ECE333 - Digital Systems Lab

– Lab Report –

Lab 2 - Universal Asynchronous Receiver Transmitter - UART

submitted by

Liaskonis Spyridon

December 4, 2022

Liaskonis Spyridon
sliaskonis@uth.gr
Student ID: 03381

Lab Report Summary

The following report consists of six sections. The first section is an introduction to the aim of this project. The sections two, three, four and five are more detailed paragraphs about each part of the assignment. They focus mostly on the implementation of each part as well as how we verified the implementation using testbenches etc. The last section is the conclusions in which we talk about the progress of the assignment in fields such as design and verification as well as some difficulties and how we overcame them.

Contents

1 Introduction

The aim of this project is the implementation of a Universal Asynchronous Transmitter Receiver (UART). The implementation is divided into 4 parts, each one developed separately and with separate test-files/testbenches. Each part focuses on a different objective however each part is very important for the implementation of the UART. The objectives of each part are the following:

- Part A: Implementation of a baud rate controller, responsible for providing the receiver and the transmitter with a baud rate that they would both use in order to communicate.
- Part B: Implementation of the Transmitter, responsible for transmitting an 8-bit value with a given baud rate.
- Part C: Implementation of the Receiver, responsible for sampling/receiving the data sent by the receiver.
- Part D: Implementation of the UART, a connection between the transmitter and the receiver for a serial transfer of data between them.

All of the objectives above were successfully implemented. In particular for the part A, the baud rate controller was implemented using a counter that counts clock cycles. The counter's highest value was then picked by a multiplexer according to the baud rate that we wanted. The highest the baud rate the smallest the counter's value would be, meaning that the transmitter/receiver would achieve highest transmitting speeds. The second as well as the third part were then implemented using two FSMs with different states according to the expected bit. Lastly, the final part was implemented by putting all the modules created in the previous parts together and join them with a wire from which they communicated.

2 Part A - Implementation of Baud Rate Controller

2.1 Implementation

The first part of the UART implementation is the implementation of a Baud Rate Controller responsible for providing the circuit with the right sampling signal ac-

cording to the chosen baud rate. The baud rates that the Baud Controller will implement are the following:

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 baud_controller module accepts 3 inputs:

- Reset: 1-bit reset signal used to initialize the circuit
- Clock: 1-bit clock signal
- Baud_select: 3 bit value used to determine the baud rate

and has one output: sample_ENABLE, a 1-bit signal that is the sampling signal that comes from the chosen baud rate. For each baud rate, a period for the sampling signal needs to be determined. This period is calculated using the formula $T_{sc} = \frac{1}{16 \cdot BaudRate}$, and after this period of time the signal sample_ENABLE will be set to one for one clock cycle.

The Baud Rate Controller consists of one sequential circuit that implements a 15-bit counter and a multiplexer. The counter here acts as a clock cycle counter and is responsible for counting the quantity T_{sc}. The multiplexer is used to determine the counter's highest value according to the baud_select. Thus, for each baud_select value there is a value that once the counter reaches it is then set back to 0 and the sample_ENABLE signal is set to 1 for one clock cycle. This way according to the baud rate chosen, we get the right period for the sampling signal.

Here is a table containing the T_{sc} and counter's value as well as the error for each baud rate:

Baud_Sel	Baud Rate	$T_{sc}(\cdot 10ns)$	Counter's value	Error (%)
0 0 0	300	20833.3333	20833	0.0016
0 0 1	1200	5208.3333	5208	0.0064
0 1 0	4800	1302.0833	1302	0.0064
0 1 1	9600	651.0416	651	0.0064
1 0 0	19200	325.5208	326	0.147
1 0 1	38400	162.7604	163	0.147
1 1 0	57600	108.5069	109	0.4524
1 1 1	115200	54.2534	54	0.4671

The counter in this case gets increased every 10ns. So in order to find the counter's highest value for each baud rate all we do is divide the T_{sc} quantity by 10. Our counter can only count integer values which leads to us getting an error since the T_{sc} quantity has also a decimal part which cannot get counted. In order to get the smallest errors we round the counter's value towards the closest integer i.e. when $T_{sc} = 162.7604$ instead of setting the counter's highest value to 162 we set it to 163 since its closer to the expected value.

2.2 Verification of implementation

For the verification of the implementation of part A a simple testbench was used. The testbench consisted of the instanciation of the baud_controller and an initial block from which we set the baud_select value to the one we needed and then observed the waveforms. In order to check if the module worked we had to see if after the counter reached its maximum value (according to the baud_select) the signal sample_ENABLE was set to 1 for one clock cycle.

