ECE333 - Lab 2
Universal Asynchronous Receiver Transmitter — UART

Nikas Ioannis-Iason

10/11/2024

Contents

1 Summary **1**

2 Introduction **2**

3 Part A - Baud controller **2**
3.1 Implementation . . . 2
3.2 Simulation . . . 3

4 Part B - Transmitter **3**
4.1 Implementation . . . 3
4.1.1 Sample counter . . . 3
4.1.2 Finite state machine . . . 3
4.1.3 Top-level module . . . 4
4.2 Simulation . . . 5

5 Part C - Receiver **6**
5.1 Implementation . . . 6
5.1.1 Finite state machine . . . 6
5.1.2 Sampler . . . 8
5.1.3 Data handler . . . 9
5.1.4 Top-level module . . . 9
5.2 Simulation . . . 9
5.2.1 Parity error . . . 10
5.2.2 Framing error . . . 10
5.2.3 Operating normally . . . 10

6 Part D - Controller **11**
6.1 Implementation . . . 11
6.1.1 Button and memories . . . 11
6.1.2 Top-level module . . . 11
6.2 Simulation . . . 12

7 Optional feature: Displaying messages on FPGA seven-segment display **13**
7.1 Implementation . . . 13
7.2 Simulation . . . 13
7.3 Live testing . . . 13

1 Summary

This report outlines the design and the implementation of a UART communication protocol. It is divided into 4 different parts: a baud controller, a transmitter, a receiver and a controller. The baud controller is responsible for setting the communication rate while the transmitter and receiver

handle the transmission and the reception of data. The receiver also implements error detection for two different kinds of errors. The controller connects the modules mentioned above and transmits a specific message as a test.

2 Introduction

The goal of this project was to implement a UART (Universal Asynchronous Receiver Transmitter) protocol, a straightforward serial communication protocol. The project is divided into four main components:

- A. Baud controller - The baud controller sets the communication speed (baud rate) between the transmitter and the receiver. The baud rate is configured in advance by the user and is not exchanged or negotiated between the devices.
- B. Transmitter - The transmitter is responsible for sending bits to the receiver in the correct format, including the start bit, data bits, parity bit, and stop bit.
- C. Receiver - The receiver waits for the transmitter to initiate communication, then captures and stores the incoming bits in its memory. It also handles error detection to ensure data integrity during transmission.
- D. UART controller - This module connects the transmitter and receiver and is used to send a static message to validate the functionality of these modules.

3 Part A - Baud controller

3.1 Implementation

For the baud controller, a single module was used for both the transmitter and the receiver. Since the receiver needs to operate 16 times faster than the transmitter to sample each bit sent by the transmitter multiple times and detect any potential errors, we must properly manage the baud controller's sample signal on the transmitter side by creating a new sample signal that pulses every 16 cycles of the baud controller's signal.

To implement the controller, a counter was used to count a specific number of clock cycles and generate a pulse. There are 8 baud rates implemented, and for each one, we need to calculate the number of cycles required before sending the pulse. The time the counter needs to wait can be calculated using the formula $T_{sc} = \frac{1}{16 \times BaudRate}$. The number of cycles required can be calculated using the formula $C = \frac{T_{sc}}{T}$, where T is the period of the clock. In our case the period of the clock is $T = 10^{-8}$.

Baud Rate	Time to Wait (T_{sc})	Cycles to Wait (C)	Rounded Cycles	Relative Error
300	$0.208\overline{33}$ms	$20833.\overline{33}$	20833	0.0016%
1200	$52.08\overline{33}\mu$s	$5208.\overline{33}$	5208	0.0064%
4800	$13.020\overline{833}\mu$s	$1302.08\overline{33}$	1302	$\approx 0.00639\%$
9600	$6.5104\overline{166}\mu$s	$651.04\overline{166}$	651	$\approx 0.0064\%$
19200	$3.255208\overline{33}\mu$s	$325.520\overline{833}$	326	$\approx 0.1472\%$
38400	$1.62760\overline{4166}\mu$s	$162.760\overline{4166}$	163	$\approx 0.1472\%$
57600	$1.08506\overline{944}\mu$s	$108.506\overline{944}$	109	$\approx 0.4544\%$
115200	$0.54253\overline{472}\mu$s	$54.25\overline{3472}$	54	$\approx 0.4672\%$

Table 1: Baud Rates and corresponding calculations

In order to select the baud rate at which the controller operates, we use a 3-bit input signal, `baud_select`, which maps to one of eight predefined baud rates. This input signal sets the maximum value the counter can reach. Each baud rate is associated with a specific `max_counter` value, and by selecting a baud rate, the `baud_select` signal ensures that the counter increments at the correct rate to produce accurate timing for communication.

3.2 Simulation

For simulating and verifying the functionality of the baud controller, a simple testbench module was used. It sets the `baud_select` signal to a specific value. By observing the waveform of the output signal, which indicates when the controller generates a sample pulse, we can verify the correctness of the controller's operation and the timing for each baud rate. Below we can see the waveforms for the baud controller working on the faster baud rate available. The time between the pulses is 540ns which is really close to the calculated wait time for the controller.

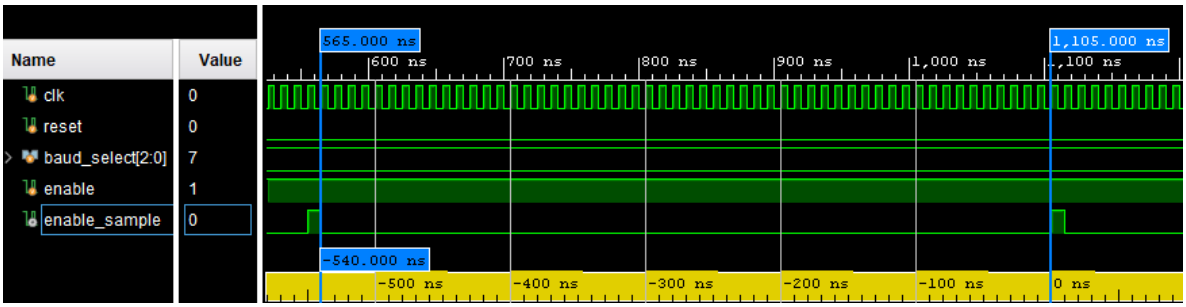


Figure 1: Sample signal of baud controller

4 Part B - Transmitter

4.1 Implementation

The implementation of the transmitter consists of three main components, each responsible for different aspects:

4.1.1 Sample counter

As mentioned earlier, the transmitter needs to operate 16 times slower than the receiver to ensure that the bits are sent correctly. Since the same baud controller is used for both the transmitter and the receiver, we need to introduce an additional counter. This counter has a maximum value of 16, and each time it reaches its maximum value and the next pulse from the baud controller the next bit is transmitted. This way we achieve a pulse 16 times slower than the controller's pulse.

