Universal Asynchronous Receiver Transmitter

CHRISTODOULOS ZERDALIS, AEM: 03531, 11/11/2024

Summary: The objective of this lab is to design and build a UART (Universal Asynchronous Receiver-Transmitter) in four key stages: the baud controller, transmitter, receiver, and finally the complete UART unit. This report provides a detailed explanation of each stage, offering insights into the design process, underlying concepts, challenges encountered, and construction strategies for each component.

Introduction: This lab aims to build a comprehensive understanding of UART design by constructing each of its key components: the baud controller, transmitter, receiver, and finally, the complete UART unit. Each stage presents unique challenges and requires careful consideration of timing, data integrity, and synchronization. The first stage included building the baud controller and calibrating the sampling.

Part A - Baud Rate Controller:

Implementation: The Baud Rate Controller is responsible for managing the timing of sampling intervals for both the Transmitter and Receiver. Since both modules operate on the same clock in this design, a single Baud Rate Controller can serve both. To generate the correct delay between sampling intervals, a counter was implemented to count the required number of clock cycles. This count was determined by dividing the sampling period by the clock period. However, because this division did not yield an exact integer, the result had to be rounded, introducing a small error in each case. Furthermore, the counter/baud rate controller is only enabled when the respective module that is in is enabled.

Additionally, to support eight different baud rates, an extra module was designed to accommodate selectable baud rates. Based on the baud_select input, this module outputs the correct cycle count for each specified baud rate, enabling the counter to produce the accurate delays for each rate.

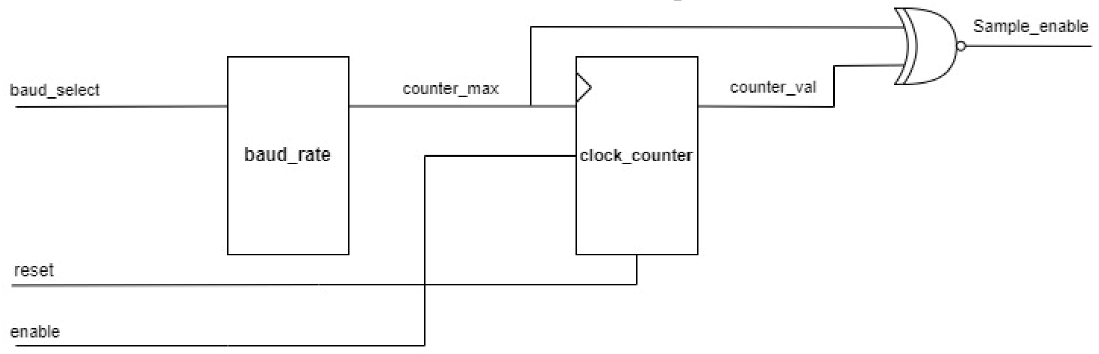
Cycle count calculation:

$$MAX\_COUNT = \frac{1}{16 \times BAUD\ RATE} \div Tclk = \frac{10^{-8}}{16 \times\ BAUD\ RATE}$$

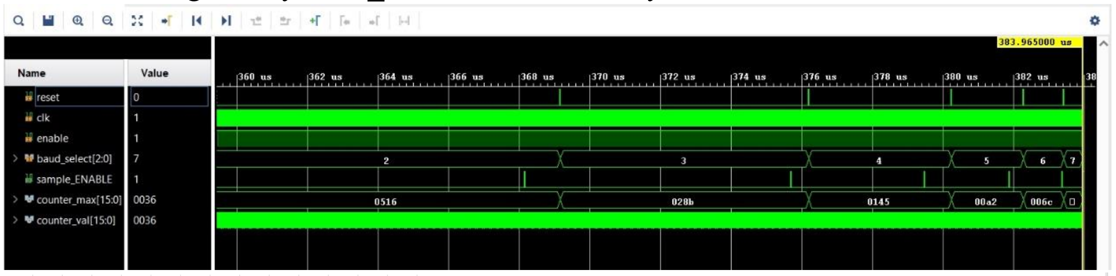
BAUD RATE	RESULT	ROUNDED	RELATIVE ERROR PERCENTAGE
0	20833.33	20833	1.60E-06
1	5208.333	5208	6.39E-05
2	1302.083	1302	6.37E-05
3	651.041	651	6.30E-05
4	325.52	326	1.47E-04
5	162.76	163	1.47E-03
6	108.506	109	4.55E-03
7	54.243	54	4.48E-03

Modules:
Baud_rate_select: This is the module that outputs the number of cycles to count(combinational)

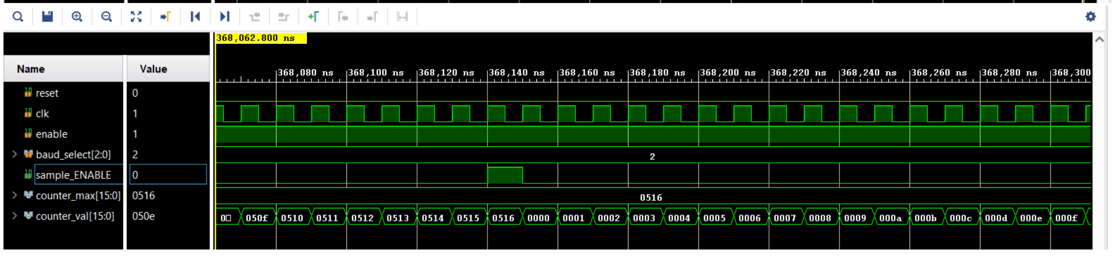
Clock_counter: This is the module that creates the delay and since it depends on the clock it is sequential.
Baud_controller: Combines both modules and outputs the signal sample_enable that indicates when the receiver or transmitter should sample.



Simulation: In the simulation every possible baud rate was tested with a testbench that went though every baud_select and reset the system between each.



Name	Value
reset	0
clk	1
enable	1
baud_select[2:0]	7
sample_ENABLE	1
counter_max[15:0]	0036
counter_val[15:0]	0036



Name	Value
reset	0
clk	1
enable	1
baud_select[2:0]	2
sample_ENABLE	0
counter_max[15:0]	0516
counter_val[15:0]	050e

Here we can see that the sample_enable is equal to 1 when the counter reaches the intended number.

