

UART Communication Protocol

1. INTRODUCTION :

UART ,or Universal Asynchronous Receiver Transmitter Protocol.is one of the most used device-to-device communication protocols.UART can work with many different types of serial protocols that involve transmitting and receiving serial data. In serial communication, data is transferred bit by bit using a single line or wire. In two-way communication, we use two wires for successful serial data transfer.

From name itself we can understand the function of UART,where **U** stand for universal which means this protocol can be applied to any transmitter and receiver, and **A** is for Asynchronous which mean one cannot shared use clock signal for communication of data and **R** and **T** refer to Receiver and transmitter.

There are Two wire in which only one wire is used for transmission and other wire is used for reception. Data format and transmission speeds can be configured here. So, before starting with the communication define the data format and transmission speed.

On comparing this serial communication of UART with parallel then it can be observed that in parallel multiple buses are required.Based on the number of lines bus complexity of UART is better but parallel communication is good in terms of speed. So, when speed is required at that time we should select parallel communication and for a low-speed application, UART must be used and the bus complexity will be less.

For UART serial communications, the baud rate needs to be set the same on both the transmitting and receiving end's. The baud rate is the rate at which information is transferred over communication channel. In the serial port context, the set baud rate will serve as the maximum number of bits per second to be transferred.

The two signals of each UART device are named:

- Transmitter (Tx)
- Receiver (Rx)

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.

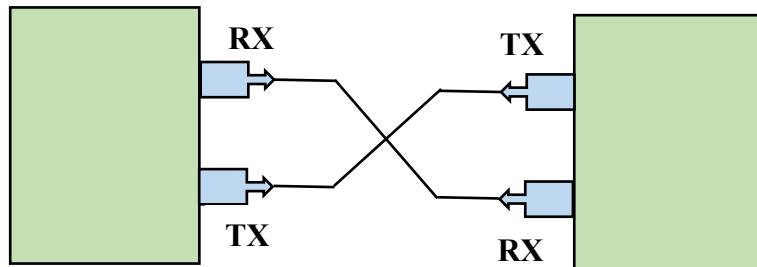


Figure 1. Two device communicate using **UART** with each other.

The transmitting UART is connected to a controlling data bus that sends data in a parallel form. From this, the data will now be transmitted on the transmission line

(wire) serially, bit by bit, to the receiving UART. This, in turn, will convert the serial data into parallel for the receiving device.

The UART interface does not use a clock signal to synchronize the transmitter and receiver devices; it transmits data asynchronously. Instead of a clock signal, the transmitter generates a bitstream based on its clock signal while the receiver is using its internal clock signal to sample the incoming data. The point of synchronization is managed by having the same baud rate on both devices. Failure to do so may affect the timing of sending and receiving data that can cause discrepancies during data handling. The allowable difference of baud rate is up to 10% before the timing of bits gets too far off.

DATA TRANSMISSION:

In UART, the mode of transmission is in the form of a packet. The piece that connects the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines. A packet consists of a start bit, data frame, a parity bit, and stop bits.

Start Bit (1 bit)	Data Frame (5 to 9 Data data Bit)	Parity Bit (0 to 1 bit)	Stop Bits (0 to 2 bit)
----------------------	--------------------------------------	----------------------------	---------------------------

Figure 2. Start bit.

Start Bit:

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Start Bit (1 bit)	Data Frame (5 to 9 Data data Bit)	Parity Bit (0 to 1 bit)	Stop Bits (0 to 2 bit)
----------------------	--------------------------------------	----------------------------	---------------------------

Figure 3. Start bit

Data Frame:

The data frame contains the actual data being transferred. It can be five (5) bits up to eight (8) bits long if a parity bit is used. If no parity bit is used, the data frame can be nine (9) bits long. In most cases, the data is sent with the least significant bit first.

Start Bit (1 bit)	Data Frame (5 to 9 Data data Bit)	Parity Bit (0 to 1 bit)	Stop Bits (0 to 2 bit)
----------------------	--------------------------------------	----------------------------	---------------------------

Figure 4. Data frame.

Parity:

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.

After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.

When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Start Bit (1 bit)	Data Frame (5 to 9 Data data Bit)	Parity Bit (0 to 1 bit)	Stop Bits (0 to 2 bit)
------------------------------	--	------------------------------------	-----------------------------------

Figure 5. Parity bit.