4.1.2 Finite state machine

The FSM of the transmitter is a Moore type FSM and consists of 13 states. The only inputs are the signal to enable the transmitter (`tx_en`) and the signal to start a transmission (`tx_wr`).

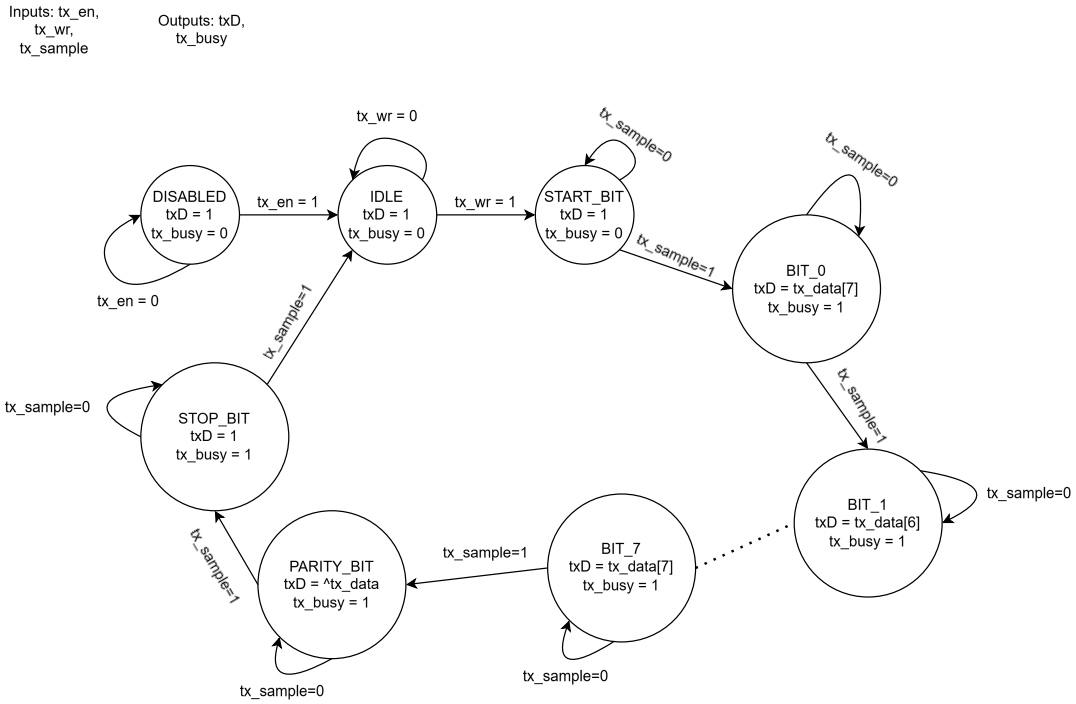
- `DISABLED` - In this state the transmitter is waiting to be enabled using the `tx_en` signal.
- `IDLE` - The transmitter is waiting to receive a pulse of the signal `tx_wr` signal to start the transmission.
- `START_BIT` - The transmitter sends the start bit (0).
- `BIT_0` - `BIT_7` - The transmitter transmits the data bits based on the `tx_data` signal.
- `PARITY_BIT` - The transmitter sends the result of XOR-ing all the data bits. This can be used by the receiver to verify the integrity of the data.
- `STOP_BIT` - The transmitter sends the stop bit (1).

Additionally the FSM produces two output signals:

- `txD` - This is the data signal sent to the receiver, carrying the transmitted data bits, parity bit, and stop bit.

- `tx_busy` - This signal indicates whether the transmitter is currently in operation, and is used by the top-level module to monitor the transmitter's status.

My initial approach was to use one state per data bit, as it offered a straightforward solution and would be easy for others to understand when reviewing the design. Below is a diagram of the FSM displaying all the states and outputs along with the transition from one state to another:



4.1.3 Top-level module

This module of the transmitter integrates the previously mentioned components, along with the baud controller from Part A, allowing the new module to function independently as a complete transmitter. Additionally, it combines the signal from the sample counter and the baud controller to create the signal `tx_sample` which is used by the transmitter. Below is a diagram depicting how the top-level module connects everything together.

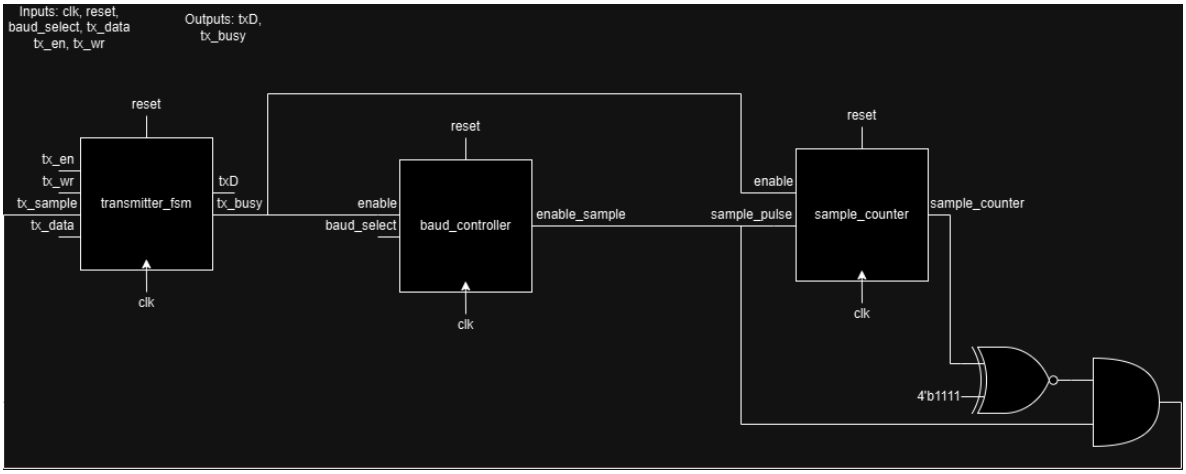


Figure 2: Data flow diagram of the transmitter module

4.2 Simulation

For the simulation a testbench module was used that sends two `tx_wr` pulses and transmits the same message twice. That way we can see if the data is being transmitted correctly and if the FSM correctly changes states. The data transmitted in the following example is `8'haa` and in the image below we can view how the bits are transmitted through the `txD` signal and how the FSM's current and next states.

