Electrical and Computer Engineering
University of Thessaly (UTH)

ECE333 - Digital Systems Lab

Fall Semester — Educational year 2023-2024

Lab02

Universal Asynchronous Receiver-Transmitter (UART)

Georgios Kapakos - AEM: 03165

Contents

Abstract

The goal of the 2nd assignment is the implementation of the two modules required for serial communication via the UART protocol, a Transmitter (Tx) and a Receiver (Rx). In this laboratory report, the code with which they were implemented is described, the way in which the two modules were synchronized with each other, and the results of the testbenches. The ultimate goal of the project is the successful transfer of four symbols (8-bit data) from the Transmitter to the Receiver.

1 Introduction

The objectives of this assignment are the implementation of a UART protocol. That is, we implement a serial, asynchronous communication protocol. The transmitter sends 8 bits of data, which it receives from the system, and the receiver samples them and stores them in an 8-bit register, bit by bit.

This specific communication protocol has the particularity that the 2 or more devices that use it for communication may have unrelated or unsynchronized clocks. The asynchronous communication of the UART is implemented through a wired connection of the data sent by the transmitter (TxD) and those received by the receiver (RxD). Through this protocol, we operate independently of the conditions prevailing in the input and output systems, so to avoid data loss, a common speed (baud rate) is selected, with which the Transmitter and Receiver will communicate via the UART. The Receiver will operate at this speed, while the Transmitter will examine each transmitted digit 16 times faster.

Since there is no guarantee regarding the relative frequency and phase between the clocks of the Transmitter and Receiver, as the communication is asynchronous, the receiver must know when the transmission of a symbol begins. The two modules are connected with a one-bit signal, which, when no transmission is occurring, has the value 1. The start signal (start bit) signals the beginning of communication for the Receiver, which, to recognize it, has the value 0. Correspondingly, the end of the symbol and the completion of communication are signaled by the stop bit, which is given the value 1, to return to the initial idle state.

To verify the correct communication of the data, they are accompanied by a parity bit, which indicates whether our symbol contains an even or odd number of ones. In case the data we have received differ from those we have sent, it will be evident in the parity of the parity bit, and the error signal Rx_PERROR will be activated. Additionally, if the sampling of any bit fails to occur at the center, then the error signal Rx_FERROR is activated. If either of these is activated, then the bits we received are not taken into account.

The UART protocol does not include flow control between Receiver and Transmitter, meaning the Receiver cannot stop the Transmitter, so both must be ready for continuous communication. Finally, for the transmission of multiple different symbols, proper management of input and output signals is required. When a transfer begins, the Transmitter does not accept a new symbol for transmission until the current one is completed. On the other hand, the Receiver maintains the values of its outputs until new communication occurs.

2 Part A - Baud Rate Controller

2.1 Implementation

In this part, we implement a Baud Rate Controller, which will be implemented internally in the circuits of the transmitter and receiver. This controller provides the receiver and transmitter with a common sampling rate, meaning they pre-agree on the speed of their communication in Baud units (bits/sec), according to the following table:

BAUD_SEL	Baud Rate (bits/sec)
0 0 0	300
0 0 1	1200
0 1 0	4800
0 1 1	9600
1 0 0	19200
1 0 1	38400
1 1 0	57600
1 1 1	115200

The above table presents the UART communication speeds and the Baud_Sel selection signal used to choose the Baud Rate at which a symbol is transmitted. The transmission period of each digit from the Transmitter corresponds to $T_{sample} = 1/\text{BaudRate}$.

However, the operation and sampling of the UART Receiver occur 16 times faster than the pre-agreed Baud Rate frequency by the Receiver. Practically, the Receiver operates at a speed of BaudRate $\times$ 16, while the Transmitter operates at BaudRate. Therefore, the Controller must send an appropriate sampling signal based on which the Transmitter and Receiver will operate. This results in the two modules operating independently of their clocks, which is desirable since it is not necessary for them to have a common clock, and in this case, there is no need to synchronize them.

We calculate for each Baud Rate how many clock cycles correspond to the respective bits/sec, using the formula: max_cycles $= f_{clk}/f_{sample} = f_{clk}/(16 \times \text{Baud\_Rate})$.

Using the board's clock, we have $f_{clk} = 100\,\text{MHz}$. Thus, from the above table and formula, the following results:

Baud Select	Baud Rate	max_cycles	Approximation	Relative Error (%)
000	300	20833.3333	20833	0.0016
001	1200	5208.3333	5208	0.0064
010	4800	1302.0833	1302	0.0064
011	9600	651.0416	651	0.0639
100	19200	325.5208	326	0.1472
101	38400	162.7604	163	0.1472
110	57600	108.5069	109	0.4544
111	115200	54.2535	54	0.4673

The relative error is calculated using the formula: $|\text{X}_{\text{approximation}} - \text{X}_{\text{actual}}|/\text{X}_{\text{actual}} \times 100\%$.