Stop Bits:

To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration.

Start Bit (1 bit)	Data Frame (5 to 9 Data data Bit)	Parity Bit (0 to 1 bit)	Stop Bits (0 to 2 bit)
------------------------------	--	------------------------------------	-----------------------------------

Figure 6. Stop bit.

UART Transmission:

- First The Transmitter received the data in parallel form from the data bus.

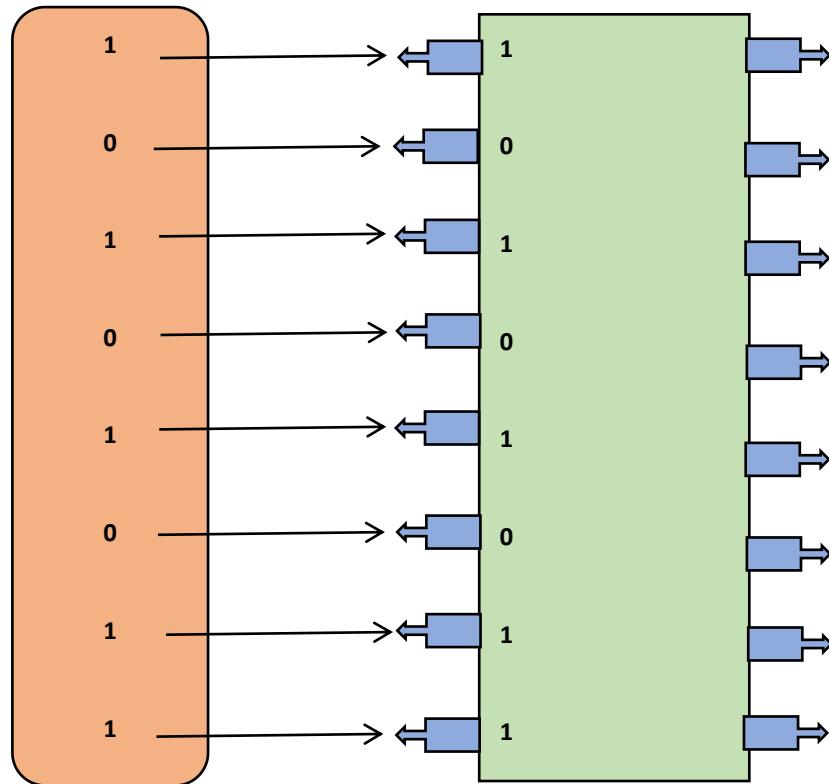


Figure 7. Data bus to the transmitting UART.

- The Transmitting UART adds the start bit, parity bit, and the stop bit to the data frame.

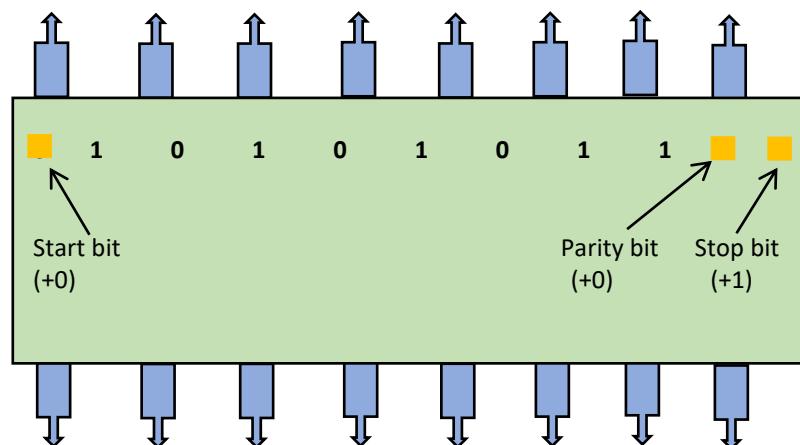


Figure 8. UART data frame at the Tx side.

3. The entire packet is sent serially starting from start bit to stop bit from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate.