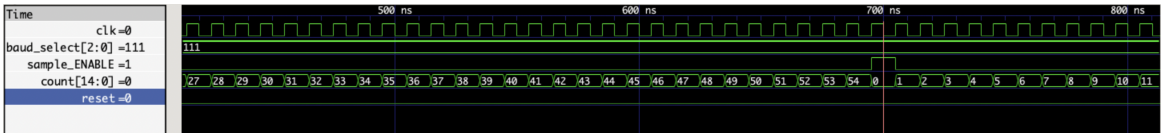

Figure 1: Testbench for part A. Waveforms for baud_select = 111.

3 Part B - Implementation of UART Transmitter

3.1 Implementation

For the implementation of the transmitter we used the four following modules:

- Baud_controller: module implemented on part A
- Check_parity: module used to check an 8-bit value for even parity
- FourBitCounter: a 4-bit counter
- DataCounterTx: module implementing an FSM used for transfering the 8 data bits and start, parity and stop bit

Here is the dataflow of the transmitter:

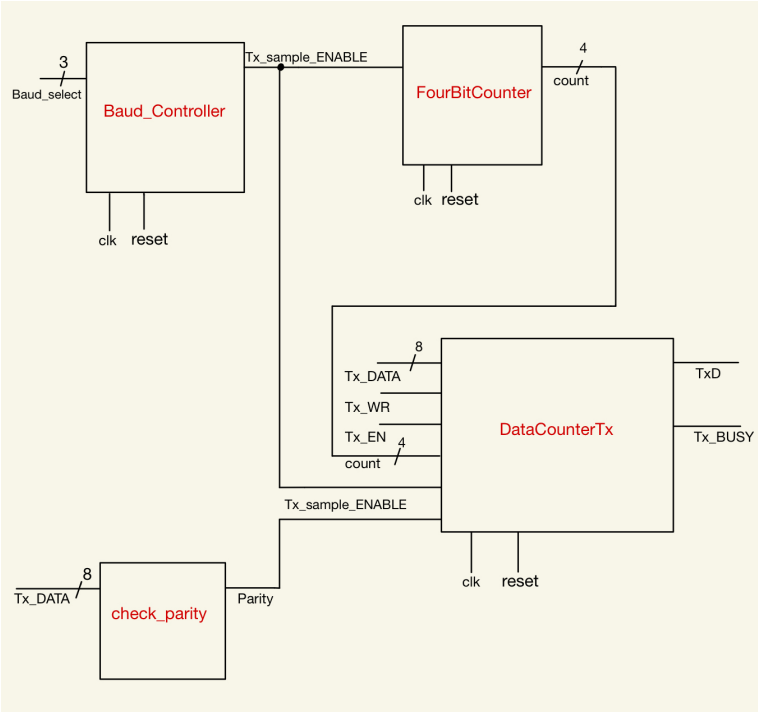

Figure 2: Transmitter dataflow

Now lets see how each module used to implement the transmitter works:

1. Baud_Controller: baud controller provides the Tx_sample_ENABLE signal that marks the active cycles for the transmitter. On the active cycles the transmitter can transmit data.

2. Check_parity: check_parity module consists of one combinational circuit that implements 7 XOR-gates. The module has one 8-bit input, that's the data we check for parity, and one 1-bit output called parity that indicates if the number of 1s is even or odd. If the number of 1s is even then the parity bit will become 0, else is will become 1. Here's how the XOR-gates are connected:

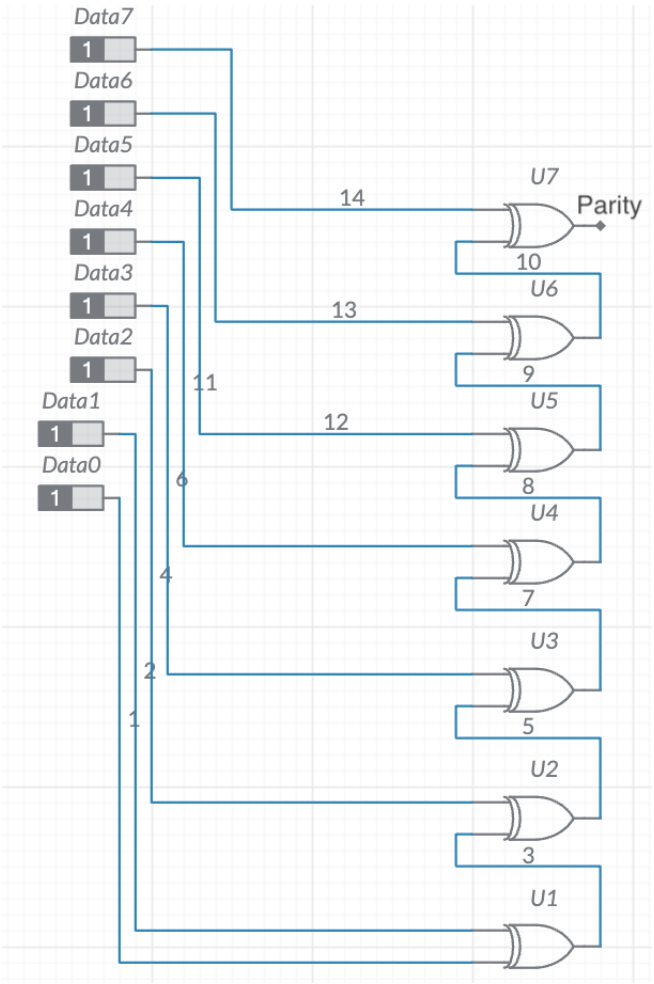

Figure 3: Circuit to check for even parity

3. FourBitCounter: the fourBitCounter module implements a simple 4-bit counter. It has three 1-bit inputs: reset, clock and enable and one 4-bit output that's the counter's value. The module consists of one sequential circuit that controls the counter's values. When the reset is activated the counter is set to 0 else when the enable system is activated the counter gets increased by one.

In this case the enable is the Tx_sample_ENABLE signal that comes from the baud_Controller and thus our counter is used to count active cycles for the transmitter (cycles that the transmitter can transmit data).

4. DataCounterTx: the dataCounterTx module is responsible for transmitting the 8 data bits as well as the start, stop and parity bits. The module has the following inputs:

   - Clock signal
   - Reset signal: used to initialize the circuit.
   - Tx_DATA: an 8-bit value containing the data that will get transmitted.
   - Tx_EN: a signal indicating when the transmitter is activated.
   - Tx_WR: a signal used to transfer the data we want to transmit from the Tx_DATA to a register from which they will get transmitted.
   - Tx_sample_ENABLE: baud_controller's output.
   - Count: fourBitCounter's module output.
   - Parity: check_Parity's module output.

   and the following outputs:

   - TxD: a 1-bit output used for the serial transfer of the data.
   - Tx_BUSY: a 1-bit signal that indicates when the transmitter is transmitting data. When it's set to 0 the transmitter is not transmitting data and a new symbol can be written in the register from which the data is getting transferred. Else when it's set to 1 it means that the transmitter is transmitting the data written in the register and new data cannot get transmitted till it is set back to 0.

   The transmitting of the data is achieved using a moore machine (FSM). First, once the Tx_WR is set to 1 for one clock cycle and while the Tx_BUSY is set to 0, the data from the Tx_DATA are transferred to a register. That is happening from the first always block in the module, which is a sequential circuit that implements a register. After that there are two more always blocks. One sequential and one combinational that implement the FSM. The first (sequential) always block is used to assign the new value to the current state. When the reset is activated the current state is set to the idle state. Else when the 4-bit counter counts 16 Tx_sample_ENABLE cycles, the value of

the next state is assigned to the current state. The second (combinational) always block is used to describe the next state and it's output, according to the current state. The FSM consists of 12 states . The current FSM has only one input which is the Tx_EN signal. When the signal is set to 1 we move from one state to the other. When the Tx_EN is set to 0 then the current state remains the same. In the context of the current assignment the Tx_EN signal will always remain at 1 until the transmission is over. Thus there is no point in changing the state if the Tx_EN signal is disabled.

The outputs of the FSM are the TxD and the Tx_BUSY signal. According to the current state a value is assigned to the TxD. This value can be 1 if the current state is the idle state or stopBit state, 0 when the current state is the startBit state, the value of the parityBit (calculated from the check_parity module) when the current state is the parityBit state or a value from the data stored in the register containing the data we want to transfer if the state is one of the 8 dataBit states. The second output of the FSM (Tx_BUSY) is set to 0 only in the idle state since that's the only state where the transmitter is not transmitting any data. In every other state it's value is set to 1 meaning that the transmitter is busy and no new data can get transmitted until the ones in the register are fully transmitted.

