

# Arquitectura de Computadoras

## TP2: UART

---

Tomás Martín - 39326227

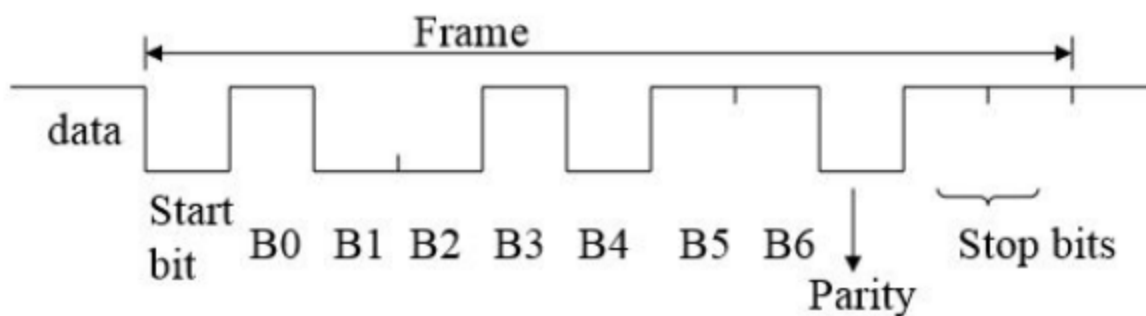
[https://github.com/tomasmartin27/TP2\\_UART.git](https://github.com/tomasmartin27/TP2_UART.git)

4 de noviembre del 2021

## Consigna

El trabajo consiste en crear un módulo de comunicación UART que se conectará a la ALU para realizar operaciones con los datos que ingresan al receptor de la UART y transmitir el resultado por el transmisor.

UART (Universal Asynchronous Receiver-Transmitter) es un protocolo de comunicación asíncrono ya que transmite información de sincronismo en el mismo cable que transmite, no es necesario otro cable para la señal de clock. Es necesario que el transmisor y el receptor sepan de antemano con qué frecuencia se va a transmitir para que el receptor conozca el tiempo que tiene que esperar entre un bit y otro.



La trama tiene un bit de Start (siempre cero) y uno o dos bits de Stop (siempre uno) para indicarle al receptor cuando comienza a recibir algo y cuando termina.

Además de saber de antemano la frecuencia de transmisión, ambos (transmisor y receptor) deben saber si la trama va a tener bit de Paridad y la cantidad de bits de Stop.

El bus de recepción está siempre en alto, al llegar un cero (bit de Start), el receptor se prepara para recibir una trama, empezando de esta forma el sincronismo.

Luego del bit de Start va a recibir entre 6 a 8 bits (tamaño de la trama), y también recibirá el bit de Paridad en caso de que este esté activado.

El bit de Paridad es igual a 1 cuando la cantidad de unos de la trama es impar e igual a 0 cuando la cantidad de unos es par. Este bit se utiliza para control de errores de la siguiente forma: si es igual a 1 y la cantidad de unos en la trama es par, puede ser que se haya recibido uno o varios bits de la trama con el valor incorrecto, o que el bit de Paridad sea incorrecto, en cualquiera de

los casos la trama se descarta. Por otro lado, si llega un 0 y la cantidad de unos en la trama es impar, se considera incorrecto por lo mismo que se mencionó en el caso anterior y la trama se descarta.

Para finalizar la recepción, si llega el bit o los bits de Stop correctamente es porque todo llegó en sincronismo, pero si tengo un cero en alguno de los casos, lo que llegó no es confiable y se descarta.

Se debe generar un módulo UART que tenga tanto receptor como receptor. El transmisor tiene que generar la trama, la salida va a ser un bit que va a mandar los datos serializados. El receptor va a tener un bit de entrada esperando por un bit de Start, va a decodificar la trama y generar el dato recibido.

Este módulo UART también debe generar una señal de sincronismo llamada ticks.

