[1]

UART in Verilog with a Chatbot implementation

Omer Al Sumeri, Zach O'Neil, and Josh Lopez

***Abstract*—UART, or Universal Asynchronous Receiver/Transmitter, is a serial communication protocol that operates between two devices. A byte is sent bit by bit between two UART components. The goal of the project is to demonstrate full duplex UART with an application of two Chatbots communicating with each other. Once the Chatbots receive the message, the message will be printed onto the terminal using ASCII translation.**

I. Introduction

The Universal Asynchronous Receiver/Transmitter (UART) serves as a serial communication protocol with the ability of exchanging data between two devices with a connection requiring only two wires.Specifically, the transmitter wire (Tx) of device 1 links to the receiver wire(Rx) of device 2, while the transmitter wire (Tx) of device 2 connects to the receiver wire(Rx) of device 1. There are multiple UART communication operations available to us for consideration. Simplex, which is the standard for one way transmitting, Half-Duplex allows the alternation of communication between the transmitter and receiver or Full-Duplex which is the operation chosen for our project allowing the simultaneous data transmission.

The key component of UART is the ability to transmit and receive data which is only possible by incorporating the standard Data Frame. This data frame, which consists of bits, transmits a certain amount of data with a sequence of a starting, ending and parity bit to ensure the proper communication between the transmitter and receiver.

As previously mentioned, UART operates asynchronously meaning the transmitter and receiver do not share a common clock signal. This asynchronization requires certain parameters of both devices to agree on a specified baud rate in order for both to communicate with each other properly. This baud rate is measured in bits per second and there are multiple or different baud rates to choose from depending on the design of the project. Not only do both UART devices have to share the same baud rate but they must also use the same data frame structure which allows real world application uses such as RS-232 interfaces, External Modems, Sensors, etc.

II. Specification and Goals

Our goal was to design the UART protocol and utilize it for sending text messages between two devices. The protocol would be defined for a fixed specific clock frequency and baud rate, meaning that each bit was to be sampled at a specific interval of 434 clock cycles per bit. When the UART module receives a complete message frame of a low start bit, 8 data bits, a parity bit for error checking, and a high end of frame as seen in Figure 1, it will then transfer the data byte over to the chat bot module. Upon receiving the byte, the chat bot will concatenate it into its current 8 byte message frame until all 8 bytes have been received. Once all 8 bytes have been received, the chat bot module will print a text message depending on the respective 8 bit ASCII value of each character.

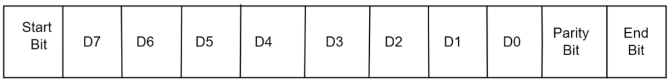

Figure 1: UART Message Frame

In addition to receiving a message from an external device, the chat bot and UART modules should also be able to send messages. This message would be loaded into the chatbot via the testbench, then serially transfer 8 bits over to the UART module until it has all 8 bytes of the message. Upon receiving each byte, the UART module will concatenate the data into the standard UART message frame. Parity checking will be used, so when there are an even number of ones in the byte, the parity bit will be set to 1 so that the amount of ones in the byte plus parity bit is odd. When there is an odd number of ones in the byte, the parity bit will be set to 0. The receiving UART module will check the byte and parity bit to see if there are an odd number of ones. If this is not the case, an error has occurred and the byte will not be sent to the receiving chat bot.

