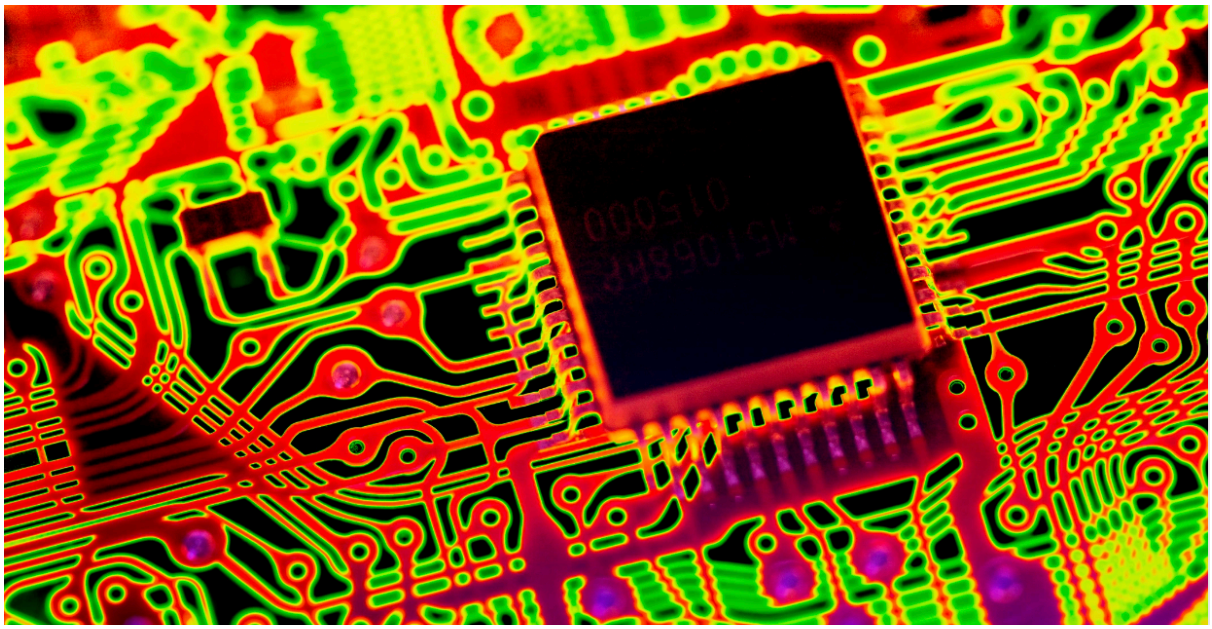
Student : Melika Keshavarzmirzamohammadi
Id : 2054633
Email : Melika.Keshavarzmirzamohammadi@studenti.unipd.it

# Bubble Sort Algorithm with VHDL implemented on FPGA

This project focuses on the implementation of the bubble sort algorithm for sorting ten 8-bit integers on a field-programmable gate array (FPGA). The primary module, referred to as the Main module, was developed and integrated with a UART transmitter and receiver component. The communication protocol was augmented with a 1-bit parity check to ensure the secure and reliable transmission of data. Additionally, a test bench was designed to simulate the performance of the implemented algorithm. This study showcases the successful implementation of the bubble sort algorithm on an FPGA platform and highlights the importance of reliable communication in embedded systems.



*FPGA*

## INTRODUCTION

### FPGA:

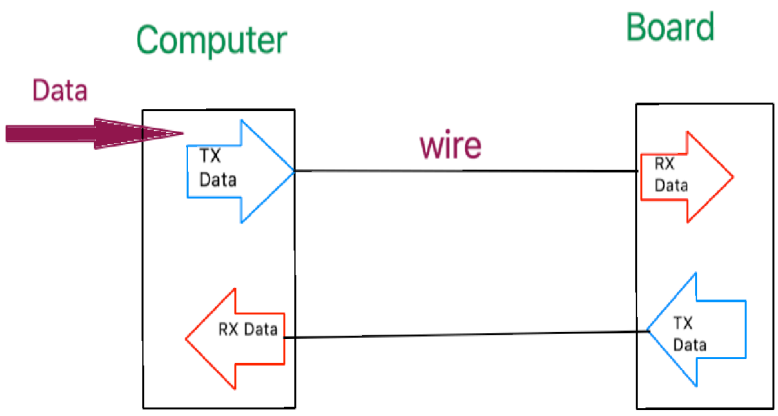
Field-Programmable Gate Arrays (FPGAs) are programmable digital devices that offer a high degree of flexibility and versatility for implementing complex digital circuits and systems. FPGAs can be programmed using hardware description languages such as VHDL, enabling the design of custom digital circuits and algorithms. FPGAs are widely used in many fields, including digital signal processing, image processing, and robotics, to name a few.