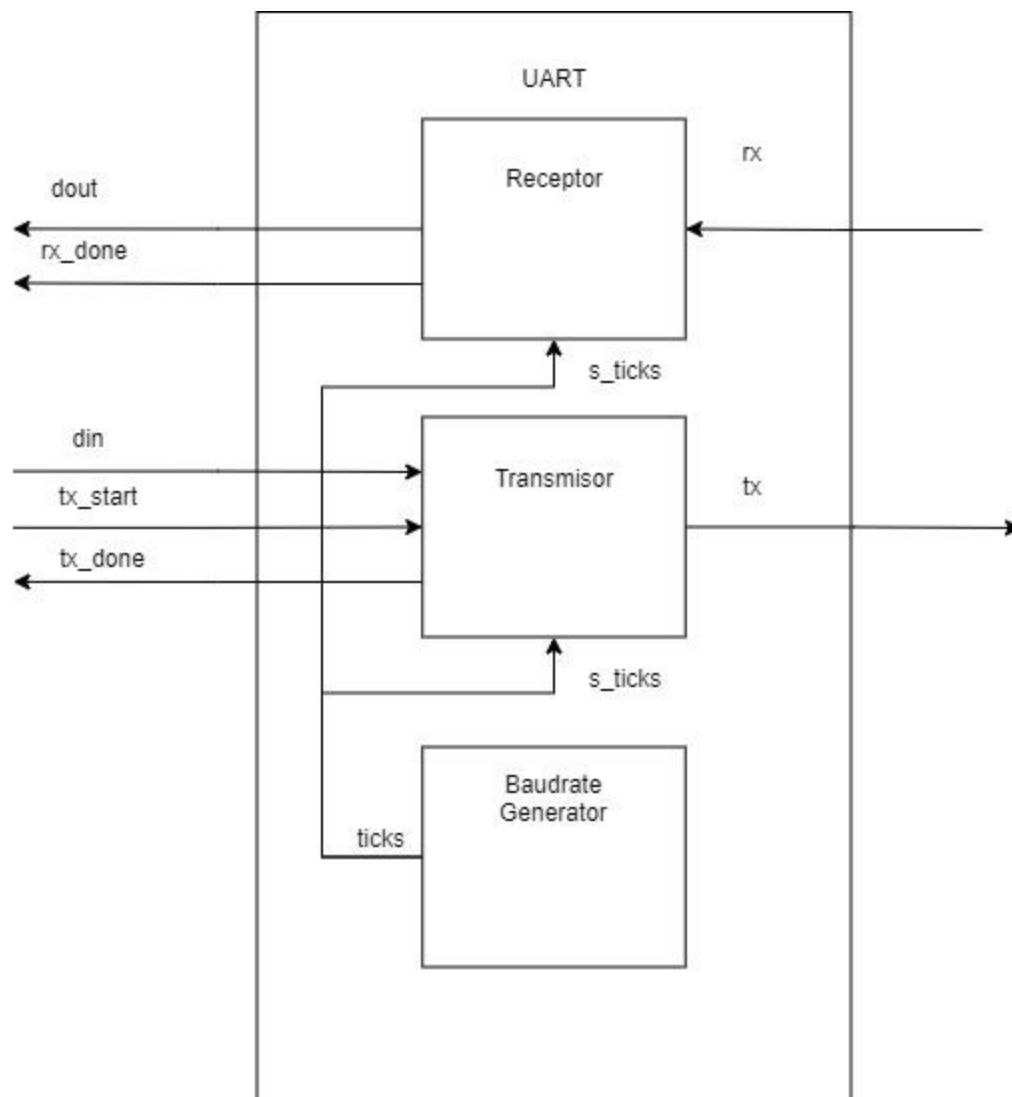
$$\frac{CLK}{Baudrate \cdot 16} = n$$

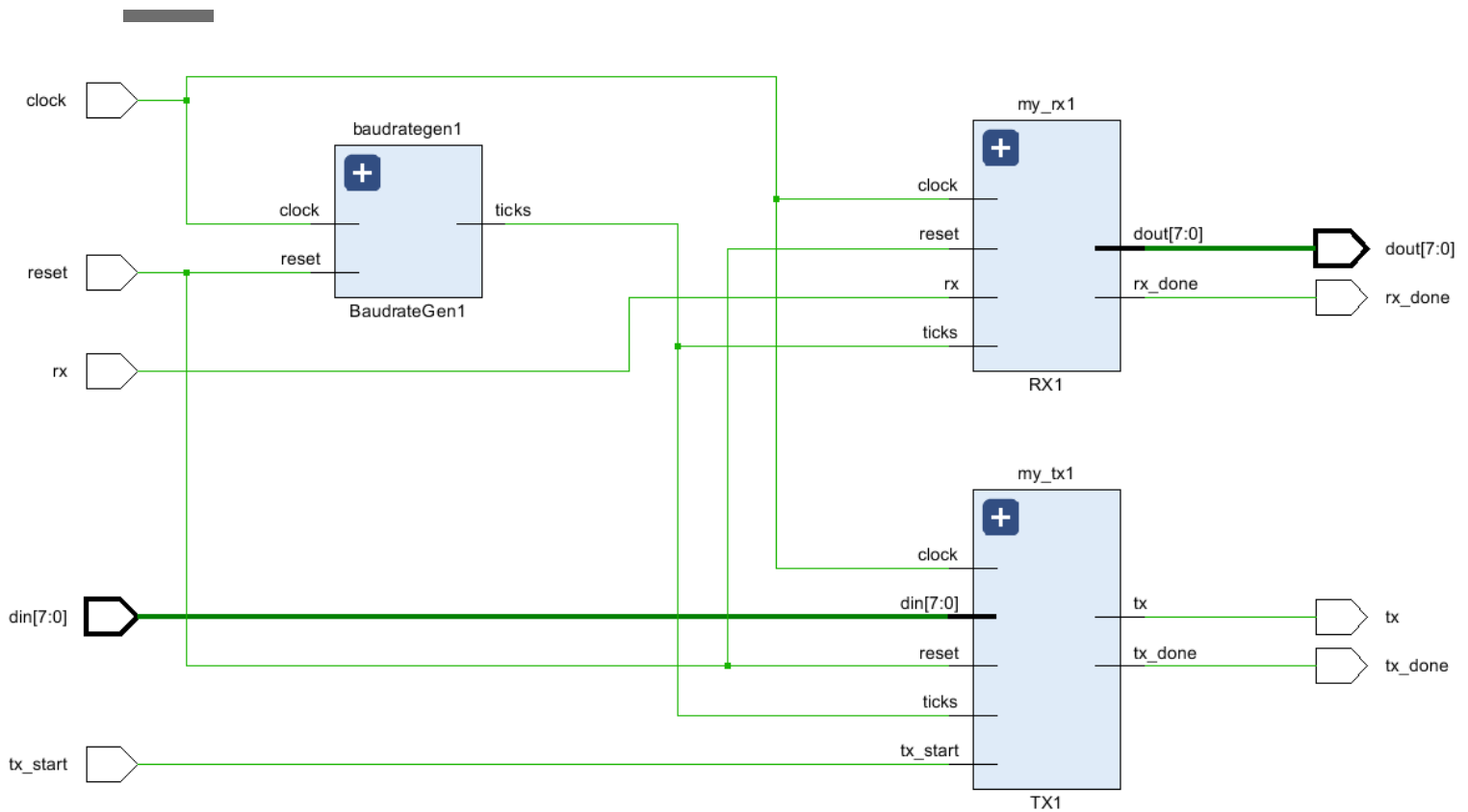
Donde n es la cantidad de ciclos de reloj que equivalen a un tick.

El generador de ticks es un contador de ciclos de clock que al llegar al valor n manda un pulso.

El ancho de un bit en la trama es de 16 ticks.

## Módulo UART





El módulo UART consiste en un transmisor, un receptor y un baudrate generator encargado de generar los ticks.

Sus entradas son:

- `clock`.
- `reset`.
- `din`: es la trama a transmitir que ingresa de forma paralela para ser transmitida en serie.
- `tx_start`: señal encargada de indicar cuándo iniciar la transmisión.
- `rx`: señal donde ingresa la trama serializada para luego decodificarla.

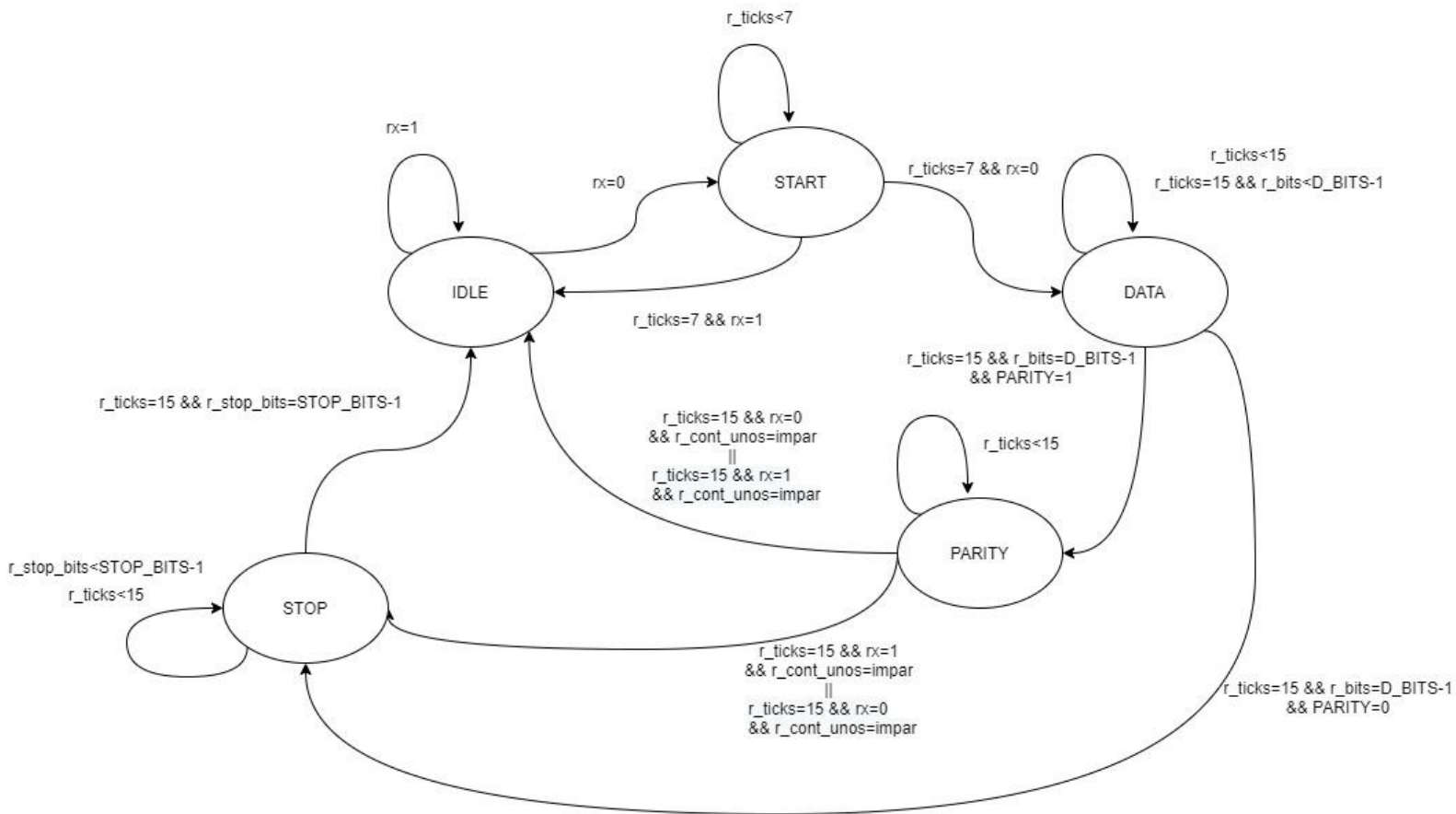
Sus salidas son:

- `dout`: salida en forma paralela del valor obtenido por `rx`.
- `tx`: señal que manda la trama ingresada por `din` de forma serial.
- `rx_done`: indica cuando el receptor terminó de recibir una trama de forma correcta.
- `tx_done`: indica cuando el transmisor terminó de enviar una trama de manera exitosa.