III. Design

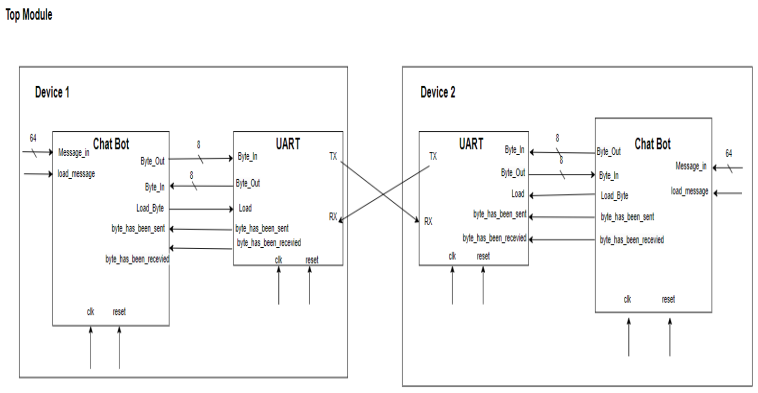

Figure 2: Complete Block Diagram of System

Our complete design was based on four state machine diagrams: one for UART receiving, another of UART transmitting, another for chat bot receiving, and the last one for chat bot transmitting. For UART receiving, the first state should be idle, which is when the receiving bit RX is reading ones serially. When the RX bit goes low, the start bit has been

[1]

detected and the state advances to the start_of_ packet. In the start_of_packet state, the interval of data sampling determined by the clock frequency and baud rate is offset so that each sample occurs at the middle of each bit. Sampling at this spot will help reduce misreading data. A clock count is incremented until it reaches 216 (representing the middle of the bit for our fixed number of clock cycles per bit), whereupon UART can start reading in the data. This occurs at the next state, bits are sampled upon a clock count of 433. Each bit is concatenated into Buffer_In until the Bit_Count_Rev reaches 9, whereupon the parity check state is advanced to. If the parity check returns a 1 (all bits of data and parity bit is XOR'd one by one) then the frame is odd as expected and no error has occurred. In this case, the data is sent to the chat bot. Otherwise, an error has occurred and data is not sent. In the stop bit state, a one should be sampled and the module goes back into idle.

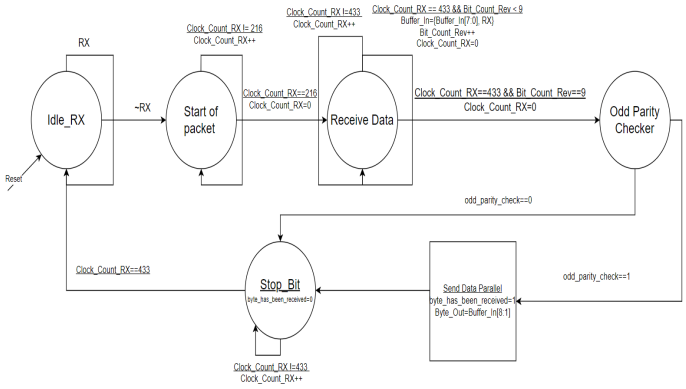

Figure 3: UART Receiving State Machine Diagram

For UART transmitting, the load bit is set low when a message needs to be sent. Otherwise the module stays in the idle state and TX transmits ones. Upon a low load bit, TX will transmit low indicating the start of the frame. Then the complete frame is constructed with a parity bit being set to whichever value makes the amount of ones present in the data portion odd. TX then transmits the data at the sampling rate until all frame bits have been sent, putting the module back into idle.

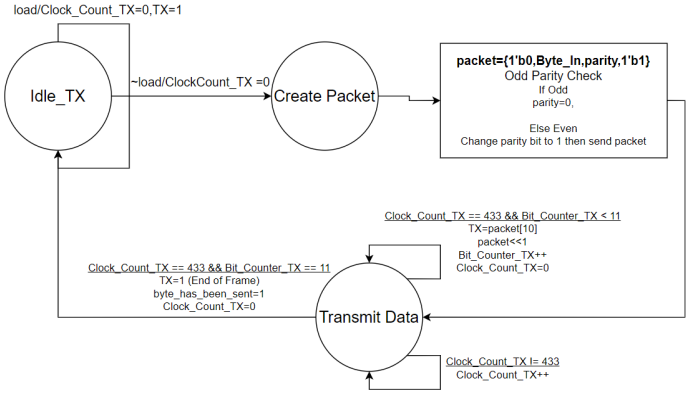

Figure 4: UART Transmitting State Machine Diagram

For chat bot receiving, the state is idle until the device's UART sends a high signal indicating a byte has been received, whereupon a byte is received in parallel and concatenated into the current message. Receiving of successive bytes happens between two states: one which concatenates the bytes into the message and another that waits for successive bytes from the UART. When the byte counter reaches 9, all 8 bytes have been received and the state advances to printing. Each character is represented by a certain byte value specified by ASCII, so the most significant byte from the complete 8 byte message is received and printed to the terminal via case statements. When the complete message has been shifted for each byte, the entire message has been printed and the chat bot returns to idle.