We observe that as we increase the Baud Rate, the cycles decrease, and the relative error increases. However, the approximation error is quite small, so in the UART implementation, errors will mostly not be observed.

For the implementation of the Baud Rate Controller, we created the following module:

Baud Controller:
It consists of the inputs:

```
always@ (baud_select) begin
    case(baud_select)
    3'b000: begin
            size <= 20_833;
            end
    3'b001: begin
            size <= 5_208;
            end
    3'b010: begin
            size <= 1_302;
            end
    3'b011: begin
            size <= 651;
            end
    3'b100: begin
            size <= 326;
            end
    3'b101: begin
            size <= 163;
            end
    3'b110: begin
            size <= 109;
            end
    3'b111: begin
            size <= 54;
            end
    default: begin
             size <= 20_833;
             end
    endcase
end
```

- **reset**: Asynchronous reset of our circuit.
- **clk**: Clock, with $f_{clk} = 100\,\text{MHz}$ frequency.
- **baud_select**: Based on this specific input, we select which baud rate we will have for the transmission of our signal, according to the table mapping baud select to baud rate. We implement this in a combinational always block, with the sensitivity list containing only baud_select. At the same time, we set a variable we call size, the number of maximum cycles, depending on the baud select choice. Additionally, based on the size of the baud select, to include all signals, we set a 15-bit counter. The counter has this size because: $2^{14} < 20833 < 2^{15}$, to be able to count all our values; 20833 is our maximum value.

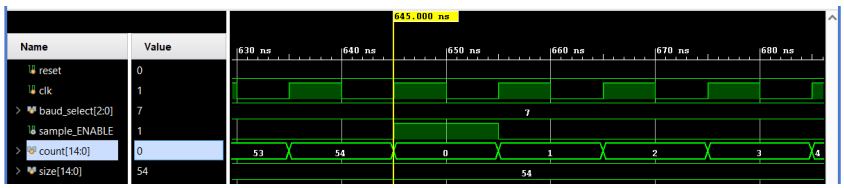
It consists of the output:

```
/*15 bit counter*/
always @ (posedge clk or posedge reset) begin
    if (reset) begin
        count <= 15'b000000000000000;
        sample_ENABLE <= 1'b0;
    end
    else if(count == size) begin
        sample_ENABLE <= 1'b1;
        count <=15'b000000000000000;
    end
    else begin
        count <= count + 15'b000000000000001;
        sample_ENABLE <= 1'b0;
    end
end
```

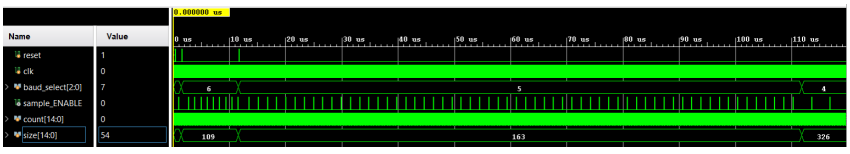
- **sample_ENABLE**: Output of our controller, which is activated each time for one clock cycle when the 15-bit counter we created reaches the selected size. Otherwise, it remains inactive, and the counter increases. This counter is implemented in a sequential always block.

2.2 Verification

For the verification of our circuit, we have created a testbench in which we examine the output signal sample_ENABLE for different values of baud_select. Before each change in baud_select, I reset the circuit for its reinitialization.



In the above waveforms, I observe that when our counter reaches the value of size, i.e., max_cycle, according to the table we have created, it activates the value of sample_ENABLE for one clock cycle. This repeats until the end of my simulation.