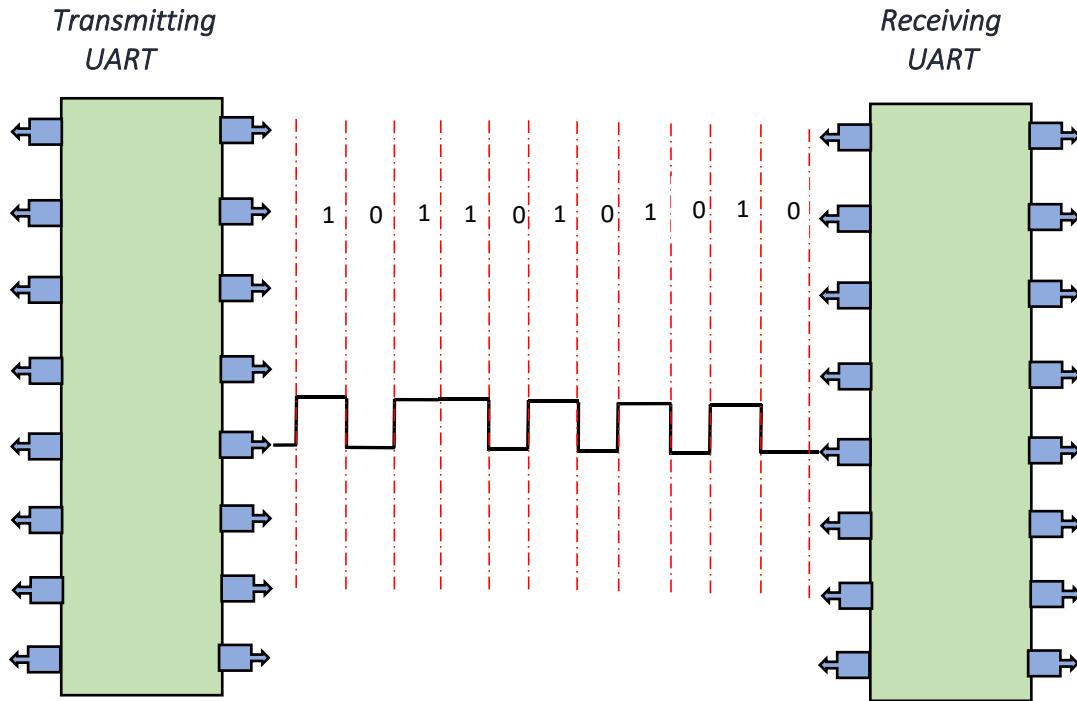


Figure 9. UART transmission.

4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame.

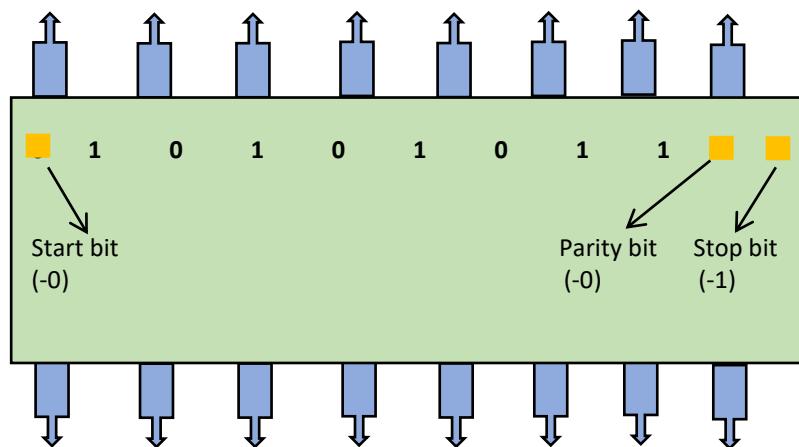


Figure 10. The UART data frame at the Rx side

5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end.