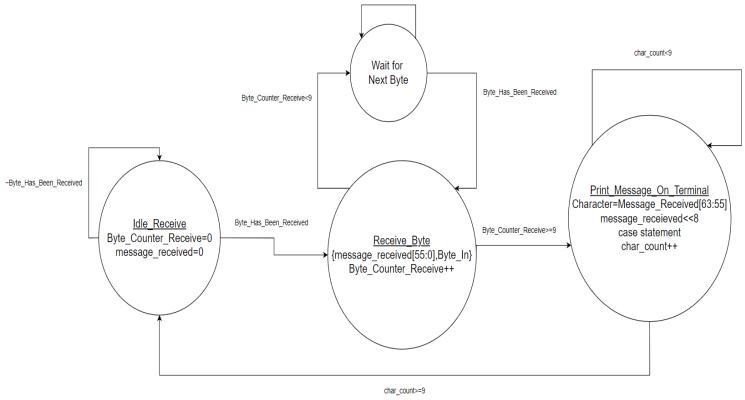

Figure 5: Chat Bot Receiving State Machine Diagram

Finally, the chat bot's transmitting state starts idle until the testbench sends a high signal indicating a message needs to be transmitted. The message specified by the testbench is loaded into the chat bot module and the message is transmitted to the UART byte by byte. This is accomplished through two states: one that loads the byte and another that waits for the UART to accept the byte, indicated by a signal sent from the UART. When the byte counter reaches 9, the entire message has been handed off to the UART module and the chatbot returns to idle.

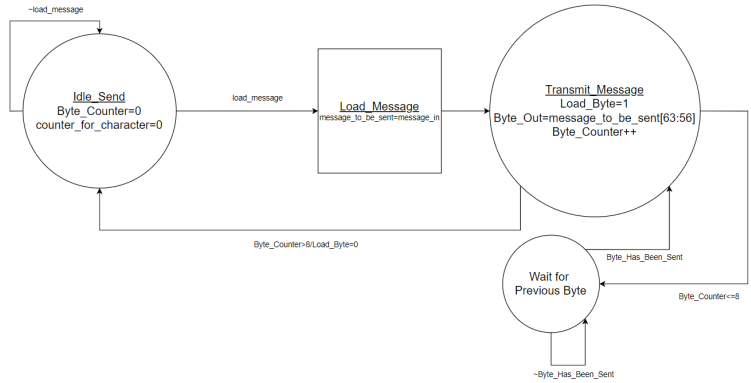

Figure 6: Chat Bot Transmitting State Machine Diagram

IV. Implementation

Verilog Implementation

The first module implementation was the UART module. The baud rate must be set in the UART module. The baud rate

implemented in this project is 115,200 bit/second. The frequency of the clock is divided by the baud rate to find the clock cycles per bit. The clock cycles per bit is then used to know when the bits should be sampled if a start bit is detected. In the UART implementation of this project, a 50 MHz clock was used to end up with 434 clock cycles per bit.

The UART module has four output registers and five inputs. Table 1 contains the port names and the purpose of each port inside of the UART module.