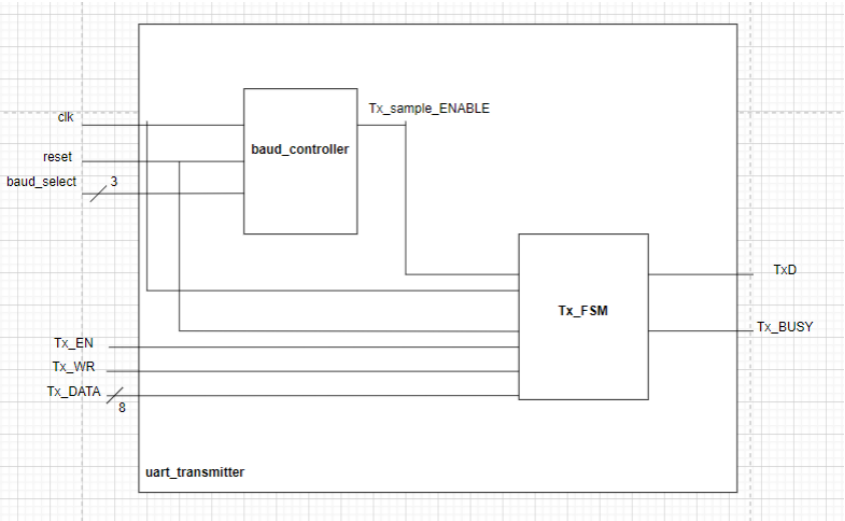
In the above waveforms, we see the frequency difference with which the output signal changes depending on the Baud Rate we have selected.

3 Part B - Implementation of UART Transmitter

3.1 Implementation

The implementation of the data flow of the UART-Transmitter:

In this part, we implement the UART Transmitter as the top-level module, uart_transmitter.



It has as inputs:

- **reset**: Asynchronous reset signal of the circuit.
- **clk**: Clock of our circuit.
- **baud_select**: Signal for selecting the sampling rate.
- **Tx_DATA**: Constitutes the symbol (8 bits) that I want to transmit.
- **Tx_EN**: Signal that activates the transmitter to start its operation and remains active until the completion of the data transfer.
- **Tx_WR**: Signal that indicates that the data are ready for transmission. The signal remains active for one clock cycle and becomes 1 again when we want to send a new symbol.

It has as outputs:

- **TxD**: Signal that is the bit-by-bit data we transmit serially to the receiver.
- **Tx_BUSY**: Signal that indicates that the transmitter is in a transmission state and cannot accept new data while in this state. Upon completion of the message transmission, the signal becomes 0 and allows the reading of new data.

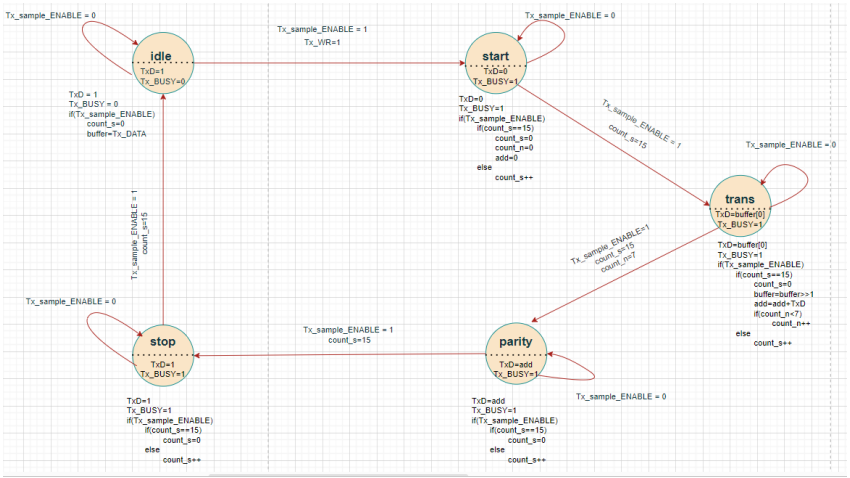
In the above data flow, it is evident that the following modules are instantiated:

- **baud_controller**: Constitutes the controller we described in Part A of our assignment. It produces as its output the signal Tx_sample_ENABLE. When this signal is active, it allows the FSM we have created to make changes to the circuit's variables.
- **Tx_FSM**: We implement an FSM in which we perform the appropriate actions for signal transmission. All inputs of our circuit are connected to this FSM, except for baud_rate, and it produces both of our output signals. The states it consists of are:

  1. **idle (000)**: In this state, the transmitter is in an idle state, so it sets the signal TxD = 1 and the signal Tx_BUSY = 0. It waits for the Tx_ENABLE and Tx_WR signals to be activated to change state; otherwise, it remains inactive.

