RS-232 es una norma mundial que rige los parámetros de uno de los modos de comunicación serial. Se encarga de estandarizar las velocidades de transferencia de datos, la forma de control que utiliza dicha transferencia, los niveles de voltaje utilizados, el cableado permitido, distancias entre equipos, conectores, entre otros.
Las comunicaciones seriales poseen líneas de transmisión (Tx), recepción (Rx) y otras líneas de control de flujo donde su uso es opcional dependiendo del dispositivo a conectar. A nivel de software, la configuración principal que se debe dar a una conexión a través de puertos seriales es la selección de velocidad en baudios (1200, 2400, 4800, etc.), verificación de datos o paridad, bits de parada luego de cada dato (ya sean 1 o 2) y la cantidad de bits por dato (7 u 8), que se utiliza para cada símbolo o carácter enviado.
En el puerto serie se tienen dos líneas, Tx y Rx, las cuales se deben cruzar para comunicar dos dispositivos. El Tx del dispositivo 1 se conecta al Rx del dispositivo 2. El Tx del dispositivo 2 se conecta al Rx del dispositivo 1. Cuando un dispositivo solo tiene el Tx es porque ese dispositivo sólo envía datos y no los recibe.
Un transmisor-receptor asíncrono universal (UART) es un conjunto de reglas para intercambiar datos en serie entre dos dispositivos, la transmisión y recepción es mediante dos cables de ambas terminales (receptor y transmisor) las cuales están conectadas a tierra, la comunicación del UART puede simplex es decir que los datos se envían en una sola dirección, semidúplex donde cada lado transmite pero solo uno a la vez o dúplex completo el cual ambos lados pueden transmitir en simultáneo, al utilizar una comunicación asíncrona debido a que no hay señal de reloj para sincronizar los bits de salida del dispositivo transmisor que va al extremo receptor .

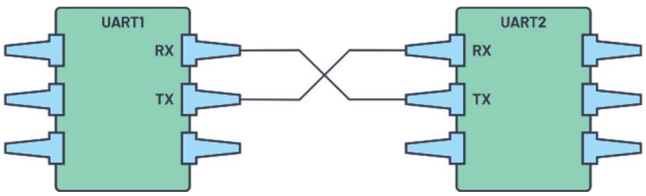


Figura 1. UART

Para enviar datos se debe usar un hardware que esté diseñado para llevar a cabo esa tarea, en este caso es el Universal Asynchronous Receiver/Transmitter (UART). Se encargará de leer los datos cuando llegan, generar y gestionar interrupciones, enviar datos y gestionar los tiempos de bit.
Lo que se debe enviar a través del puerto serie es la combinación de ceros y unos, lo primero que se debe hacer es invertir el orden de los bits, el primero pasa a ser el último y el último será el primero, esto se debe a que en este protocolo primero se envía el bit menos significativo. La línea de transmisión siempre estará en reposo en nivel alto. Para iniciar la comunicación se debe enviar un bit de start que siempre será un 0 y se mantendrá durante un tiempo que se llama “tiempo de bit”, el cual es el tiempo en el que se mantiene un bit en la línea de transmisión, su cálculo es muy sencillo, solo se debe dividir 1 sobre el baud rate. Pasado este tiempo se comenzará a enviar los datos, posteriormente se debe indicar que ya se ha terminado el proceso, por lo que se debe enviar

un bit de stop, se debe enviar un 1 durante el tiempo de bit. Después del bit de stop la línea se quedará en espera en nivel alto.

Análisis y descripción del problema

Se deben desarrollar dos máquinas de estados, una encargada de hacer la transmisión o envío de información, y otra encargada de recibir el paquete de información enviado por el transmisor.
La máquina de estado de la transmisión está descrita por la siguiente imagen, misma que es una máquina interna de estados finitos para transmitir el carácter completo de 10 bits, bit a bit.