Para todo el desarrollo del trabajo se utilizó un clock de 50MHz y un baudrate 19200.


$$\frac{50MHz}{19200.16} = 163,163 \text{ ciclos de clocks representan a un tick.}$$

## Máquina de estados del receptor



Arranca en el estado IDLE donde espera la llegada del bit de Start ( $rx=0$ ). Al llegar este bit pasa al estado START, en el cual se cuentan 7 ticks para posicionarse a la mitad del bit de Start y se verifica que este siga siendo cero porque puede haber sido ruido. Si este sigue siendo cero se pasa al estado DATA para iniciar la recepción de la trama, en cambio, si  $rx$  es 1 significa que no era el bit de Start y se vuelve al estado IDLE a seguir esperando por un bit de Start.

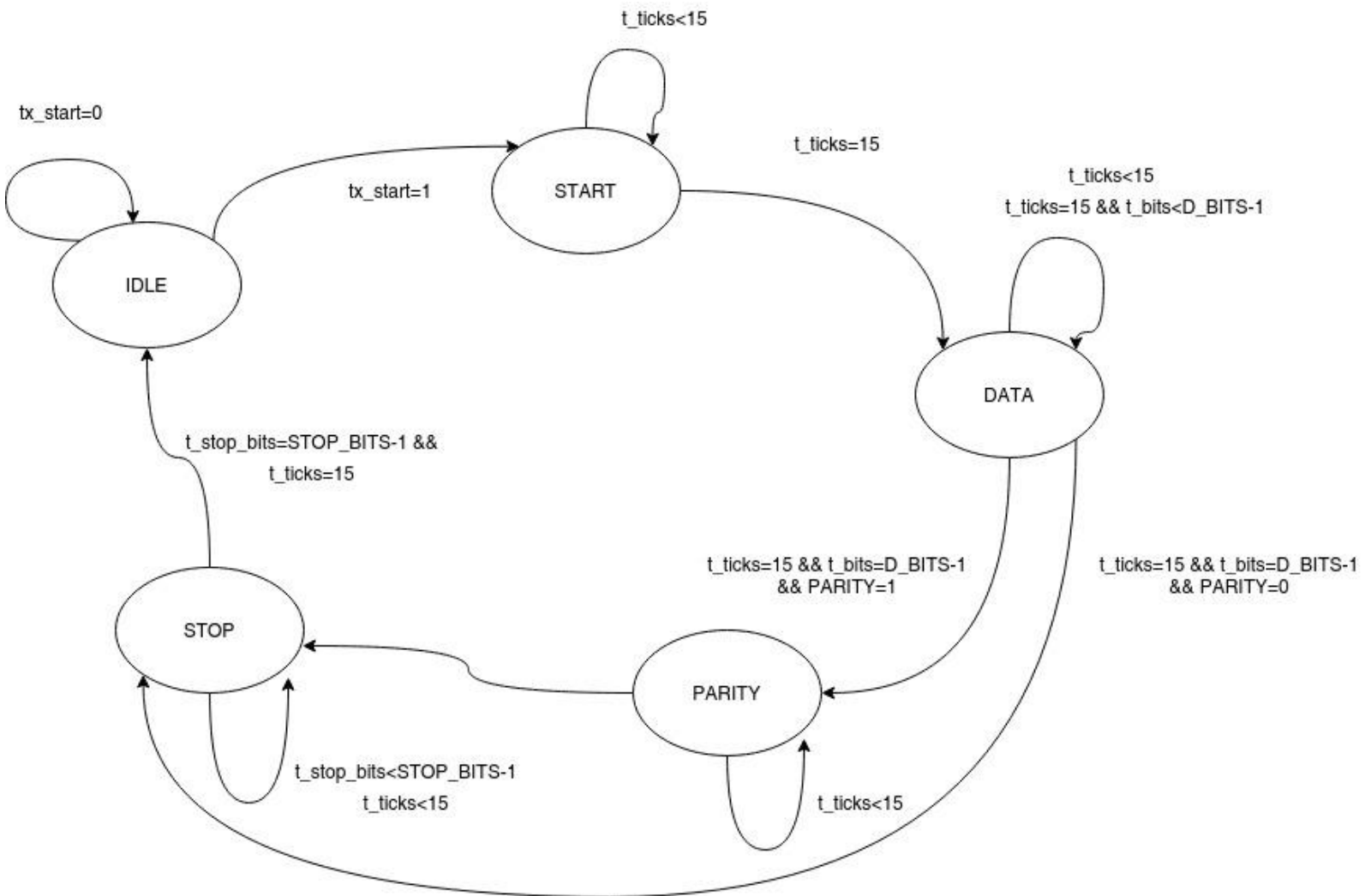
Luego de haber comprobado correctamente el bit de Start, en el estado DATA se esperan 16 ticks para posicionarse a la mitad del primer bit de la trama y guardarlo. Se repite este paso hasta registrar todos los bits de la trama. También se cuenta la cantidad de unos que llegaron para poder verificar la paridad.



Si se tiene configurado el envío y recepción del bit de Paridad (PARITY=1) se pasa al estado PARITY. En este estado se recibe el bit de Paridad esperando 16 ticks para pararse a la mitad de este y se comprueba que sea correcto. El chequeo de la paridad se realiza viendo si la el contador de unos es par o impar y si el bit que llegó es el correcto. Si el contador es par y se recibió un cero, o si el contador es impar y se recibió un uno, entonces la trama se recibió correctamente y se pasa a al estado de STOP, en caso contrario se pasa al estado de IDLE y se descarta la trama.

En el estado STOP se comprueba que lleguen uno o dos bits de Stop, es decir que lleguen uno o dos unos, dependiendo de cómo esté configurado. Si llegan los unos esperados, se pone en alto el bit de rx\_done y se lee el valor de la trama correctamente por la salida en paralelo dout. En caso de que llegue algún cero, la trama se descarta y se pasa al estado IDLE.

## Máquina de estados del transmisor



Se ingresa el valor en paralelo por din y se da inicio a la transmisión con tx\_start. Hasta que esto suceda se encuentra en el estado IDLE, cuando llega el tx\_start se pasa a estado START y se envía el bit de Start por tx.

En el estado START se cuentan 16 ticks correspondientes al bit de Start. Cuando pasan estos 16 ticks, se pasa al estado DATA y se manda por tx el primer bit de din.

Se mandan todos los bits de din cada 16 ticks en el estado DATA y se va contando la cantidad de unos para ver cual tiene que ser el bit de Paridad que se debe enviar, en caso de que esté configurada la paridad. Cuando termina de mandar todos los datos, si el parámetro PARITY es igual a uno, manda el bit de Paridad y pasa al estado PARITY, sino se pasa al estado STOP.