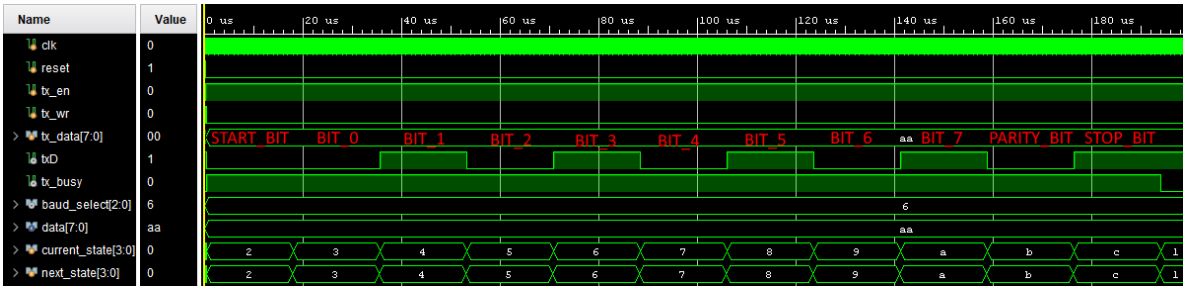


Figure 3: Waveforms showing the operation of the transmitter when transmitting the byte `8'hAA`. The figure illustrates the transitions of the txD signal, which carries the transmitted data, and the associated changes in the FSM states as the data is sent, including the start, data bits, parity bit, and stop bit.

Additionally, we can view how the transmitter changes state when the next pulse of the signal `tx_wr` arrives and it retransmits the message.

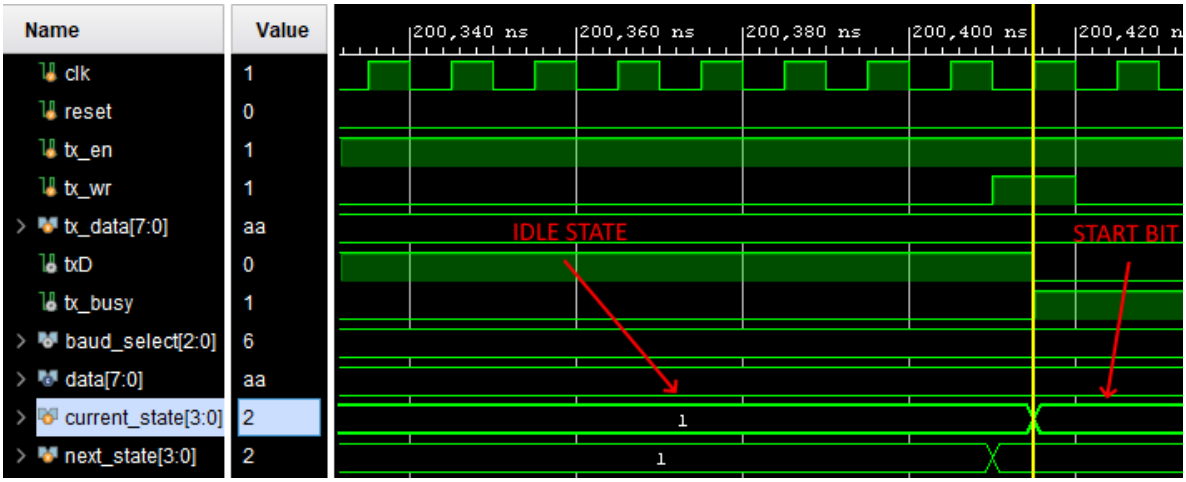


Figure 4: FSM state transitions triggered by the `tx_wr` signal. The figure shows how the FSM moves through its states when a new `tx_wr` pulse arrives, starting a new transmission sequence. This pulse initiates the transition from the IDLE state to the START_BIT state, followed by the data transmission states.

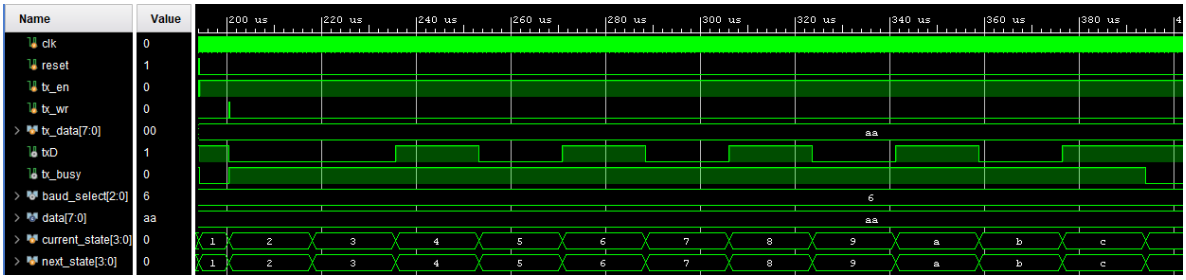


Figure 5: Waveform showing the retransmission of the byte 8'hAA. After completing the first transmission, a new transmission sequence is initiated.

The same testbench was used for 4 different values (`8'haa`, `8'h55`, `8'hcc`, `8'h89`) allowing us to draw reliable conclusions about the functionality of the transmitter. The results were promising so I moved on to the receiver.