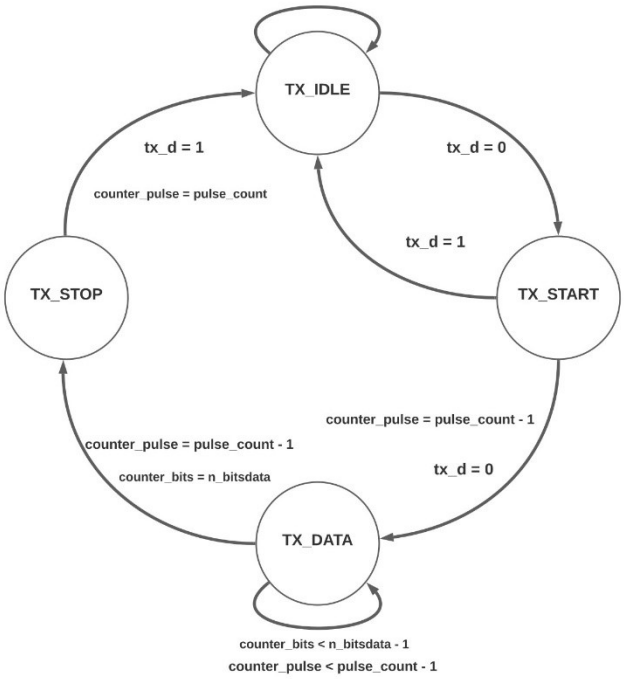


Figura 2. Máquina de estados Transmisor UART.

En este caso, se tienen cuatro estados en el transmisor TX_IDLE, TX_START, TX_DATA y TX_STOP.
TX_IDLE: cuando se restablece el UART la máquina de estado se encontrará en este estado. El módulo transmisor esperará al pulso proveniente de tx_d, que indica que los datos se encuentran listos para ser transmitidos. Esto habilita el pulso proveniente del procesador y se sincroniza con el reloj serial.
TX_START: en este estado, el módulo transmisor enviará un bit de inicio ‘0’ en la línea de datos en serie.
TX_DATA: en este estado, la máquina de estado carga el registro interno con datos a transmitir. Para la primera transmisión, el registro se cargará con una dirección de destino. La máquina transmite la dirección poco a poco, empezando por el bit menos significativo en [0] hasta el bit más significativo [7].
TX_STOP: Cuando se ha transmitido con éxito el paquete de datos, la máquina de estado entrará en este estado para completar la transmisión enviando un bit de parada de ‘1’. Luego verifica si hay otro byte de datos, si lo hay, la máquina de estado irá nuevamente a TX_START y repite el proceso, en caso de estar vacío significa que no hay más bytes de datos para transferir, por lo tanto, la máquina de estados entrará en TX_IDLE y esperará otro pulso que indique TX_START.

Durante la recepción el UART estará en modo escucha. Se detecta un bit de inicio cuando hay una transición de ‘1’ lógico a ‘0’ lógico en la línea de datos en serie. Capturará el byte de dirección y comprobará el bit de parada. Si el bit de parada es correcto, la dirección capturada se comparará con la dirección de transmisión y la propia dirección UART determinará si recibir o ignorar los bytes de datos entrantes. El módulo receptor también comprueba la integridad del bit de inicio y el bit de parada. Si un bit de inicio en falso o un bit de parada se detecta, la recepción se interrumpirá y una interrupción se enviará al procesador para leer el resto de los datos, si están disponibles, en el buffer RX.

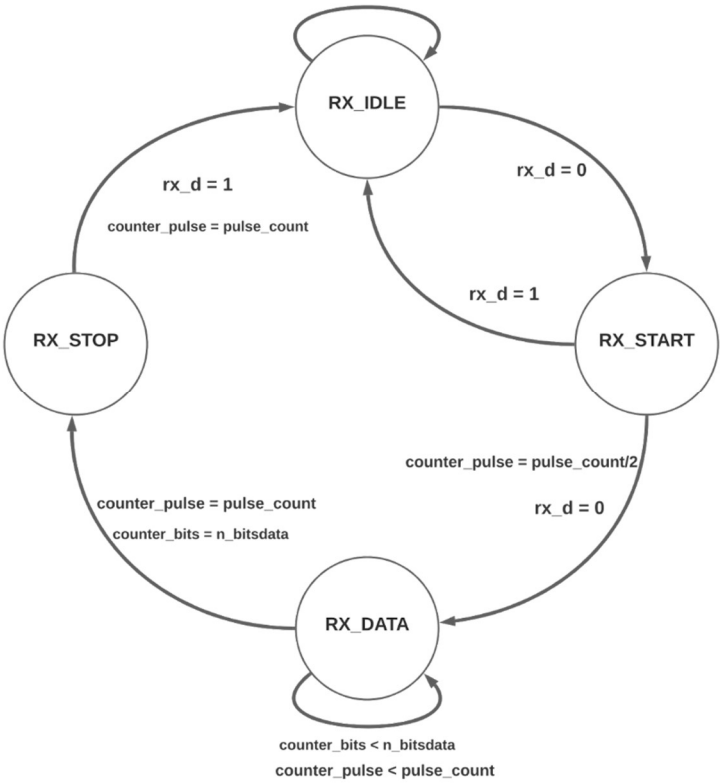


Figura 3. Máquina de estados Receptor UART.