Experiment: There was no experiment on the FPGA on this part.

Part B – UART Transmitter:

Implementation: The transmitter is responsible for sending data to the receiver in a precise manner, coordinated by the sample_enable signal from its baud controller. The process begins when the transmitter is enabled, and, upon activation of the Tx_WR signal, the module initiates data transmission.

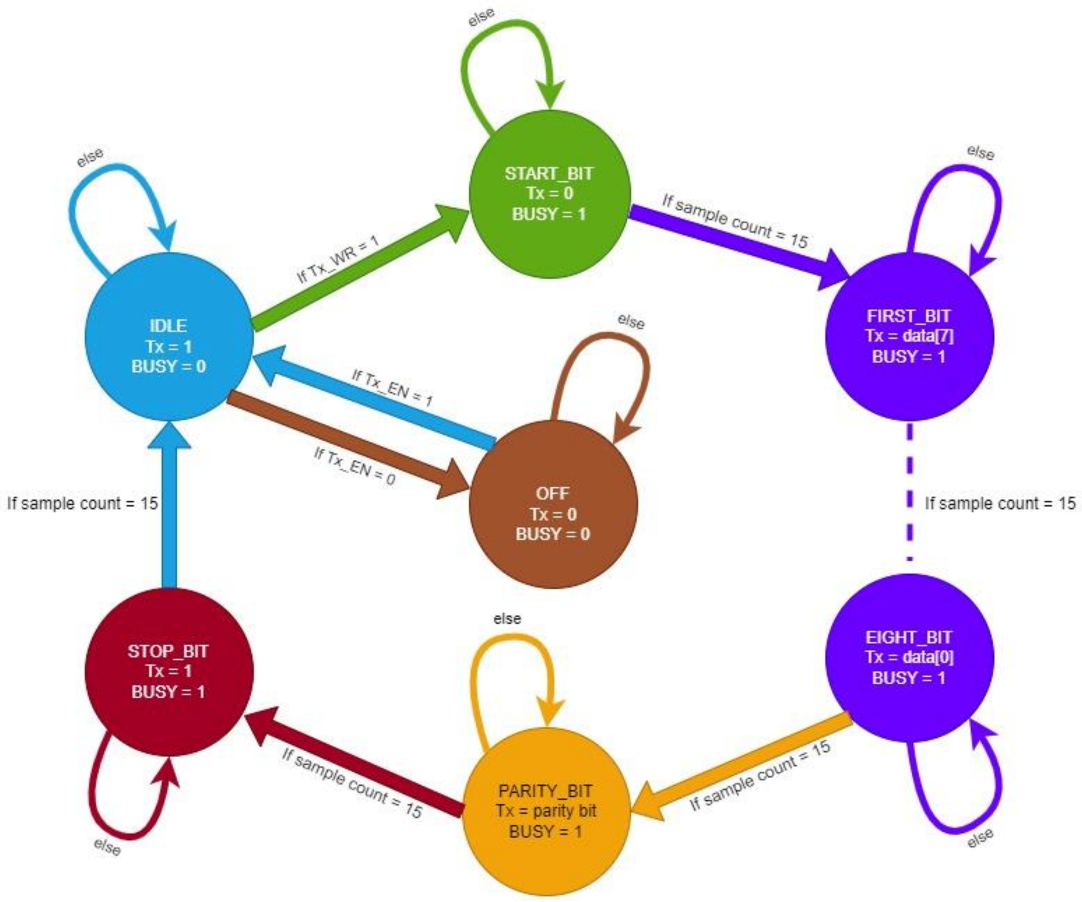
The transmitted data consists of a start bit, which indicates the beginning of transmission, followed by 8 data bits, a parity bit for error checking, and finally a stop bit to signal the end of the transmission. This behavior will be simulated using a state machine, where each state corresponds to a specific bit in the transmission sequence.

A baud controller regulates the timing of each bit, ensuring that each is transmitted precisely every 16 pulses of the sample_enable signal. To achieve this, a 4-bit counter is used to track these pulses, outputting the pulse count. When the Tx_WR signal is enabled, the counter resets to zero, as the transmission restarts from the beginning.

Lastly, the parity bit is the LSB from the addition of every data bit or the result of adding every data bit in a one bit variable.

State machine:

Here is the flow diagram of the state machine

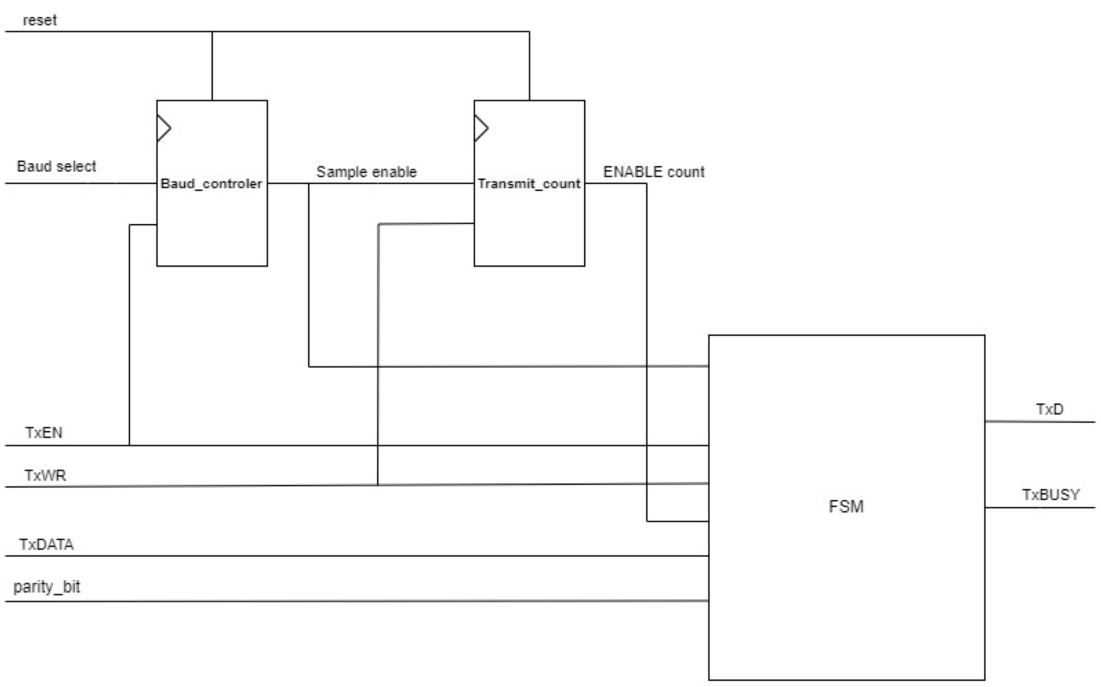


Initially, while the module is disabled, it remains in the OFF state. Once enabled, it transitions to the IDLE state, where it waits for the Tx_WR signal to indicate the start of a transmission.

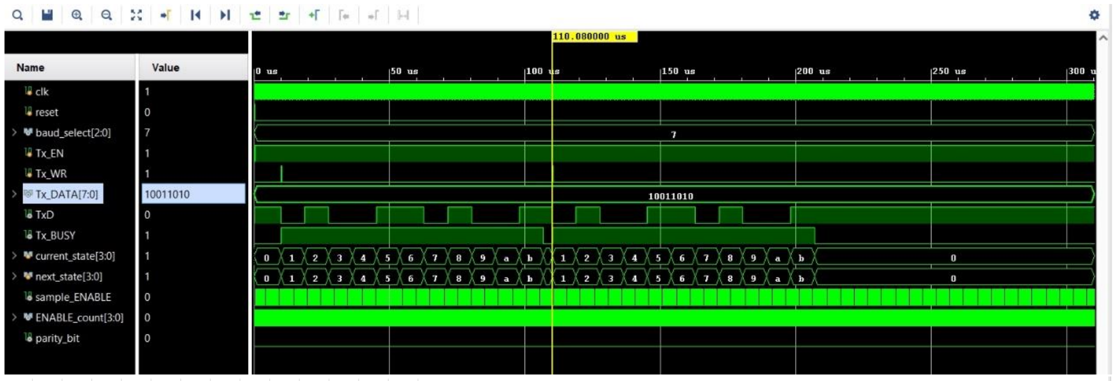
When transmission begins, a new bit is sent every 16 pulses of the sample_enable signal, continuing this process until the stop bit is reached. At that point, the state returns to IDLE, ready to detect the next rising edge of Tx_WR or to respond to the deactivation of Tx_EN, if the module is disabled.

This design is implemented using a Moore state machine comprising a sequential circuit that controls the state variable, along with a combinational circuit that uses a case statement to execute the specific actions associated with each state.

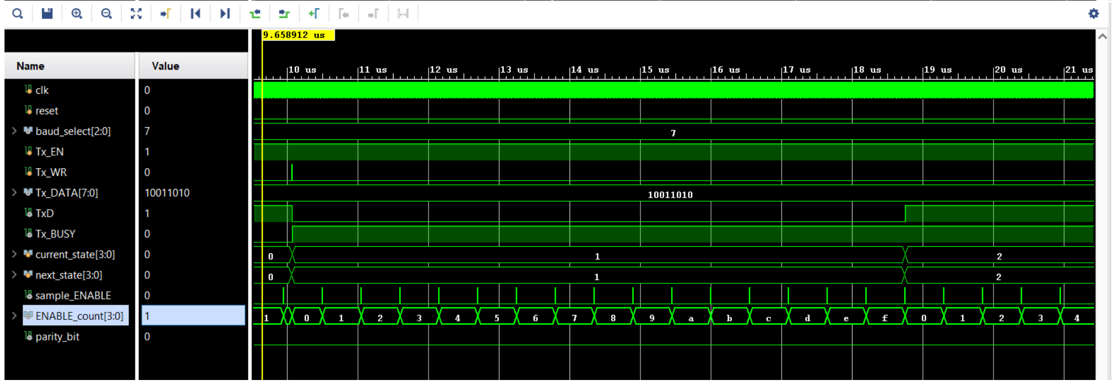
Modules:
Baud_controller (same as before)
Transmit_counter: The module that counts the pulses of the sample enable(sequential)
uart_transmiter: The module that combines the two above with the state machine.



Simulation: For the simulation a testbench is used that enables the module and begin the transmission of some data two consecutive times.



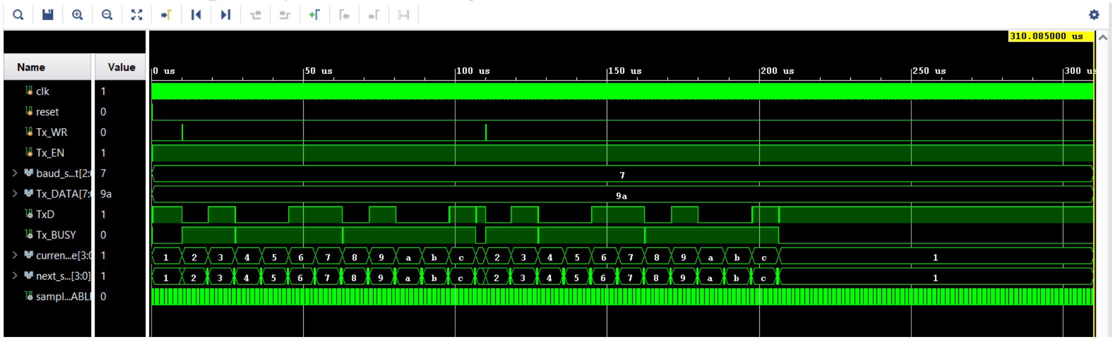
Name	Value
clk	1
reset	0
baud_select[2:0]	7
Tx_EN	1
Tx_WR	1
Tx_DATA[7:0]	10011010
TxD	0
Tx_BUSY	1
current_state[3:0]	1
next_state[3:0]	1
sample_ENABLE	0
ENABLE_count[3:0]	0
parity_bit	0



Name	Value
clk	0
reset	0
baud_select[2:0]	7
Tx_EN	1
Tx_WR	0
Tx_DATA[7:0]	10011010
TxD	1
Tx_BUSY	0
current_state[3:0]	0
next_state[3:0]	0
sample_ENABLE	0
ENABLE_count[3:0]	1
parity_bit	0

Here the behavior of the design seems correct, in the first image the general behavior can be seen and in the second, which is the first roomed, it can be seen that the states actually change every 16 pulses of the sample enable. Also its evident that the transmission starts when the Tx_WR is activated for one clock period.