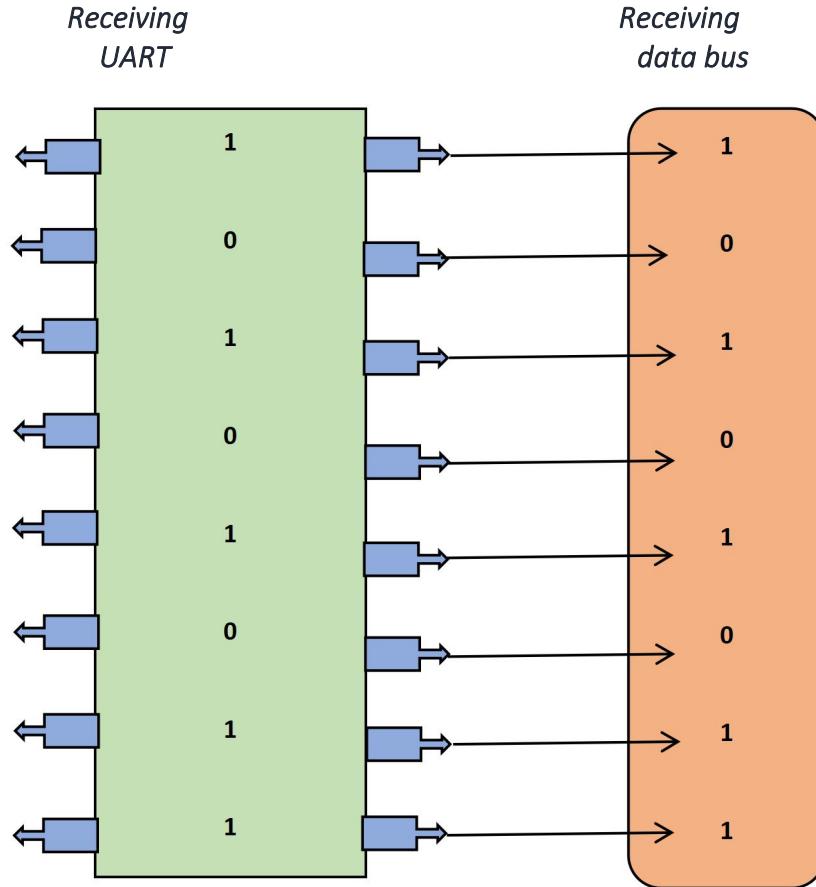


Figure 11. Receiving UART to data bus.

PROJECT IMPLEMENTATION:

1. Baud Rate Generator:

Baud rate=115200 bit/sec

Internal clock frequency =25MHZ

Bit Time: Amount of duration of time which is allocated to single bit which is given as follow.

$$\text{Bit Time} = \frac{1}{\text{Baud Rate}}$$

$$\text{Bit time} = \frac{1}{115200} = 8.681\mu\text{s}$$

So baud rate always inversely proportional to bit time as baud rate increases bit time decreases.

Numbers of cycle to send the single bit at desired baud rate is given below.

$$\text{No of cycle's} = \text{bit time} \times \text{frequency}$$

$$\text{No of cycle's (for transmitter)} = \frac{25 \times 10^6}{115200} = 217 \text{ cycle's}$$

\therefore for sending 1 bit 217 cycle are required to achieve 115200 baud rate

\therefore **At master (transmitter)** required counter to count 0 to 216 cycle for sending single bit and again it reset to 0 for sending other bit.

$$\text{No of cycle's (for receiver)} = \frac{25 \times 10^6}{115200 \times 16} = 13.5 \approx 14 \text{ cycle's}$$

At the receiver, the number of clock cycles required to receive a single bit is 14 cycles. The difference between the transmitter and receiver timing occurs because, in UART, there is no shared clock between the transmitter and receiver. Due to possible baud-rate mismatch and clock drift, the receiver cannot rely on exact bit timing like the transmitter.

To avoid missing data, the receiver uses a $16\times$ oversampling technique. It samples the incoming data multiple times within one bit period and selects the sample near the center of the bit. Because of start-bit detection and internal synchronization, the effective sampling of each bit happens after about 14 cycles instead of 16, which improves reliability and ensures correct data reception.

\therefore **At slave (receiver)** required counter to count 0 to 14 cycle for receiving single bit and again it reset to 0 for re other bit.

In the code, we use an 8-bit tx_counter to count from 0 to 216 because the transmitter directly generates the baud rate and must hold each bit for the full bit duration. On the other hand, the receiver uses a 4-bit rx_counter to count from 0 to 14 because it operates with oversampling ($16\times$) and samples the incoming data near the center of each bit.

