HY430 - Digital Systems Lab

Lab assignment 2

13/11/2019

Βαγενάς Αναστάσης

This report describes the implementation of the 2nd laboratory work where it is divided into 4 parts and in each of them we show its implementation in code / logic, the verification of its operation and the results of the experiment on FPGA.

Prelude

The aim of the laboratory exercise is the implementation of a serial communication system, which will implement the UART protocol. The system will consist of a Sender and Receiver, who will transfer data from the first to the second through a serial connection of a signal.

Part A

Implementation/Verification

In part A we are called to implement a Baud Rate controller which will provide the appropriate sampling signal depending on the selected Baud Rate, ie the transmission rate of the bits.

So we want to find the number of clock cycles that corresponds to T_sc = $\frac{1}{16\text{xBaud Rate}}$ so we can say that we want to find the ratio λTclk = T_sc where T_sc changes each time and where Tclk the FPGA clock period (2Ons) .

So if we want to find eg λ for Baud Rate = 300bits/s ,then T_sc = 1/4800 and Tclk = 20ns we have λ = $\frac{50000000}{4800}$ = 10.416,666 (infinitely). But λ is not an integer we need, because it is a maximum value of a counter that only works with integers.

For this reason we round to the nearest integer and we have λ = 10.417. In addition, the error will be essentially the absolute difference between the initial and final value with respect to the initial value of λ. So we have :

$E_{err} = \frac{|10.416,666 - 10.417|}{10.416.666} = 3.2*10^{-5}$

We do the same for the other Baud Rates and we have the following table:

Baud Rate	Counter Max Value	Error
300	10417	$3.2*10^{-5}$
1200	2604	$6.4*10^{-5}$
4800	651	$6.4*10^{-5}$
9600	326	0.001472
19200	163	0.001472
38400	81	0.004672
57600	54	0.004672
115200	27	0.004672

(The above values are calculated in detail in the excel document that is in the PartA folder)

As it seems, it is logical with the increase of the Baud Rate to have an increase of the error since with the rounding that we apply the relative difference starts and increases, with the result that the ratio itself increases.

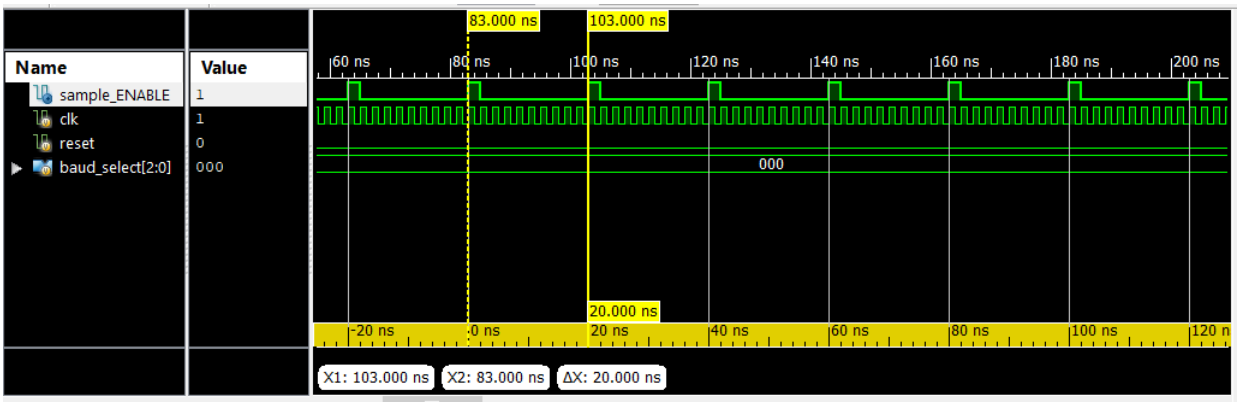
In the implementation code section below we always use a section which based on the value of COUNT_MAX_xxx and the Baud Rate we have chosen to give us the sample_ENABLE.

In the code, however, the value of each COUNT_MAX_xxx will be reduced by one, compared to what we calculated, since we also count for the value 0 of the meter which adds another cycle.

```
module baud_controller(reset,clk,baud_select,sample_ENABLE);
.....
always @(posedge clk or posedge reset) begin
  if(reset == 1'b1) begin  //signal not stable yet or 0
     cycle_counter<=0;
     sample_ENABLE<=0;
  end
//we reached designated Maxed out counter for this baud rate
  else if(baud_select == 3'b000 && cycle_counter == COUNT_MAX_000) begin
     cycle_counter<=0;
     sample_ENABLE<=1;
  end
……
  else
    cycle_counter <=cycle_counter +1;   //counter hasn't reached max yet
.....
endmodule
```

To simulate and verify the circuit we create a testbench which displays the baud_controller module and sets the clk, the reset and a specific baud rate. For convenience we use a random baud rate where we have its COUNT_MAX set to 10.