Here is also the post synthesis timing simulation:



Results are the same except from same spikes that last some pico seconds.

Experiment: There was no experiment on the FPGA.

Part C – UART Receiver:

Implementation: The receiver is responsible for receiving data at precise times, validating its accuracy, and detecting any errors during transmission. It samples incoming data at a rate 16 times faster than the transmitter's sending rate to ensure accurate and precise readings.

The receiver detects two types of errors:

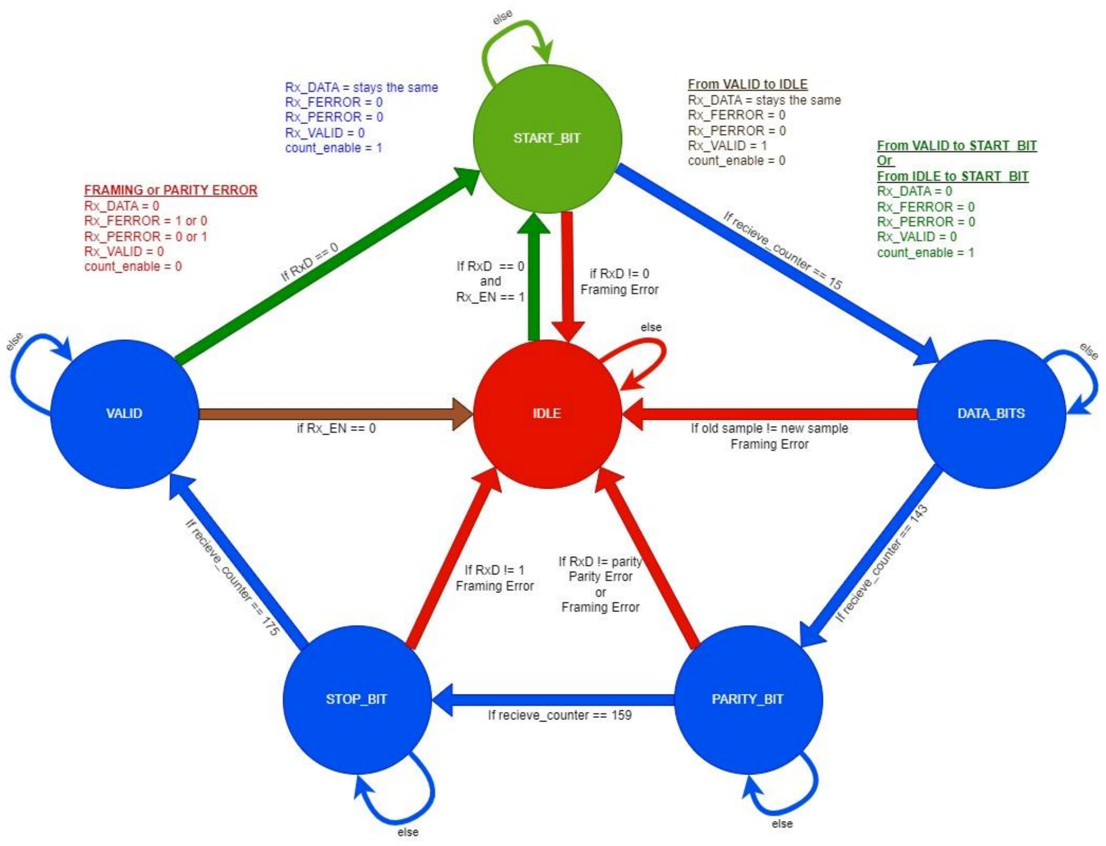
1. *Parity Error*: This occurs when the received data has a parity bit that does not match the transmitted parity bit, indicating possible data corruption.

2. *Framing Error*: This error occurs if the sampled data unexpectedly changes state, suggesting a misalignment in timing, as the transmission should remain stable during this interval.

To manage timing, an 8-bit counter is used to count sample_enable pulses and is enabled every time the data collecting starts. Since the transmitter sends each bit every 16 pulses, the receiver can be calibrated to read data every 16 pulses and to perform additional checks in between to detect framing errors.

This functionality is implemented with a state machine of higher computational complexity than the transmitter's. The data collection process starts with the detection of the start bit and concludes with the stop bit. If no errors are detected, the received data is considered valid.

State machine:



The receiver module is designed using a Mealy state machine. Initially, while the module is disabled, it remains in the IDLE state, which also serves as the OFF state. When the Rx_EN signal is activated, the receiver prepares to detect the start bit. Upon detecting this start bit, it transitions to the START_BIT state, enabling the counter and beginning continuous sampling to check for any framing errors. If no framing errors occur, and the receive counter reaches 15, the module advances to the DATA_BITS state.

In the DATA_BITS state, the receiver samples the new data bit at counts 16, 32, 48, ..., up to 128. For each sampled bit, the value is checked against subsequent samples that should belong in the same state of transmission. If a discrepancy is detected, a framing error is flagged. Otherwise, when the counter reaches 143, the receiver moves to the PARITY_BIT state.