2. **start (001)**: In this state, each time Tx_sample_ENABLE is activated, our counter will increase, and when it counts 16 cycles, we move to the next state, initializing variables we will use in the next step. It also activates the start bit, which signifies the start of transmission in our circuit, and we set the outputs to TxD = 0 and Tx_BUSY = 1.
3. **trans (010)**: In this state, the output signal of our bits will take the value of the appropriate bit from the signal we used as input. This will be implemented by right-shifting the bits of Tx_DATA by 1, which we have stored in a buffer. At the same time, we increase the value of add, which controls the parity bit in the next state. This happens every 16 clock cycles when Tx_sample_ENABLE is active. Finally, upon counting the 8 bits to be transmitted, we move to the next state.
4. **parity (011)**: In this state, we set the value of TxD equal to the sum of all digits to be transmitted from the signal in the previous state; we call it the parity bit, and it checks the parity of the bits.
5. **stop (100)**: In this state, we arrive to declare the end of the transmission of our bits, sending the stop bit that signals the end of the symbol transmission (TxD = 1).

For the implementation of the FSM, we use two always blocks:



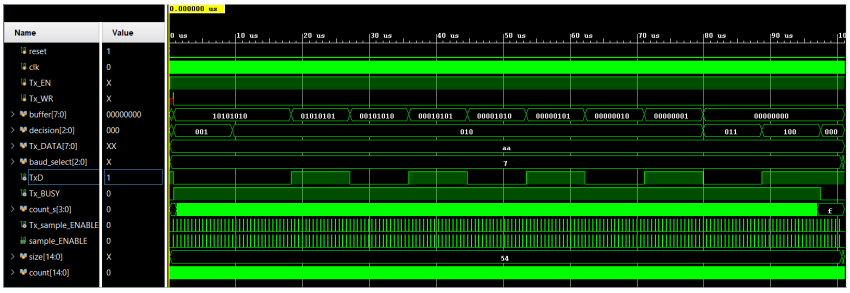
The first is a sequential always block and is responsible for assigning values to the current state by equating it to the next state. Each time a variable changes, it is assigned to the current state. Otherwise, if reset is pressed, we have an initialization of all values to 0.

The second is a combinational always block and modifies the next states with the activation of Tx_sample_ENABLE, and when our counter reaches the value 15, then we allow changes to the variables and the transition from one state to another.

Below, we present the FSM model schematically as a Moore model:

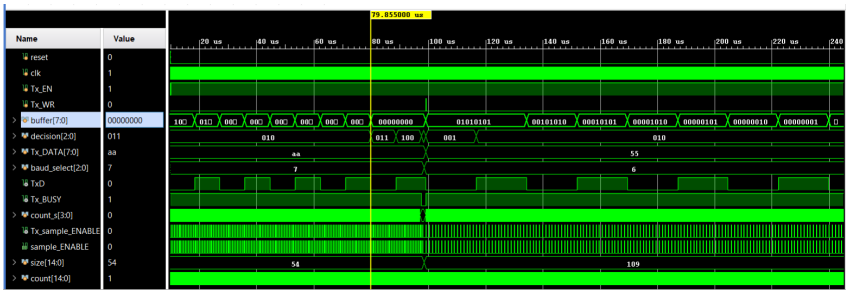
3.2 Verification

In the testbench, we set baud_select = 111, which is the fastest of the given speeds, and Tx_DATA = 10101010. We activate the Tx_WR signal for one clock cycle when it is active and the Tx_sample_ENABLE signal for the transition from the idle state to the start state. The waveform is as follows:

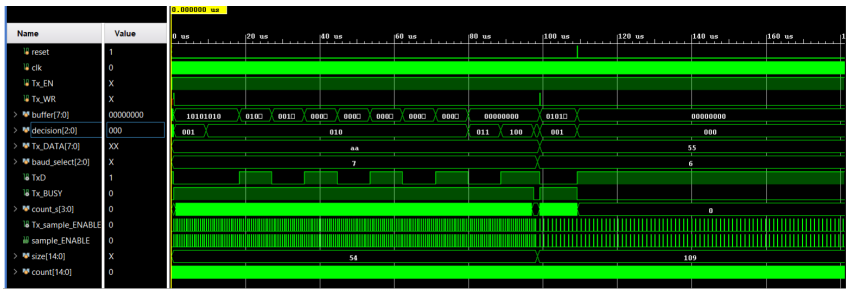


In the above waveform, we observe that the states change from idle to stop, one after the other. From 000 to 100 (5 states). The TxD signal is initially 1 in the idle state, but after transitioning to the start state, it becomes zero, and then in the transmission state (trans), it takes the value of the least significant bit (lsb) of the buffer we created, to which we apply a right shift, so we see its most significant bits move one position to the right each time. In the parity check state (parity), we observe that it takes the value of the sum of all digits to be sent, in this case, 0, i.e., even, so the digits were transmitted correctly because our signal has an even number of ones and zeros. Finally, we reach the state where we stop the transmission (stop); there, TxD becomes 1 again, this state signifies the end of transmission and returns to the initial idle state until the Tx_WR signal is activated again for sending a new message.