Παρακάτω εικόνα της προσομείωσης:

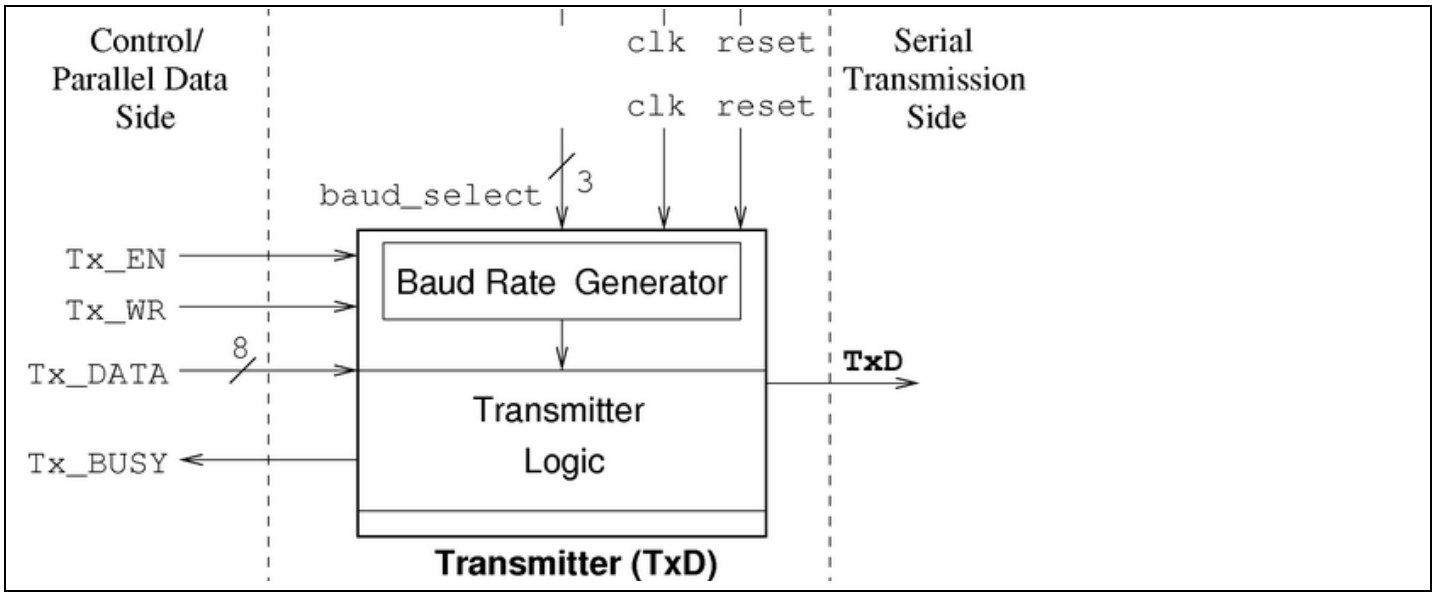


In the image of the simulation it appears that sample_ENABLE becomes positive every 10 cycles when the cycle_counter reaches its maximum value. The duration from one sample_ENABLE to the other is 20ns (the clock has a period of 2ns so we have a correct operation).

Part B

Implementation

In part B we are called to create the UART sender based on the specifications requested.



As shown the main signals of the module are TxEN, Tx_WR, Tx_DATA, Tx_BUSY, TxD.

After receiving data from Tx_DATA (from the system Tx_WR being the activation signal) then we raise the flag Tx_BUSY which marks the start of transmission of the sender. To do all this the sender must be active through the flag Tx_EN .

First, we need to receive the data, which is done if the sender is active and available. To do this we use the following condition in the code:

```
always @(posedge clk or posedge reset) begin
.....
 else if(Tx_EN == 1 ) begin
   if(Tx_WR == 1 && Tx_BUSY==0) begin
      DATA_Symbol[8:1] <= Tx_DATA;
      DATA_Symbol[0] <= ^Tx_DATA;
    end
  end
end
```

DATA_Symbol is the temporary register we use to store data and parity bit.

Then we need 3 registers (state, counter, trans_act) of which:

The counter reg is the one that will count 16 cycles from the Baud Rate when the signal Tx_sample_ENABLE is active.