Table 1. Ports of UART Module

Port Name	Size of port (bits)	Input or Output	Purpose of Port
TX	1	Output	Signal to transmit the packet bit by bit
Byte Out	8	Output	The received byte from other UART module that will go to chat bot
Byte has been sent	1	Output	The signal to chat bot saying a byte has been sent
Byte has been received	1	Output	The signal to chat bot saying a byte has been received
RX	1	Input	Signal to receive the packet bit by bit
clk	1	Input	The clock of module, will be set to 50 MHz
reset	1	Input	Signal for reset to initialize state machines and reset all registers to 0
Load	1	Input	Signal from Chat Bot to start sending a byte
Byte In	8	Input	The byte that will be received from Chat Bot to transmit

The first state machine that was implemented was the state machine that receives a byte. The sensitivity list for that state machine is the negative edge of the clock and the positive edge of the reset button. If the reset is 1, then all registers that are used in the always block are reset and the state machine enters the idle state.
There are five states in the state machine for receiving a byte. The states are: Idle, Start of Packet, Receive Data Serially, Parity Checker, Send Data Parallel, and Stop Bit.

The state machine stays in the Idle state until RX is a 0. When RX is 0, registers used in the receiving state machine are reset and the next state is entered, Start of Packet.

The first bit of the packet is the Start of Packet. In the Start of Packet state, a counter is incremented starting from 0. Once the counter reaches clock cycles per bit minus 1 divided by 2, the counter is reset and the next state is entered, Receive Data Serially. What is happening is that the middle of the start bit is found using the clock cycles per bit number. The purpose of finding the middle of the start bit is to then increment clock cycles per bit amount every time to sample every incoming bit.

Once the Receive Data Serially state is entered, the counter is incremented until clock cycles per bit minus 1 is reached. Once that number is reached, there is another condition check. There is another counter, bit counter receive, to check how many bits have been received up until that point. If the number is less or equal to 8, then the incoming bit is sampled and put to the end of the buffer, the bit counter receive is incremented, clock cycles counter is set to 0, and the same state is entered again.

Once 9 bits have been received, which is the byte of data and the odd parity bit, the Parity Checker State is entered. In this state, there is a wire that checks the odd parity of the bits received and the parity bit by XORing all the bits together. If the result is 1, then an odd number of bits is detected and the next state is Send Data Parallel. If not, then an error was detected so the data is not sent out.

Once in the Send Data Parallel State, the signal byte received is sent High and the upper 8 bits of the 9 bits received are sent as Byte Out. The next state is the Stop Bit. Once in the Stop Bit state, the counter is incremented until the end of the stop bit is reached and the state machine goes back into the idle state.

The second state machine that is implemented is the interface for transmitting a byte. Following the same procedure as in

our first state, the sensitivity list for that state machine is the negative edge of the clock and the positive edge of the reset button. If the reset is 1, then all registers that are used in the always block are reset and the state machine enters the idle state.

There are three states in the state machine for transmitting a byte. The states are: Idle, Create Packet and Transmit Data.

The state machine stays in idle until load is a 1. When load is 1, registers used in the transmitting state are reset and the next state is entered, Create Packet.

When beginning the creation of the 11-bit packet, we must first consider the importance of maintaining data frame consistency across all communication devices. This is achieved by following the UART standard. The first bit read transmits as a 0, indicating the preparation of our starting frame. Following this, there are n bits of data, in our case, 8 bits. We have incorporated a parity bit checker into our data frame, which works by XORing each data bit. If the result is 1, we set the parity bit to 0, indicating that the data is successfully an odd amount of 1's. Otherwise, we set the parity bit to 1. Finally, the end bit remains as 1. Following this data frame protocol allows us to simulate standardized UART interfaces seen today. Once this packet is set, we then move to the next state, Transmit Data

Once in the Transmit Data state, a counter is incremented, starting from 0. When this counter reaches the successful clock cycles per bit, another condition is checked. There is an additional counter, bit counter transmit, which continues to increment until its condition is true. If the counter is less than 10, we perform the transmission of our packet serially until it reaches the end bit of our data frame. Once the full byte has been sent, we then send a signal indicating that the byte has been transmitted. After sending the signal indicating that the byte has been transmitted, we move to the next state, Idle, where we wait until the transmitter is notified to create a new packet.

The next module implementation was the Chat Bot module. This module will act as a communication device between two users where they are allowed to send characters to each other which we incorporated following the ASCII table.