Se utilizó una máquina de estados finitos para implementar los pasos mencionados anteriormente, contiene cuatro estados RX_IDLE, TX_START, RX_DATA y RX_STOP.

RX_IDLE: cuando se reinicia el módulo receptor, la máquina de estado estará en este estado, esperará para la detección de bits de inicio en la línea de datos en serie. Un bit de comienzo se identifica cuando hay una transición de caída en la línea de datos en serie del estado inactivo ‘1’ al ‘0’ lógico. Para no detectar un bit falso, solo considerará rx_d señal que se ha sincronizado con el serial de reloj.

RX_START: En este estado se llega luego de haber corroborado que existe un bit de start equivalente a ‘0’, para luego poder pasar al siguiente estado correspondiente.

RX_DATA: tomará una muestra del carácter recibido en el momento más ideal, que es en la mitad de cada bit. Luego, cada bit se almacena en un registro “register_dataRx” para formar el paquete de datos completo de 8 bits.

RX_STOP: tomará una muestra del bit de parada en el punto medio. Detendrá la comprobación de errores de bits y la dirección de comparación, si no detecta ningún bit de parada, ese es el bit

muestreado ‘0’ lógico, el módulo receptor interrumpe el proceso de detección de errores, comparación de direcciones de omisión y la máquina de estado pasará a RX_IDLE. Si detecta un bit de parada correcto, procede con la coincidencia de direcciones para comparar el byte de dirección recibido con su propia dirección.

Desarrollo de la propuesta de solución

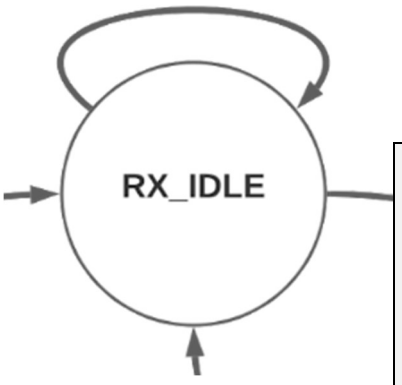
Para la implementación de nuestro UART se decidió realizar un código en VHDL en la herramienta de Model Sim la cual nos ayudara de igual manera para la simulación del mismo código. Se tuvo que realizar un código para cada uno de los sistemas los cuales son receptor y transmisor, así como también se decidió crear un testbench para cada uno para realizar su simulación correspondiente.

Para comenzar con el desarrollo del UART se comenzó por diseñar el **receptor** el cual tendrá como entradas definidas en el código:

clk : señal de reloj
reset: control de reset.
rx_d: recepción de datos.
data_package: vector de datos para el guardado de los mismos.

Para la construcción de la arquitectura y como se vio en la parte anterior, esta estará compuesta de estados los cuales a su vez serán Rx_Idle, Rx_Start, Rx_Data y Rx_Stop, así como también se utilizarán contadores para poder realizar paros para la correcta lectura de datos. A continuación, se procederá a describir cada uno de los estados.

RX_IDLE

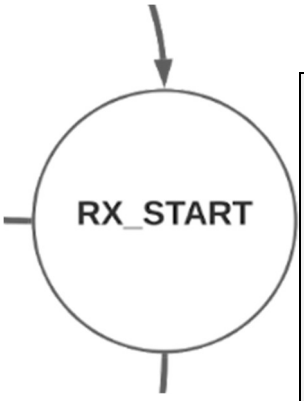



```
when Rx_Idle =>
        counter_bits    <= (others => '0');
        counter_pulse   <= (others => '0');
        register_dataRx <= (others => '0');
        if rx_d = '0' then
        nextstatus <= Rx_Start;
        end if;
```

Este estado es el primero en ejecutarse el cual es el encargado de la detección de los bits de inicio de nuestra línea de datos, el cual se inicializa cuando hay una caída en la línea de ‘0’ a ‘1’. Cuando

se capta un bit de inicio este nos llevara al estado “Rx_Start”, cabe mencionar que al momento de reiniciar el sistema es en este estado en donde somos dirigidos.