The state is the register based on which we drive the data to TxD properly and keeps the transmission states for each bit we transmit.

Trans_act is the register that with the registration of new data we will have an activation signal for the state finite state machine and will remain active until it changes .

Below the code we just described:

```
  always @(posedge clk or posedge reset) begin
.....
   if(Tx_EN == 0)
       counter<=0;
   else if(Tx_BUSY==0)
       counter<=0;
   else if(Tx_sample_ENABLE==1)
       counter<=counter+1;
.......
end
```

Above is the counter that actually increases only if we have high Tx_sample_ENABLE from the module of baud_generator and the module is in active transmission.

```
always @(posedge clk or posedge reset) begin
  if (reset==1)
    trans_act<=1'b0;
  else begin
    if(Tx_WR == 1 && Tx_BUSY==0 && Tx_EN == 1)
      trans_act<=1'b1;
    else if(state!=IDLE && Tx_WR==0)
      trans_act <=1'b0;
  end
```

Above is the trans_act register activated by a new data transmission from Tx_WR, Tx_DATA.

Finally we have the main part where the data transmission and the situation management takes place:

```
always @(posedge Tx_sample_ENABLE) begin
case(state)
 IDLE:if(trans_act==1) state<=START_BIT;
 START_BIT:if(counter==COUNT_MAX) state<=DATA_8;
.....
end
```

Here is basically a part of the state status machine which is the way we manage the data transmission. Every time the counter becomes COUNT_MAX then we have a change of state.

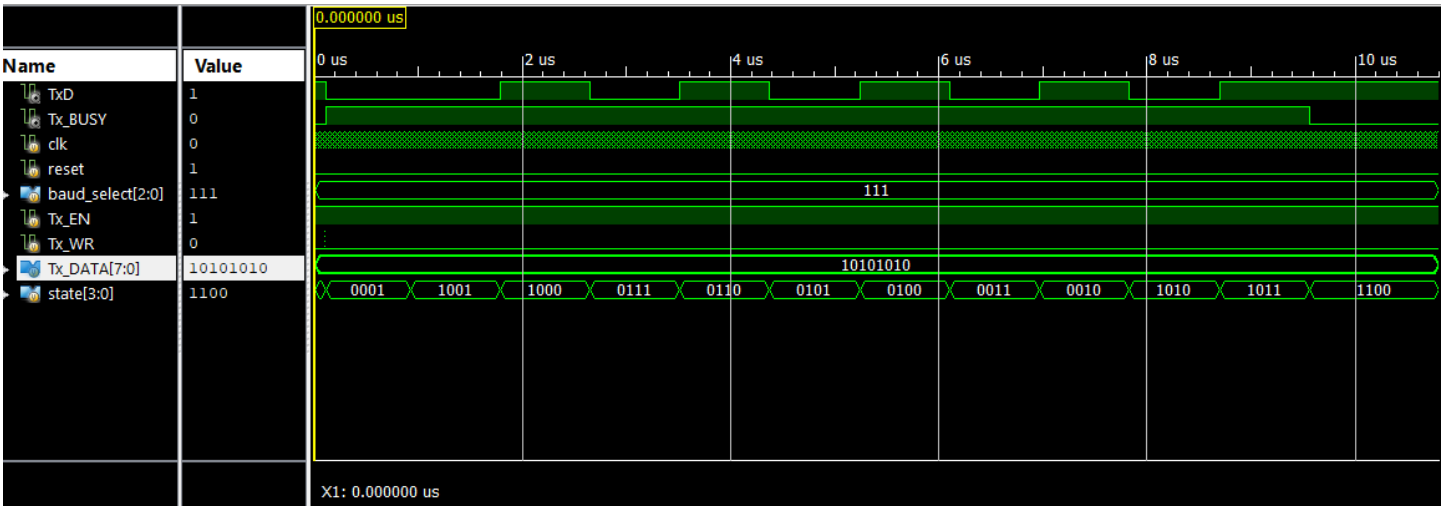
```
always @(state) begin
case(state)START_BIT:TxD<=1'b0;
DATA_8:TxD<=DATA_Symbol[0];
DATA_7:TxD<=DATA_Symbol[1];
.....
end
```

Finally, we have the data transmission that takes place essentially every time we change state in *state* and pass the data to the TxD channel.

