PROJECT2 REPORT

1. **TITLE**

Universal Asynchronous Receiver Transmitter — UART

GAROUFALI MARIA NEFELI, AEM: 3129, 12/1/22

1. **SUMMARY**

The 2nd lab project of the course is presented in this report. This project concerns the construction of a communication system. Its implementation took place in stages, which will be detailed. Each stage is described by Implementation, Verification, and Experiment.

1. **INTRODUCTION**

The objective of the 2nd lab project is the implementation of a serial communication system, which will use the UART protocol (Universal Asynchronous Receiver Transmitter). The system consists of a UART Sender and a UART Receiver, which transfer data in one direction, from the Sender to the Receiver, through a serial connection of one signal. The UART that will be implemented will be used for serial transfer at least of a sequence of four different 8-bit symbols, from Sender to Receiver.

The UART is a serial, asynchronous communication protocol, and related circuit that implements it, which allows the transfer of data between two (or more generally) devices, which may have independent and uncorrelated clocks. UART is widely used (RS232), due to its simplicity in implementation, practicality ease of use, and general application.

1. **PART A – Baud controller**

* **IMPLEMENTATION**

In this part, the construction of the Baud Controller Unit took place.

For the correct sample rate so that there are no lost or multiplied data in the UART protocol, Sender and Receiver agree on the speed of communication between them in Baud units (bits/sec). This pre-agreement is not part of the communication protocol and is done at a higher level. Several possible baud rates can be used for communication. Instead of using one specific baud rate we make a decoder and let the user decide among 8 baud rates.

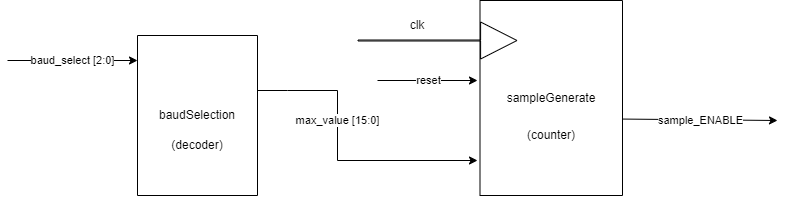
The Baud Controller Unit will be used internally in the Sender and Receiver modules. It aims to provide the appropriate sampling signal, depending on the one selected Baud Rate.

For the implementation of the Baud Controller Unit, two other units were used. Firstly, the *baudSelection* unit which is a decoder. As a decoder, this Unit is a combinational system, and blocking assignments were used for its construction. This Unit takes as input the *baud\_select[2:0]* from the user and gives as output a *max\_value[15:0].* Baud rate is a frequency. From this frequency, we take the period (T) that corresponds to the time a bit needs to be transferred. However, to sample the bit values in the middle of the bits, we compute a 16 times smaller period (T16). The output of the baudSelection Unit represents the number of 10ns clock pulses that are contained in the final period T16. The calculations for all these max\_values are shown in detail below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Baud Rate** | **T= 1/Baud Rate** | **T16= 1/(Baud Rate\*16)** | **Np = T16 / 10 (ns)** | **Relative error** |
| 300 Hz | 3333.33 μs | 208.33 μs | 20833.3 pulses | 1.6e-5 |
| 1200 Hz | 833.333 μs | 52.083 μs | 5208.3 pulses | 6.4e-5 |
| 4800 Hz | 208.333 μs | 13.020 μs | 1302.08 pulses | 6.3e-4 |
| 9600 Hz | 104.166 μs | 6.5104 μs | 651.04 pulses | 6.3e-4 |
| 19200 Hz | 52.0833 μs | 3.2552 μs | 325.52 pulses | 0.0016 |
| 38400 Hz | 26.0416 μs | 1.6276 μs | 162.76 pulses | 0.00467 |
| 57600 Hz | 17.3611 μs | 1.0850 μs | 108.50 pulses | 0.00467 |
| 115200 Hz | 8.68055 μs | 0.5425 μs | 54.25 pulses | 0.00467 |

The max\_value output of this Unit goes as input in the 2nd Unit, the sampleGenerate. This unit is a counter and consequently a sequential system. As sequential system non-blocking assignments were used for its construction. This counter resets its value to 0 whenever a reset signal or the counter has reached the max\_value. Otherwise, it increments its value as a counter must do with the pulse of the clock. Finally, every time the counter is equal to the max\_value a signal sample\_ENABLE is set to 1. This signal is the output of the module and is a pulse corresponding to the period T16. So, for every T16 this pulse becomes 1.

Below is shown the dataflow of this part.



* **SIMULATION**

The functionality of this part was checked with the simulation method. For this reason, a testbench was made. There I made the clock of 10ns and instantiated the module of baud\_controller. After I reset the system, I selected a specific baud rate by giving the appropriate value to baud\_select. Then I checked that for every baud rate the sample\_ENABLE pulse was generated properly. Screenshots of some of the simulations are shown below. The first screenshot shows that the first pulse is generated after 208.335μs which matches the value we computed for this baud rate. As a verification, we examine the second pulse which is generated in 416.685μs. With basic calculations, we understand that this value is double the previous. So, the pulse is generated properly. The second screenshot is the same concept for another different baud rate. After the simulation process, I synthesized and implemented this module. Both the synthesis and the implementation were completed without any problems or serious warnings.

* **EXPERIMENT**

This part couldn’t be checked experimentally in the FPGA.

**PART B - Transmitter**

* **IMPLEMENTATION**

The UART Transmitter should be switchable, communicate with the system to receive the symbol to be transferred, and also produce an availability indication or not, depending on whether it is performing a transfer or waiting for the next data. The Tx\_EN signal is the enable signal of the Transmitter. This signal should remain active, by the system (or testbench), if the Transmitter is active or if it has not yet completed the current transfer. The Baud Rate should be set before enabling the Transmitter. To receive data to be transmitted, we use the Tx\_WR signal, where the latter will become 1 for one cycle, while the data of the symbol to be transmitted will be at Tx\_DATA[7:0]. The Tx\_BUSY signal, directed from the Transmitter to the system will indicate, as long as it remains 1, that the Transmitter is transmitting, so (1) Tx\_EN is not allowed to drop and (2) no new data is allowed to be transferred to it via pair Tx\_WR, Tx\_DATA[7:0].

For the implementation of the Transmitter, a Moore state machine was used. In a Moore stet machine, the outputs of the machine are completely independent of the inputs of the FSM. They depend only on the current state of the machine. This is quite clear in the code of this FSM too. Right below every state, there is the definition of the outputs in that state. After that there is an if statement which when is true, we are going to the next state. If it is false, we remain in the same state. The state machine contains 12 states, each for every bit and one for the IDLE or initial state. At this point, it would be useful an explanation of how the Transmitter works. The data we transfer have a particular form. First, we transfer 8-bit symbols each time. In these 8 bits of the message, we add however 3 more bits. These are the start bit, the parity bit, and the stop bit. The value of the stop bit is always 1 by default. The value of the parity bit depends on the data we transfer. More specifically if the 8-bit message has no number of 1s or an even number of 1s, the parity bit is 0. In case the number of 1s in the message is odd the parity bit becomes 1. The parity bit is used to make sure that all bits of the message are transferred properly. This is achieved by comparing the parity bit the Transmitter sends with the parity bit of the Receiver, but this will be discussed later. The value of the start bit is always 0 and informs the Receiver that a message is coming. That is why in the IDLE state TxD signal is 1. Because we want to distinguish the start bit. If TxD was 0 we couldn’t do that.