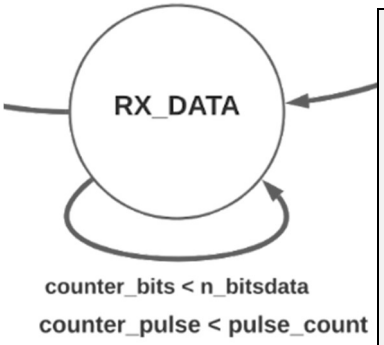
RX_START.




```
when Rx_Start =>
    if counter_pulse <  (pulses_count/2) then
    counter_pulse <= counter_pulse+1;
    else
    counter_pulse <= (others=>'0');
   if rx_d ='0' then
   nextstatus <= Rx_Data;
   else
   nextstatus <= Rx_Idle;
   end if;
   end if;
```

En el estado de Start se realiza la verificación del contador de pulsos e ir incrementando uno hasta que esta llegue en la mitad del bit, esto lo podemos notar en el primer IF en el cual se compara que “counter_pulse” sea menor que “pulses_count/2”, cuando el contador tiene la misma cantidad que a la cantidad de pulsos dividida entre 2 se le asignara el valor de ‘0’ puesto que los pulsos fueron sensados, enseguida un IF verifica que el valor de rx_d sea ‘0’ y esto corrobora que no hubo un falso start.

RX_DATA.

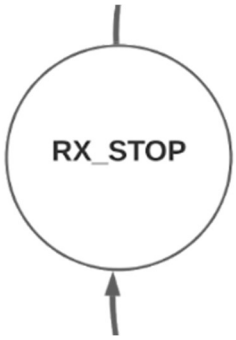



```
when Rx_Data =>
        if counter_pulse   <  pulses_count then
        counter_pulse   <= counter_pulse+1;
        else
        counter_pulse   <= (others=>'0');
        if counter_bits <   n_bitsdata then
        register_dataRx <= rx_d & register_dataRx(n_bitsdata-1 downto 1)
        counter_bits <= counter_bits +1;
        else
        counter_bits <= (others =>'0');
        nextstatus <= Rx_Stop;
        end if;
        end if;
```

Este estado es el encargado de recibir y almacenar los bits de datos que serán ingresados de manera serial. Se puede observar en el primer IF en el cual se hace una comparación, si “counter_pulse” es menor a “pulse_count”, la cual sirve para empezar el contador, este contador nos servirá poder ir recibiendo los datos por medio de “rx_d”, esto se hace a través del segundo IF el cual compara si

“counter_bits” es menos a “n_bitsdata”, se irán almacenando los bits en “register_dataRx”, mientras que en este IF se cumpla quiere decir que todavía no se han recibido los 8 bits correspondientes a la cadena de datos. Cuando se termina e recibir los 8 bits correspondientes y por medio de la concatenación de estos, se tendrá que el primer bit recibido pasara a la posición 0.

RX_STOP




```
when Rx_Stop =>
        if counter_pulse < pulses_count then
        counter_pulse <= counter_pulse+1;
        else
        counter_pulse <= (others=>'0');
        buffer_Rx <= register_dataRx;
        nextstatus <= Rx_Idle;
        end if;
```

Este último estado el cual es el e Stop, el cual es el encargado de la detección del bit de stop ‘1’, el cual se encuentra mediante el monitoreo de “rx_d”, una vez detectado el contador es puesto a ‘0’ y el bus de datos se pasa a almacenar a “buffer_Rx” para por último regresar al estado de reposo, con esto nuestro sistema de recepción esta completo.

Transmisor.

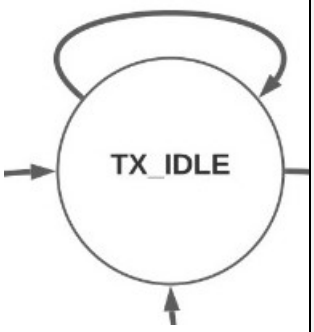
Para la implementación del código, se definen las entradas y salidas las cuales son las siguientes.

clk: pulso de reloj.
reset: señal de reset.
sending: señal para inicial la transmisión.
tx_d: transmisión de datos en forma serial.
n_bitsdata: cantidad de bits de nuestro bus.
baud_rate: velocidad de transmisión.