Here is the FSM and all its states:

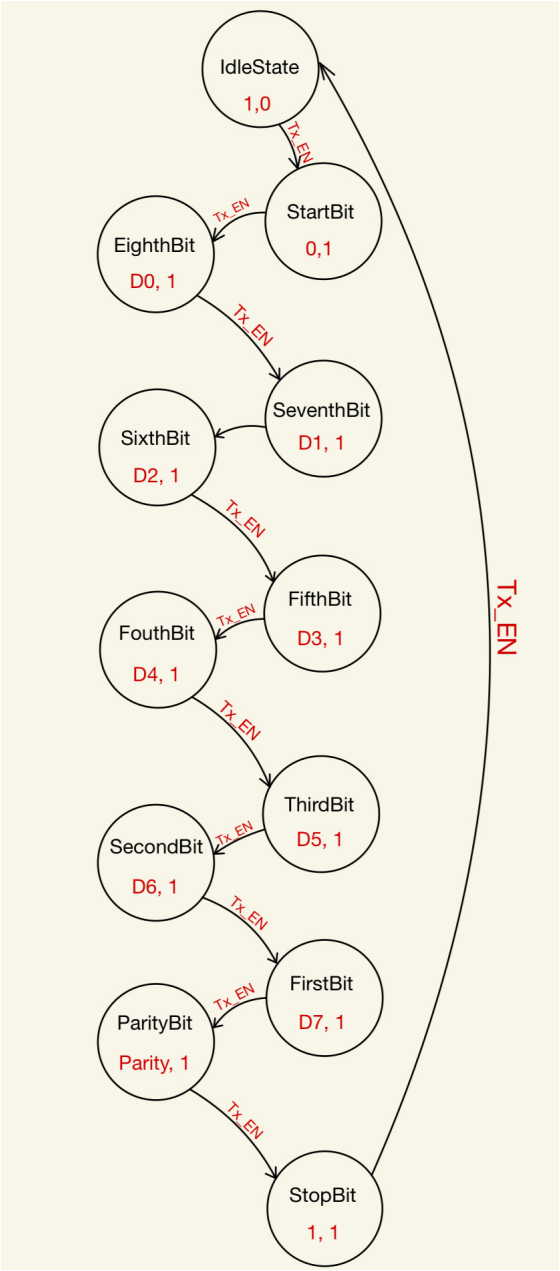

Figure 4: FSM for the transimitter

3.2 Verification of implementation

For the verification of the implementation of the transmitter a similar testbench to the one of the part A was used, consisting only of the instantiation of the top module

and an initial block. In the initial block we first set a value for the baud_select and a value for the Tx_DATA and the we activate the reset. After that we set the Tx_EN to 1 so that the transmitter gets activated and also we set the Tx_WR to 1 for one clock cycle so that the data from the Tx_DATA get written in the register. After that we check the waveforms and observe the values TxD, currentState and count in order to see if the transmitter worked properly. Here are some screenshots of the simulation:

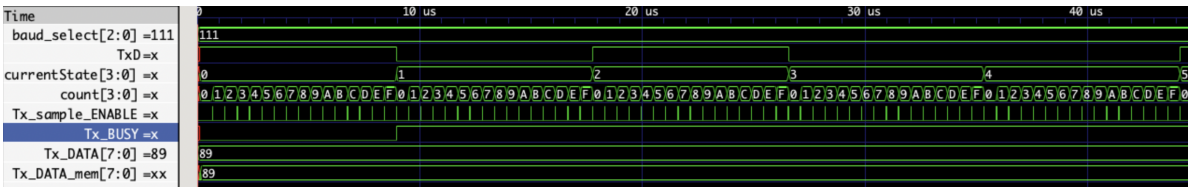

Figure 5: Testbench part B - Waveforms

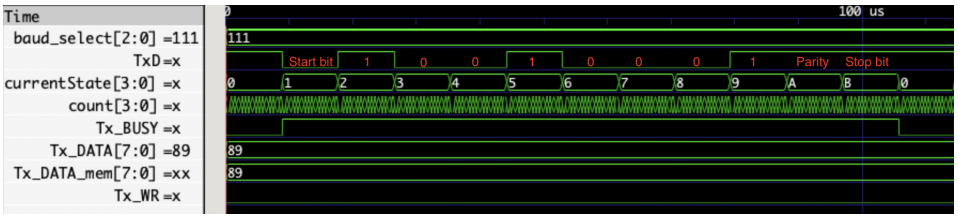

Figure 6: Testbench part B - Waveforms

4 Part C - Implementation of UART Receiver

4.1 Implementation

For the implementation of the receiver we used the following modules:

- Baud_Controller: module implemented on part A.
- Sync_input: a module used to sync the asynchronous serial input.
- Shift_register: a module used to (right) shift data to an 8 bit register.

- Check_parity: module used to check an 8-bit value for even parity.
- DataCounterRx: module implementing an FSM used for sampling and receiving the data.
- Write_data: a module used to write the data received on a register.