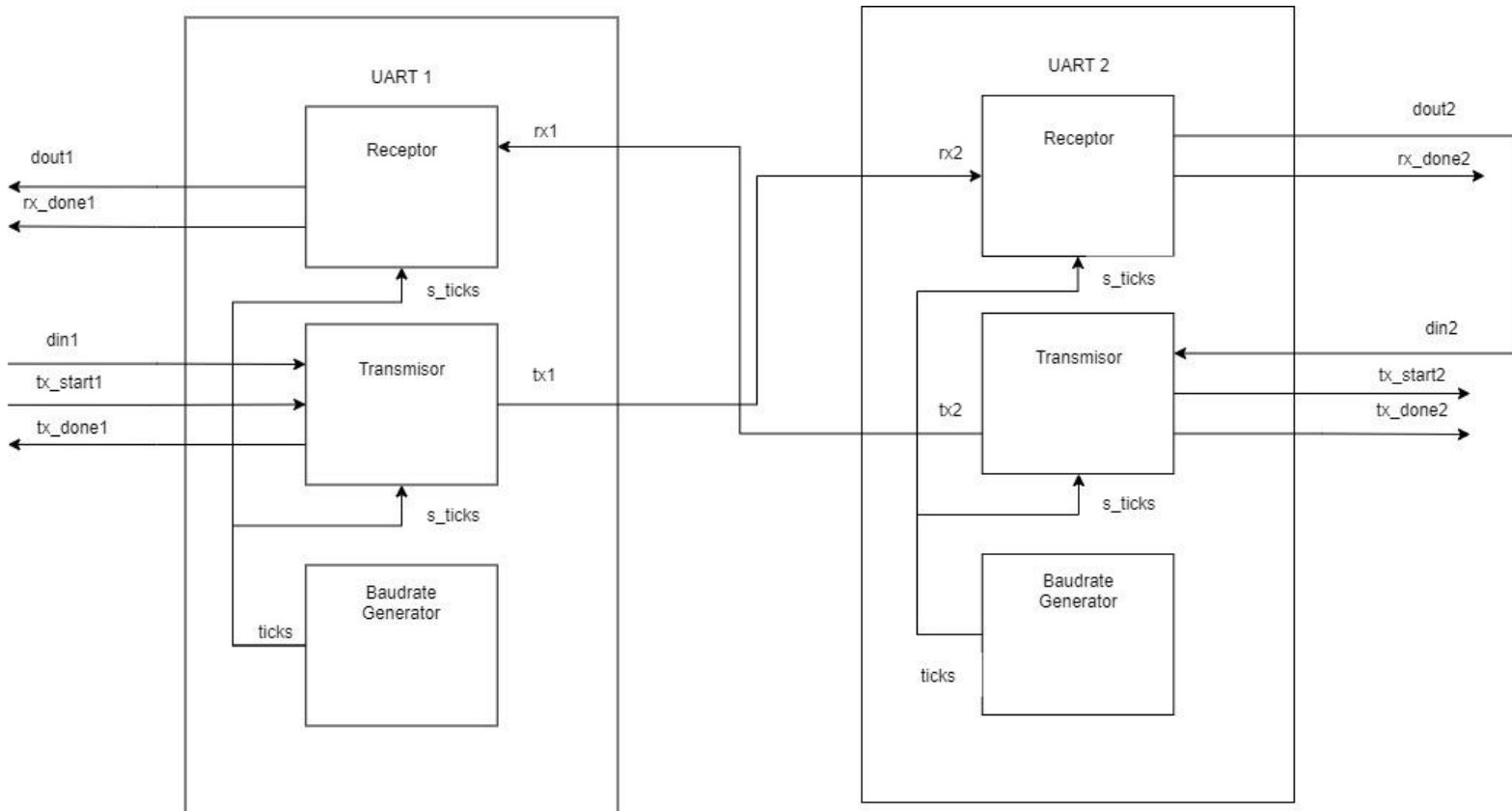
El bit de Paridad que se envía es uno si la cantidad de unos de din es impar y cero si la cantidad de unos es par.



Para finalizar se envían el o los bits de Stop y se pone en alto el bit de tx\_done.

## Testbench del Módulo UART

Para probar este módulo se instancian dos UART y se conectan como indica la figura de abajo.



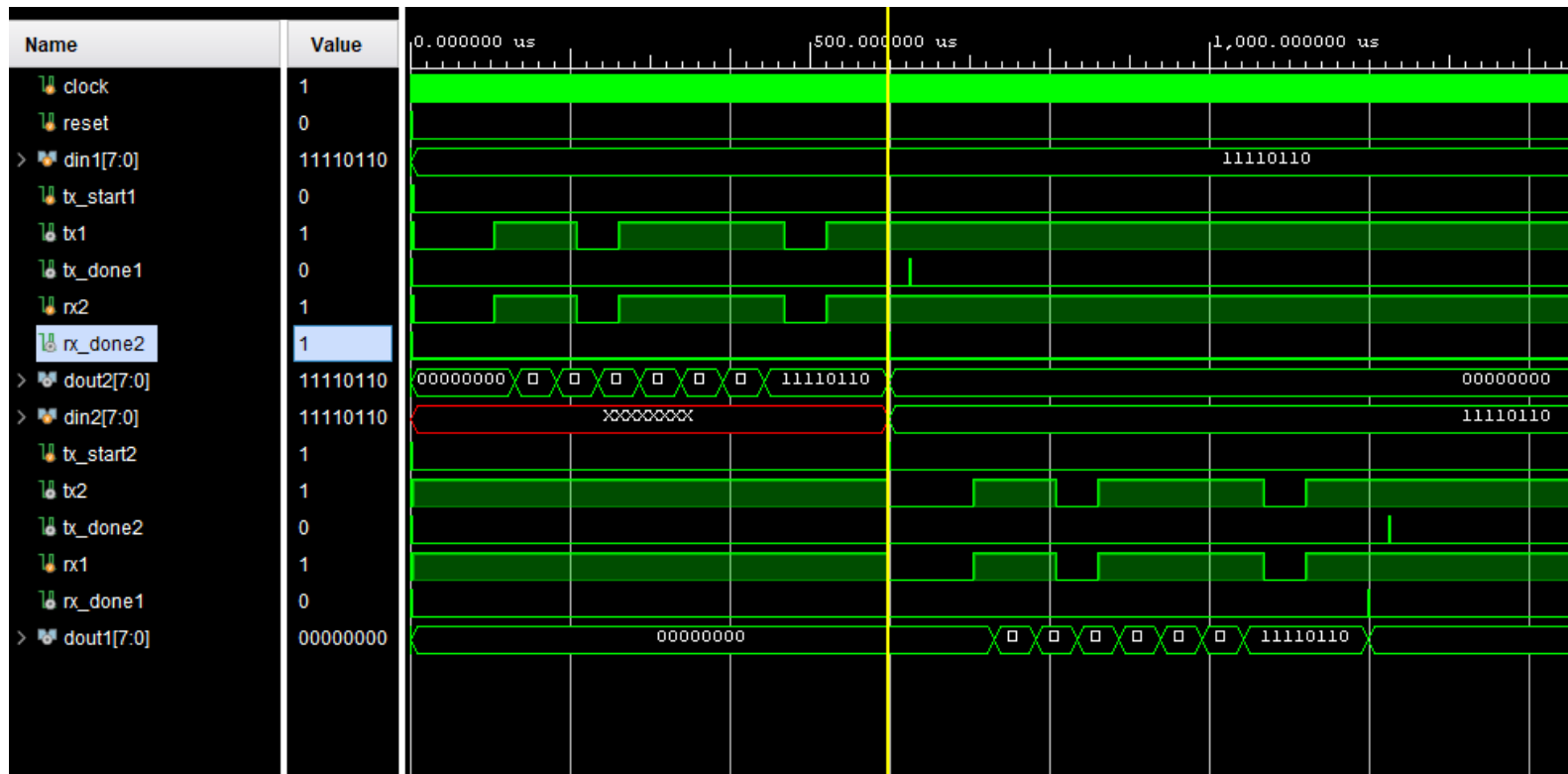
Se ingresa un valor aleatorio por `din1` y se pone en alto `tx_start`, el transmisor de la UART1 va a comenzar a transmitir `din1`. Al tener conectado `tx1` con `rx2`, el receptor de la UART2 comenzará a recibir el valor transmitido por la UART1. Cuando el transmisor1 termina de transmitir correctamente manda un pulso por `tx_done1` y cuando el receptor2 termina de recibir manda un pulso por `rx_done2`, indicando que ya tiene el valor listo en `dout2`,

Cuando `dout2` está listo se lo ingresa por `din2` y se pone en alto `tx_start2` para que el transmisor de la UART2 comience a transmitir y el receptor de la UART1 comience a recibir.

Si todo funciona correctamente cuando se ponga en alto el bit `rx_done1`, el valor `dout1` debe ser igual al ingresado por `din1`.

## Behavioral Simulation del Módulo UART

Se configuró para que tenga bit de Paridad y dos bits de Stop.