A continuación, se procederá a realizar una descripción de cada uno de nuestros estados, por lo cual.

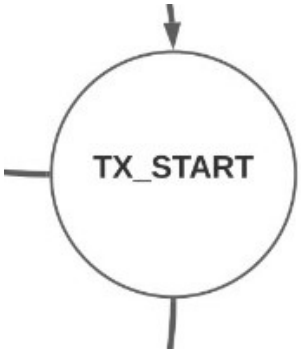
RX_IDLE

En este estado al igual que en el receptor, corresponde al estado de reposo, en este estado para la verificación de continuidad se manda la señal de “tx_d” a ‘1’, al momento que a “sending” sea uno este empieza la transmisión.




```
when Tx_Idle =>
        tx_d <= '1';
        if sending='1' then
        register_dataTx <= message(indice);
        if indice<message'length-1 then
        indice <= indice + 1;
        else
        indice <= 0;
        end if;
        counter_pulse <= (others => '0');
        counter_bits  <= 0;
        nextstatus <= Tx_Start;
        end if;
```

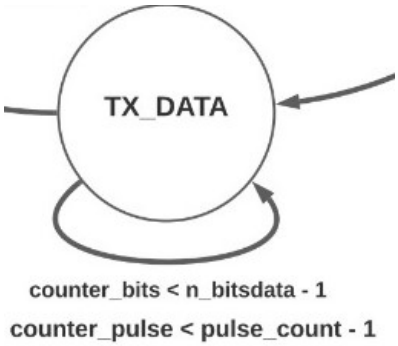
TX_START




```
when Tx_Start =>
        tx_d <= '0';
        nextstatus <= Tx_Data;
```

Este estado es el encargado de dar el bit de inicio correspondiente a ‘0’, por lo tal este estado se encarga de sincronizar el transmisor y el receptor.

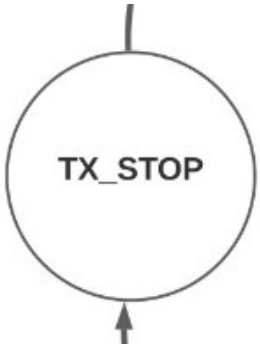
TX_DATA




```
when Tx_Data =>
        if counter_pulse < pulses_count-1 then
        counter_pulse <= counter_pulse+1;
        else
        tx_d <= register_dataTx(counter_bits);
        counter_pulse <=(others => '0');
        if counter_bits < n_bitsdata-1 then
        counter_bits <= counter_bits+1;
        else
        counter_bits <= 0;
        nextstatus <= Tx_Stop;
        end if;
        end if;
```

El estado TX_DATA es el encargado de la transmisión del paquete de datos, haciendo uso de una comparativa entre el contador de pulsos y los pulsos contados podemos generar el ancho del bit, con ayuda de la señal “tx_d” a la cual se le asignan los bits de “register_dataTx”, por lo cual con la ayuda del subíndice a “counter_bits” podemos ir asignando el bit correspondiente al valor de “counter_bits” . En el siguiente IF se realiza la verificación del envió correcto de todos los bits.

TX_STOP




```
when Tx_Stop =>
        if counter_pulse < pulses_count-1 then
        counter_pulse <= counter_pulse+1;
        else
        tx_d <= '1';
        counter_pulse <= (others => '0');
        buffer_Tx <= register_dataTx;
        nextstatus <= Tx_Idle;
        end if;
```

En este estado se realiza el envió del dato final ‘1’ el cual indica que el bus de datos se a terminado de enviar, con esto último nos regresa al estado de reposo para poder enviar otro bus de datos.

Simulaciones.

Como se mencionó anteriormente se realizó un testbench para cada uno de nuestros códigos para poder simular su funcionamiento, obteniendo en estas los siguientes resultados.

Receptor.