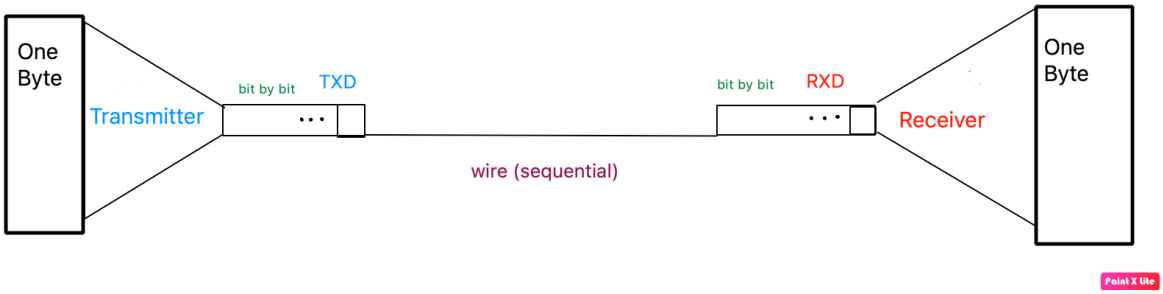
The communication between FPGAs and other devices is a crucial aspect of many FPGA applications. The Universal Asynchronous Receiver-Transmitter (UART) is a popular digital implementation used for serial communication between two devices. The UART provides a simple and reliable method for transmitting and receiving data, making it an ideal choice for communication between an FPGA and external devices.

## UART:

When connecting a computer to an FPGA through a UART transmitter, the computer typically sends data to the FPGA through the transmitter, and The FPGA's UART receiver then reads these bits and reassembles them into the original data stream.

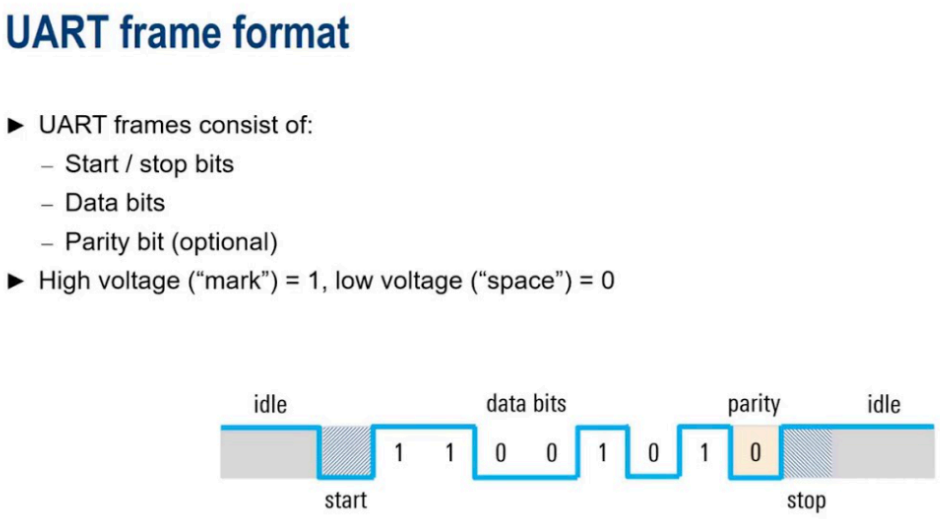
Similarly, when the FPGA needs to send data to the computer, it uses its UART transmitter to send a series of bits over the communication line, and the computer's UART receiver reads these bits and reassembles them into the original data stream.



## UART FORMAT:

The first received and sent bit in UART communication is always a ‘0’, known as the start bit. This bit changes the Idle to zero, to inform the system that the data is being sent. After the start bit, the data is sent in bits, which are sometimes followed by a parity bit to make sure the communication is reliable. In the end, a stop bit is generated to again inform the system that the first chunk of data in bits has been received or sent completely and to make sure the system is ready for receiving or sending the next chunk.