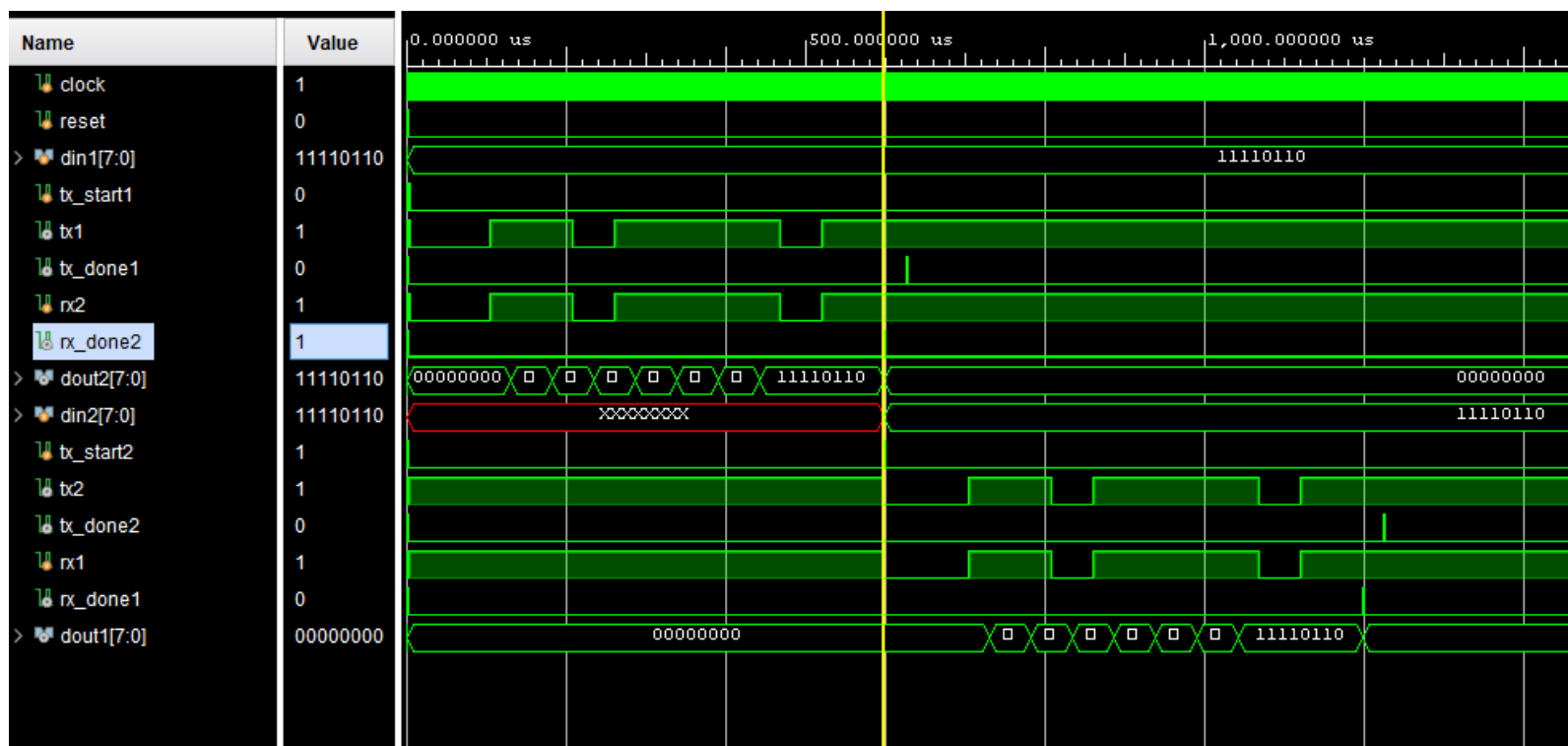


Se puede observar que tx1 comienza a enviar el valor din1 = 11110110. Este arranca enviando el bit de Start, luego envía la trama, el bit de Paridad que es cero porque tiene 6 unos y dos bits de Stop.

El rx2 va recibiendo lo mandado por tx1 y cuando termina (rx\_done2=1) se observa que se tiene el mismo valor que din1 en dout2. A este valor se lo ingresa a din2.

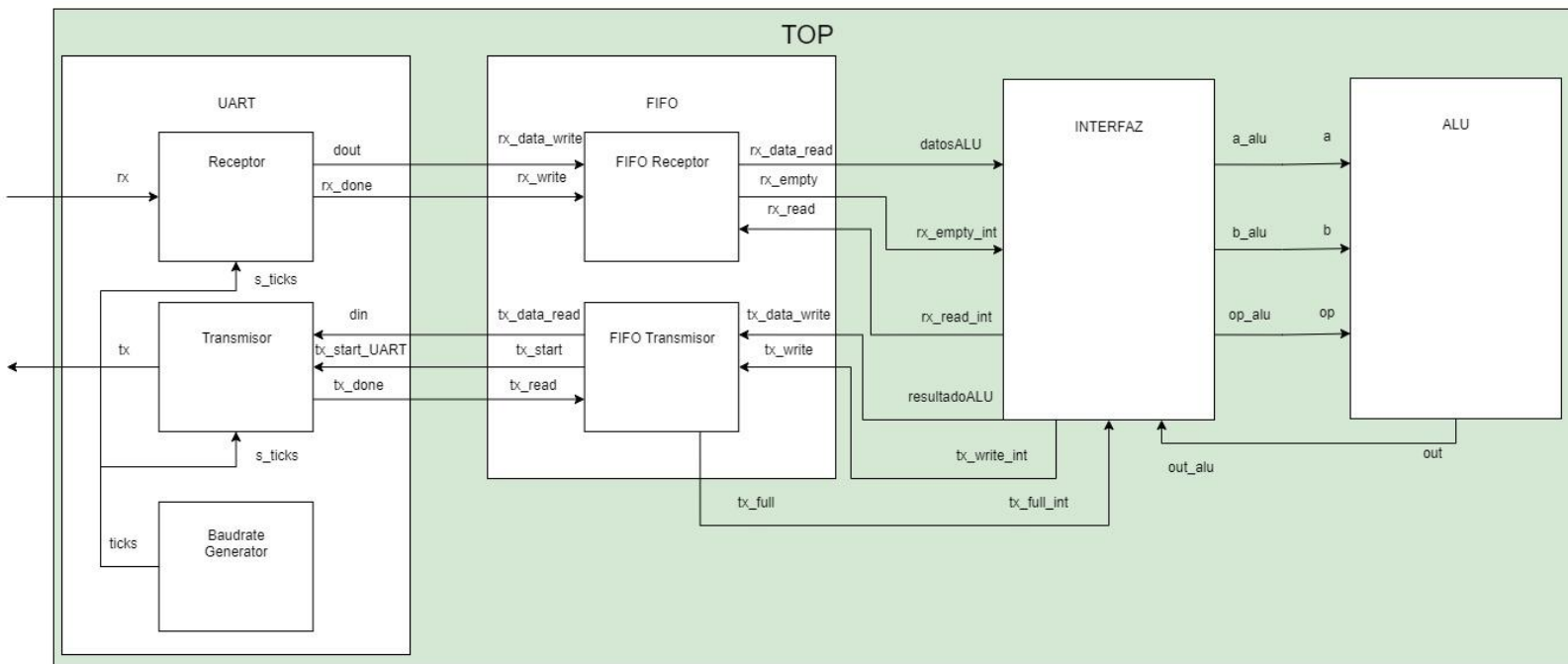
Sucede los mismos pasos anteriores y se obtiene en dout1 el valor ingresado en din1 comprobando el correcto funcionamiento del módulo UART.

## Post-Synthesis Timing Simulation del Módulo UART



De la misma forma que el Behavioral se comprueba el funcionamiento correcto.

## Módulo Top



El módulo Top está compuesto por: un módulo UART, como el explicado anteriormente, un módulo FIFO, un módulo Interfaz y un módulo ALU.

Por rx entran los datos de forma serial para realizar una operación con la ALU, estos datos son el primer operando (a), el segundo operando (b) y el código de operación (op). Cuando el receptor termina de recibir uno de estos datos, pone en alto el bit rx\_done y este le indica a la FIFO del receptor que el dato está listo para ser escrito.

Cuando la FIFO del receptor tiene escrito algún valor, indica con un cero en el bit rx\_empty que no está vacía. Al indicar que tiene un valor, la interfaz lo lee para asignarlo al dato a, b u op según corresponda.

Los valores de la operación son pasados a la ALU para que realice la operación y devuelva el resultado que será capturado por la Interfaz.