Throughout the transmission duration, the Tx_BUSY signal is 1; upon completion of the transmission, it becomes 0 when it reaches the idle state, as seen in the waveforms.



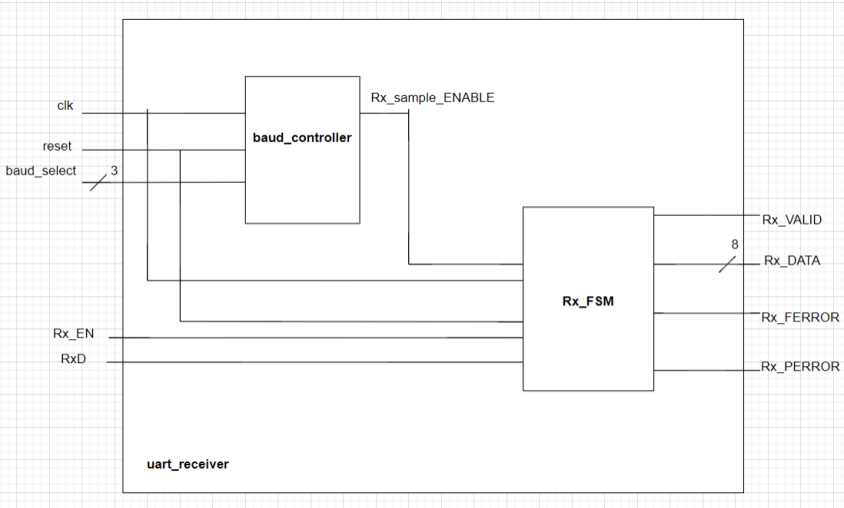
Here, we see that when the Tx_WR signal is activated again, our system is ready to receive the new signal for transmission, which it sends with a different baud rate, performing the same operations.



Finally, in these waveforms, we see that after the activation of the reset, the circuit returns to its idle state.

4 Part C - Implementation of UART Receiver

4.1 Implementation



The implementation of the UART Receiver is shown in the following figure:
In this part, we implement the UART Receiver as the top-level module, uart_receiver.
It has as inputs:

- **reset**: Asynchronous reset signal of the circuit.
- **clk**: Clock of our circuit.
- **baud_select**: Signal for selecting the sampling rate.
- **Rx_EN**: Signal that activates the receiver to start its operation and remains active until the completion of the data transfer.
- **RxD**: Signal that receives the data from TxD and samples them bit by bit.

It has as outputs:

- **Rx_DATA**: The data we received from the transmitter, provided it sent them without errors.
- **Rx_FERROR**: Signal that indicates that the receiver has performed incorrect sampling either at the start of the signal (start bit) or at the end (stop bit).
- **Rx_PERROR**: Signal that indicates that the receiver has received a different parity bit from the one we calculate based on the symbol we have sampled.
- **Rx_VALID**: Signal that, when it becomes 1, indicates that the data we received arrived successfully without errors. It becomes 1 for one clock cycle.

In the above data flow, it is evident that the following modules are instantiated:

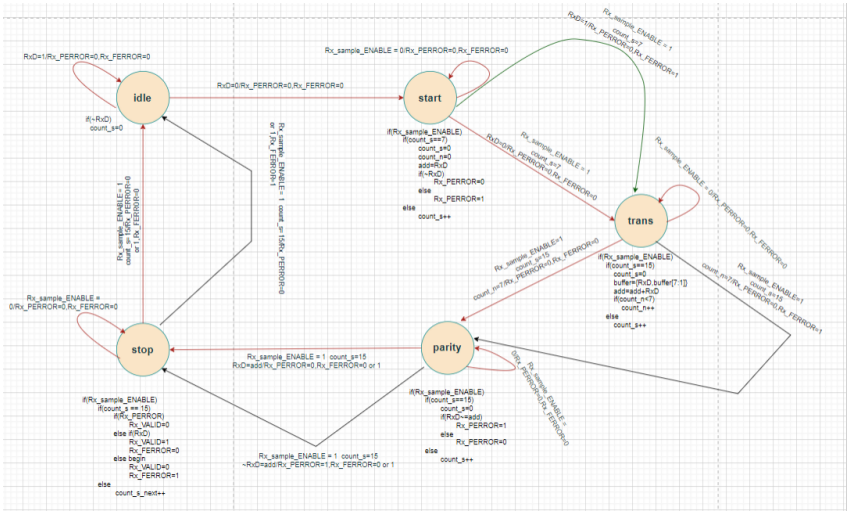
- **baud_controller**: Constitutes the controller we described in Part A of our assignment. It produces as its output the signal Tx_sample_ENABLE. When this signal is active, it allows the FSM we have created to make changes to the circuit's variables.