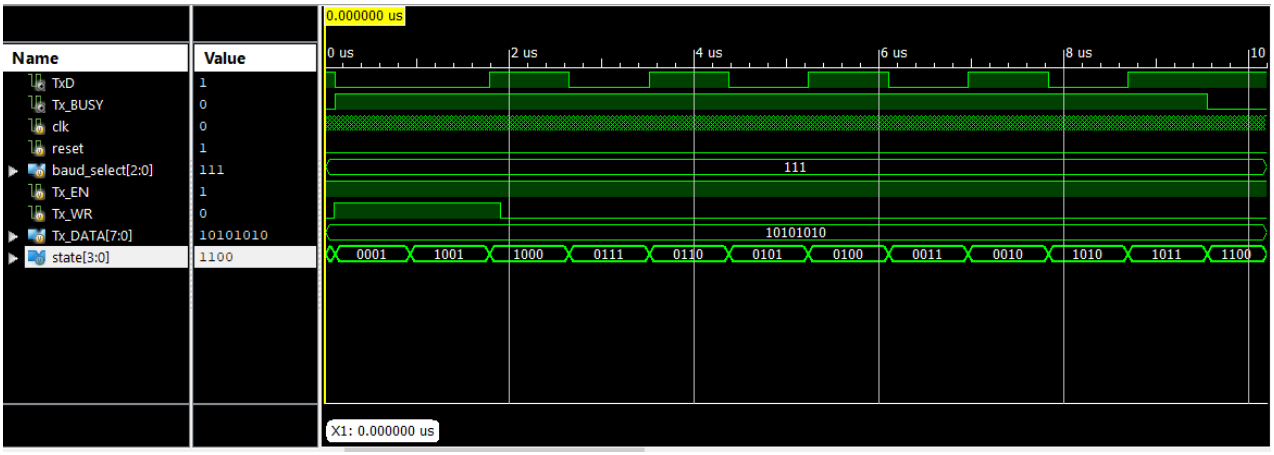
Verification

To verify the UART sender we have used an appropriate testbench that activates it via Tx_EN, provides the Tx_DATA data and writes it to it with the Tx_WR signal.

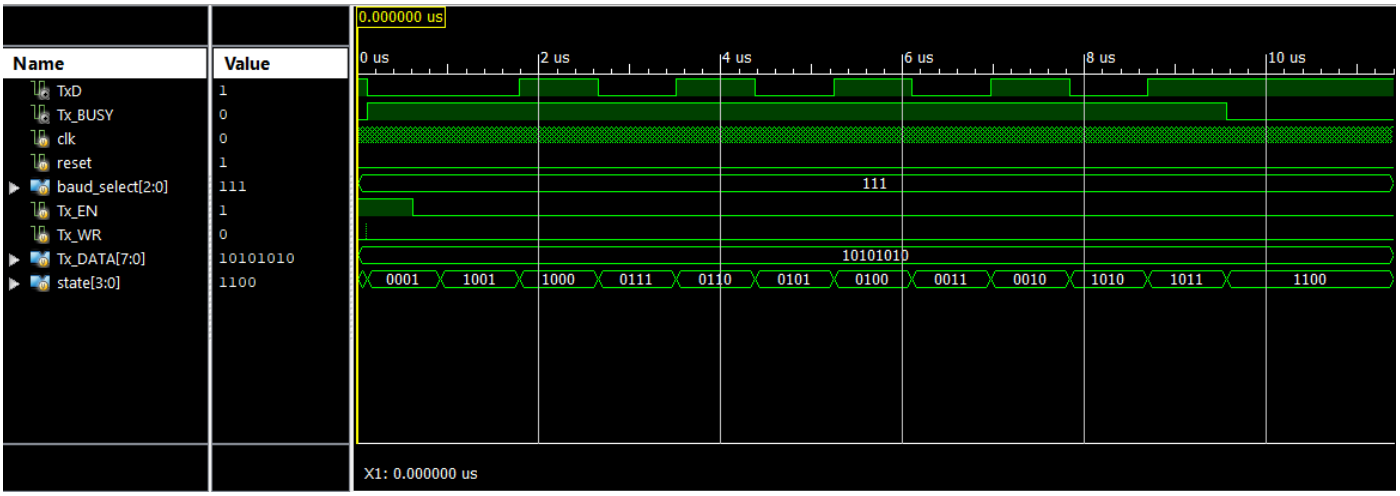
So, in this example, the testbench gives us Tx_DATA = 10101010, ie the AA character.



The simulation image shows the transfer of the character (AA) to the sender's TxD channel. Specifically, the character is inverted since we send from lsb to msb of Tx_DATA that has been given to us. With each status change we send a different appropriate bit. Additionally as the transition progresses it appears that the Tx_BUSY signal remains active until the transfer is complete.