## Parity Generator and Parity Checker:

Parity is a method to make sure every bit which has been transmitted and received is communicated in the most reliable way. It is an additional bit transmitted after the data bits and is used to indicate whether the number of ‘1’s in the data being transmitted is odd or even to make sure none of the bits has been flipped.

The parity bit is generated by the transmitting device by implementing a “xor” operation between the transmitting data with ‘0’ or ‘1’ in the case of even or odd parity, respectively. The calculated parity bit gets inserted into the data stream, and the receiving device checks the parity bit, by implementing another “xor” between the received data, the transmitted parity bit, and ‘0’ for even parity or ‘1’ for odd parity. If the result of the xor calculation is ‘0’ it means the data transmitted and received did not face any changes and it is reliable. However, if the XOR result is not

'0', it indicates that the received data has been corrupted during transmission, and the communication is not considered reliable.

$$P_{checkeven} = A \oplus B \oplus C \oplus P_{even}$$

$$P_{checkodd} = \overline{A \oplus B \oplus C \oplus P_{odd}}$$ (not= parity(0) )

For even parity:

**Even Parity Generator**

| A | B | C | P |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Paritry generator for even aprity

| A | B | C | P | Parity Check (Error) |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Parity checker for even parity

# Methodology:

In this project, a bubble sort algorithm is developed and designed in Vivado environment which was tested and implemented on an Arty7 FPGA (xc7a35tcsg324-1).

This implementation had 4 main modules, 3 of which were designed in a hierarchical way, and the last module was a test bench.

## 1. Transmitter:

The algorithm receives Clk, Rst, Start, Parity, and Din as inputs. Rst, Start, and Parity are controlled by the higher-level module Main as needed. Din is a parallel data input that must be converted into a sequential bit stream for transmission over a wire.

The module includes a constant named Full, which is derived from two generics: Frequency, which represents the frequency of the FPGA's clock, and Baud, which represents the number of bits transmitted per second. Full represents the number of clock cycles required to receive or transmit a single bit.

The Transmitter module is a state machine composed of seven states: St_Idle, St_Start, St_Full, St_Data, St_Parity, St_Stop, and St_Return. The Idle state is the default mode of the algorithm, during which it transmits a sequence of high bits unless the Start input port is set to one by the high-level module. When the Start input is set to one, the algorithm transitions to the next state, the Start state. The Start state initializes the transmission process by sending the first low bit, commonly referred to as the start bit, and then transitions to the Full state. In the Full state, the algorithm determines the completion of the start bit transmission using the 'Full' constant and initiates the transmission of the first data bit while simultaneously transitioning to the Data state.

Subsequently, the algorithm enters a section similar to a loop in the Data state, transmitting the 8-bit data sequentially. Upon completion of the data transmission, the transmitter sends the parity bit if the parity check is enabled. Then, the state machine transitions to the Parity state where the transmitter finishes sending the parity bit and starts sending the stop bit. In the next state, the Stop state, the stop bit is sent and the algorithm gets finalized by moving to the Return state, in which if the Start input port is low the state machine returns to the Idle state and waits until the Start input port again is activated for sending the next byte.

## 2. Receiver:

The algorithm implemented by the receiver module exhibits significant similarities with that of the transmitter, albeit with the key distinction that the receiver undertakes the inverse operation of the transmitter, where it receives data as a sequential bit stream and subsequently outputs it in parallel form as a byte.

## 3. Main:

The algorithm is designed using the entity-architecture structure in VHDL. The entity "Main" has five ports: Clk, Rst, RXD, TXD, and LED. The Clk and Rst ports are inputs, RXD is the input port to receive data from an external device, and TXD is the output port to transmit data to an external device. The LED port is used to display some status information.

The architecture "Behavioral" defines the internal behaviour of the system using a set of components, signals, and a state machine. Two components are used to implement the receiver and transmitter modules. The receiver component receives data and stores it in an internal memory array. The transmitter component sends data stored in the memory array to the external device.