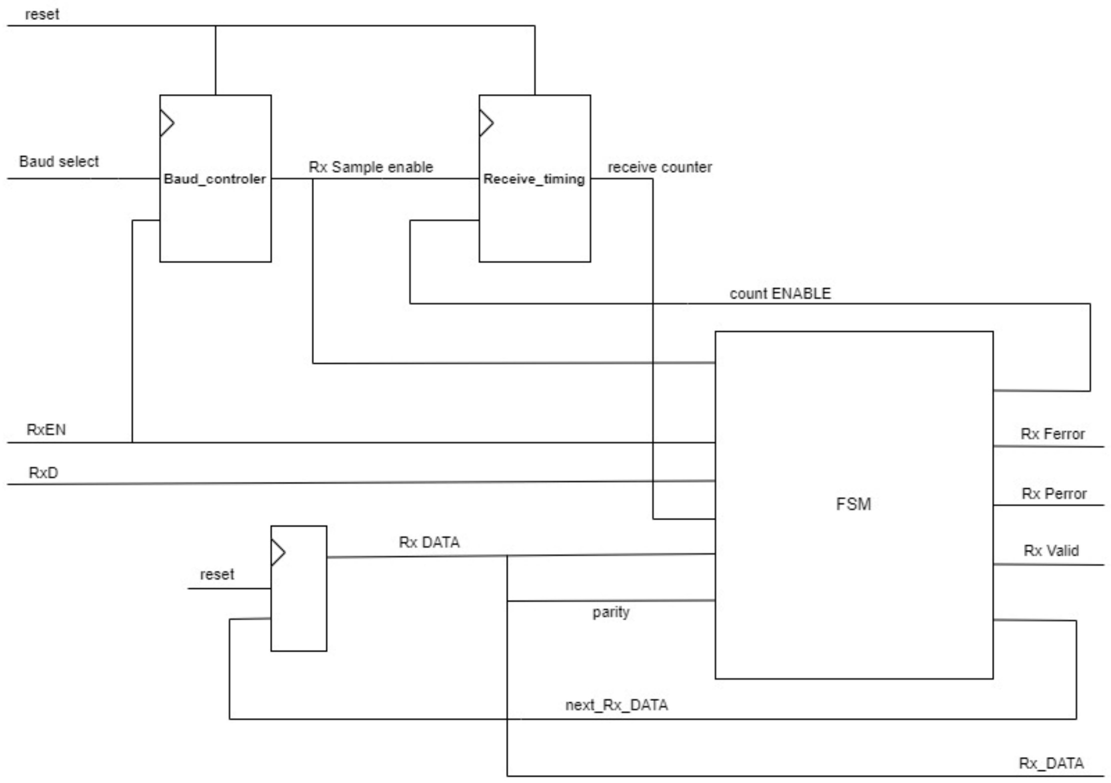
In the PARITY_BIT state, the receiver samples the first parity bit at count 144. If this bit matches the parity expected from the received data, there is no parity error. The receiver then continues sampling to monitor for any framing errors during the next 15 counts.

Finally, if no framing error occurs, the receiver enters the STOP_BIT state at count 159. Here, the stop bit is expected to remain at 1 without any changes for 15 samples, confirming the end of transmission. If no errors are detected during this period, the receiver transitions to the VALID state at count 175, marking the data as valid and disabling the counter. The receiver remains in this state until a new start bit arrives, or the receiver is powered off.

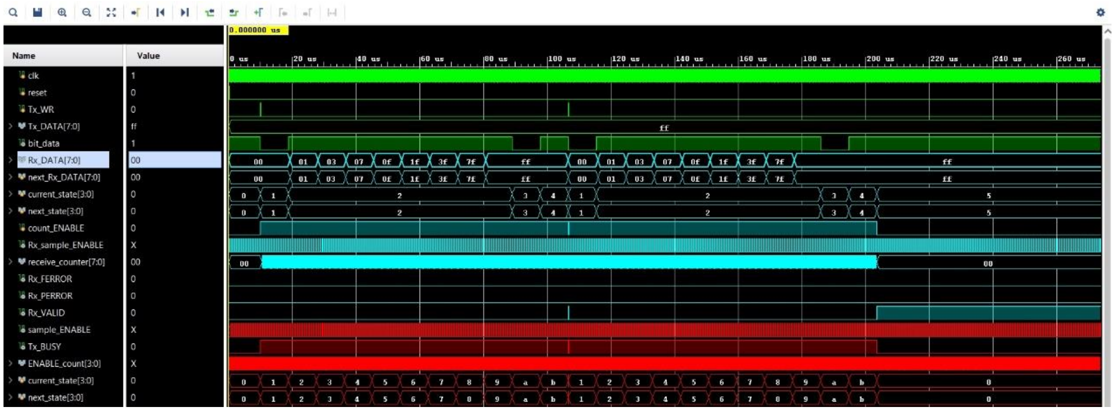
If a framing or parity error occurs at any point, the receiver returns to the IDLE state, outputting an error signal for one clock cycle. The user is responsible for the systems reset to realign the data, as continued operation without resetting may lead to false data and even erroneous valid signals.

This state machine is also implemented with a sequential part that changes the states, and a combinational part for the login of every state. Also, in order to avoid a latch while saving the incoming data a next_Rx_DATA variable is used is the combinational part and the Rx_DATA variable is changed along side with the state variable inside the sequential always block.

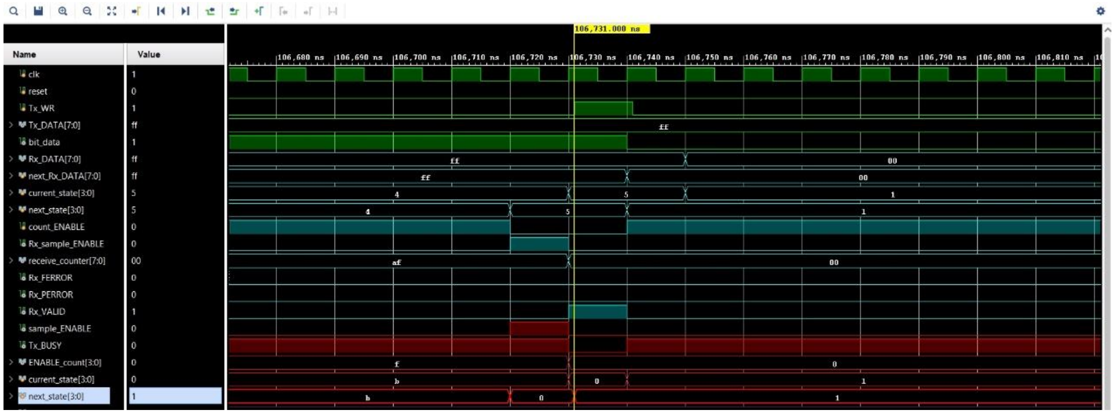
Modules:
Baud_controller: (same as above)
Receive_timing: The module that implements the receive counter (sequential)
Uart_receiver: The module that combines the above with the state machine and the parity bit calculation.