The Chat Bot module has two output registers and seven inputs. Table 2 contains the port names and the purpose of each port inside of the Chatbot Module.

Table 2. Ports of Chat Bot Module

Port Name	Size of port (bits)	Input or Output	Purpose of Port
Load Byte	1	Input	Signal from Chat Bot to start sending a byte
Byte In	8	Input	The byte that will be received from Chat Bot to transmit
Load Message	1	Input	The signal to get out of idle and load a message
Byte has been received	1	Input	The signal to chat bot saying a byte has been received
Message in	64	Input	Message that is 8 bytes worth of characters
clk	1	Input	The clock of module, will be set to 50 MHz
reset	1	Input	Signal for reset to initialize state machines and reset all registers to 0
Byte Out	8	Output	The received byte from other UART module that will go to chat bot
Byte has been sent	1	Output	The signal to chat bot saying a byte has been sent

The first state machine that was implemented was the state machine that sends a byte. The sensitivity list for that state machine is a positive edge of the clock. If the reset is 1, then all registers that are used in the always block are reset and the state machine enters the idle send state.

There are four states in the state machine when sending a byte.

The states are: Idle Send, Load Message, Transmit Message and Wait for Previous Byte to Finish.

The state machine stays in the Idle send state until load is set to 1 where we then move to the next state Load Message, otherwise we stay inside the Idle send state.

Inside the Load Message state is where we load the 8 bytes of characters which will act as our message to be sent and then move to the next state Transmit Message.

Once inside the Transmit Message state we enter our first condition, a byte counter is then incremented until it reaches 8. Inside this condition block we send a load byte signal to 1 and then proceed to send byte by byte the characters to UART. Which we accomplish by incrementing our byte counter and shifting 8 bits to read each character inside our 64 bits worth of message. Once this is complete we then move onto Wait for Previous Byte To Finish State.

The second state machine that is implemented is to receive a byte. Following the same procedure as in our first state, the sensitivity list for that state machine is the positive edge of the clock. If the reset is 1, then all registers that are used in the always block are reset and the state machine enters the idle receive state.

There are four states in the state machine when sending a byte. The states are: Idle Received, Receive Byte, Wait for next byte to send and Print Message on Terminal.

Inside the Idle Receive state we check our condition signal if 1 we have received a byte and move onto Receive Byte state otherwise we stay inside the Idle Receive state.

Once inside the Receive Byte state, the byte counter is incremented until it reaches the value of 8, allowing the procedure of receiving a byte onto our 64-bit receiver register. Each increment of the counter moves us to the next state, Wait For Next Byte, ensuring that each byte has been received. We then loop back to the Receive Byte state until the condition of our byte counter is fulfilled. After the byte counter is complete, we move on to the state Print Message on Terminal.

In the Print Message on Terminal state, we have a counter that increments until each character is displayed on our terminal. This is accomplished by using the ASCII table as a reference inside a case statement and incrementing the shift of each byte to read each character to the terminal.

Testing and Waveforms

After implementing the UART and Chatbot modules in Verilog, the next step is the testing portion.

The UART module was tested individually to ensure that a single byte can be received and transmitted. **Figure 1** shows the top level module created for the individual test of UART.

Figure 2 shows the testbench created to test the UART module.

```
`timescale 1 ns/100 ps
module Top_Level  (
    output [7:0] Byte_Out_Device1, Byte_Out_Device2,
    input Load_Device1, Load_Device2,
    input Reset_Device1, Reset_Device2,
    input [7:0] Byte_In_Device1, Byte_In_Device2,
    input clk_Device1,clk_Device2
);

wire w1;
wire w2;


UART_Module Device_1 (
    .TX(w1),
    .Byte_Out(Byte_Out_Device1),
    .RX(w2),
    .clk(clk_Device1),
    .Byte_In(Byte_In_Device1),
    .load(Load_Device1),
    .reset(Reset_Device1)
);


UART_Module Device_2 (
    .TX(w2),
    .Byte_Out(Byte_Out_Device2),
    .RX(w1),
    .clk(clk_Device2),
    .Byte_In(Byte_In_Device2),
    .load(Load_Device2),
    .reset(Reset_Device2)
);


endmodule
```