In the next simulation we keep Tx_WR active for more than 1 cycle and all the rest in the same state. We see that the system is not affected due to the signal.



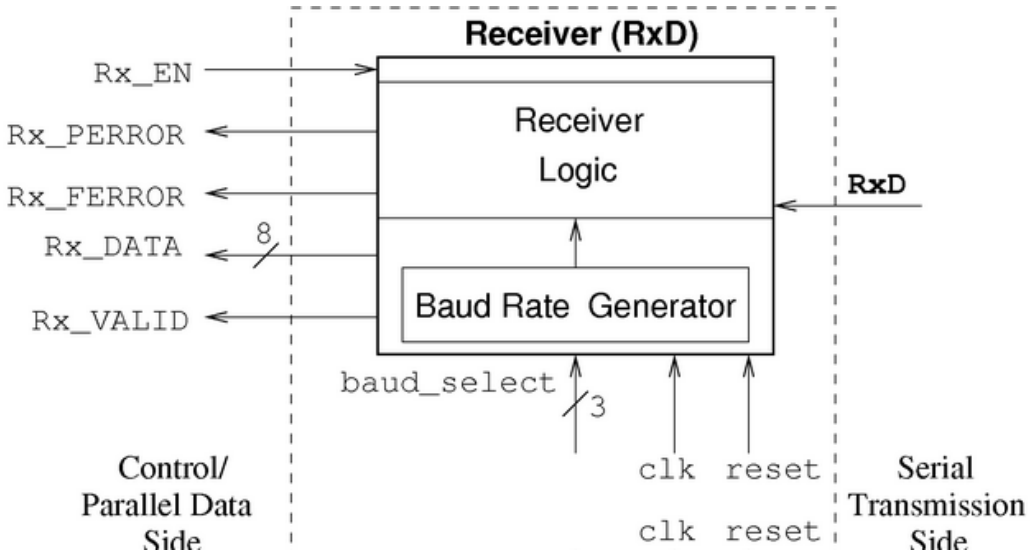
In the next simulation we change the status of Tx_EN and turn off the sender during transmission and see that the system is still not affected and the transmission continues.

Part C/D

Implementation

In this part we implement the UART receiver based on the specifications given to us.

Then we implement the appropriate testbench where we connect the sender and the receiver and check the system.



Above we can see the main signals we are interested in which are Rx_PERROR, Rx_FERROR, Rx_VALID, Rx_DATA and TxD.

We receive the serial data from the RxD channel, which we pass to Rx_DATA when all is received.

At the end of the transmission we check if we received the Stop bit and if the data in Rx_DATA matches the parity bit.

If there is an error we raise to logic 1 Rx_PERROR or Rx_FERROR, depending on the error, at the same time driving Rx_VALID to 0. If there is not then the reverse happens with the error signals remaining at 0 and Rx_VALID going to 1.

In the implementation of the receiver we work with the same situation machine that we had in the transmitter, that is, we will have the same situations since the receiver does the reverse process from the transmitter.

In addition we will use 5 auxiliary registers, 2 counters that we use to change states, one that activates active data reception and 2 more where we save the stop bit and the parity bit.

```
always @(posedge clk or posedge reset) begin
  if(reset==1)
    counter_s<=0;
  else begin
    if(state!=START_BIT)
      counter_s<=0;
    else if(Rx_sample_ENABLE==1)
      counter_s<=counter_s+1;
  end
end
```

Above is the 3-bit counter that we use to do the correct sampling of the start bit. Since we detect a change in state, we start and count 8 cycles to reach the middle of the start bit. After we go to another state the counter is deactivated.

```
always @(posedge clk or posedge reset) begin
  if(reset==1)
    counter<=0;
  else begin
    if(state==IDLE ||state==START_BIT)
      counter<=0;
    else if(Rx_sample_ENABLE==1)
      counter<=counter+1;
  end
end
```

Above the counter we use for the other situations. From the start bit onwards we count 16 cycles to reach the center of the next transmission bit for it and it is 4-bit in size.

```
always @(negedge RxD or posedge Rx_sample_ENABLE) begin
   if(state==IDLE && Rx_EN == 1 && RxD==0)
       trans_act<=1;
   else
        trans_act<=0;
```

```
end
```