Before we examine the state machine, we should see all the combinational logic that is connected to the state machine. First, we have the instantiation of the baud\_controller unit. This unit as it was previously said is used to generate a pulse signal that will help us sample the data. The pulse that is generated from this Unit goes as input in another Unit the *counter.* Except for that input, this Unit takes as inputs the clock (*clk*), the *reset,* and *Tx\_WR.* This Unit is sequential, so it is constructed with non-blocking assignments. Its functionality is to increment (or count) every time Tx\_sample\_ENABLE is 1. The counter is set to 0 every time the reset is 1 or every time Tx\_WR is 1. We reset the counter with the Tx\_WR signal just to make sure that every time we start a new transmission the counter will start counting from 0. The output signal of the counter goes as input in another Unit called *bits\_counter.* This Unit takes also as inputs the clock (*clk*), the Tx\_sample\_ENABLE, and reset. It is also a sequential Unit (counter) and is constructed with non-blocking assignments. Its functionality is to give as output a new counter that increments on every 16 pulses of the previous counter. In other words, this Unit counts the time that needs one bit to be transferred. Remember that Tx\_sample\_ENABLE is 16 times smaller than the period of one bit. At this point, I would like to comment that this extra counter is not necessary. The functionality we want could be achieved with only 1 counter, which would make our implementation less power-consuming, quicker, more efficient, and smaller. However, I decided to leave this extra counter because it would be very helpful in the debugging process. Finally, Transmitter contains another Unit called *parity\_maker.* This Unit takes as input the Tx\_DATA and produces as output the parity bit. It is a combinational Unit, so it is made with blocking assignments. This Unit performs the addition of all the bits of the Tx\_DATA and stores the result in a 1-bit variable. By doing that we make an XOR between the data because we are taking only the sum of the half adder and not the carry. Again, instead of using one half-adder, we could just make an XOR from the beginning which would be more efficient.

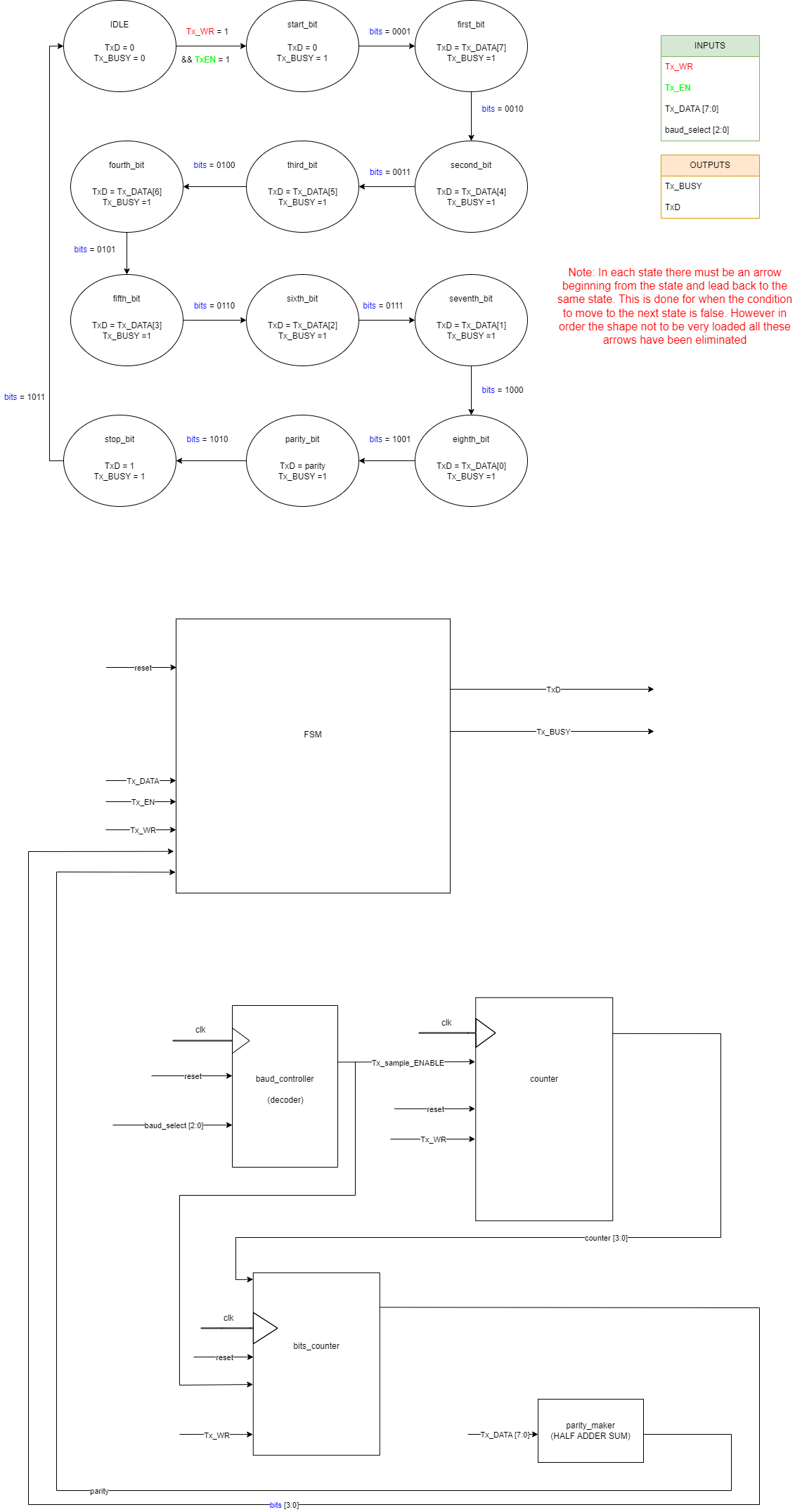
Now let’s examine the state machine in detail. When the system is reset, the Transmitter is in the IDLE state. There the Tx\_BUSY is 0 indicating that the Transmitter is not transmitting any data and TxD is 1. TxD is the wire that connects the Transmitter with the Receiver and transfers bit by bit the data to the Receiver. We are in the **IDLE state**. As the pulse Tx\_WR comes (or Tx\_WR =1) and if the Transmitter is active meaning Tx\_EN is 1 we are moving to the next state (start\_bit). Now we are in the **start\_bit state**. In this state we have started the transmission, so the Tx\_BUSY is 1. The bit we are transmitting is the start\_bit which is by default 0. This means that in this state the output of the Transmitter TxD is 0. From this state to move to the next state we wait until the value of the bits\_counter changes. So, if bits = 4’b0001, we move to the next state (first\_bit). Now we are in the **first\_bit state.** The logic for each of the 8 bits of the message is similar. The output Tx\_BUSY in every state from **first\_bit** to **eighth\_bit** is 1 because we are transmitting data. The only thing that changes is the output TxD. This takes the value of the particular bit we are transmitting at that moment. For example, if we are in the first\_bit state, TxD = Tx\_DATA[7]. If we are in the second\_bit state TxD = Tx\_DATA[6] and so on till the eighth\_bit state where TxD = Tx\_DATA[0]. At this point, I should comment that I have deviated from the description of the project. Instead of transmitting the less significant bit first and the most significant bit last, I did the opposite. I transmit the most significant bit first moving to the less significant bit that is transmitted last. The condition to move to the next state in each of these states (first\_bit – eighth\_bit) is also similar. When the *bits\_counter* changes its value, we are moving to the next state. The table below shows in detail which value of bits (the output of *bits\_counter*) corresponds to which state.