Luego que la Interfaz tiene el valor del resultado, lo escribe en la FIFO del transmisor, si no se encuentra llena. Si está llena, la interfaz espera que se vacíe para escribir el resultado. De todas formas como el receptor tiene que recibir tres datos para la siguiente operación y el transmisor solo tiene que mandar uno, no habrá problemas de que se llene la FIFO del transmisor.

Una vez que la FIFO del transmisor tenga un valor escrito, pone en alto el bit tx\_start para que el transmisor comience a transmitir el valor que está primero en la FIFO. Cuando termina de

transmitir el valor, lo indica con el bit tx\_done que marca como leído a ese valor en la FIFO y esta apunta al siguiente valor.

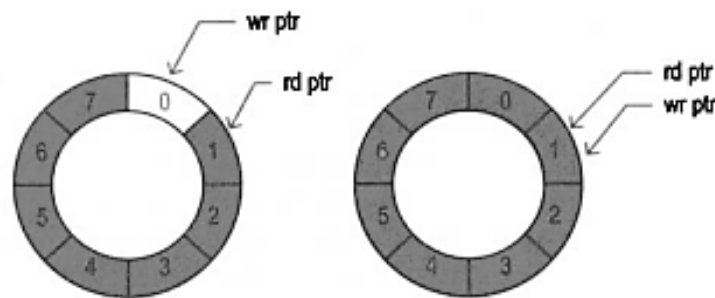
Las entradas de este módulo son:

- clock.
- reset.
- rx: entrada serial para recibir los datos.
- tx: salida serial para obtener el resultado.

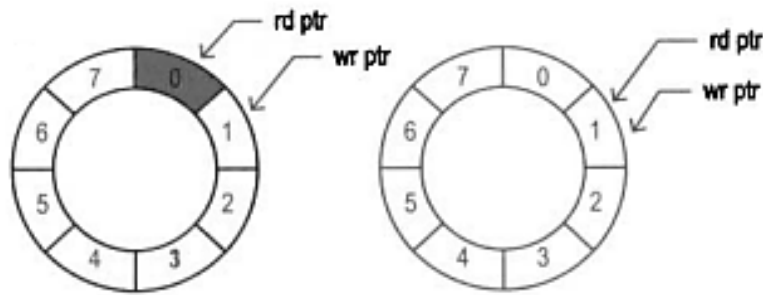
## Módulo FIFO

El módulo FIFO consiste en dos FIFOs una para el receptor y otra para el transmisor.

Estas FIFOs se tratan de buffers circulares.



Si llega un write y la FIFO no está llena, el valor que entra se guarda en la posición de la memoria indicada por el puntero wr\_ptr actual y se incrementa el puntero en uno para apuntar a la siguiente posición. Si esta posición es igual al puntero actual de lectura, entonces la FIFO se encuentra llena, esto se puede observar en la figura de arriba. También, al momento que se realiza una escritura podemos decir que la FIFO no se encuentra vacía.

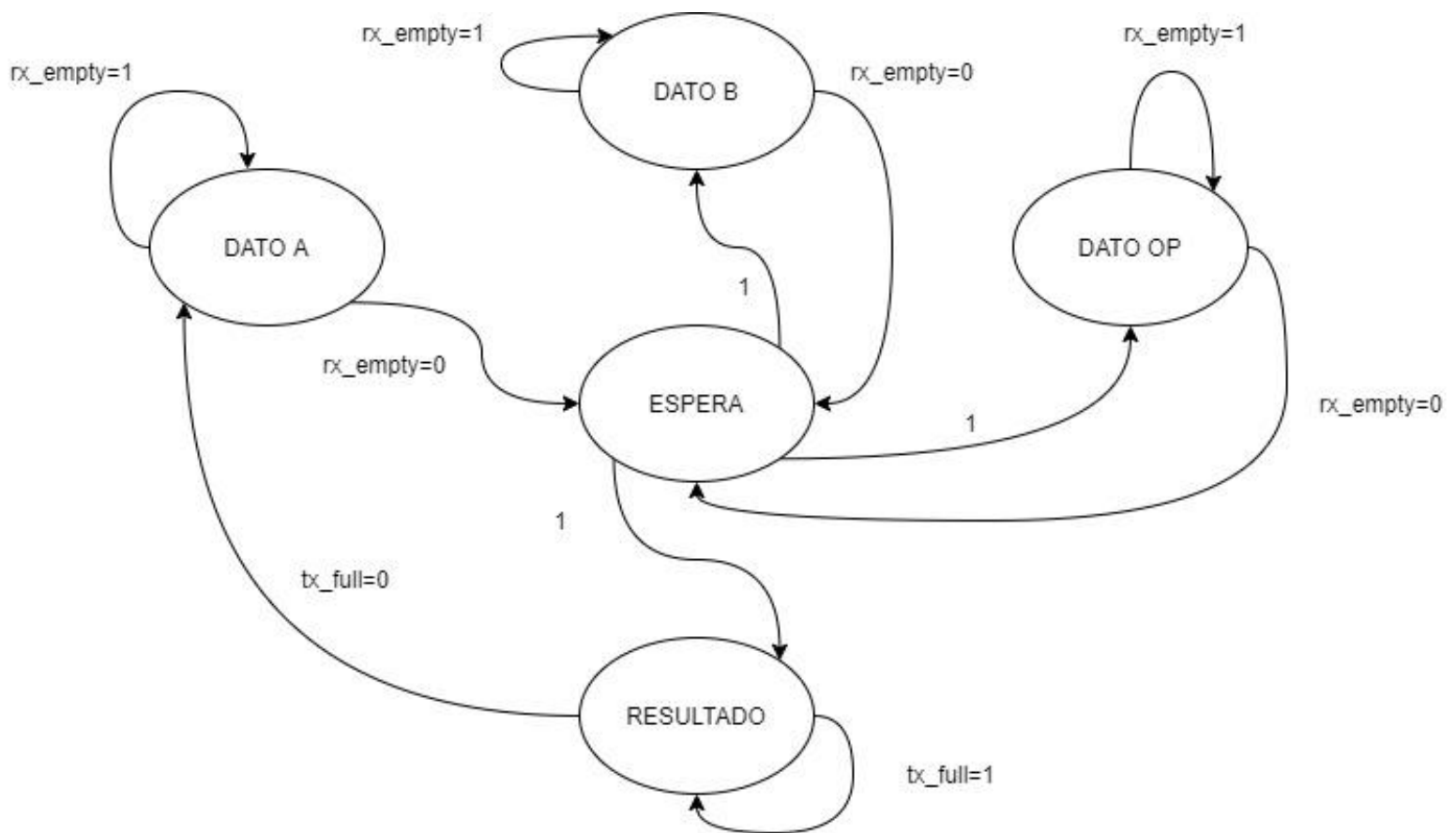


Si llega un read y la FIFO no se encuentra vacía, se lee el valor indicado por el puntero *rd\_ptr* y se actualiza el puntero para que apunte al siguiente incrementándolo en uno. Si el puntero de lectura siguiente es igual al de escritura actual, podemos decir que la FIFO se encuentra vacía, como se puede visualizar en la figura de arriba. Además, si la lectura se puede realizar, se puede indicar que la FIFO no está llena.

## Módulo Interfaz

El módulo Interfaz es el encargado de tomar los datos de la FIFO del receptor y de escribir el resultado en la FIFO del transmisor.

## Máquina de estados de la interfaz



Se arranca en el estado DATO A, en el cual se queda si la FIFO del receptor se encuentra vacía ya que no hay ningún valor para tomar. Si hay un valor disponible, se lo asigna al dato a y se marca como leía la FIFO. A medida que se guardan los datos se pasan al módulo ALU.

Luego se pasa siempre al estado ESPERA para que se espere un ciclo de clock y le dé tiempo a la FIFO a apuntar al siguiente valor. Si no se espera este ciclo de clock y se pasa a tomar el dato b este guarda el valor anterior porque no se actualizó el puntero de lectura.

Después del estado de ESPERA se pasa al estado DATO B para guardar el dato b, si la FIFO del receptor está vacía se espera por el dato, y por la misma razón que se mencionó anteriormente se pasa al estado de ESPERA.