Above is the register that detects the negative edge on the RxD channel and activates for exactly 2 Rx_sample_enable pulses, so that we have enough time to go from IDLE to START_BIT state.

```
always @(posedge Rx_sample_ENABLE) begin
                    case(state)
                      IDLE:if(trans_act) state<=START_BIT;
                      START_BIT:if(counter_s==COUNT_HALF) state<=DATA_8;
                      DATA_8:if(counter==COUNT_MAX) state<=DATA_7;
                      ……..
                      DATA_1:if(counter==COUNT_MAX) state<=PARITY;
                      PARITY:if(counter==COUNT_MAX) state<=STOP_TRANS;
                      STOP_TRANS:if(counter==COUNT_MAX) state<=IDLE;
                    Endcase
end
```

Here we see the state finite state machine that is activated only if and as we have said, to go from IDLE to START must pass 8 cycles Rx_sample_ENABLE and for the other states 16 cycles.

```
always @(posedge Rx_sample_ENABLE or posedge reset) begin
//we reset signals when transmission starts
  if(reset==1 || state==START_BIT) begin
        Rx_FERROR<=0;
        Rx_PERROR<=0;
   End
   else begin //and we calculate them every time transmission ends
          if(state==STOP_TRANS && counter==COUNT_MAX) begin
                    if(stop_bit == 1'b1) //Stop bit correct or not
                      Rx_FERROR<=1'b0;
                    else
                      Rx_FERROR<=1'b1;
                    if(parity_bit == ^Rx_DATA) //parity bit correct or not
                      Rx_PERROR<=1'b0;
                    else
                      Rx_PERROR<=1'b1;end
```

Here we see the handling of the error signals of the receiver. They are initialized to 0 every time we do a reset or we see a start bit. While at the end of each transmission we calculate their values, depending on the condition that must be satisfied for each.

```
always @(posedge Rx_sample_ENABLE or posedge reset) begin
                    case(state)
                      IDLE:Rx_DATA<=Rx_DATA;
```

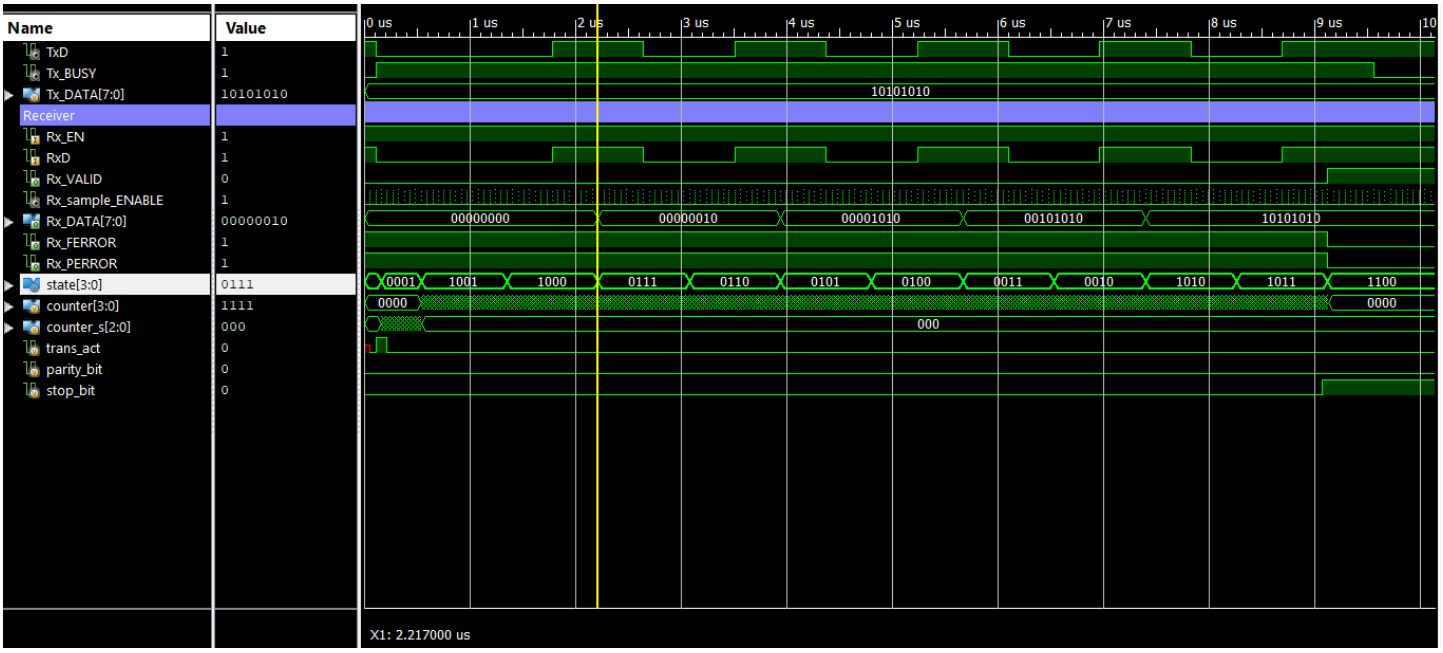
```
                START_BIT:Rx_DATA<=Rx_DATA;
                DATA_8:if(counter==COUNT_MAX) Rx_DATA[0]<=RxD;
                .....
                DATA_1:if(counter==COUNT_MAX) Rx_DATA[7]<=RxD;
                PARITY:if(counter==COUNT_MAX) parity_bit<=RxD;
                STOP_TRANS:if(counter==COUNT_MAX-1) stop_bit<=RxD;
            Endcase
end
```