The state machine has 12 states, including Idle, Start, Rec0, Rec1, Rec2, Proc0, Proc1, Send0, Send1, Send2, Send3, and Return. The system starts in the Idle state, where all components are reset. In the Start state, the receiver component is enabled, and the LED is set to 001. In the Rec0, Rec1, and Rec2 states, the receiver component receives data and stores it in the memory array. In the Proc0 and Proc1 states, the data is bubble-sorted by swapping the elements in the memory array. In the Send0, Send1, Send2, and Send3 states, the transmitter component sends the processed data to the external device. Finally, in the Return state, the system returns to the Idle state. Various signals are used to control the operation of the system. For example, the RX_Rst, RX_Start, RX_Busy, and RX_Error signals are used to control the receiver component. The Tx_Rst, Tx_Start, TX_Busy, and TXD signals are used to control the transmitter component. The Mem signal is used to store data in an internal memory array. The Cnt0 and Cnt1 signals are used to count iterations and implement the bubble sort algorithm. The Number signal is used to track the current position in the memory array.

## 4. Main_tb:

This code described a test bench for our digital design which will test and simulate our design to make sure it works without any problems. Simulation is very important since it makes debugging and understanding the code very easy.

This module consists of three module components: Main, Transmitter, and Receiver. In VHDL, the term UUT (Unit Under Test) refers to the module or entity that is being tested. In this case, Main is the UUT.

The interesting part is that the RXD input of the receiver component is connected to the TXD output of the transmitter component, and the TXD output of the transmitter is connected to the RXD input of the receiver, allowing for bi-directional communication.

Bi-directional communication is a type of communication where data can flow in both directions between two devices. In other words, both devices can both send and receive data. Here bi-directional communication is achieved by connecting the TXD signal of the transmitter to the RXD signal of the receiver, and vice versa. This allows the receiver to receive data from the transmitter, and also to send data back to the transmitter when needed.

The Clk_process is responsible for generating the clock signal for the modules. It uses a wait statement to specify the period of the clock signal, which is determined by the Freq parameter. The process uses a signal to keep track of the current state of the clock signal and toggles it between high and low states on every clock period.

The next part of the code defines a process called stim_proc that is responsible for generating the simulation. The process begins by opening a text file TestData.txt in read mode, which contains the test data. It then initializes the inputs and waits for 10 clock cycles. After that, it reads the data from the file, converts it into a binary string, and sets the Tx_Data input to this value. When the TX_Start is ‘1’ the system waits for one cycle, while for TX_Busy, it makes sure as long as it is ‘1’ or ‘0’, it waits for as many clock cycles as it needs to finish in order to prevent transmitting a new piece of data while the transmitter is busy. It then indicates that the TX_Start remain at ‘0’ for two clock cycles.

After transmitting all the data, the process enters another loop, where it sets the Rx_Start to '1' to initiate the reception. It waits for the Rx_Busy signal to go low, indicating that the receiver is ready to receive data. It then waits for the Rx_Rdy signal to go high, indicating that the receiver has received the data without any errors. It then sets the Rx_Start input back to '0' and waits for 2 clock cycles before repeating the process for the next set of data.

in the stim_proc process, there are several wait-for statements, which pause the simulation for one clock cycle. This is done to ensure that the signals have enough time to propagate through the design and settle to their final values before the next step in the simulation is executed.

Similarly, the wait for Clk_period*10 statements is used to pause the simulation for 10 clock cycles, which is enough time to allow the reset signal to be asserted and the design to settle into a known state before the simulation proceeds to the next step.

Finally, the process enters an infinite loop where it waits indefinitely. This is because the simulation will continue indefinitely until manually stopped.