|  |
| --- |
| **If** |
| bits == 1 🡪 |
| bits == 2 🡪 |
| bits == 3 🡪 |
| bits == 4 🡪 |
| bits == 5 🡪 |
| bits == 6 🡪 |
| bits == 7 🡪 |
| bits == 8 🡪 |
| bits == 9 🡪 |
| bits == 10 🡪 |
| bits == 11 🡪 |

|  |  |
| --- | --- |
| **bits** | **CurrentState** |
| 0 | start\_bit |
| 1 | first\_bit |
| 2 | second\_bit |
| 3 | third\_bit |
| 4 | fourth\_bit |
| 5 | fifth\_bit |
| 6 | sixth\_bit |
| 7 | seventh\_bit |
| 8 | eighth\_bit |
| 9 | parity\_bit |
| 10 | stop\_bit |

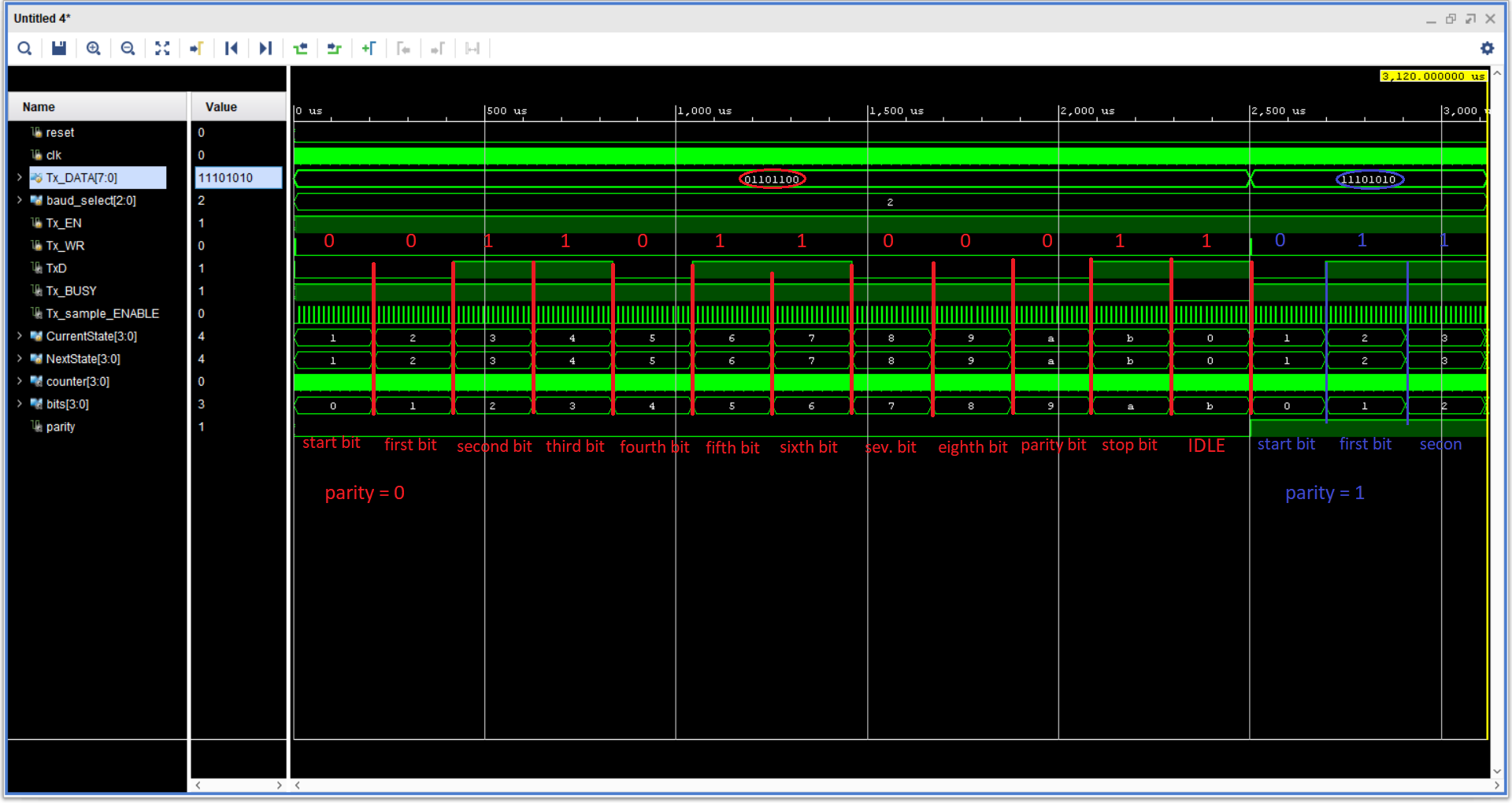
|  |
| --- |
| **NextState** |
| first\_bit |
| second\_bit |
| third\_bit |
| fourth\_bit |
| fifth\_bit |
| sixth\_bit |
| seventh\_bit |
| eighth\_bit |
| parity\_bit |
| stop\_bit |
| IDLE |

Now let’s say that we are in the **parity\_bit state.** The output Tx\_BUSY is once again 1 because the transmission continues. The TxD output takes its value from the parity\_maker Unit that produces the parity. We explained this Unit in detail above. We move to the next state once the bit = 10 as it is shown in the table above. Now we are in the **stop\_bit state.** In this state, both Tx\_BUSY and TxD are 1. I think the reason for Tx\_BUSY is now obvious. TxD is set to 1 because the stop bit that we are transmitting is always by default 1. From this state, if bits\_counter increments once again we move to the IDLE state. For the sake of correctness and latch avoidance, I described a default case too. There TxD is 1, Tx\_BUSY is 0 and the next state is the IDLE state. Below is shown the data flow of this part. There we see the combinational logic, the FSM as well as their connection.