Finally we have the assignment of the values, based on the situation, to Rx_DATA, parity_bit and stop_bit.

Verification

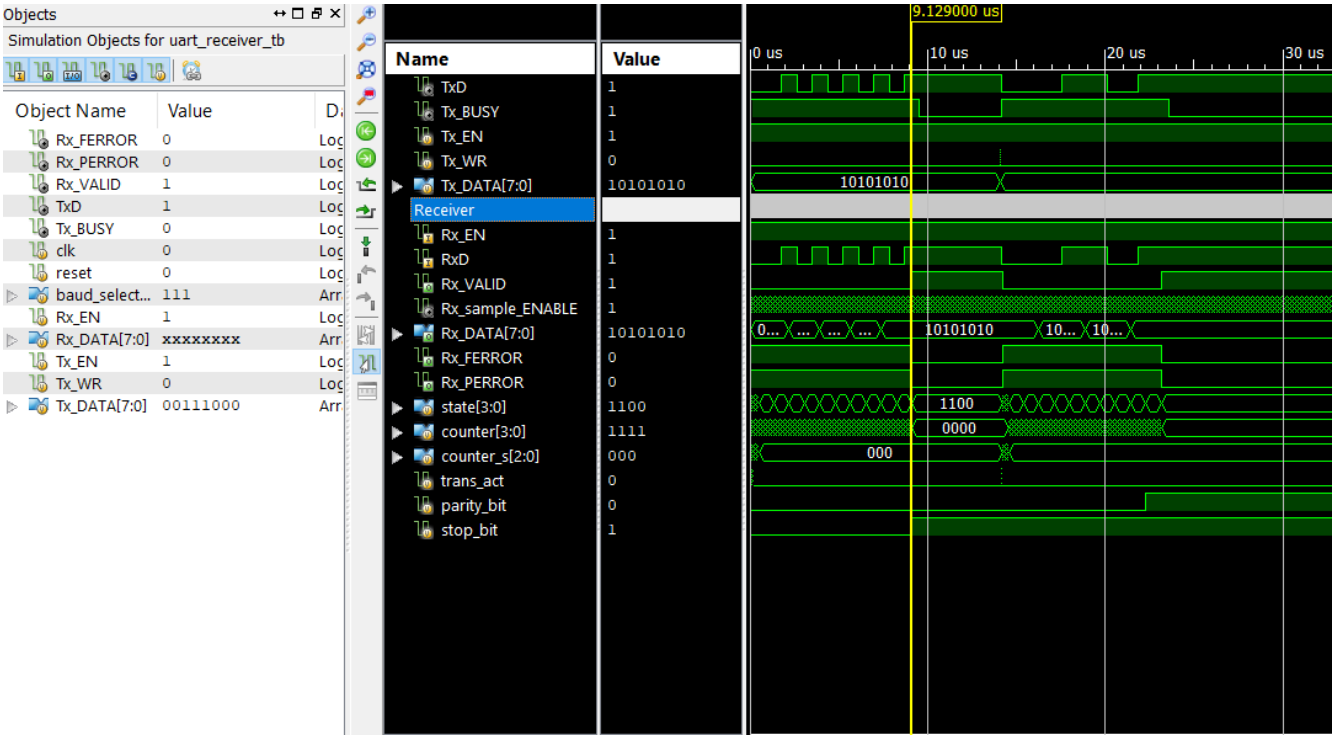
To verify the correct operation of the UART receiver we use a suitable testbecnh. Because we have checked the operation of the transmitter and the receiver to be checked we must provide data, we connect the two subsystems in a complete UART communication system.

In this way, part D of the laboratory exercise is implemented and verified at the same time.