Simulation: For the simulation to check if the errors actually work three different receiver modules are tested. One that does not calculate the parity correctly, one that works on a different baud rate from the transmitter and one that operates normally. In the testbench these three modules are instantiated and a transmitter that provides the data. The first image shows the simulation of the normal receiver. The variables in blue correspond to the receiver and the ones in red to the transmitter.

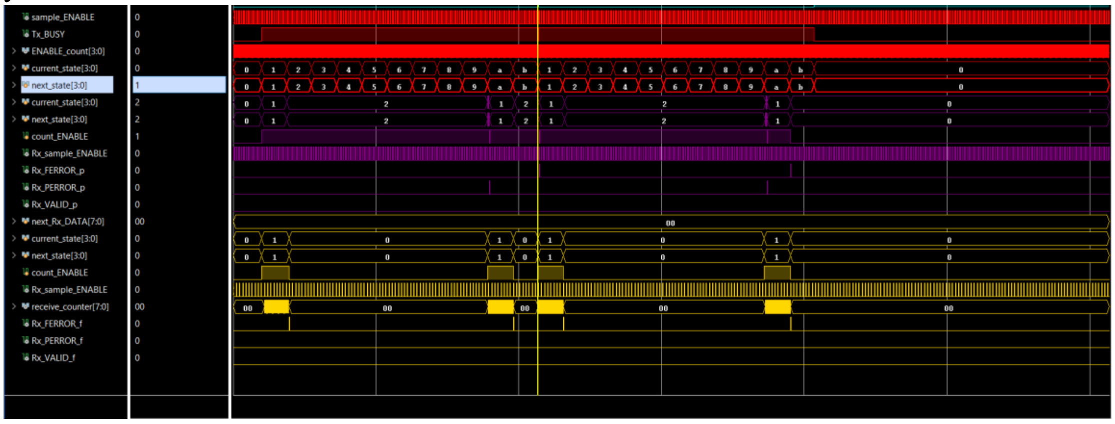


For the red: State 0 is IDLE, 1 is START_BIT, 2 to 9 are the data bits, a is the PARITY_BIT and b is the STOP_BIT.
For the blue: state 0 is IDLE, 1 is START_BIT, 2 DATA_BITS, 3 PARITY_BIT and 4 STOP_BIT
The data transmitted is ‘hFF. This test demonstrates how the states of the transmitter and the receiver are aligned and how the valid variable is activated when the transmission is over. The first valid activation is so short to demonstrate a borderline case of the Tx_WR activating the soonest that it is possible. Also, it is evident how the Rx_DATA changes during the DATA_BITS stage of the receiver.

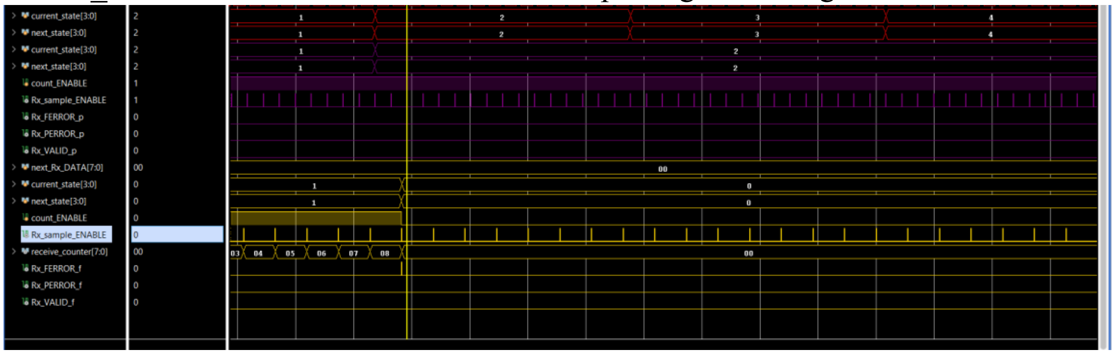


This is the borderline case ,where the Tx_WR is enabled as soon as the transmitter is at the IDLE state, which does not affect the functionality of the receiver and the valid signal remains high for one clock cycle giving the user the chance to read the data.

In the next image in purple is the module with a faulty parity calculation and in yellow the one that works with a different baud rate.



At the PARITY_BIT state the purple receiver detects a parity error as it should and after that, because the transmission starts again due to Tx_WR even if the receiver is not in a position to handle that, it detects a framing error and that is the reason why the system should be reset after an error. Furthermore, the yellow receiver detects a framing error as soon as the transmission state changes from the START_BIT to the FIRST_BIT as it runs slower and is not still expecting that change.



Here is why the framing errors occurs, even though the transmitter moved to state 2 the receiver is still in 1 and at the first Rx_sapmle_enable that the transmitter is at 2 and the receiver is at 1 the FERROR flag activates.

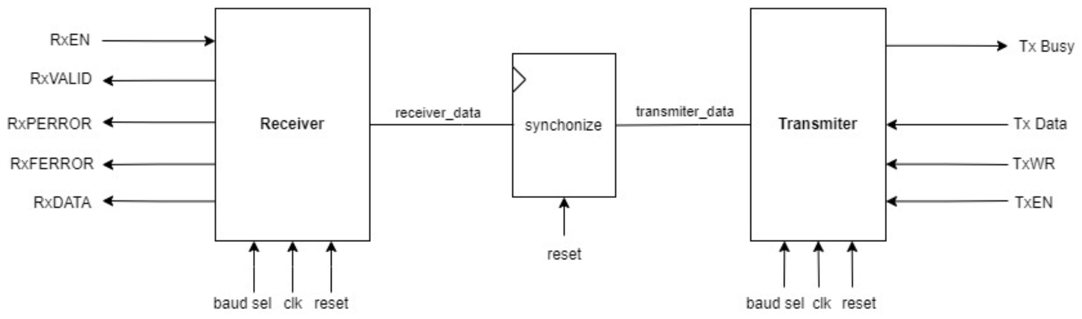
This test has not simulated after a synthesis as a top module to combine a four modules was not created, but that does not matter a the UART will be synthesized and simulated in the next part.