As shown, in the top module, the baud_controller module is instantiated, operating in the same way as described in the Transmitter. The logic and outputs of the module arise from the third module (Rx_FSM):

- **The Rx_FSM module**: Implements an FSM whose states are as follows:

  1. **idle (000)**: This state is the waiting state. We wait to initially receive the Rx_EN signal, which indicates the receiver's operation, and then the start bit from the transmitter, which signifies the start of sampling our signal.
  2. **start (001)**: This state begins the sampling of our message. Our counter counts up to half, i.e., 8 cycles, and then we activate the next state; to be able to sample at the center of the message, we count 8 cycles in this state.
  3. **trans (010)**: This state signals the start of sampling the symbol we sent via the transmitter. Sampling occurs every 16 cycles, again at the center of the pulse. Each time 16 cycles are completed when Rx_sample_ENABLE is active, we add the bits together in add for later checking if the parity bit is incorrect. Additionally, we right-shift the buffer every 16 cycles, our message into a buffer, which we will ultimately assign to Rx_DATA.
  4. **parity (011)**: This state contains the check of whether the signal that reached the receiver and we sampled arrived correctly. If it arrived correctly, after 16 cycles, Rx_PERROR becomes 0; otherwise, 1, and we proceed to the final state change.
  5. **stop (100)**: This state, after the 16 cycles when Rx_sample_ENABLE is active, checks if Rx_PERROR was previously activated and sets Rx_VALID to 0. On the other hand, if, when I reach the stop bit, it has the value 1, it means I sampled correctly. I set Rx_VALID = 1, and Rx_DATA takes the buffer data; if the sampling is done incorrectly, then Rx_FERROR = 1 is activated, and Rx_VALID = 0, returning to the initial idle state.

The implementation diagram of the FSM is as follows:

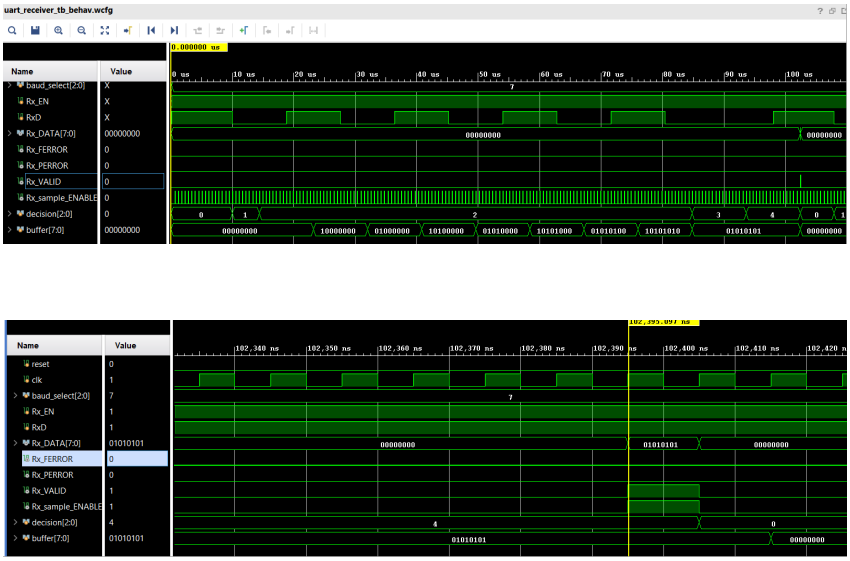


Additionally, in the code, we use two more combinational always blocks: one to assign a value to Rx_VALID depending on the values taken by the Rx_PERROR and Rx_FERROR signals. That is, as long as we do not encounter errors and neither of these two signals is 1, then Rx_VALID becomes 1; otherwise, 0. The other always block assigns the value of the buffer

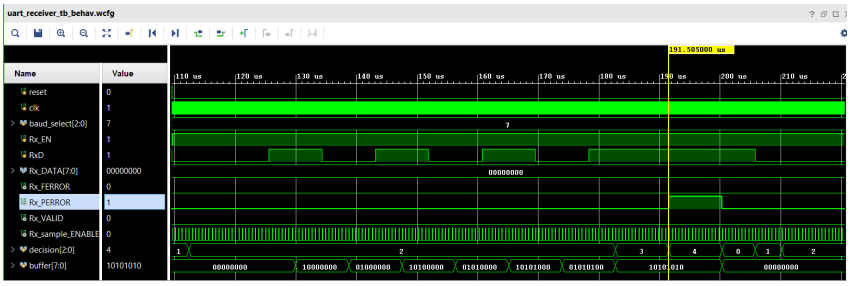
we calculated in the FSM to Rx_DATA; if there are no errors in receiving the symbol, then Rx_VALID will be active; otherwise, we do not assign a value to Rx_DATA because the data to be stored are incorrect.