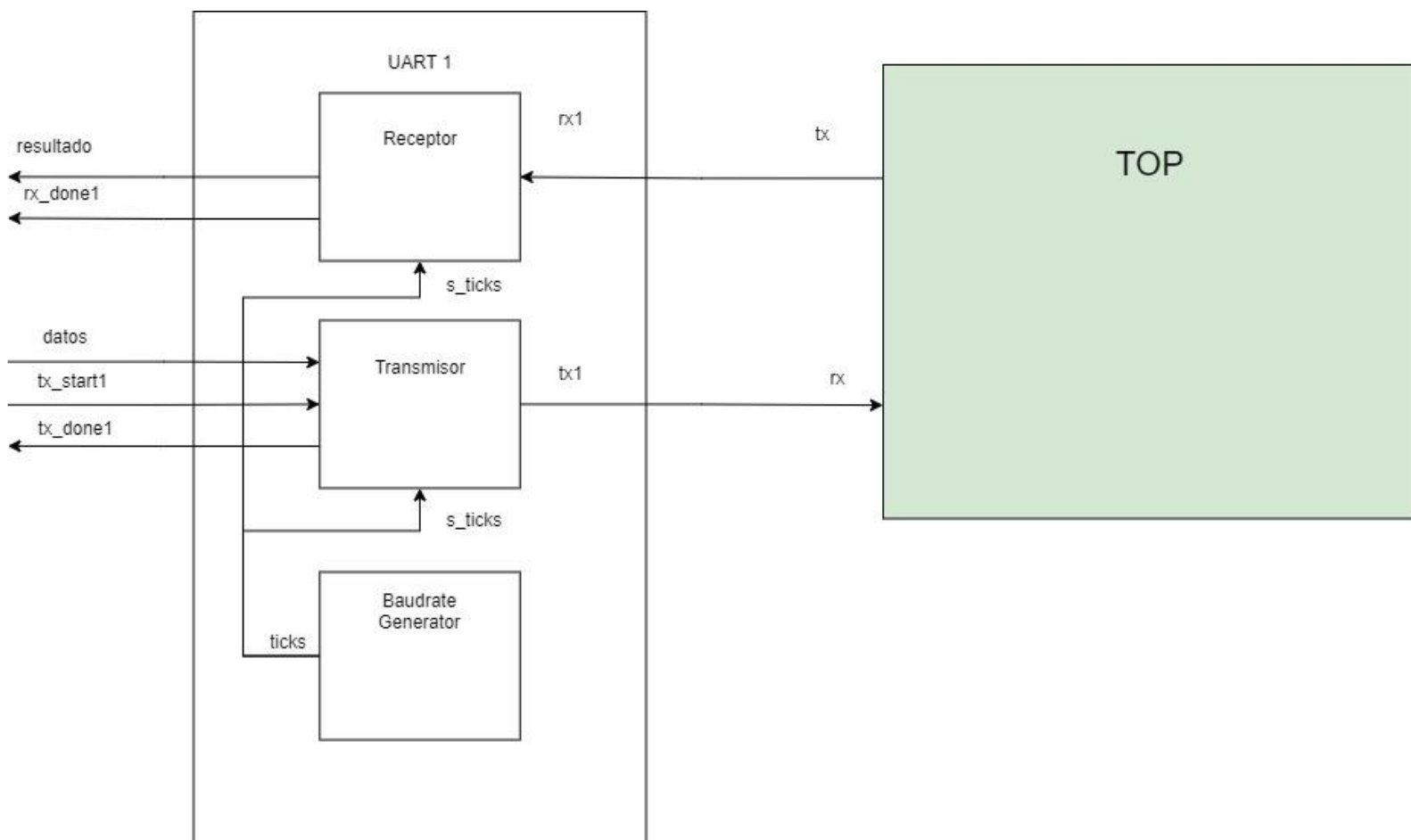
De la misma manera que los otros se guarda el dato op, para luego pasar al estado ESPERA y darle un ciclo de clock para que el resultado que viene de la ALU sea el correcto.

Para finalizar se pasa al estado RESULTADO, donde, si la FIFO del transmisor está llena, se espera a que se vacíe para escribir el resultado para ser transmitido. Cuando se logra escribir en la FIFO se vuelve al estado DATO A.

## Testbench del Módulo Top

Para probar el funcionamiento correcto del módulo Top se lo instanció junto con un módulo UART.

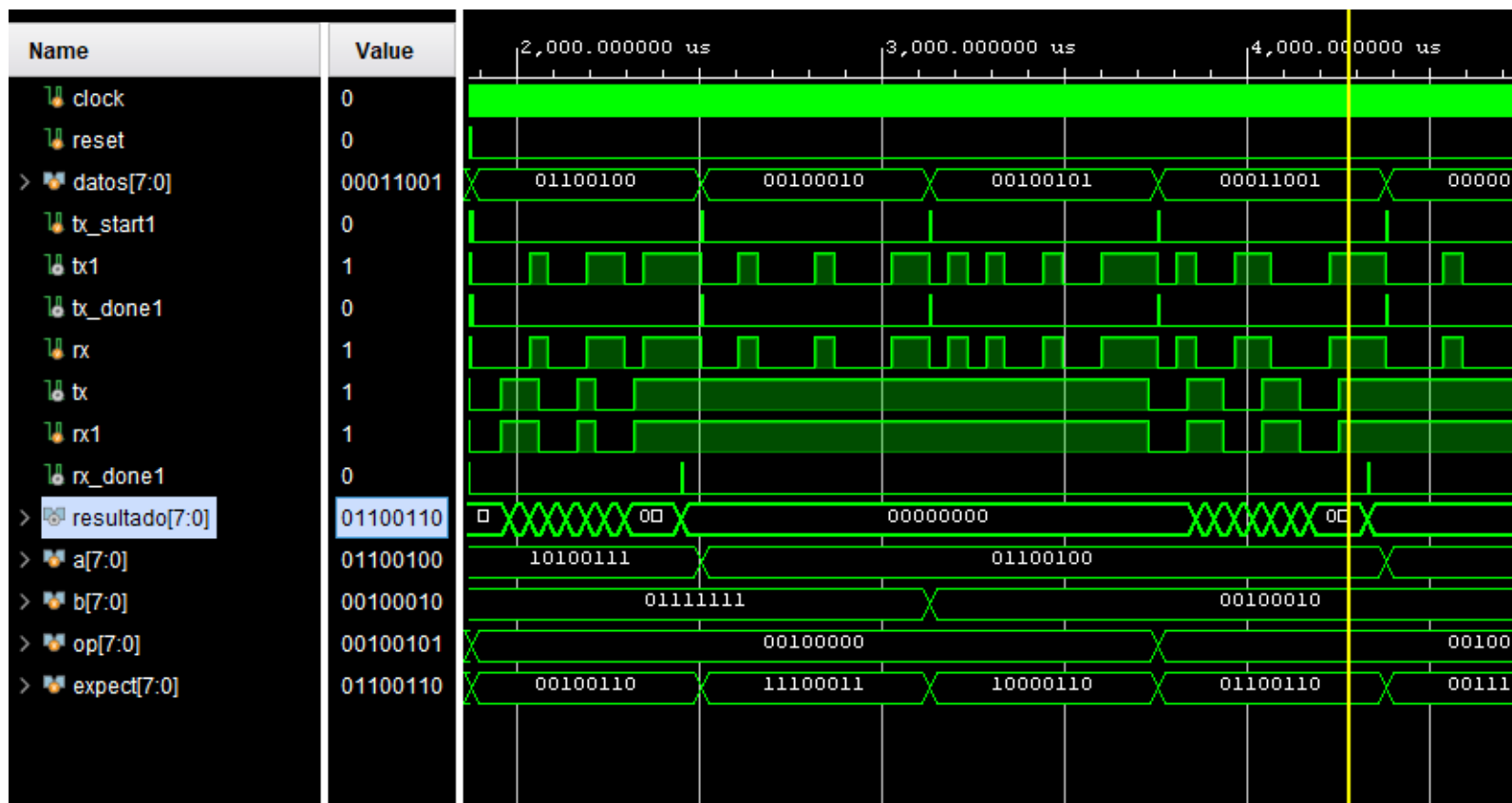
Se ingresan los datos por el transmisor de la UART y se conecta la salida del transmisor tx1 con la entrada rx del Top. También, se conecta la salida del transmisor tx del Top con la entrada rx1 de la UART para recibir el resultado.