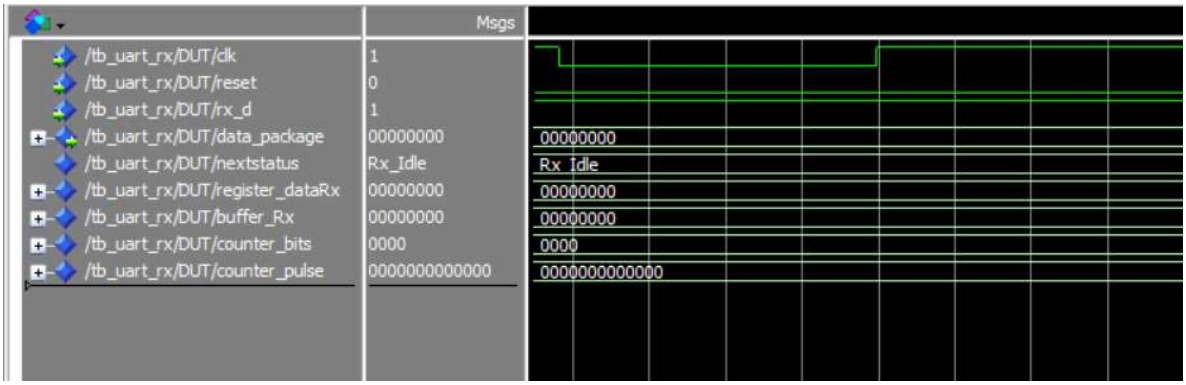


Figura 4. Estado Rx_Idle

Podemos verificar que el envió del bus de datos no ha empezado dado que rx_d esta en ‘1’ por lo cual no a recibido el bit de inicio. Por lo cual todo se mantiene en ‘0’.

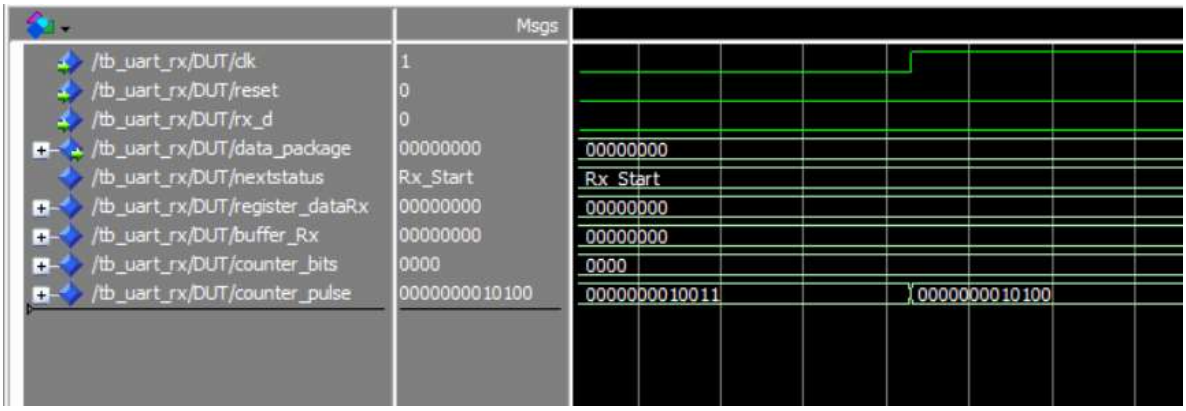


Figura 5. Estado Rx_Start

En esta parte es evidente que se recibió el bit de inicio '0', lo cual hace que el counter_pulse comience a actualizar, por lo cual a su ves se verificara la existencia de un bit falso de start.

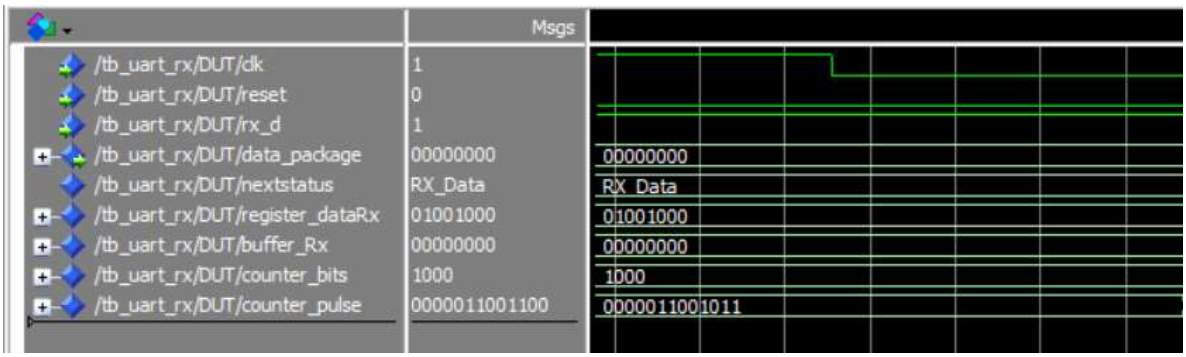


Figura 6. Estado Rx_Data

Se pude notar que en el estado de data como se describio previamente este se encargara de almacenal los valores recividos. Podemos compobar que en este estado de la simulacion Data ha recibido toda la cadena de 8 bits de datos y se puede verificar con el valor del contador de bits el cual esta en '1000'.