4.2 Verification

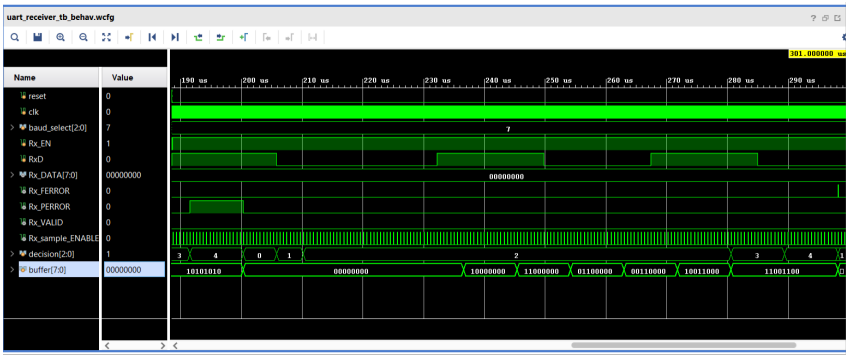
In this testbench, I want to check if sampling is done correctly in the middle of the period of the transmitted digits, whether the error signals are activated or not, and finally, whether the output values are as expected, which must be maintained even after the end of communication.



From the above figure, we observe that the Rx_DATA output, as long as the FSM has not completed its checks, does not yet take a value and will only take a value if everything operates correctly and no error messages are activated in the circuit, upon the end of the stop bit. This value is stored in Rx_DATA for one clock cycle, as long as Rx_VALID is active. In the above circuit, there are no errors, so the corresponding error messages are not activated.



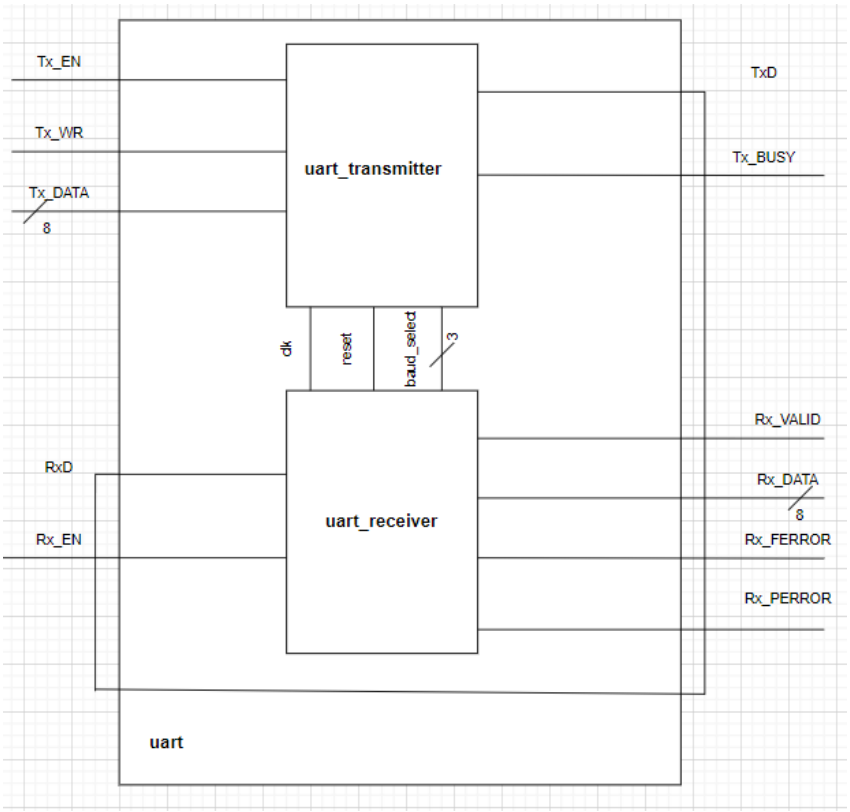
In the above waveforms, we see the case of an error in our circuit. We have PERROR, so the error signal is activated, while the valid signal remains 0, resulting in Rx_DATA not receiving data from the system and remaining 0. We see that the Rx_PERROR signal is activated after the parity stage.