5 Part C - Receiver

5.1 Implementation

For the receiver the implementation is broken into four different parts:

5.1.1 Finite state machine

The FSM of the receiver is also a Moore FSM but is more complex and contains additional states because, in addition to reading the transmitted bits, the receiver must also perform error checking. The inputs of the FSM are the following:

- `rx_en` - Signal given by the user to enable the receiver.
- `rxD` - The bus connected to the transmitter.
- `sample_mid` - Signal used on the parity bit state to check the integrity of the data received.
- `sample_done` - Signal given by the sampler when the sampling is done and the receiver can proceed to the next bit.
- `sample_diff` - Signal given by the sampler that indicates whether the bit received a cycle before is different that the bit that is currently being sent. If this is 1 the FSM will go into an errorred state.

The states of the FSM have some similarities with the transmitter's FSM but also contain extra states for the errors that can occur.

- `DISABLED` - In this state the receiver is waiting to be enabled using the `rx_en` signal.
- `IDLE` - The receiver hasn't yet received any messages and is waiting for a start bit to be sent by the transmitter.
- `START_BIT` - The start bit has now been detected by the FSM.
- `BIT_0` - `BIT_7` - The receiver reads the data bits sent by the transmitter and stores them.
- `PARITY_BIT` - The receiver reads the parity bit sent by the transmitter and uses it to check against the previously received data bits. If the parity bit doesn't match the XOR of all the received bits the FSM's state changes to an error state.
- `STOP_BIT` - The receiver reads the stop bit.
- `COMPLETED` - The receiver has successfully read a message from the transmitter.

- `PERROR` - A parity error has occurred and the FSM will stay in this state until the user disables and re-enables the receiver
- `FERROR` - A frame error has occurred while sampling a received bit. The FSM will stay in this state until the user disables and re-enables the receiver

Additionally because the receiver is more complex there are now 5 outputs:

- `rx_valid` - This signal indicates whether the received bits are valid and can be read by the user.
- `rx_ferror` - This signal indicates whether a framing error has occurred.
- `rx_perror` - This signal indicates whether a parity error has occurred.
- `rx_busy` - This signal indicates whether the receiver is currently in operation, and is used by the top-level module to monitor the receiver's status.
- `rx_busy_data` - This signal indicates whether the receiver is currently in operation and is specifically busy reading data bits.

My initial design for the error signals was a bit different because it used flip-flops to keep the errors on until the next transmission but after some thinking I decided to add new states to the FSM that signify the errors and use them to drive the error signals. I think the new solution is an improvement because it makes the whole process of error detection and handling way easier and also uses 3 less flip-flops. Below is a diagram of the FSM displaying all the states and outputs along with the transition from one state to another:

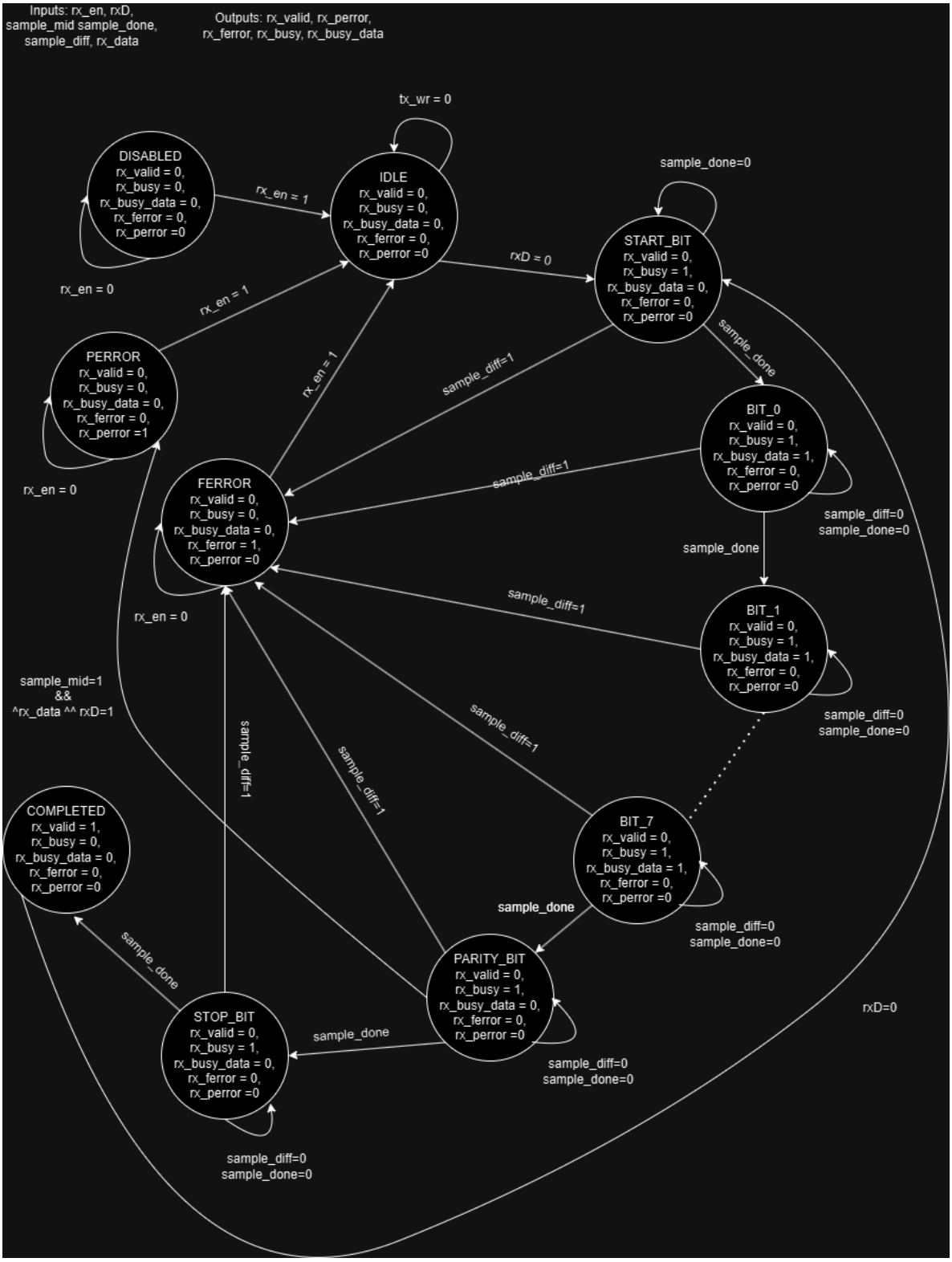


Figure 6: FSM diagram of the receiver

5.1.2 Sampler

The sampler as the name suggests handles the sampling of the data that is read by the receiver and produces outputs that are useful to the FSM in order to have correct timing and also to detect any mistakes. In order to do the sampling it uses an instance of the same sample counter module from

Part B and the sample signal from the baud controller (`sample_pulse`). It samples a single received bit 8 times in order to have a certain degree of certainty about the errors that may have occurred.