# Results

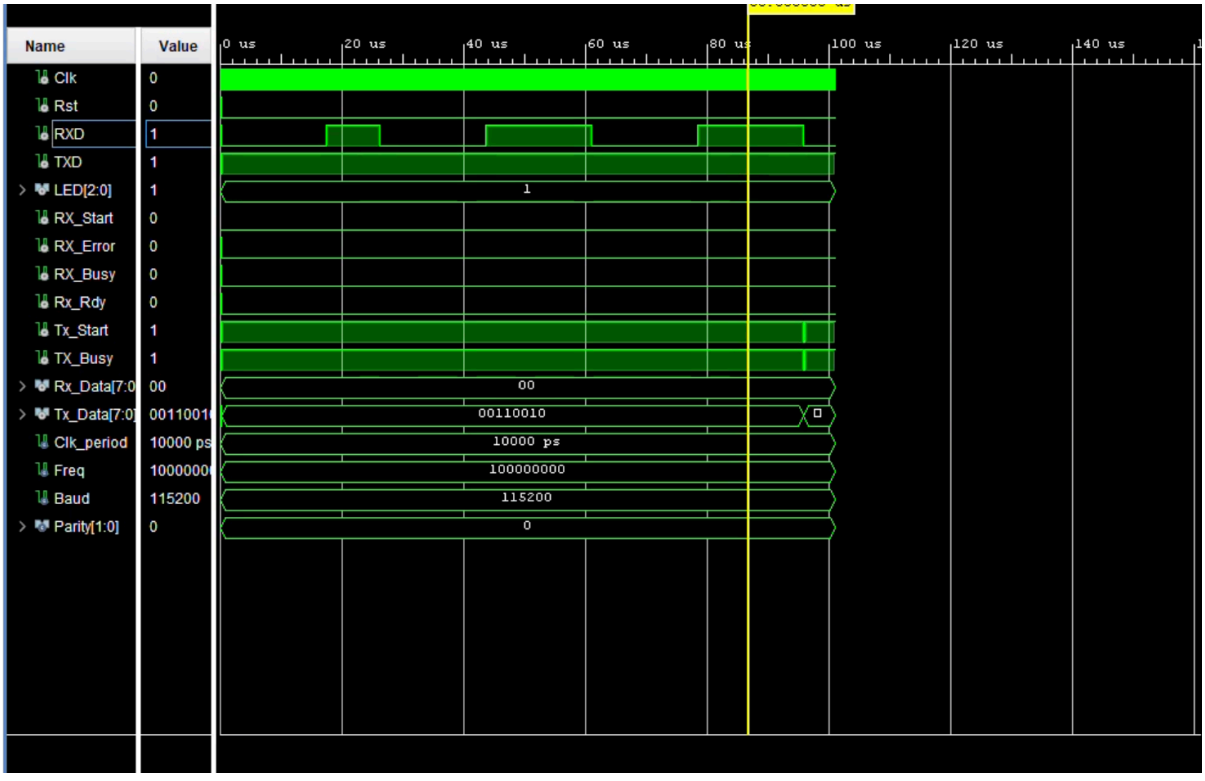
## A. Simulation Result:

The sorting algorithm performed very well under the simulation without any errors. The simulation is run for every 100 **µs**, due to the calculation below.

$$Time_{(perbit)} = \frac{1}{115200} = 8.6\mu s \qquad Time_{(11bits)} = 11 \times 8.6\mu s = 100\mu s$$

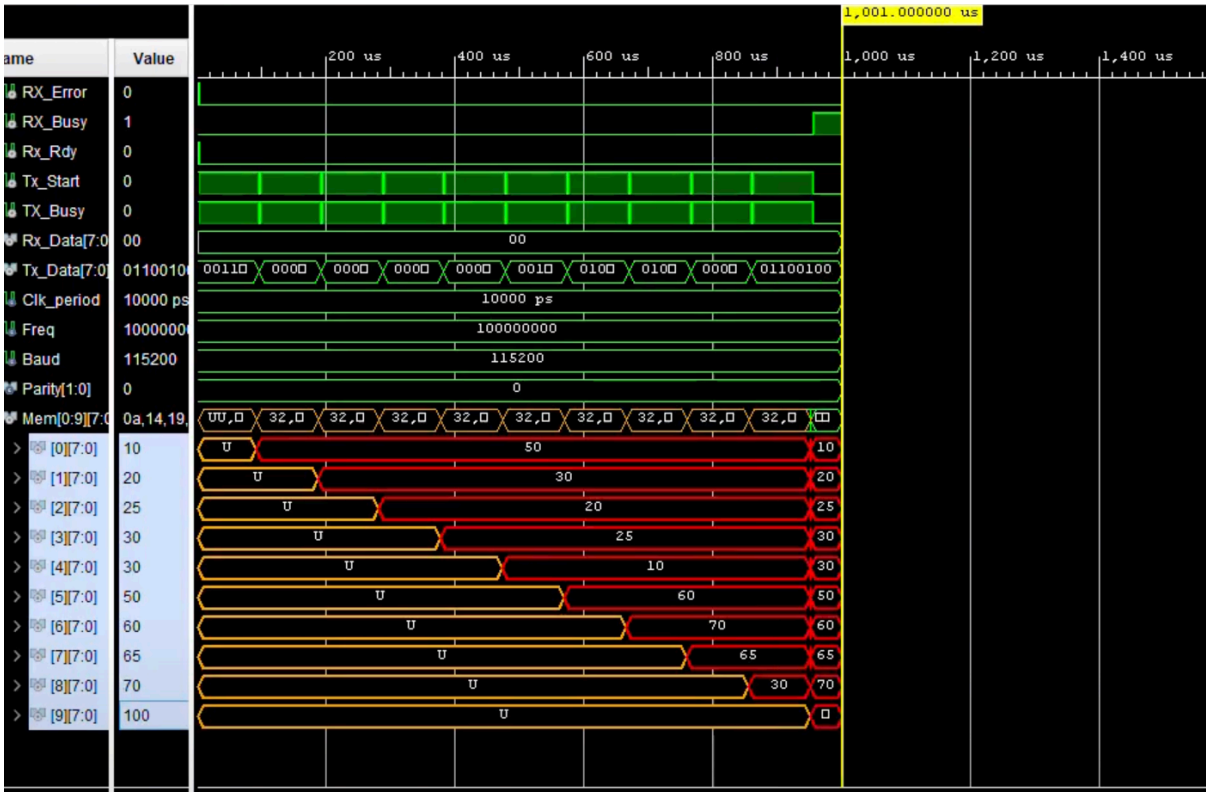
### 1. TX active:

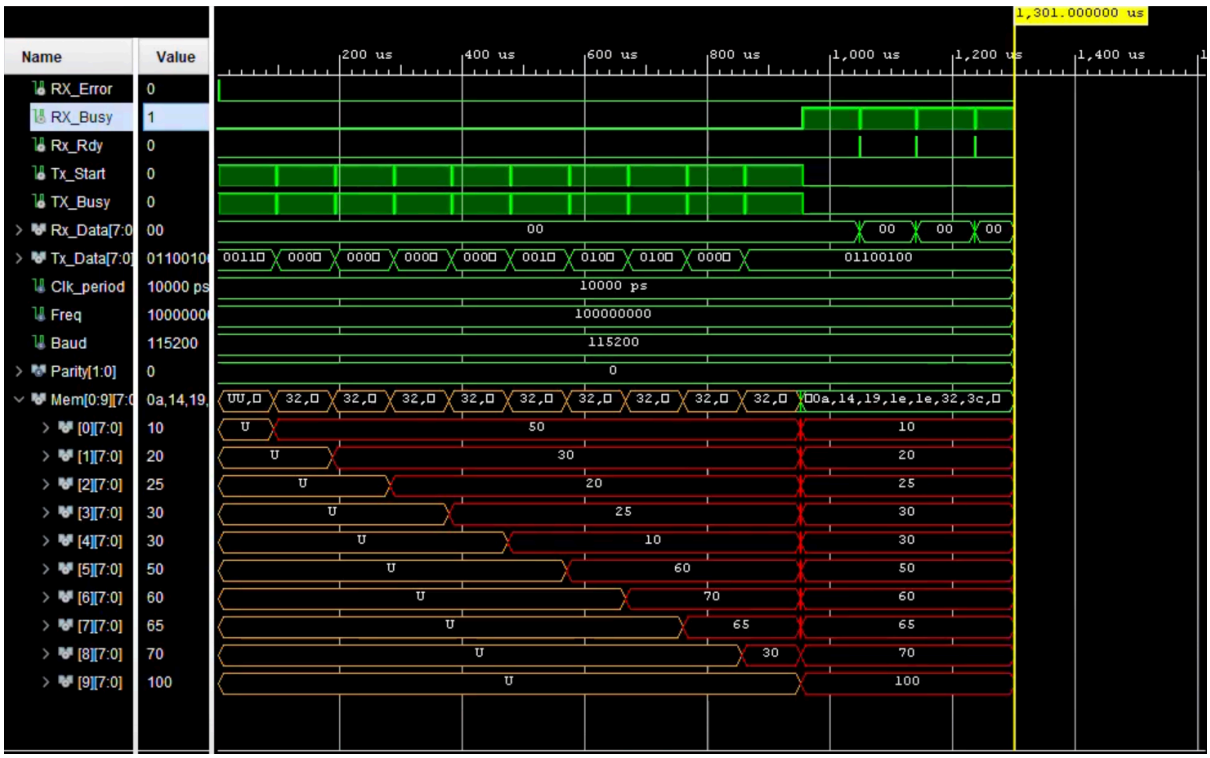
The picture below shows the waveform configuration for receiving one byte. The first received data is 50; therefore, RXD receives two zeros at first, one for the start bit and the other for the data itself, and ends with one bit even parity, which in this case is '1' and a single stop bit.