Transmisor.

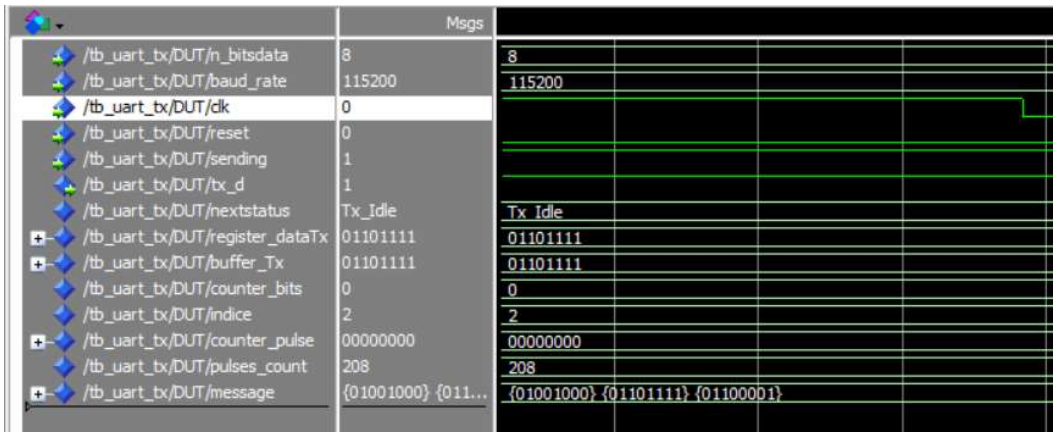


Figura 7. Estado Tx_Idle

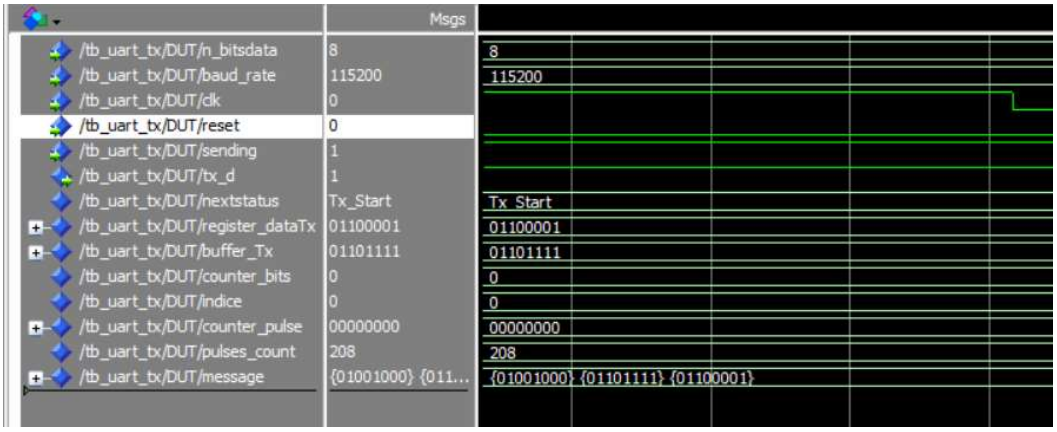


Figura 8. Estado Tx_Start

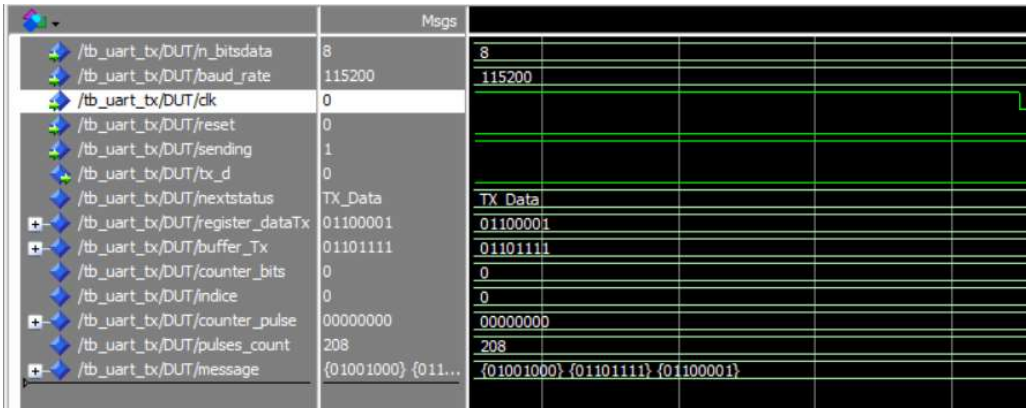


Figura 9. Estado Tx_Data

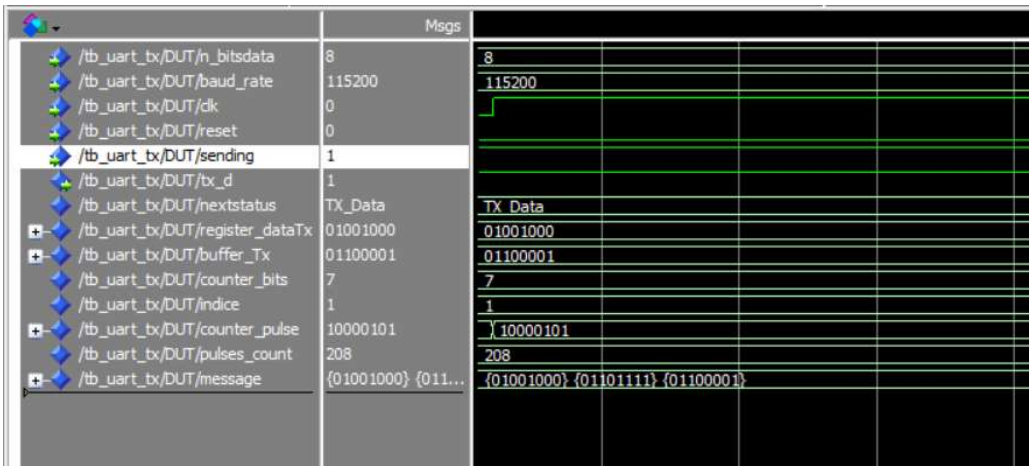


Figura 10. Estado Tx_Data

En las anteriores imagene podemos ver de manera clara el envio de un bus e datos en cada estado en el que el transmisor ejecuta, se puede observar como nuestros contadores juegan un papel sumamente importante a la hora de enviar la informacion ya que estos son los encargados de que si se envie el correcto numero de datos asi como tambien su orden de envio.

Conclusión.

La implementación del protocolo UART para la comunicación serial entre dos dispositivos utilizando el lenguaje VHDL es de sumo aprendizaje, ya que en este pudimos implementar máquinas de estados, así como también contadores para llegar a un resultado adecuado de comunicación. Se tiene que mencionar la importancia que tiene la comunicación entre dispositivos dado a esto, se debe de tener un conocimiento claro de la implementación de un protocolo destinado a esto. Es interesante y de gran valor el conocer como la arquitectura de una máquina de estados en VHDL dado que se utilizaron distintas directivas y tipos de datos destinados a un propósito, así como también su correcto uso y adicionalmente poder traducir un mapa de estados a un código funcional, nos permite conocer mejor su funcionamiento.

Bibliografía consultada

- 3.4.2 Estándar RS232. (s. f.). Recuperado de http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro27/342_estndar_rs232.html
- Cómo funciona el Puerto Serie y la UART. (2017, octubre 31). Rincón Ingenieril. https://www.rinconingenieril.es/funciona-puerto-serie-la-uart/
- UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices. (s. f.). Recuperado de https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html
- Los UART garantizan comunicaciones industriales confiables de larga distancia a través de las interfaces RS-232, RS-422 y RS-485. (s. f.). Recuperado de https://www.digikey.com.mx/es/articles/uarts-ensure-reliable-long-haul-industrial-communications
- International, R. & S. (s. f.). Entendiendo el UART. Recuperado de https://www.rohde-schwarz.com/lat/productos/prueba-y-medicion/osciloscopios/educational-content/entendiendo-el-uart_254524.html

Anexo: códigos VHDL

En la siguiente liga se puede encontrar el código completo del UART, tanto como código del transmisor y emisor. También los testbench utilizados para la simulación y prueba de estos.

https://github.com/lapr98/Evidencia_1.git