Experiment: There was no experiment on the FPGA.

Part D – UART module:

Implementation: This module integrates both the transmitter and receiver to simulate communication between them without the need for a physical cable. The only additional component is a synchronizer, which syncs the TxD signal with RxD, as the transmitter and receiver may sometimes operate at different frequencies.

Modules:
uart_transmiter: (same as above),
uart_receiver: (same as above),
uart: combines the previous two and syncs the data transferred between them.

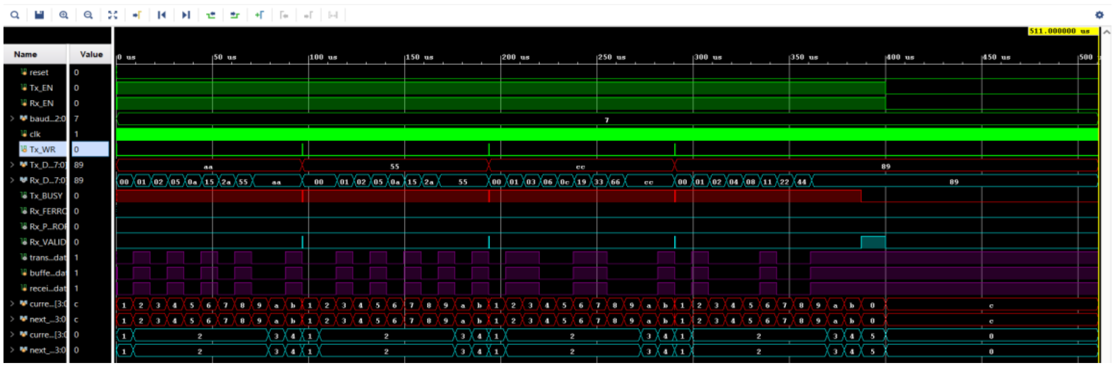


Simulation: In this simulation two test benches are designed. One that works with an FSM and one that works with just an initial block. The correct way to approach this simulation is the FSM test bench but the second test bench is useful for the simulation of the synthesized design.

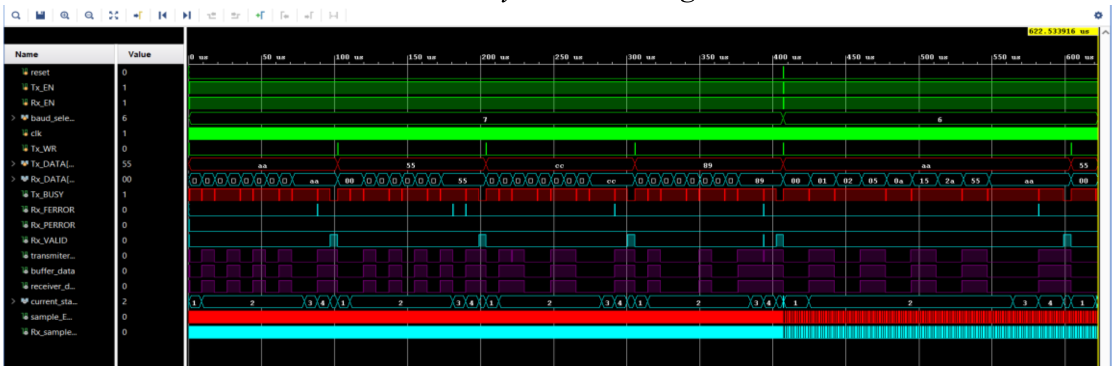
The FSM test transmits four data strings and if it detects an error or when the data is been transmitted it stops the simulation, the other test transmits the same data but for all possible baud rates.

Normal uart

FSM functional simulation



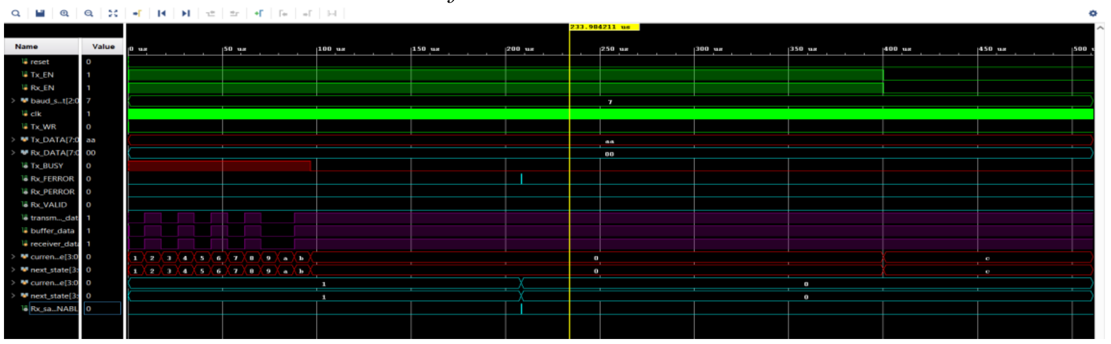
Second test Post synthesis timing simulation



the ferror signals are just spikes not actual ferrors.