It also takes as input the bus (`rxD`) that the transmitter sends the data to and which uses to detect framing errors. It uses 2 flip-flops to store the `current_bit` that is received and the `previous_bit` that is the bit received one cycle before. Additionally using the sample counter it produces 2 output signals that are used by the FSM in order to know when to transition to the next state (`sample_done`) or when the middle of sampling in a state has passed (`sample_mid`).

5.1.3 Data handler

The data handler is a simple module that handles the `rx_data` bus and stores the read bit in the correct place. As input it uses the signal the sampler produces to know when the sampling has reached the middle to read the value of a bit because that's the time where the transmitted bit has a high chance to be stable. Additionally it uses the output of the FSM `rx_busy_data` to know when the receiver is actually in the state of receiving data bits that need to be stored.

5.1.4 Top-level module

The receiver's top module correctly connects all the previously mentioned parts, along with the instance of the baud controller and also produces the `sample_diff` signal by XOR-ing the `previous_bit` and `current_bit` produced by the sampler. Below is a diagram depicting those connections:

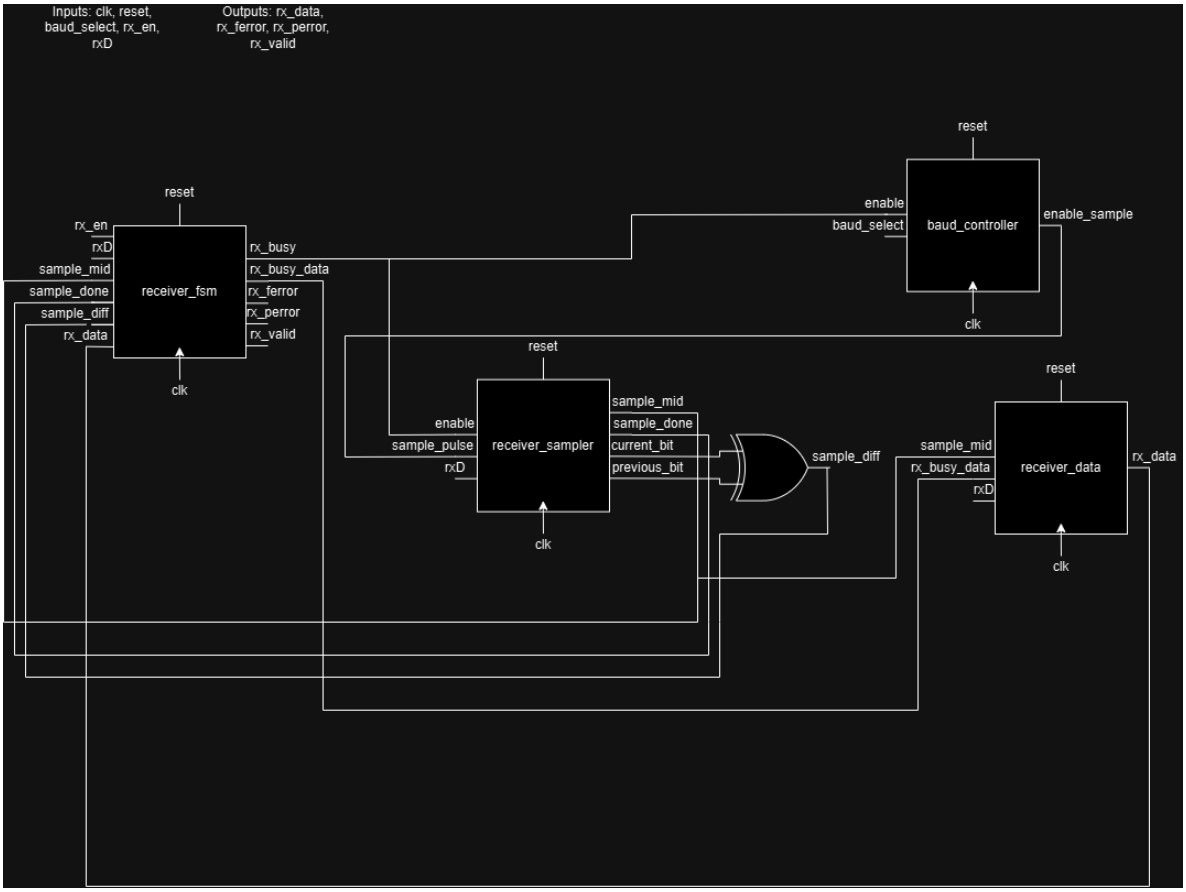


Figure 7: Diagram of the receiver module

5.2 Simulation

The simulation here was done using 3 different testbench modules: a. one for testing the parity error b. one for testing the framing error and c. one to test a successful transmission of a message.

5.2.1 Parity error

This testbench module intentionally sends a flipped parity bit to the receiver in order to test the error states of the FSM and the general behaviour of the module. Below you can see the moment a receiver reads the parity bit in the middle of sampling and compares it to the previously received data. The receiver changes state one clock cycle later and the `rx_valid` signal goes to high.