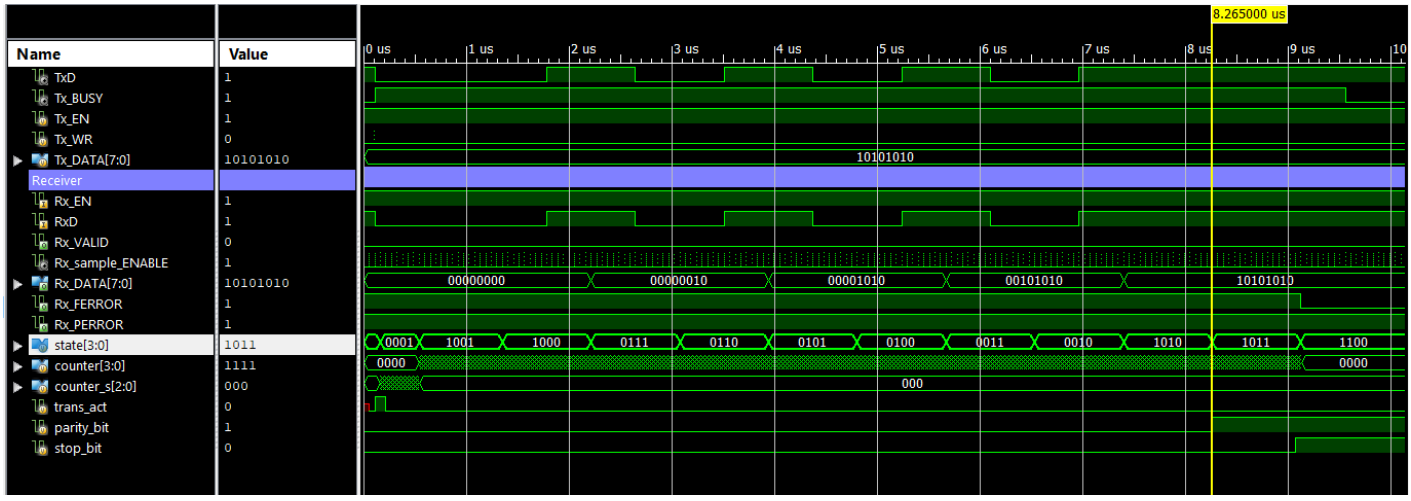


In the image of the simulation we see 1 transmission in the UART channel that we have created. We can see the complete status of the receiver, specifically:

1. *We see the alternation of state states, with the start at the beginning of the start bit and the change of each in the middle of each bit that we sample (as shown in the point where the yellow cursor is).*
2. *We have the active state of counter_s that counts only during the first state for 8 Rx_SAMPLES, then the counter is deactivated and activated for the rest of the transmission (counts for 16 Rx_SAMPLES).*
3. *Finally, in the middle of the stop bit we switch to idle mode and the receiver outputs the control signals Rx_VALID, Rx_PERROR, Rx_FERROR. Since we had a correct transmission Rx_VALID goes up to 1.*



In the next image we see 2 transmissions in the UART channel that we have created. First we transfer 10101010 and then 00111000. We can give the correct transmission of the characters in each case, the increase of Rx_VALID to 1 because of this after the via bit bit in In addition, we can see that Rx_VALID remains active as long as the sender does not provide data to the receiver and this will only fall when we reach the beginning of a start bit.



In the next simulation we check the error signal Rx_PERROR. To do this in the parity bit transmission state from the sender, we always transmit 1 regardless of the transmission data (Tx_DATA). So while we send the character 10101010 with parity_bit = 0, we send parity_bit = 1.

So at the end of the transmission we should have Rx_PERROR at 1 and Rx_VALID at 0. At the point where the yellow cursor is (in state PARITY_BIT) we get the parity bit. So, at the end of the transmission (in the middle of the stop bit) we update the values of Rx_PERROR and Rx_FERROR with 1 and 0 respectively. So we will have Rx_VALID stay at 0 and the data we received is unacceptable from the system side.

Conclusion

In this laboratory exercise we wanted to implement the UART protocol in circuit form in Verilog.

Based on the simulations that have been done above, we have confirmed the correct operation in terms of behavior, which was also the request of the laboratory exercise 2 (The optional part of the exercise has not been done).

Difficulties that arose concerned only the selection of the appropriate values in the counters used for each part, the selection of the correct number of states, their appropriate assignment and the timing control with the new clock from the baud_rate module.

But after corrections on the code, the help of the laboratory assistants and our theoretical knowledge all the problems were solved and the laboratory exercise was successful.