Here is the dataflow of the receiver:

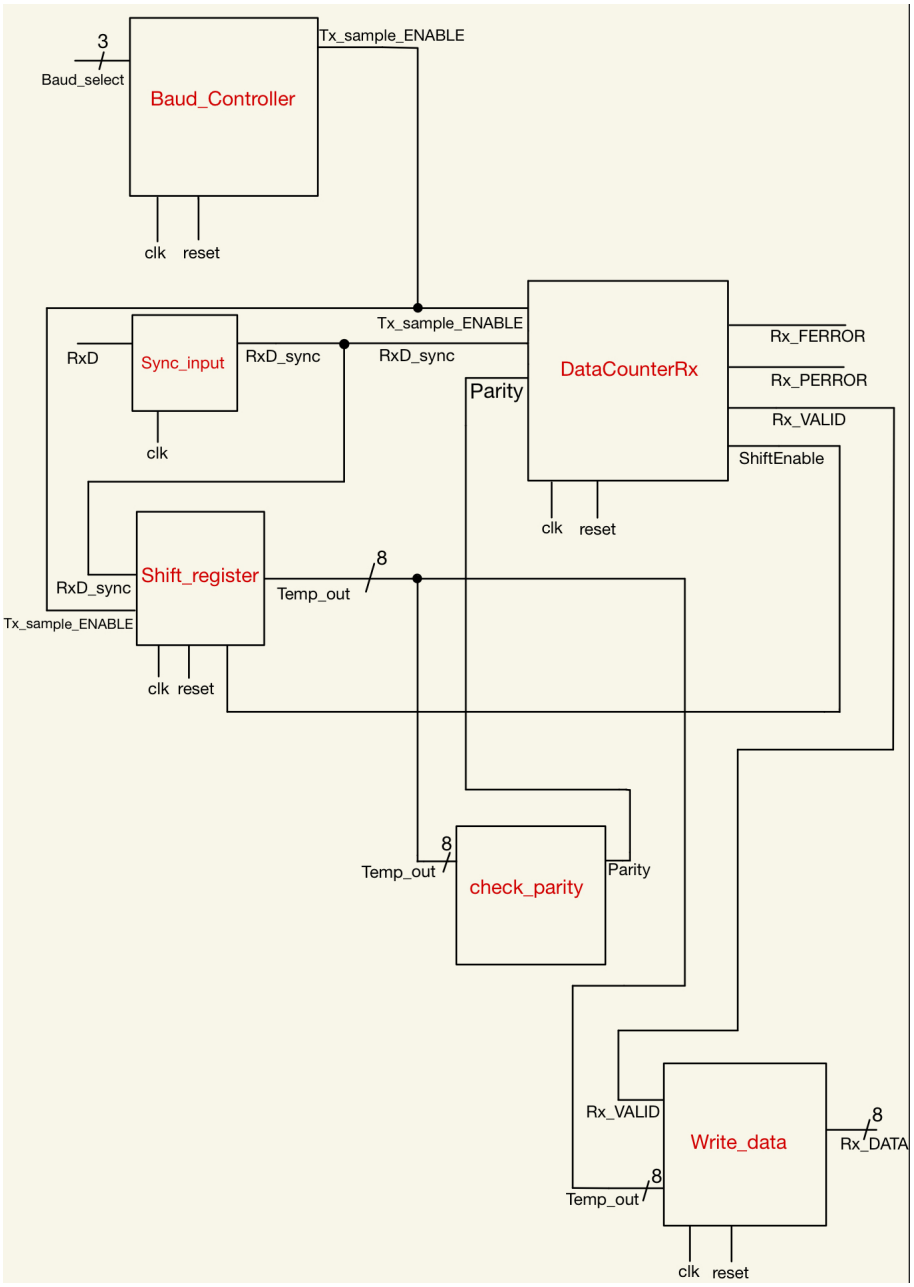

Figure 7: Receiver dataflow.

Now lets see how each module used to implement the receiver works. Here we are not going to talk about the baud_controller and check_parity modules since they are the same ones used in the previous part and they are used for the same purpose in the receiver as well.

1. Sync_input: sync_input module has two 1-bit inputs, clock and signal, and one 1-bit output signalSync. The module consists of two flip flops that are used to synchronise the signal input with the clock.

2. DataCounterRx: the dataCounterRx module is the most important module of the receiver since it's the one responsible for the sampling of the data. It has the following inputs:

   - Clock signal
   - Reset signal: 1-bit signal used to initialize the circuit.
   - Rx_sample_ENABLE: baud_controller's output, the signal indicating the cycles that the receiver should sample the data.
   - RxD_sync: the synced RxD signal after it went through the sync_input module
   - Parity: the check_parity module's output that the dataCounterRx will compare with the one received from the RxD_sync and determine if the data were correct or not.

   and the following outputs:

   - Rx_FERROR: a 1-bit signal used to inform the system if a framing error was occurred, meaning that while sampling the data the start or stop bit wasn't correctly sampled.
   - Rx_PERROR: a 1-bit signal used to inform the system if a parity error was occurred, meaning that the parity that got received wasn't the one expected.
   - Rx_VALID: a 1-bit signal indicating if the data received was correct. This signal is set to 1 at the end of the sampling only if there were no errors and it keeps the value 1 until the next start bit is received, meaning that new data are coming.