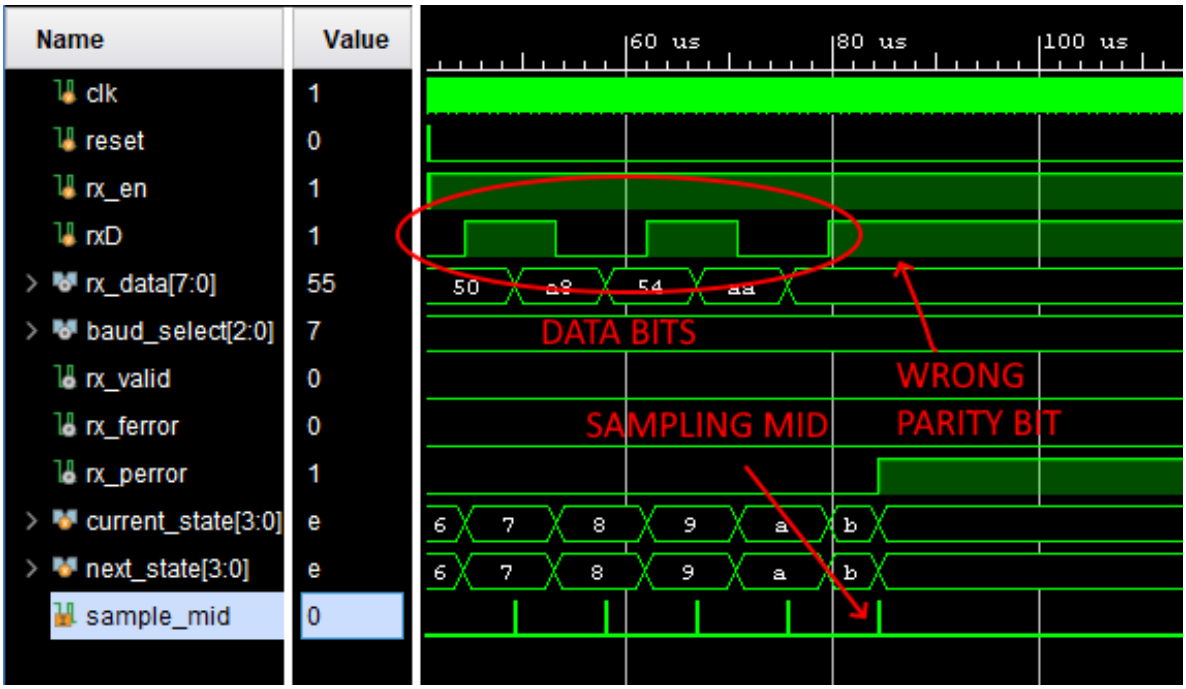


Figure 8: Receiver detecting a parity error

5.2.2 Framing error

This testbench module intentionally uses the wrong `baud_select` for the receiver which should cause it to detect a framing error. Below we can view the waveforms of the receiver when using the wrong baud rate. As we can see there's a possibility to not detect the error write away due to way the are sent on the channel. After some states the sampler finds different bits on the `current_bit` and the `previous_bit` and causes the FSM to go into the `FERROR` state.

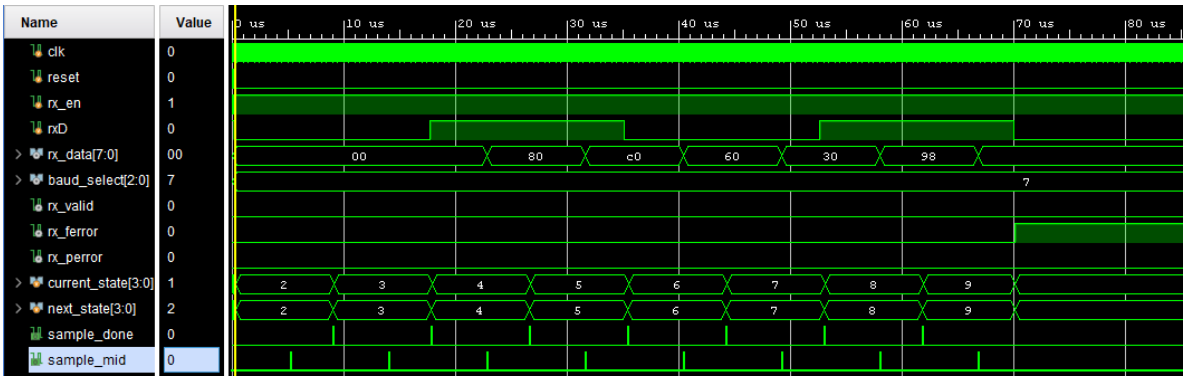


Figure 9: Receiver detecting a framing error

5.2.3 Operating normally

Finally another testbench module is used to check the correct functionality of this module and tries to transmit a simple message and check if it has been successfully received. Below is an image of the

waveforms that show the receiver going through different states and finally makes `rx_valid` high when it ends and the data is ready for reading.

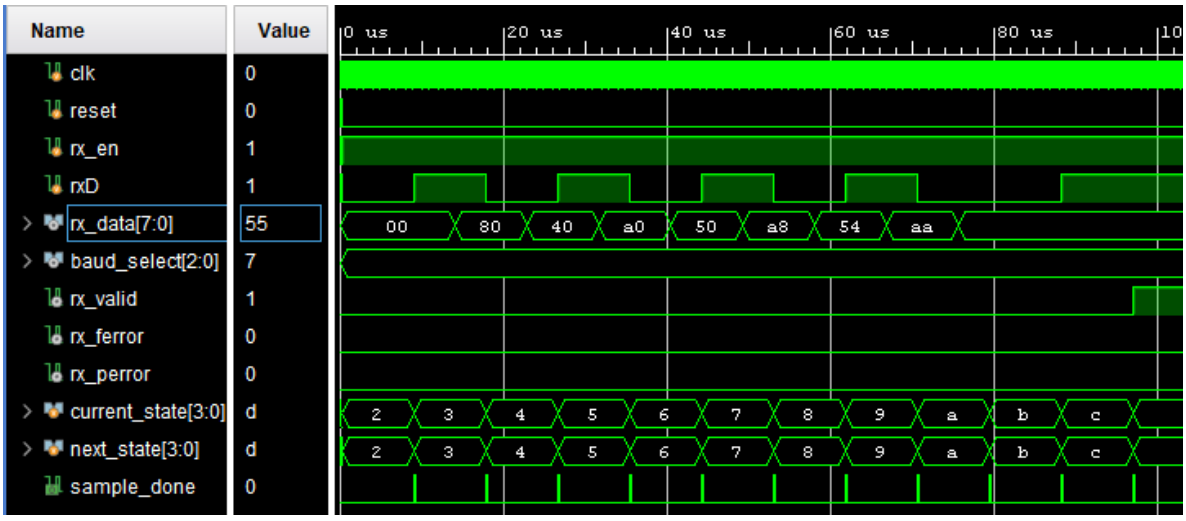


Figure 10: Receiver successfully storing a message

6 Part D - Controller

6.1 Implementation

The controller consists of a transmitter and a receiver memory that are used for reading and writing respectively and a message counter to count the messages that have been sent by the transmitter to the receiver.