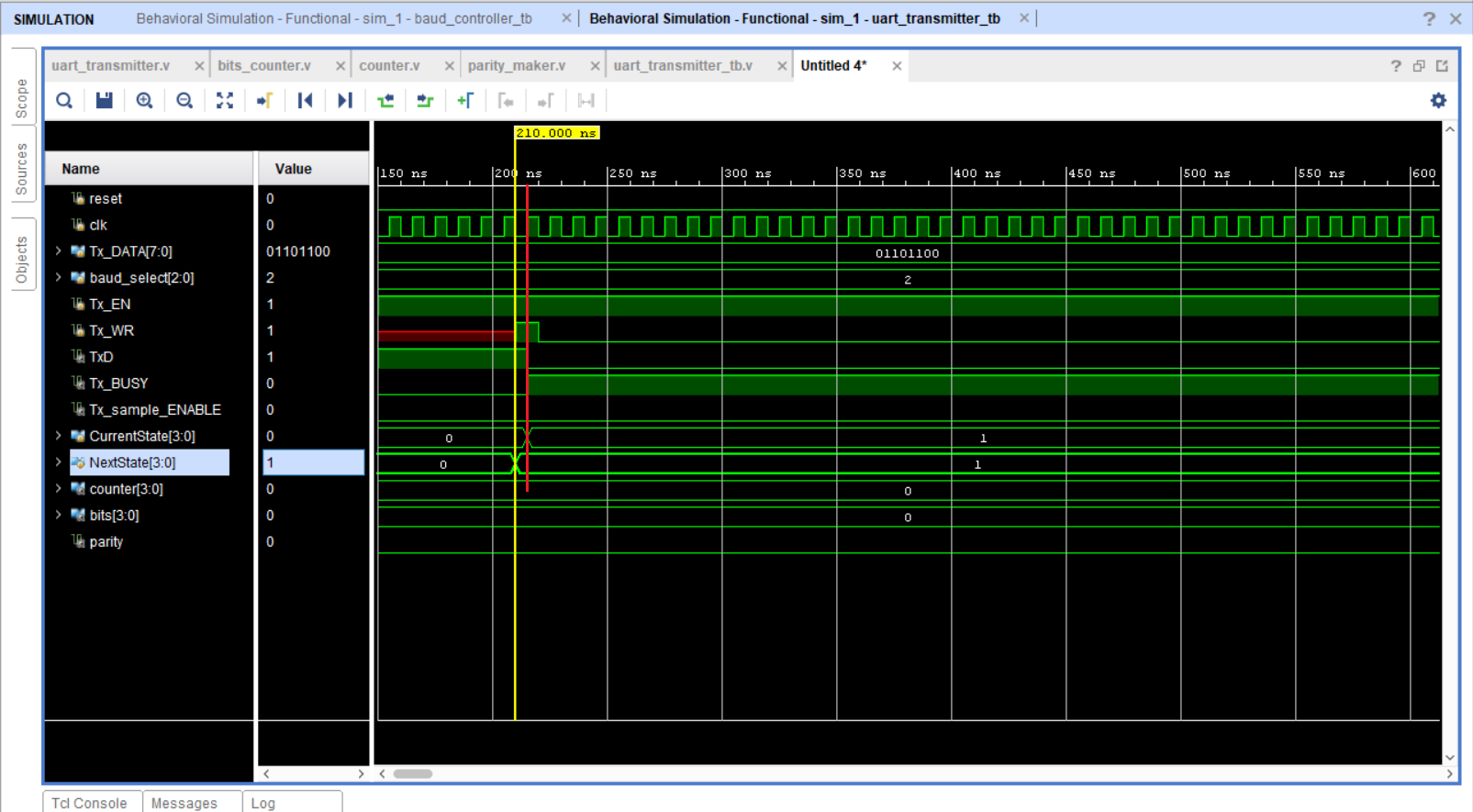


* **SIMULATION**

The functionality of this part was checked with the simulation method. For this reason, a testbench was made. There I made the clock of 10ns and instantiated the module of uart\_transmitter. After I reset the system, I selected a specific baud rate by giving the appropriate value to baud\_select. Then I set a value to Tx\_DATA to be transferred and the signal Tx\_EN to 1 to activate the Transmitter. After that, I created a pulse by setting Tx\_WR to 1 for a clock cycle. Then I created a delay to transmit the data and then I sent a second message with a similar process. For the simulation, I added all the extra signals that were necessary for debugging. The debugging process pointed out several errors in my implementation. For example, I had not predicted to reset the counter when I wanted to transmit a new message. This had as result the counter to continuous increment, the states to change wrong, and the message to be sent wrong. So, I decided to reset the counter every time a new message comes for transmission. Now let’s examine in detail the waveforms from the testbench. Here is the screenshot from the simulation.



The screenshot is edited to make clearer the basic points we will focus the analysis on. First, we see that the transmission starts at the time the Tx\_WR pulse comes. Since this pulse comes too early it is not clear that the initial state of both NextState and CurrentState was 0. We will see that however in the next screenshot. After the Tx\_WR we moved from state 0 to state 1 which is the state “start\_bit” as it is described in the code, in the table above and in the screenshot. We see whenever the bits increment the states change too. Above the TxD it is noted its value in every state. We notice that the first message is properly transferred as the TxD has the right values. First, it takes the value of the start bit (0), then the values of the bits of the message starting from the most significant bit to the less significant bit as I mentioned earlier. Finally, it takes the value of the parity bit which is 0 because there is an even number of 1s in the message and the value of the stop bit which is always 1. After the stop bit, it returns to the IDLE state and TxD remains at 1 as it should. After some time in the IDLE state, a pulse Tx\_WR comes, and we start transmitting the next message. The logic is the same and we can confirm that the Transmitter works properly. Here we see the parity is equal to 1, and that is because the next message consists of an odd number of 1s. In this next screenshot, we see the CurrentState and the NextState being in the state 0 before Tx\_WR comes. We can also see that the CurrentState delays changing its value one clk cycle, compare to NextState which changes immediately when Tx\_WR comes. This is right because the output is assigned with a blocking assignment in NextState while in CurrentState the output is assigned with non-blocking.



After the simulation, I made synthesis and implementation. Synthesis pointed out that there were latches in the design. I searched about the latched and when are they made. I understood there were 3 cases for these to happen. 1) There was a missing signal in the sensitivity list of the FSM. This was a common mistake for me, and sometimes except for making a latch it resulted in the transmitter not working. 2) There was an assignment from a signal to the same signal. 3) There were missing cases. So, I corrected the latches, and everything was fine.

* **EXAMPLE**

I did not check this part experimentally in the FPGA alone.