- ShiftEnable: a 1-bit signal that is used for shifting the data received in a shift_register (as an enable signal meaning that when its value is set to 1 whatever data is received from the RxD will be shifted in the register).

The module dataCounterRx consists of three always blocks. Two sequential and one combinational. The first sequential always block together with the combinational one implement a mealy machine (FSM) while the second sequential block implements a 4-bit counter that's responsible for aligning the sampling at the middle of each bit received. The counter is set to 0 when reset is activated or when countAligned is activated. Else the counter starts counting when the signal countEnable and Rx_sample_ENABLE are both set to 1. We will explain the origin and use of the countAligned and countEnable signals in the next paragraph where we will talk about the implementation of the FSM.

The FSM implemented in this part consists of 12 states that are the same with the ones on the FSM from part B. The sequential part of the FSM here is responsible for assigning the new value on the current state. Thus, when the reset is activated the FSM's current state is set to the idle state. Else when the Rx_sample_ENABLE signal is set to 1 the currentState takes the nextState's value. The combinational part of the FSM is then responsible for assigning a value to the outputs of the FSM according to the current state and also for describing the next state according again to the current one.

Here is the FSM:

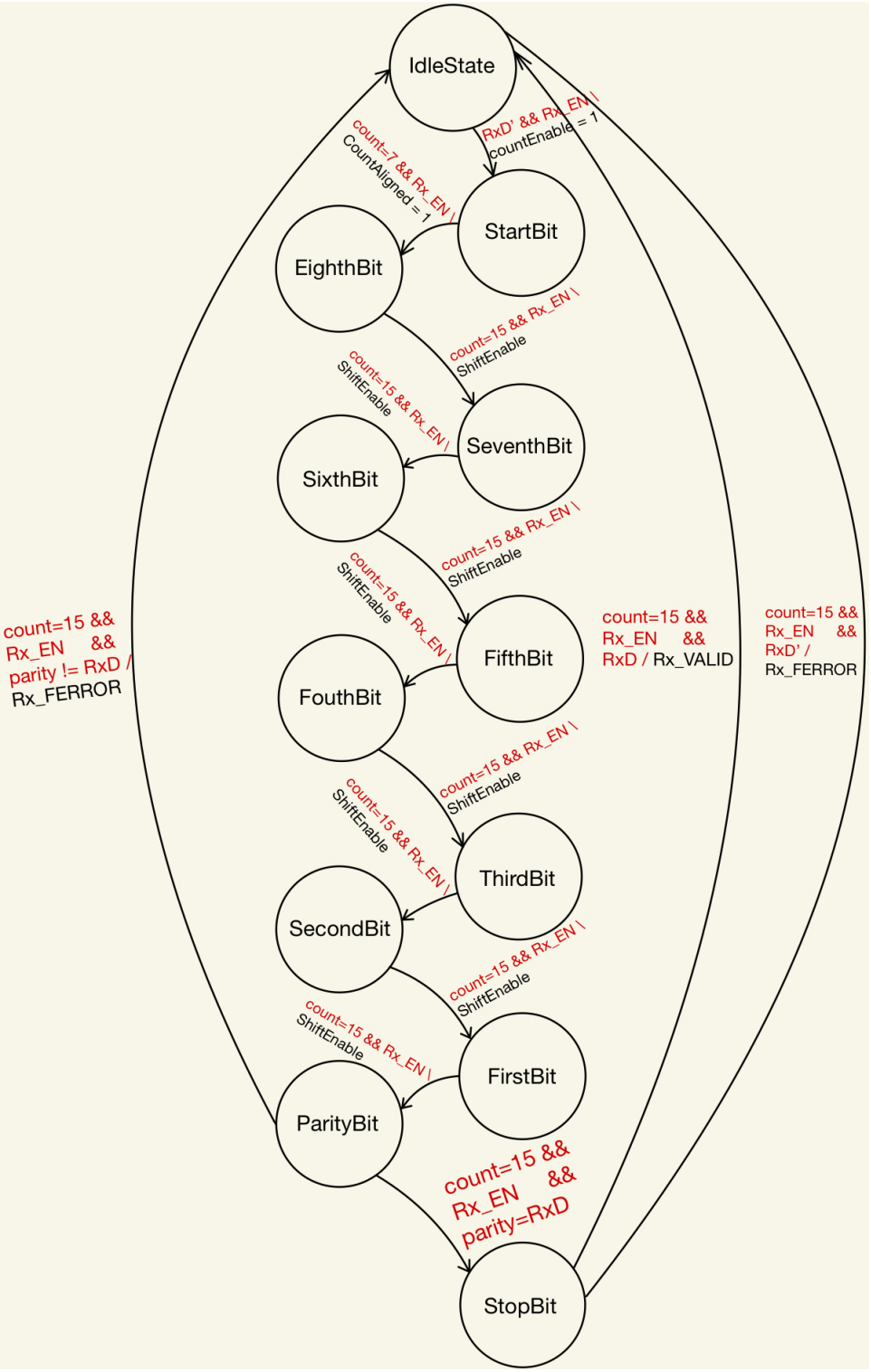

Figure 8: FSM for the receiver.

The inputs of the FSM here are the counter's value, the RxD, the parity and the Rx_EN signal that indicates that the receiver is activated. Like said above once the reset is activated the receiver goes to the idle state. There once the start bit comes (the RxD is set from 1 to 0) the FSM goes to the next state and activates the signal countEnable which makes the counter start counting. The counter counts Rx_sampe_ENABLE cycles. So, when the counter counts 8 cycles (thus its value is equal to 7) we are in the middle of the start bit. At that point we move to the next state and the signal countAligned is set to 1. Once this signal is set to 1 the counter resets it's value to 0 and starts counting again. Since in this point we are at the middle of the start bit, in order to get in the middle of the next bit and start the sampling we have to count 16 Rx_sample_ENABLE cycles (since each bit is transmitted for 16 cycles). Once we count those cycles and we are at the middle of the eighth bit we set the signal shiftEnable to 1 which lead to the shifting of the value of the RxD at that moment into a register. Also at the same time we move to the next state and repeat this process. This process wil be repeated for al the 8 bits as well as the parity and the stop bit.