Figure 1: Top Level Module for Individual test of UART

```
initial

begin
clk_Device1= 0; clk_Device2 =0; Reset_Device1=1; Reset_Device2 =1;

#10

Reset_Device1=0; Reset_Device2 =0;

#10

Load_Device1 = 1; Byte_In_Device1 = 8'h67;
#5

Load_Device1 = 0;
end

always
#10 clk_Device1 = ~clk_Device1;


always
#10 clk_Device2 = ~clk_Device2;
```

Figure 2: Testbench for Individual test of UART

In the test bench, both devices are reset to initialize them and the clock for both devices change every 10 nanoseconds to make a 50 MHz clock. Device 1 will send a message to Device 2. The message that Device 1 will send is hexadecimal 67. **Figure 3** shows the waveform from the testbench.

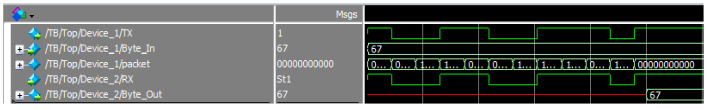

Figure 3: Waveform of the testbench for individual test for UART

The byte in for Device 1 is hexadecimal 67. The packet is seen

created and there are changes to the packet since the MSB is transmitted and a left shift operation happens to ensure the MSB is always sent. The RX of Device 2 has the same signal of TX of Device 1 which means that the signal packet is being transmitted correctly. After the packet has been transmitted, Byte Out of Device 2 has the value of hexadecimal of 67 which is the Byte In of Device 1, meaning the byte has been the UART module works to its specification.

The second testbench created was to test the Chatbot module sending a message containing 64 bits. **Figure 4** and **Figure 5** contain the instantiations for Device 1 and Device 2. Each device has a chatbot and a UART module.

```
wire [7:0] w1,w2;
wire w3,w4,w5;

UART_Module Device_1_UART (
    .TX(TX_Device_1),
    .Byte_Out(w1),
    .RX(RX_Device_1),
    .clk(clk_Device1),
    .Byte_In(w2),
    .load(w3),
    .reset(Reset_Device1),
    .byte_has_been_sent(w4),
    .byte_has_been_received(w5)
);

 Chat_bot Device_1_Chat_Bot(
        .Byte_Out(w2),
        .Load_Byte(w3),
        .Byte_In(w1),
        .load_message(load_message_Device1),
        .byte_has_been_sent(w4),
        .byte_has_been_received(w5),
        .Message_In(Message_in_Device_1),
        .clk(clk_Device1),
        .reset(Reset_Device1)
                );
```

Figure 4: Instantiations of Device 1

```
wire [7:0] w1,w2;
wire w3,w4,w5;

UART_Module Device_2_UART (
    .TX(TX_Device_2),
    .Byte_Out(w1),
    .RX(RX_Device_2),
    .clk(clk_Device2),
    .Byte_In(w2),
    .load(w3),
    .reset(Reset_Device2),
    .byte_has_been_sent(w4),
    .byte_has_been_received(w5)
);

 Chat_bot Device_2_Chat_Bot(
        .Byte_Out(w2),
        .Load_Byte(w3),
        .Byte_In(w1),
        .load_message(load_message_Device2),
        .byte_has_been_sent(w4),
        .byte_has_been_received(w5),
        .Message_In(Message_in_Device_2),
        .clk(clk_Device2),
        .reset(Reset_Device2)
);
```