**PART C – Receiver**

* **IMPLEMENTATION**

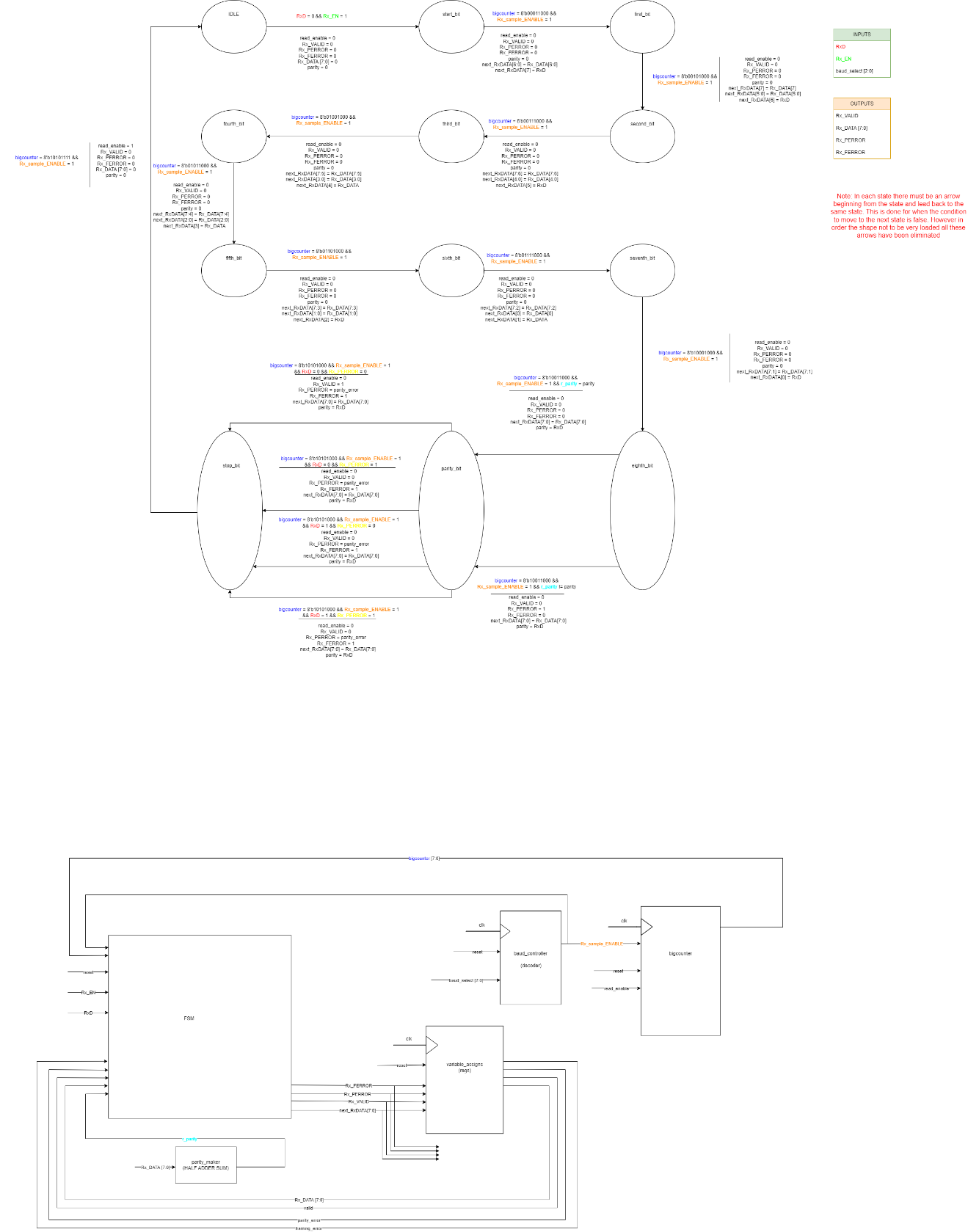
The Receiver should be active in the same way as the Transmitter and additionally indicate to the system that there is a symbol available for reading. The Rx\_EN signal is the activation signal of the Receiver, with similar behavior to the Transmitter. Baud controller Unit works in the same way as before. On completion of receiving by the Receiver, if an error in parity or serial framing is detected of the data, the corresponding error signals should be set to 1, Rx\_PERROR, and Rx\_FERROR. A parity error can occur if the parity the Receiver calculates is different from what it receives. A framing error happens if the value of the stop bit is 0 instead of 1. Alternatively, if there is no error, the symbol received must appear in Rx\_DATA[7:0], and for at least 1 cycle to raise the Rx\_VALID flag, indicating that the data is valid and can be read by the system. The Receiver also has two states, inactive, where it is waiting for data, and the receiving state, where serial reception of bits has started. The Receiver enters the receiving state by detecting the Start bit. It is important that the Receiver correctly samples the serial data, so, for maximum reliability, the Receiver should be sampled in the intended middle of the next bit. The specific prediction, and the appropriate alignment of sampling, are done concerning the Start bit, counting the appropriate number of active cycles, according to the Baud Rate. At the Receiver, active cycles are signaled from the relevant Rx\_sample\_ENABLE signal. In detail, as we want to sample in the middle of each bit, at the time we detect the start bit we wait for 24 Rx\_sample\_ENABLE. Remember that if sample\_ENABLE is for 16 times 1, a period of one bit has passed. So, by counting 24 Rx\_sample\_ENABLE we are moving from the start of the start\_bit to the middle of the first\_bit. Since we have done that, we continue counting by 16 Rx\_sample\_ENABLE to move every time in the middle of the next bits.

For the implementation of the Receiver, a Mealy state machine was used. A Mealy state machine uses a little different logic from the Moore. In this machine, the outputs depend on the input of the FSM. So, every state does not have the outputs defined right below the state. On the contrary, every state defines the outputs for the next state inside the if statement that leads to the next state. If the statement is false, there is an else statement that secures us from creating a latch and describes the outputs for the state we are in. The state machine contains 12 states, each for every bit and one for the IDLE or initial state. At this point, it would be useful an explanation of how the Receiver works. The Receiver connects with the Transmitter through a wire. The TxD of the Transmitter connects with the RxD of the Receiver. So RxD is the input of the Receiver. There will appear the data to be received. Each time we are receiving an 8-bit symbol. However, we receive 3 more bits that are useful for the communication between Transmitter and Receiver. These are the start bit, the parity bit, and the stop bit. The use of these bits was explained before in detail.

Now before examining the FSM let’s see all the combinational logic that is connected to the state machine. First, there is the baud\_controller Unit. Its functionality and implementation have been reviewed earlier. This Unit produces as output Rx\_sample\_ENABLE which goes as input in the bigcounter Unit. This Unit is a counter and consequently a sequential system. As a sequential system, it is constructed with non-blocking assignments. This Unit takes also as inputs the clock, the reset, and another signal, read\_enable which is a signal indicating when the Receiver can read data from RxD. In other words, read enable is 1 in the IDLE state, and 0 in each other. If this signal is 0 AND Rx\_sample\_ENABLE is 1 then the bigcounter can increase its value. We do this because we want the counter to count only when we are receiving data, and not in the IDLE state when we are waiting for the data. This counter resets its value when reset is 1 or when it has reached the maximum value and Rx\_sample\_ENABLE is 1. We added this extra condition for reset because we want every time, we start receiving a new message to count from 0 and not from a random value. This counter counts from 0 to 175. In other words, it counts 16 times for all the 11 bits we are receiving. Another Unit inside the Receiver is the parity\_maker Unit which is the same Unit the Transmitter contains. It is a combinational system and consequently, it is constructed by blocking assignments. Its functionality has been explained in detail before. Finally, the Receiver contains one more sequential Unit called variable\_assigns. This Unit is several Flip Flops needed to keep the previous values of some variables without creating a latch. As a sequential system, it is constructed by non-blocking assignments.