In the above waveforms, we see the case of an error in our circuit. We have FERROR, so the error signal is activated, while the valid signal remains 0, resulting in Rx_DATA not receiving data from the system and remaining 0. The FERROR signal is activated momentarily, as shown above.

5 Part D - Implementation of UART Transmitter-Receiver for Serial Data Transfer

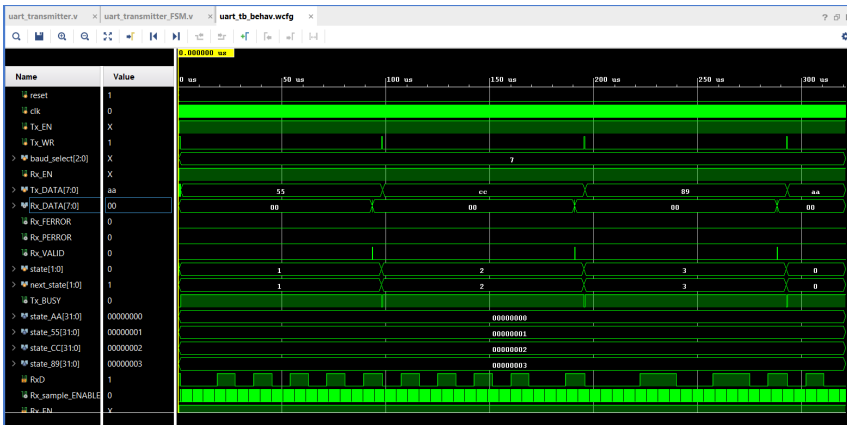
5.1 Implementation

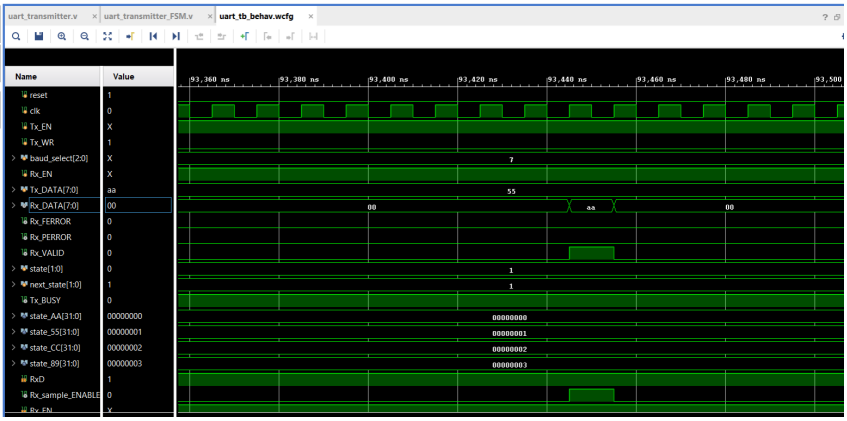


5.2 Verification

Having created a complete UART channel, we combine the Transmitter and Receiver modules described in Parts B and C. I create a testbench that will check the following four consecutive words:

1. 10101010 (AA)
2. 01010101 (55)
3. 11001100 (CC)
4. 10001001 (89)





As seen in the above waveforms, Rx_DATA takes the value of Tx_DATA as soon as it completes its internal check at the end, for one clock cycle, so the value of Rx_DATA is not distinguishable. Also, the symbol to be transmitted appears delayed in Rx_DATA, which is the output of our symbols. The communication is successful; no errors appear to occur in the communication between the transmitter and receiver.

We implemented the testbench with a very simple FSM that simply, if Tx_BUSY is 0, activates Tx_WR and moves to the next state; for each state, the same happens until the end of my simulation.

6 Conclusion

This laboratory assignment successfully achieved the implementation of a UART protocol, encompassing a Transmitter (Tx) and Receiver (Rx) for serial asynchronous communication. The Baud Rate Controller effectively synchronized the modules, allowing reliable data transfer across a range of baud rates (300 to 115200 bps) with minimal error, as evidenced by the low

relative errors in cycle approximations. The Transmitter accurately serialized 8-bit symbols with start, parity, and stop bits, while the Receiver sampled and validated them, detecting errors via Rx_PERROR and Rx_FERROR signals when present. The final integration in Part D demonstrated the successful transfer of four 8-bit symbols (AA, 55, CC, 89) with no observed communication errors, meeting the project's ultimate goal. This implementation highlights the robustness of UART for asynchronous systems and provides a solid foundation for further exploration of serial communication protocols.