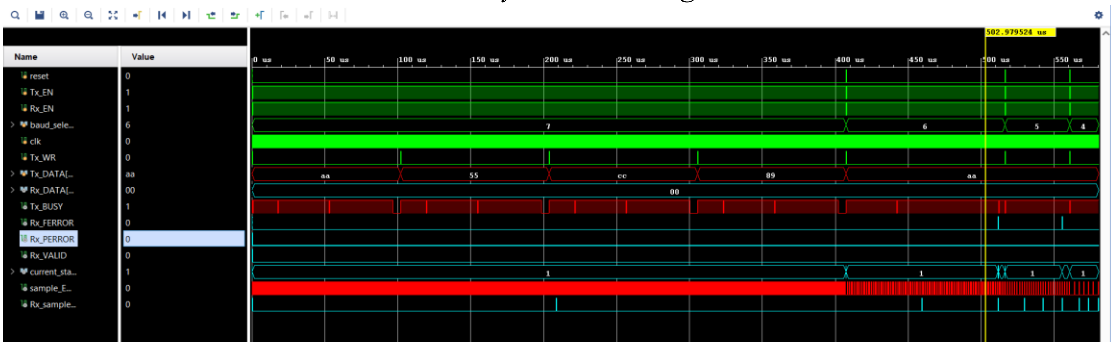
Ferror uart

FSM functional simulation



When the first rx_sample_enable is pulse activates the receiver immediately outputs a ferror signal

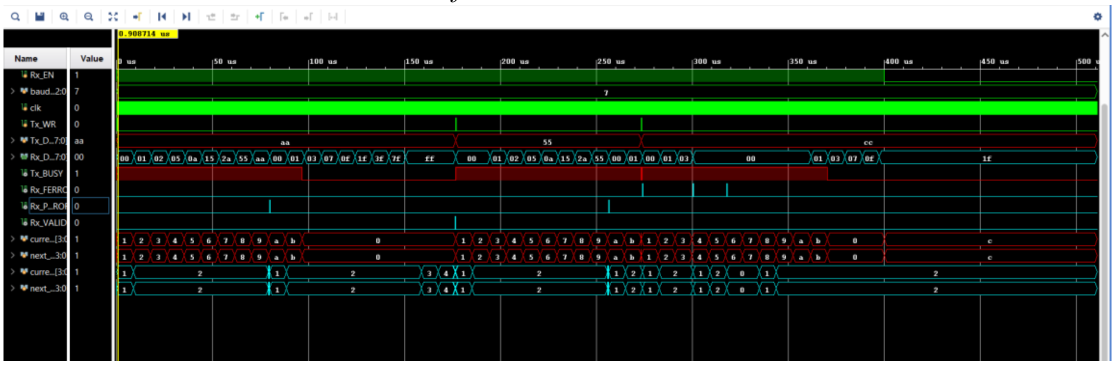
Second test Post synthesis timing simulation



Here by chance the first two samples of the receiver do not flag a ferror as the bits there happen to be 0 like the start bit (the receiver is at the start bit state) but at the third it detects the framing error. The data is not shown as the synthesized design does not provide this signal.

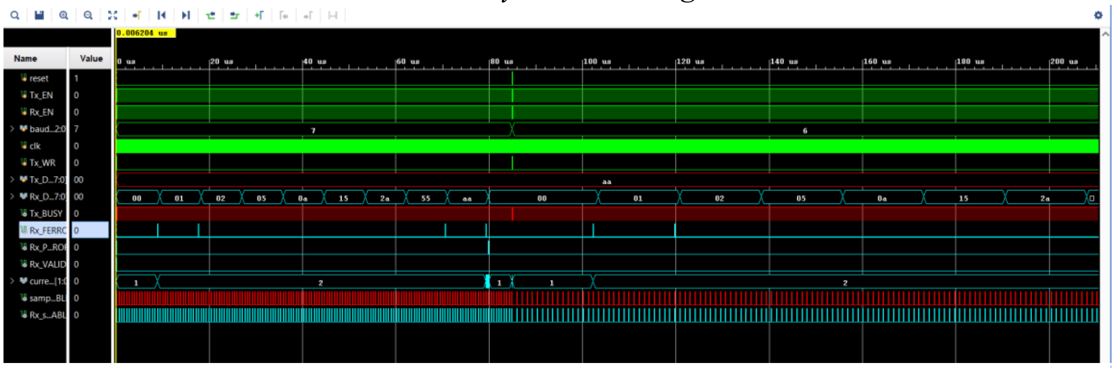
Perror uart

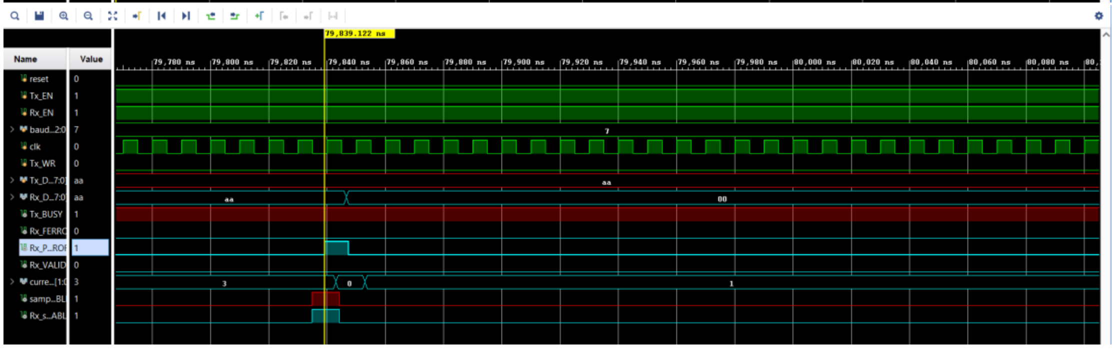
FSM functional simulation



When the parity_bit state arrives the receiver flags a parity error but because the system is not reset an erroneous valid signal occurs.

Second test Post synthesis timing simulation





Experiment: There was no experiment on the FPGA.

Conclusion: This project presented several significant challenges, including timing and synchronization between the receiver and transmitter, as well as the design of the receiver's state machine. However, the most difficult challenge I faced was creating the testbench for Part D (UART). The issue arose because, for some reason, the FSM testbench did not work with the synthesized design. Through testing and debugging, I discovered that no testbench using an always or forever block would work correctly. Consequently, I created a second test, as I found that the synthesized design would only function properly when the simulation used an initial block.