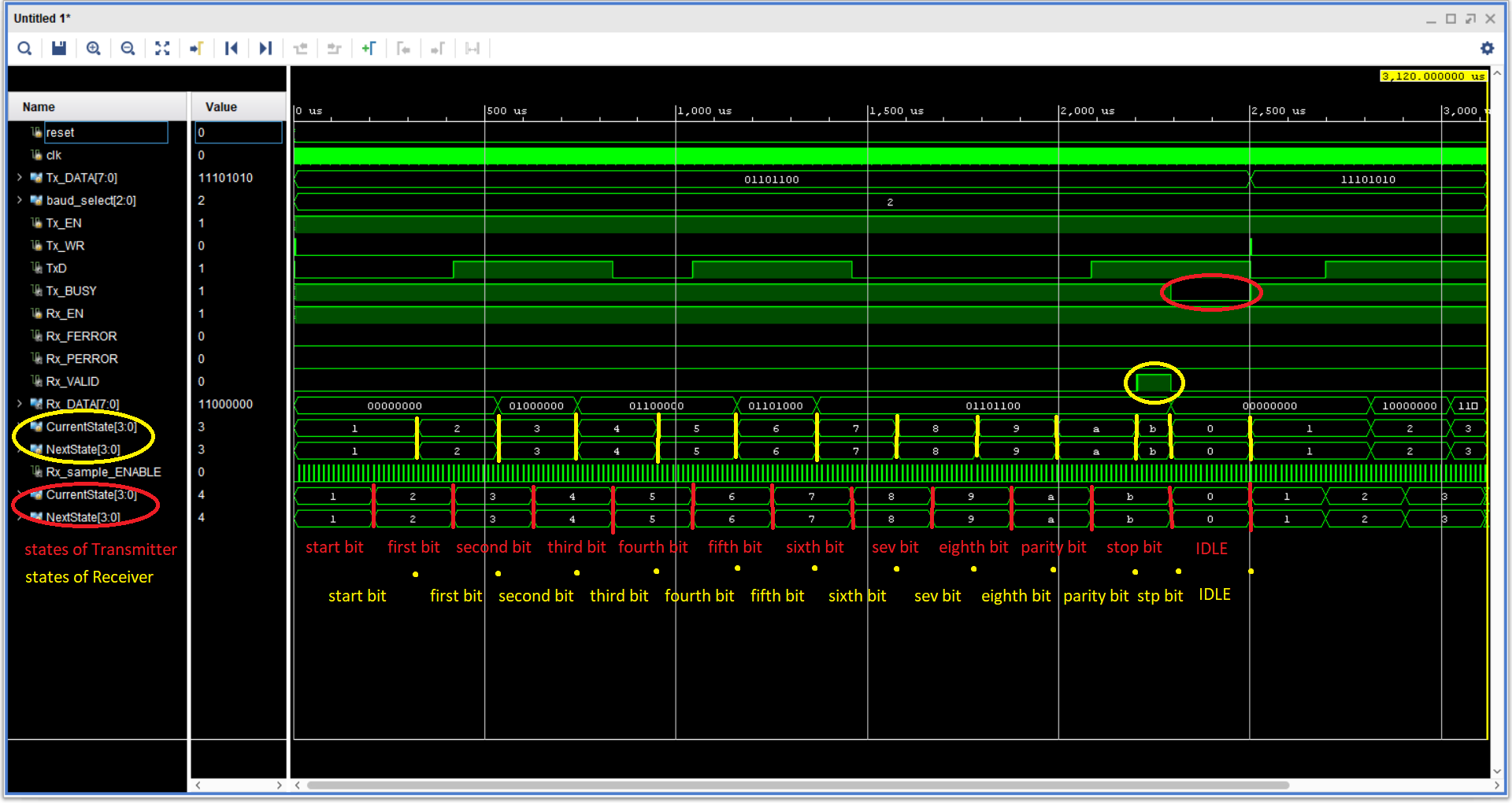
Now let’s examine in detail the FSM. When the system is reset, the Receiver is in the IDLE state. The condition to move to the next state is RxD to be 0 and Rx\_EN to be 1. In other words, to move to the next state from the IDLE state and start receiving data, we must detect the start bit from the Transmitter and the Receiver must be active. As we implement a Mealy state machine, we don’t have standard outputs for the state as we did earlier in the Transmitter. On the contrary, as the outputs depend on the input if the condition is true, we set the outputs of the FSM for the next state and we move to the next state. If the condition is false, then we set the values of the outputs to the state we are and we remain in that state. In the IDLE state read\_enable is 1, and all the other outputs (Rx\_VALID, Rx\_PERROR, Rx\_FERROR, parity, Rx\_DATA) are all 0. So now we are in the **start\_bit state.** In this state, the Receiver has started receiving data. In this state read\_enable is 0 resetting the counter, Rx\_DATA are also 0 since we haven’t received any bit of the message and all the other outputs are 0 too. The condition to move to the next state is the counter to have the value 24. This is done to move in the middle of the first bit of the message. If this condition is true, then we set the outputs with the next state’s values and we move to the next state. Else we stay in the same state, and we give the values of this state to the outputs. At this point, it is very important to comment on how the states are changing. The states of this state machine are not starting from the start of a bit and ending at the end of that as it happens in the Transmitter. Every state from the first\_bit to the parity\_bit starts in the middle of that bit and ends in the middle of the next. I did that so I could have a constant output for each state. The state of the start\_bit as well as the state of the stop\_bit is a little different from the others. The **start\_bit state** lasts one and a half period of the bit, while the **stop\_bit state** lasts half a period of the bit. All this explanation will be much clearer with the waveforms in the simulation process. Continuing the analysis of the FSM, now we are in the **first\_bit state.** In this state, the Receiver has started receiving the data of the message. So read\_enable remains 0 as well as all the other outputs except the Rx\_DATA output. This time Rx\_DATA[7] takes the value of the first bit. (I commented earlier that I have deviated from the project’s description, and I am transmitting and receiving the data in the opposite order, from MSB to LSB). The condition to move to the next state is the counter’s value to be equal to +16 from before. If this condition is true, then we set the outputs with the next state’s values and we move to the next state. Else we stay in the same state, and we give the values of this state to the outputs. In this way, we will be in the middle of the next bit. Now we are in the **second\_bit state**. As before read\_enable is 0 and all the other outputs except the Rx\_DATA are 0. This time Rx\_DATA[6] takes the value of the second\_bit. The next states have exactly the same logic, so it is not necessary to explain them. So, let’s move to the eight\_bit state. At this state, if the condition to move to the next state is true, we have a different output setting for the parity\_bit state. More specifically, in the parity\_bit state, we are sampling the parity bit and checking if it is equal to the parity bit the Receiver calculated for the data it received. From that comparison, we raise or not the RxPERROR. As for the parity\_bit state, here if the condition is true, we sample the stop\_bit and raise or not the Rx\_FERROR. Having the values of Rx\_FERROR and Rx\_PERROR we must determine the value of Rx\_VALID too. We are doing all this stuff inside the condition that moves us to the next state.

Below is shown the dataflow of this part. We see the FSM, the combinational logic, and their connection.



* **SIMULATION**

The functionality of this part was checked with the simulation method. For this reason, a testbench was made. I thought to use the transmitter’s testbench and make the connection with the receiver in the instantiation instead of sending manually the data bit by bit at the right time. In this way, I could check if the Receiver receives the data from the Transmitter. So, the testbench is actually quite similar to the Transmitter’s that was explained earlier. After the testbench was ready I added all the necessary signals to see if the Receiver was working. The simulation pointed out several problems with the implementation. I struggled many times while making the counter. I started by making a counter of 4 bits. Then, due to some problems of this implementation, I decided to make two counters. One to count till 8 to go me to the middle of the bit and another that counts to 16, to go me from the middle of each bit to the middle of the next bit. This solution worked at the beginning but as soon as I simulated to receive 2 messages there appeared errors. Finally, I decided to make a big counter of 16\*11. This counter would need to be reset one time at the beginning of each message receiving. For this reason, I created a signal read\_enable which as I explained before indicates when the Receiver receives data. Now let’s examine in detail the waveforms of the testbench.



I have added the current and the next state of the Transmitter to notice the difference that was described with the current and next states of the Receiver. The states of the Transmitter are shown in red while the states of the Receiver are shown in yellow. We see hear that the start\_bit state of the Receiver lasts 3/2-bit period and that the stop\_bit state lasts half the bit period. We also see that the message is gradually been making and by the stop bit is ready to be read if it is valid. Since our message is valid the Rx\_VALID is 1 for the rest of the stop\_bit state. Another thing we can observe is that the Tx\_BUSY is low at the IDLE state. Now that we have the waveforms let’s explain once again the change of states for the Receiver. The start\_bit state lasts for 24 Rx\_sample\_ENABLES or 3/2 bit period. Then we are in the middle of the first bit. At this point, we sample the value of the first bit, and we move to the next state. From now and then we are moving every 16 counter’s pulses or a bit period. Each time we are in the middle of the bit we want to sample in the next state. So, we sample it and then move to the next state. As we saw in the Transmitter waveforms, the current state delays changing its value one clock cycle and that is because it is assigned with non-blocking assignments. After the simulation, I made synthesis and implementation. Synthesis pointed out latches in the design. To solve that problem as I wanted to keep some variables’ values from one state to another, I created a Unit that was actually Flip Flops for these variables’ assignments.

* **EXAMPLE**

This part could not be checked experimentally in the FPGA.