## Behavioral Simulation del Módulo Top

Se configuraron las UARTs para que tengan bit de Paridad y dos bits de Stop.





Se ingresa el primer dato (a), generado aleatoriamente, por datos y se comienza a transmitir con el pulso tx\_start1. tx1 arranca con el bit de Start, luego manda la trama 01100100 y finaliza mandando el bit de Paridad, en este caso 1 porque hay tres unos, más dos bits de Stop.

Cuando llega tx\_done1 se genera el siguiente dato (b), también de manera aleatoria, que en el caso que se observa en la figura es igual a 00100010. Al tener dos unos, el bit de Paridad es igual a cero.

Para el tercer dato (op) se elige una operación de manera aleatoria, se observa que se eligió 00100101, la cual es una OR.

Se observa que rx recibe lo mismo que fue mandando tx1.

Los datos se van guardando para calcular el valor esperado (expect) que debería llegar por tx.

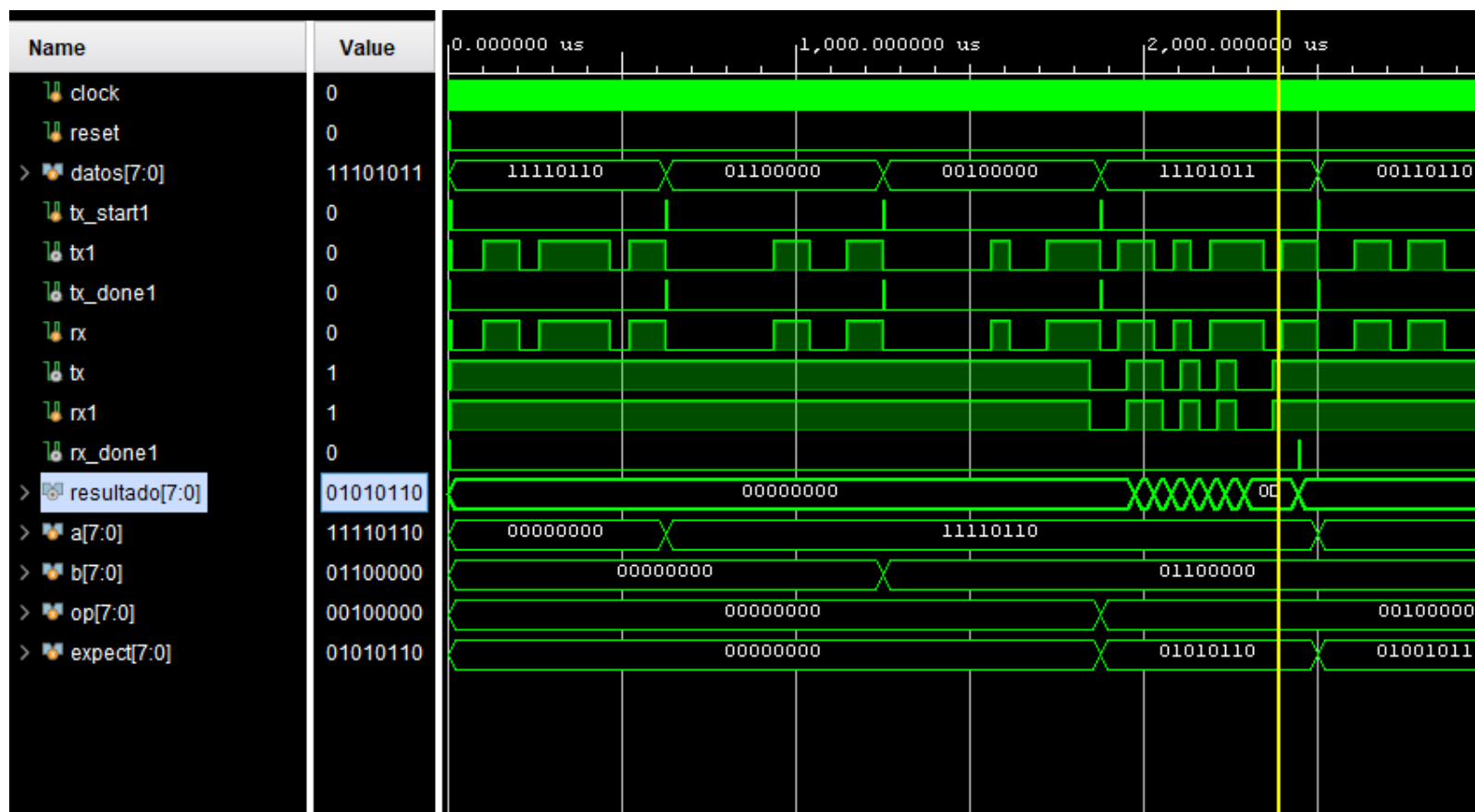
Cuando se ingresan los tres valores, se realiza la operación y tx comienza a transmitir el resultado, que es recibido por el receptor rx1 de la UART instanciada en el testbench.

Cuando el bit rx\_done1 se pone en alto, se compara el resultado con el valor esperado, que se puede comprobar en la imagen que resultado y expect son iguales. Este valor es 01100110 que es el resultado de hacer 01100100 | 00100010.

```
# run 1000ns
datos = xxxxxxxx          0
datos = 10100111          30
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Top_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:05 . Memory (MB): peak = 3042.207 ; gain = 0.000
run 12 ms
datos = 01111111          625950
datos = 00100000          1251870
datos = 01100100          1877790
resultado = 00100110       2451550
datos = 00100010          2503710
datos = 00100101          3129630
datos = 00011001          3755550
resultado = 01100110       4329310
datos = 00000100          4381470
datos = 00000011          5007390
datos = 11101001          5633310
resultado = 00000001       6207070
datos = 10100110          6259230
datos = 00100111          6885150
datos = 10011110          7511070
resultado = 00010000       8084830
datos = 10010000          8136990
datos = 00100111          8762910
datos = 01000101          9388830
resultado = 01100001       9962590
datos = 01100001          10014750
datos = 00000011          10640670
datos = 00010110          11266590
resultado = 00000000       11840350
Test Pass
```

Luego de haber pasado cierto tiempo de simulación y si todos los resultados fueron iguales al expect, se imprime por consola Test Pass, indicando que todos los resultados fueron correctos.

## Post-Synthesis Timing Simulation del Módulo Top



De la misma forma que en el Behavioral se comprueba que todo funciona como es esperado.

En la imagen tenemos la siguiente operación:  $11110110 + 01100000 = 1\ 0101\ 0110$ , que es igual al resultado y al expect sin el bit más significativo ya que todo está configurado con un tamaño de 8 bits ( $D\_BITS = 8$ ).

██████████

```
# run 1000ns
datos = xxxxxxxx          0
datos = 11110110          30
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Top_tb_time_synth' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:09 ; elapsed = 00:00:12 . Memory (MB): peak = 3042.207 ; gain = 8.590
run 11 ms
datos = 01100000          625950
datos = 00100000          1251870
datos = 11101011          1877790
resultado = 01010110       2448290
datos = 00110110          2503710
datos = 00100111          3129630
datos = 11010100          3755550
resultado = 00000000       4326050
datos = 00101110          4381470
datos = 00100100          5007390
datos = 11011101          5633310
resultado = 00000100       6203810
datos = 00000010          6259230
datos = 10101100          7511070
resultado = 00110111       8081570
datos = 11011101          8136990
datos = 00100110          8762910
datos = 10000111          9388830
resultado = 01110001       9959330
datos = 11111010          10014750
datos = 00100100          10640670
run: Time (s): cpu = 00:01:54 ; elapsed = 00:04:16 . Memory (MB): peak = 3042.207 ; gain = 0.000
run 1 ms
datos = 10010001          11266590
resultado = 10000010       11837090
Test Pass
```

Se observa que todos los resultados fueron correctos.