6.1.1 Button and memories

The memories consist of 4 8-bit entries. The transmitter memory is reset to the values that are represent specific values to be displayed later in the optional part. The message counter is used for both the transmitter and the receiver and plays the role of the memory address that each of them read or write. After it reaches `2'b11` which is the final message the counter stops incrementing and if a transmission occurs again it re-sends the last message. As an input to the counter a `next` signal is used where the counter is incremented only when this signal goes from 0 to 1.

6.1.2 Top-level module

The top-level module connects each memory and each message counter to its corresponding module (receiver, transmitter) and adds de-bouncers for each input that is a user input. It outputs 3 signals:

- `tx_end` - Indicates whether all the messages have been transmitted.
- `errorred` - Indicates whether an error has happened during the transmission.
- `rx_valid` - Forwarded signal from the receiver module.

Below is a diagram depicting how the modules are connected to each other:

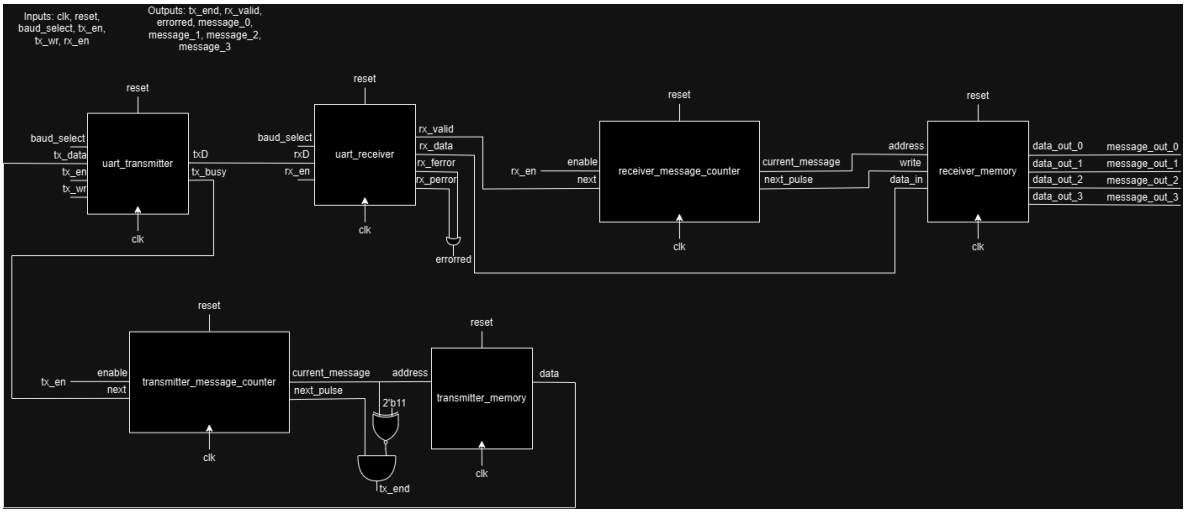


Figure 11: UART controller data flow

6.2 Simulation

To verify and test this part I created a testbench module that is based on the outputs of the module and is an FSM itself. It starts by first sending a `tx_wr` pulse and then waits until either an error is reported by the controller or the `rx_valid` is high where it sends another `tx_wr` pulse to send the next message. When the `tx_end` is high and no errors have been reported the FSM goes into a end state where all the messages have been transmitted. With this we can verify that all the signals have correct timing and that the messages are being transmitted correctly. Below we can view an image showing the state of the testbench's FSM changing and sending a `tx_wr` pulse and we can view all the messages being transmitted and stored into the receiver's memory.

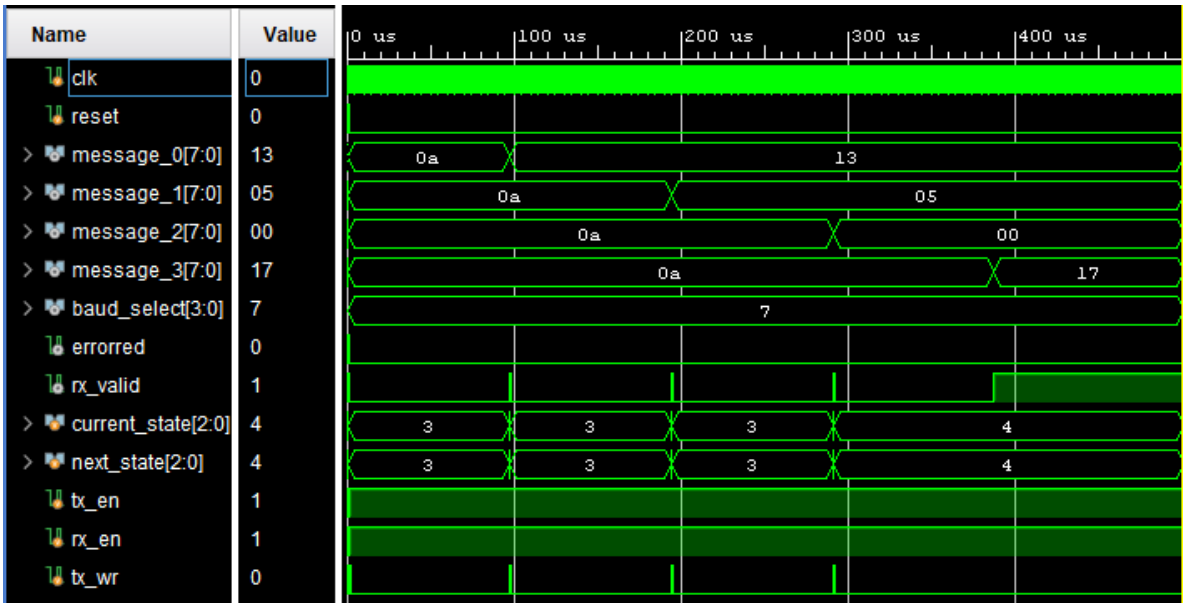


Figure 12: Controller working as expected

After running implementation on the modules some warnings were present. Those were about not using all the ports on some modules. These modules were de-bouncers which also produce an output that is a pulse of the de-bounced input. There were cases where I only needed the de-bounced output and not a pulse of it so that's why in some instances I didn't use all the ports.