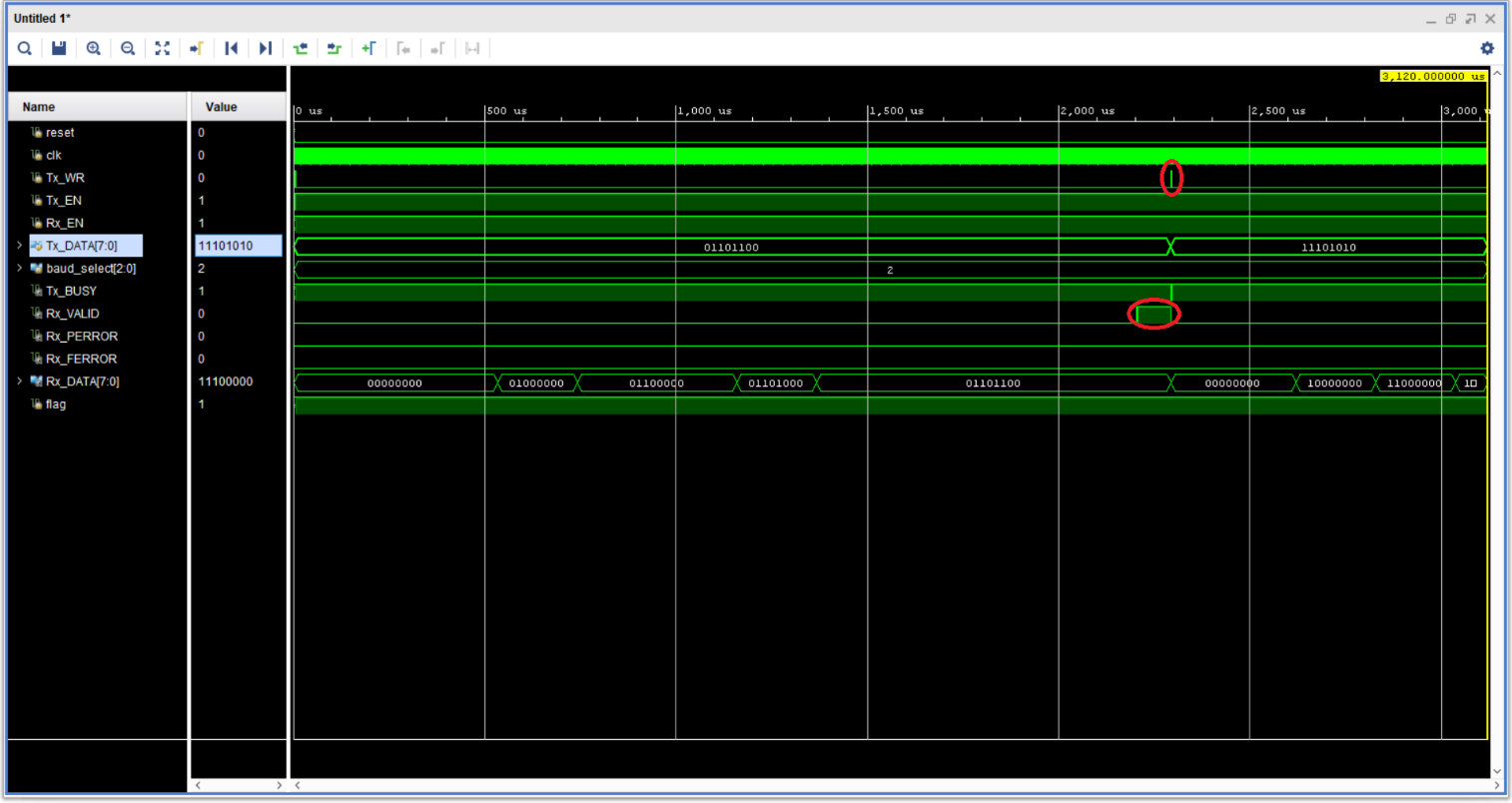
**PART D – UART Top**

* **IMPLEMENTATION**

For this part, I should connect the Transmitter and the Receiver to create the UART connection. Before connecting them, I should make a synchronizer. That is because the Transmitter and the Receiver might work with different clocks. So, there is a chance to be created metastability from the Transmitter to the Receiver. This could happen if for example the data from the Transmitter was given to the Receiver close to the setup time or to the hold time. The synchronizer Unit is a sequential system consisting of 2 Flip Flops. This is what we need to avoid metastability. After implementing the synchronizer, I connected the Transmitter’s TxD with the synchronizer and then the Receiver’s RxD with TxD.

* **SIMULATION**

For the simulation, I had to make a testbench based on the signals of the Transmitter and the Receiver and not any specific delay or counter. So, I made an initial block to initiate the values. Then as I wanted to transmit 2 messages, I created a flag and an always block. Inside the always block I was sending a Tx\_WR pulse whenever the Transmitter was not busy (Tx\_BUSY = 0). However, this created the problem of what message would be sent. For this reason, I used the flag to know which data I should assign at each time.

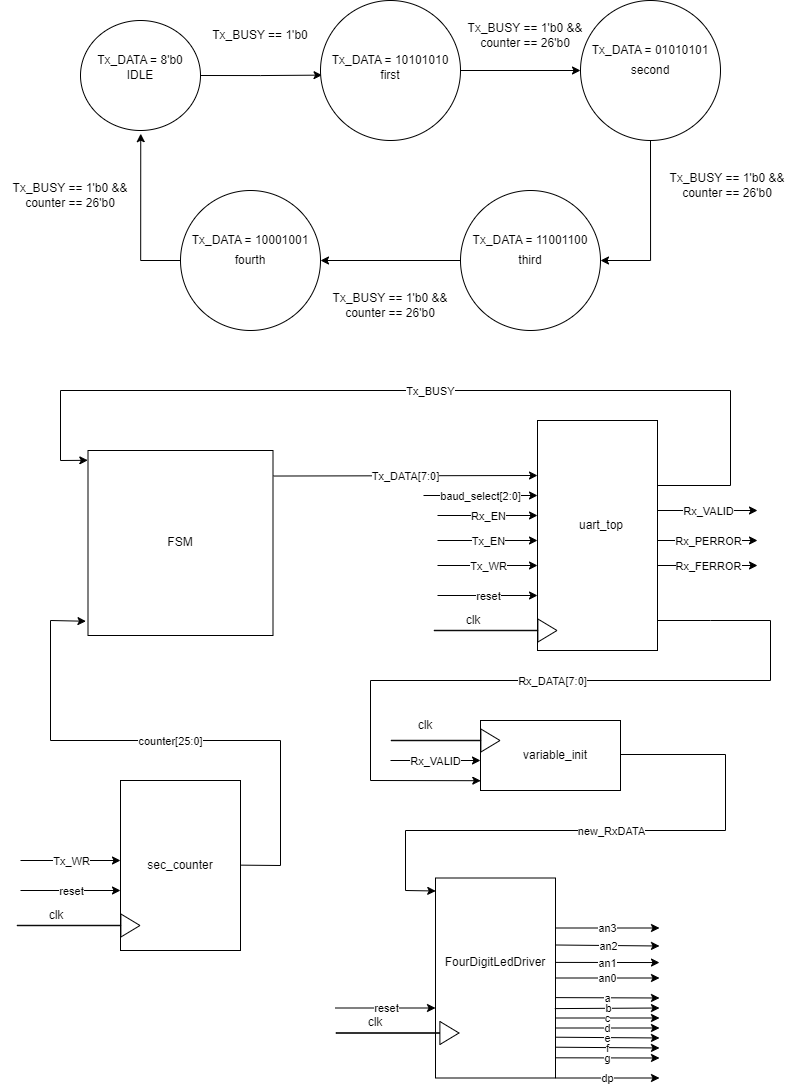


We see in the waveforms the Rx\_DATA that is gradually constructed, the Rx\_VALID that is 1 at the end of the first message, as well as the Tx\_WR pulse that is sent to begin transmitting the next data. We can also see the Tx\_BUSY that is 1 during the transmission and is 0 for just one pulse at the same time the Tx\_WR is sent.