Figure 5: Instantiations of Device 2

Figure 6 contains the Top Level module for the two devices communicating with each other.. Device 1 and Device 2 are instantiated in the Top Level Module.

```
`timescale 1 ns/100 ps
module Top_Level (
    input Reset_Device1, Reset_Device2,
    input clk_Device1,clk_Device2,
    input load_message_Device1, load_message_Device2,
    input [63:0] message_in_Device1, message_in_Device2
);

wire w1,w2,w3;


Device_1 D1(
        .Reset_Device1(Reset_Device1),
        .clk_Device1(clk_Device1),
        .load_message_Device1(load_message_Device1),
        .RX_Device_1(w2),
        .Message_in_Device_1(message_in_Device1),
        .TX_Device_1(w1)
                );

Device_2 D2(
        .Reset_Device2(Reset_Device2),
        .clk_Device2(clk_Device2),
        .load_message_Device2(load_message_Device2),
        .RX_Device_2(w1),
        .Message_in_Device_2(message_in_Device2),
        .TX_Device_2(w2)
                );


endmodule
```

Figure 6: Instantiations of Top Level Module

Figure 7 contains the testbench of the two devices communicating with each other. Both devices are reset to initialize them. Device 1 will send a message to Device 2. The message register for Device 1 will be loaded with the hexadecimal value 68656c6c6f202020. The message using ASCII translation is “Hello “.

```
initial

begin
clk_Device1= 0; clk_Device2 =0; Reset_Device1=1; Reset_Device2 =1;

#10 Reset_Device1=0; Reset_Device2 =0;

#10 load_message_Device1=1; message_in_Device1= 64'h68656C6C6F202020;

#100 load_message_Device1=0;

end

always
#10 clk_Device1 = ~clk_Device1;


always
#10 clk_Device2 = ~clk_Device2;
```

Figure 7: Testbench for two devices communicating with each other

Figure 8 contains the waveform of running the test bench created. The Byte Out signal Chat Bot, Device 1 is each byte of the message being sent with the most significant byte being sent first. The waveforms have the TX signals of Device 1 and RX signals of Device 2. The signals are the same showing the bits are being transmitted correctly. After one byte has been sent, it can be seen that a byte has been received since the Chat Bot for Device 2 receives that byte. **Figure 8** shows Device 2 receiving every byte that Device 1 sends in the sequence of the most significant byte being sent.

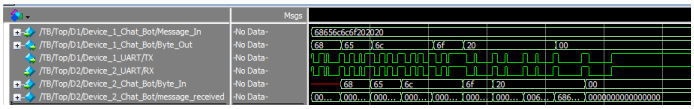

Figure 8: Waveforms of the testbench

Figure 9 shows the message, “Hello “, printed onto the terminal after around 960 microseconds.

```
# reading C:/intelFPGA/16.1/modelsim_ase/win32aloem/../modelsim.ini
# Loading project Project
# End time: 10:24:28 on Dec 12,2023, Elapsed time: 0:12:03
# Errors: 0, Warnings: 6
ModelSim> vsim -gui work.TB
# vsim -gui work.TB
# Start time: 10:42:08 on Dec 12,2023
# Loading work.TB
# Loading work.Top_Level
# Loading work.Device_1
# Loading work.UART_Module
# Loading work.Chat_bot
# Loading work.Device_2
add wave -position insertpoint sim:/TB/Top/D1/Device_1_UART/*
add wave -position insertpoint sim:/TB/Top/D1/Device_1_Chat_Bot/*
add wave -position insertpoint sim:/TB/Top/D2/Device_2_UART/*
add wave -position insertpoint sim:/TB/Top/D2/Device_2_Chat_Bot/*
VSIM 20> run
run
run
run
run
run
run
run
run
 hello  run
run
```

V. Conclusion

In conclusion, the Universal Asynchronous Receiver/Transmitter (UART) plays a pivotal role in facilitating serial communication between devices through a simple two-wire connection. The versatility of UART is highlighted by its support for various communication modes, including Simplex, Half-Duplex, and Full-Duplex, At the core of UART's functionality is the standard Data Frame, composed of bits with starting, ending, and parity bits to ensure accurate communication. Operating asynchronously, UART requires devices to agree on a specified baud rate, measured in bits per second, and to have a consistent data frame structure. This project has taught the importance of choosing the right clock edges and timing.

This project implements the UART communication protocol in Verilog and has an application of two devices communicating with each other.