Notice that due to lack of space not all the outputs are presented in the FSM. We only present the ones who's values change when the state changes. Also for the same reason there are only arrows pointing to one way. When the state is not changing (thus when the inputs aren't the ones shown on the figure 5) the state remains the same so the arrows missing would point at the same state and would output the same values.

3. Shift_register: the shift_register module is responsible for shifting the values of the RxD into a register. It has 4 1-bit inputs and one 8-bit output. The inputs are the clock, the reset, the Rx_sample_ENABLE signal and the Rxd_sync. The output is an 8-bit value called tempOut that contains the data that were shift in the register. The module consists of one sequential module. When the reset is activated, the tempOut is set to 0. Else when shiftEnabe and Rx_sample_ENABLE is set to 1 the data from the RxD_sync is shifted in the register (right shifted since the data that comes first is the LSB of the data transfered).

4. WriteData: the writeData module is used to write the data received from the tempOut to the Rx_DATA register. It has three 1-bit and one 4-bit inputs, the clock, reset, the Rx_VALID signal and the tempOut and one 8-bit output,

the Rx_DATA. This module consists of one sequential module. When the reset is activated the Rx_DATA gets the value 0. Else when Rx_VALID is set to 1 (meaning that the receiver has successfully received the 8 bits) the data stored in tempOut will be written in the Rx_DATA.

Pretty much everything on the implementation went well except one thing. After running the synthesis on Vivado for the current part, we got a warning about a latch created by the Rx_VALID signal on the FSM. However this problem was not solved since we couldn't find a way to keep the Rx_VALID enabled until a new start bit would arrive.

4.2 Verification of implementation

For the verification of the implementation of the receiver a similar testbench to the testbench from part B was used. Again the testbench consisted of the instanciation of the top module and an initial block. In the intitial block we first chose a baud_rate (in this case the baud_select was 111), set RxD to 1 and set the reset to 1. After that we started changing the RxD value each 8800 ns, the time that the transmitter also used between each digit. After that we observed the waveforms. Here are the results of the simulation for the current test:

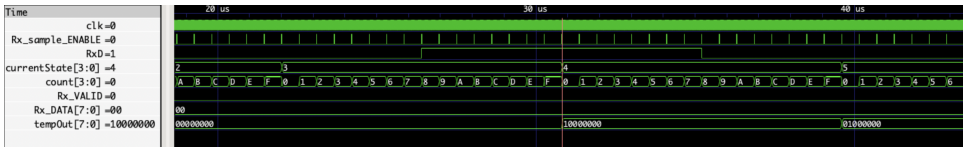

Figure 9: Testbench Part C - Alignment of the sampling

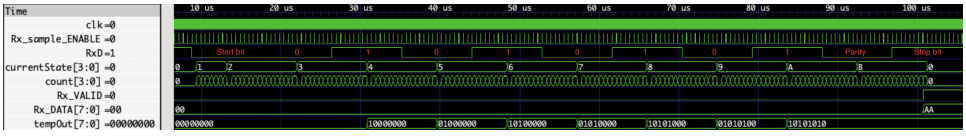

Figure 10: Testbench Part C - Receiving the value "AA"