## 2. RX active:

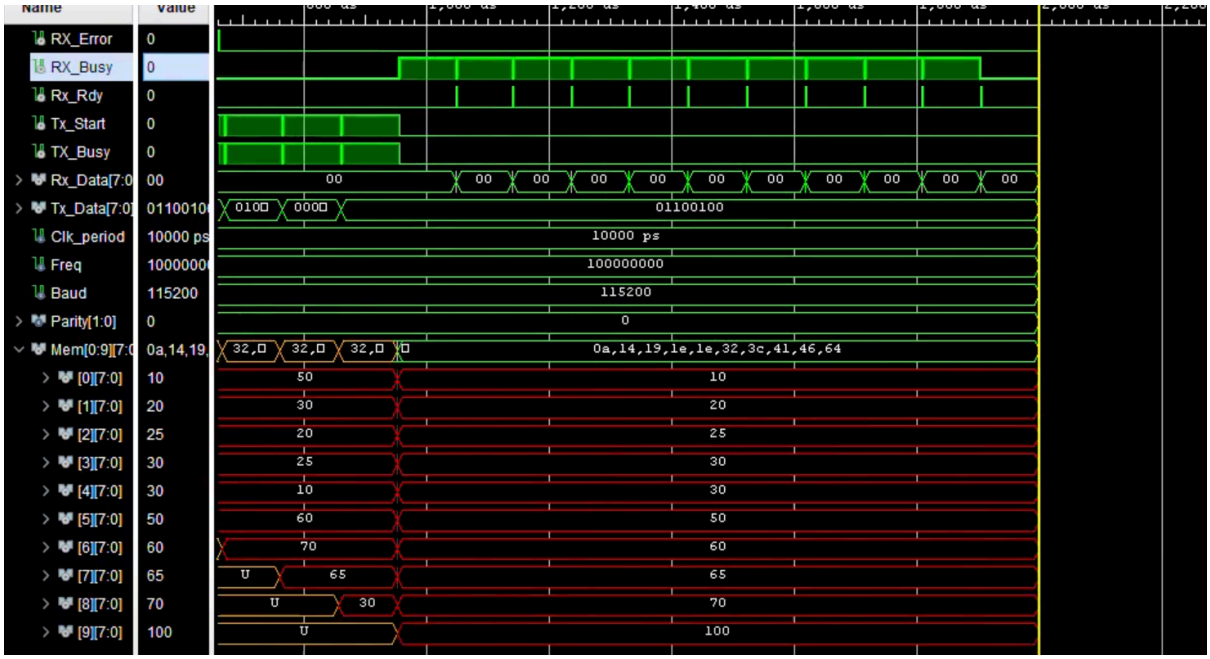
In the two pictures below, the transmission of the 10-byte data has been finished. The sorting section of the algorithm is very fast that we have to zoom in, in order to see the bubble sort operation on the 10-byte data. Right after the sorting the data is ready to be sent; therefore, the Receiver gets activated.



### 3. RX and TX deactivate:

In this picture, the sorted data is sent completely without any problems.




## B. FPGA Result:

For programming the FPGA and feeding data outside the simulation, I wrote a Python code to open the USB gate and feed a list of 10 randomly generated 8-bit integer numbers, and receive them in order, after getting sorted by the bubble sort.

In the Pyserial Notebook, the time it takes to bubble sort an array of 10-byte integers is calculated both in Python and FPGA. The outcome of timing for FPGA does not square with reality, since it is dependent on Python execution, and the UART performance. Therefore, the time for the bubble sort algorithm is directly measured in the Vivado simulation.


Bubble sort time:

Bubble sort in Python : 60 ms

Bubble sort in FPGA : 358 ns