7 Optional feature: Displaying messages on FPGA seven-segment display

7.1 Implementation

The implementation of the UART-led connector is a simple module that connects the module from part D with the slightly modified version of FourDigitLEDdriver module from the first project. It uses the output memory of the UART controller as an input memory for the character that should be displayed on the seven segment displays. Below is a diagram depicting this connection of the 2 modules:

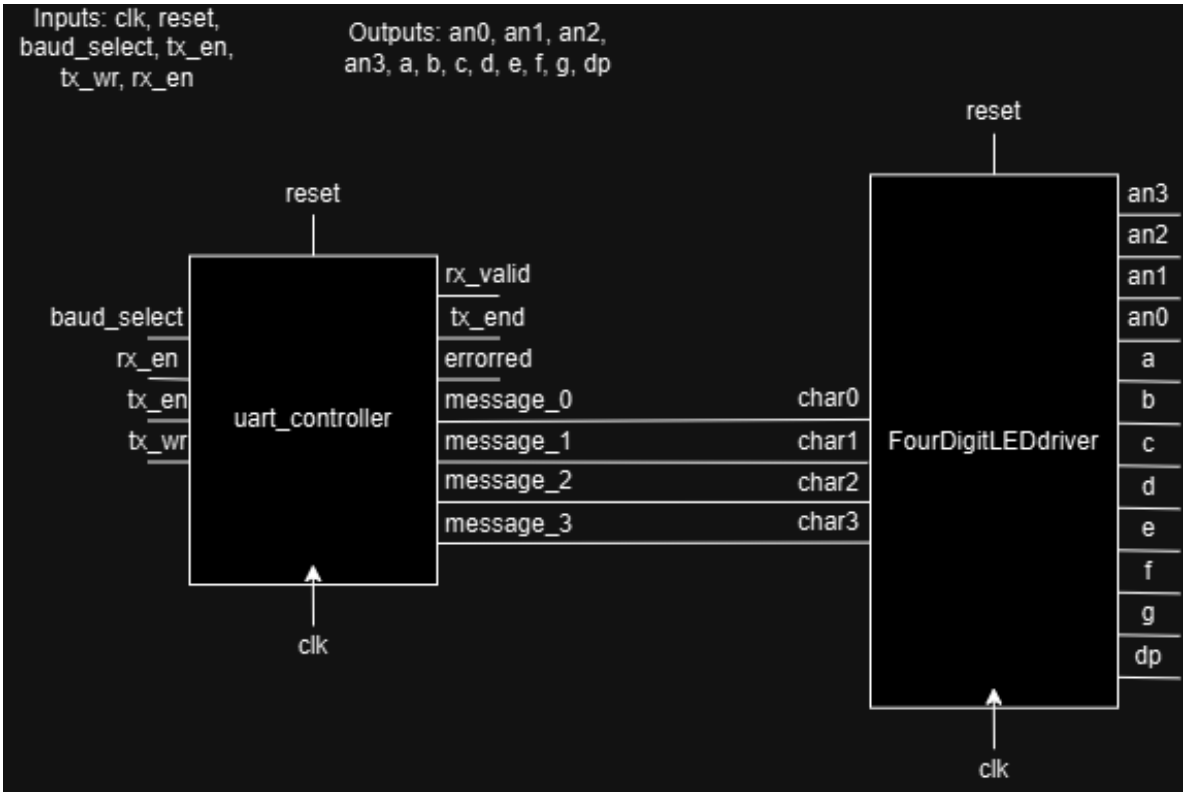


Figure 13: Data flow of the led_uart_connector module

7.2 Simulation

The simulation for this part was easy once I got the previous module from part D to work correctly. I used the same testbench module from part D and I only had to check that the anode signals and the character signals were driven correctly and displaying the desired value. The implemented design still had some warnings but most of them were the same as part D. The other warning was about the `dp` signal being a constant 1 which is not a problem since we don't want to use the decimal point on the display. Below is an image displaying how the characters change when a new message is received.

7.3 Live testing

I think this part was a moment of redemption where I got to see the UART protocol in action. I initially made `tx_en` and `rx_en` a constant 1 and used the switches to select the desired baud rate. The up button was connected to the `tx_wr` which was then send through a de-bouncer and made into a single pulse for the transmitter. Now after each button press the messages that were sent got displayed on the seven segment display. After I made sure that everything was displayed correctly I made some changes to the constraints and connected the enable signals to another 2 switches which allowed me to play around a bit more and find out what happens when either one is closed. You can view a small

video showing the functionality of the module here. Additionally here is a screenshot of the module running on the FPGA after all 4 messages have been received.

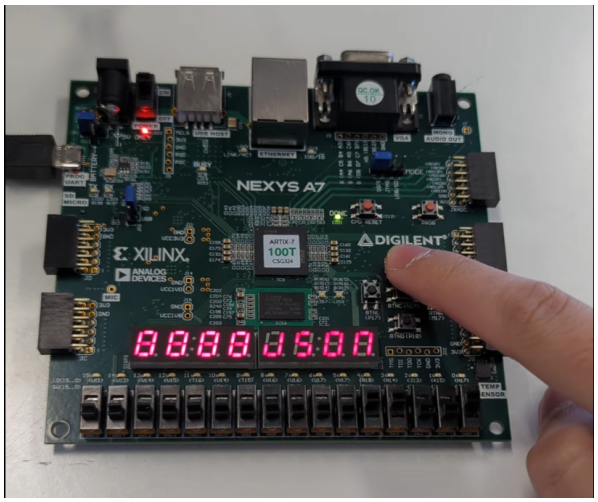


Figure 14: UART-led connector module running on the FPGA