**OPTIONAL PART**

* **IMPLEMENTATION**

To implement the optional part I needed to create a new project and copy all the files from the previous and the current project to that. Then I made some changes to the previous project to adapt it to my needs. More specifically, I turned off 2 of the 4 digits because at this time I want to transmit 8-bit messages and not 16-bit messages. The 8-bit messages correspond to 2 digits. I also created a top module where I instantiated the uart and the FourDigitLeddriver. This module contained 2 other Units. The first is a sequential system which is a Flip Flop. This was created to hold the message when it is ready and valid, and change whenever another valid message came. The output of this Unit goes as input to FourDigitLedDRIVER Unit to display the message. As a sequential Unit, it is constructed with non-blocking assignments. The second Unit is a counter. This counter has 26 bits and is made to count something more of a second. By doing this the message will have enough time to be displayed. This Unit is also sequential, so it is constructed with non-blocking assignments. In the top module, I also made a Moore FSM. This Fsm has 5 states. One for the IDLE state and the other four for each message. So in every state, I assign the Tx\_DATA. In the IDLE state Tx\_DATA is 0. In the first state, Tx\_DATA takes the value of the message I want to send first namely 10101010(AA). In the second state, Tx\_DATA takes the value of the second message, that is 01010101(55). In the third Tx\_DATA is 11001100(cc). Finally, in the last state Tx\_DATA’s value is 10001001(89). The condition to move from one state to the next is Tx\_BUSY to be 0 and counter to be 26’b0. In other words, we want the transmission of the current message to be completed and a time of approximately 1 second to have passed. This is very important so that we can see the messages in the FPGA. Below is shown the dataflow of this part with the combinational logic, the FSM, and their connection.



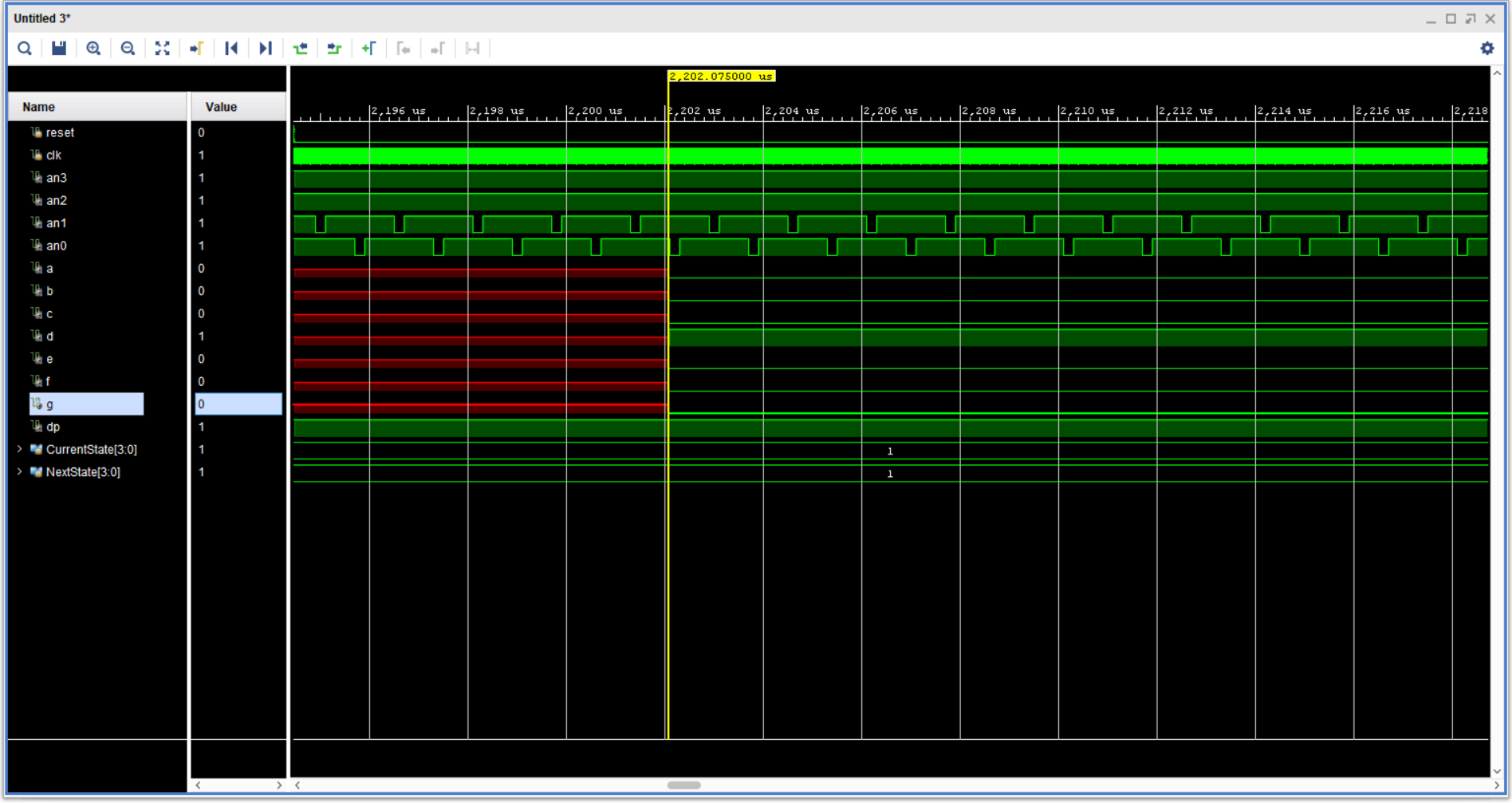
* **SIMULATION**

After finishing the implementation, I made a testbench for the simulation. This was easy to create because all the process with the message assignment is done in the top module. So, the only thing I had to do was to initiate the variables, create the clock and instantiate the module. Below we see the waveforms for this simulation. Note that to create the waveforms and see more than the first message I had to remove the counter. That is because the counter runs for a lot of time and the simulation practically could not be created.

Εικόνα που περιέχει κείμενο, πίνακας αποτελεσμάτων

Περιγραφή που δημιουργήθηκε αυτόματα

In the figure above we can see that initially the digits in the first state, have not received any value. As we said before this is done because we want only the final value of Rx\_DATA and not the intermediates. We can also see that every message appears at the end of each state. This is because at this time the valid is 1 for the first time. After that, the value of the message remains until the valid becomes again 1 and the message replaces with the new. In the zoomed screenshot below we see how the an3 and an2 which correspond to the first 2 digits of the FPGA are turned off and how the an1 and an0 which correspond to the 2 other digits turn on periodically.



* **EXPERIMENT**

This part was tested in the FPGA experimentally. On the first try, there wasn’t the wanted result in the FPGA. There appeared 00 and not any of the characters of the message I want. This is the message that I had given to the IDLE state. This led me to think that the system maybe needed more time to display the messages. For this reason, I created the counter and put it as a condition to move to the next state to have passed 1 second in the current. After that change, the wanted result appeared on the FPGA. At this point, I want to comment that the message of the IDLE state does not appear at the FPGA. That is because we stay in the IDLE state only for a pulse. Since we return to the IDLE state and Tx\_BUSY becomes 0, at the next cycle we make the Tx\_WR 1 and we move to the next state. The message is rotating because all the states are connected in a circle. Also, to run the project in the FPGA I needed to create an xdc file with the constraints. This file I created was the same as the previous’ project.