5 Part D - Implementation of UART Transmitter-Receiver for serial data transfer

5.1 Implementation

The UART Transmitter - Receiver pair was implemented by combining the uart_transmitter and uart_receiver modules in one. The top module this time is called UART and it consists of just the instanciation of the two modules. Here is the dataflow of the UART:

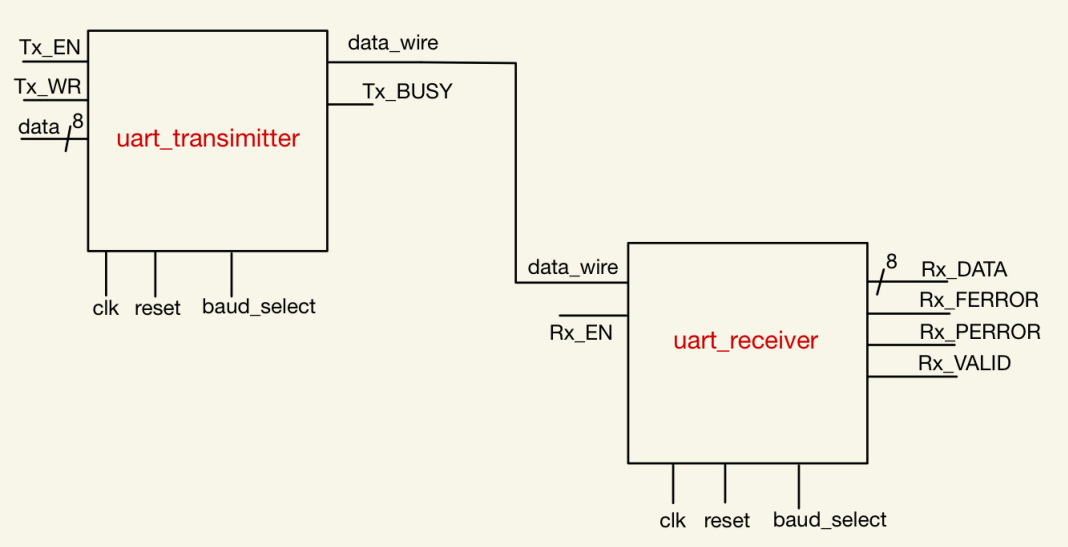

Figure 11: UART dataflow.

Also as shown above the connection between those two modules was made with a wire from which the transmitter sends the data and the receiver samples it. Both the receiver and the transmitter in this case work with the same baud rate.

5.2 Verification of implementation

For the verification of the implementation of the UART we used again a similar testbench to the other two ones. The testbench consists again of the instatiation the the top module and one initial block. In the initial block we select the baud rate, reset the circuit and set a value to the data that we want to transmit. Then we set the Tx_WR to 1 for one cycle so that the data is written from the Tx_DATA

to the register. After that we enable the receiver and the transmitter by setting the Rx_EN and Tx_EN signals to 1. Now the first symbol is getting trasmitted. Once its done with the first symbol we change the data to the next symbol. The message that we transfer consists of 4 symbols and its the following: "10001001(89) 11001100(CC) 01010101(55) 10101010(AA)". In order to know when each symbol has been transmitted we use the wait function that waits until the Tx_BUSY is set to 0 and the Rx_VALID is set to 1, meaning that the transmitter is not busy and the receiver has successfully received the first symbol. Here are the results of the simulation:

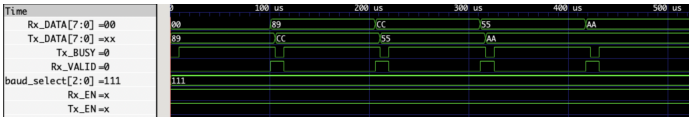

Figure 12: Testbench Part D - Data transmitted and received using the UART

6 Conclusions

The purpose of the current project was the implementation of a Universal Asynchronous Transmitter Receiver (UART). The progress of the implementation of the UART was divided in four different parts. First, we had to begin by building the baud rate controller that would then be used by both the transmitter and the receiver in order to agree to a certain baud rate and sent/receive the data. After that we had to start building the transmitter that would be responsible for sending the data that we wanted as well as the receiver, who would have to sample the data the transmitter would send and store them. Finally we had to join this two parts together and implement the UART. In terms of testing, we used different testbenches for each part and observed our circuits behaviour through its waveforms. Not many problems occurred and the ones occurred got fixed after debugging our circuits using the testbneches.