CODE:

```

module baud_rate_gen(
    input clk,
    input rst,
    output tx_enb,
    output rx_enb
);

// TX Counter: 25MHz / 115200 = 217. Count 0 to 216.
reg [7:0] tx_counter;
// RX Counter: 25MHz / (115200 * 16) = 13.5 (~14). Count 0 to 13.
reg [3:0] rx_counter; //16 is sampling factor

always @ (posedge clk) begin
    if (rst)
        tx_counter <= 8'd0;
    else if (tx_counter == 8'd216)
        tx_counter <= 8'd0;
    else
        tx_counter <= tx_counter + 1'b1;
end

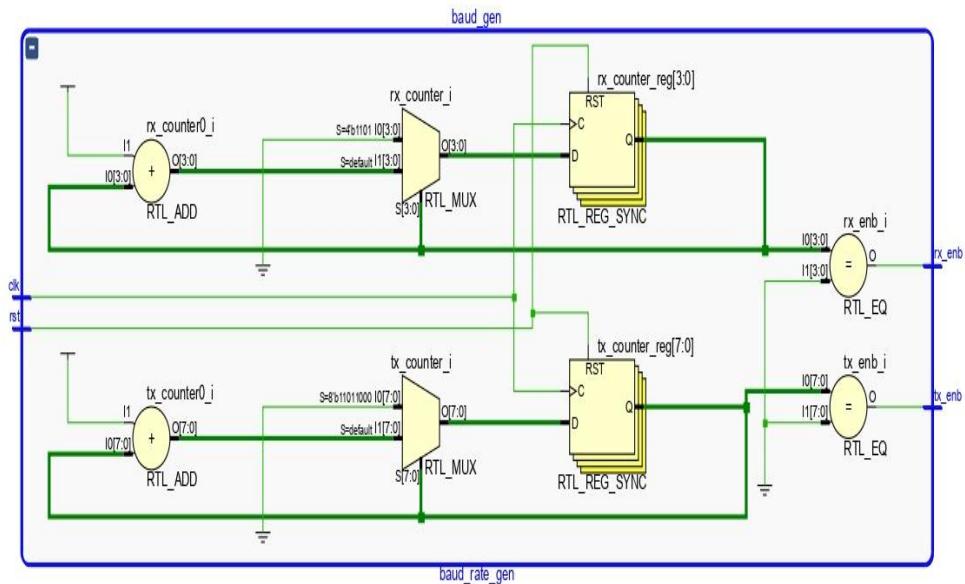
always @ (posedge clk) begin
    if (rst)
        rx_counter <= 4'd0;
    else if (rx_counter == 4'd13)
        rx_counter <= 4'd0;
    else
        rx_counter <= rx_counter + 1'b1;
end

assign tx_enb = (tx_counter == 8'd0);
assign rx_enb = (rx_counter == 4'd0);

endmodule

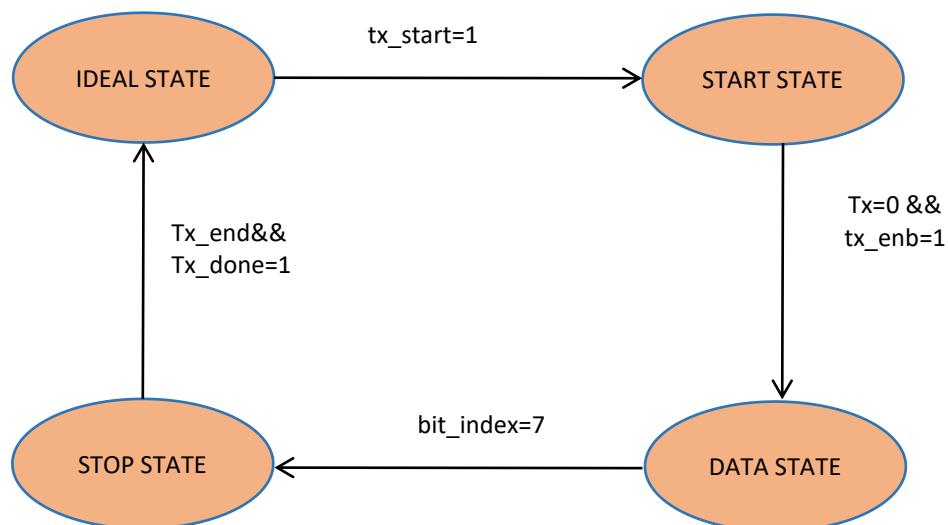
```

Schematic;



Transmitter:

FSM:

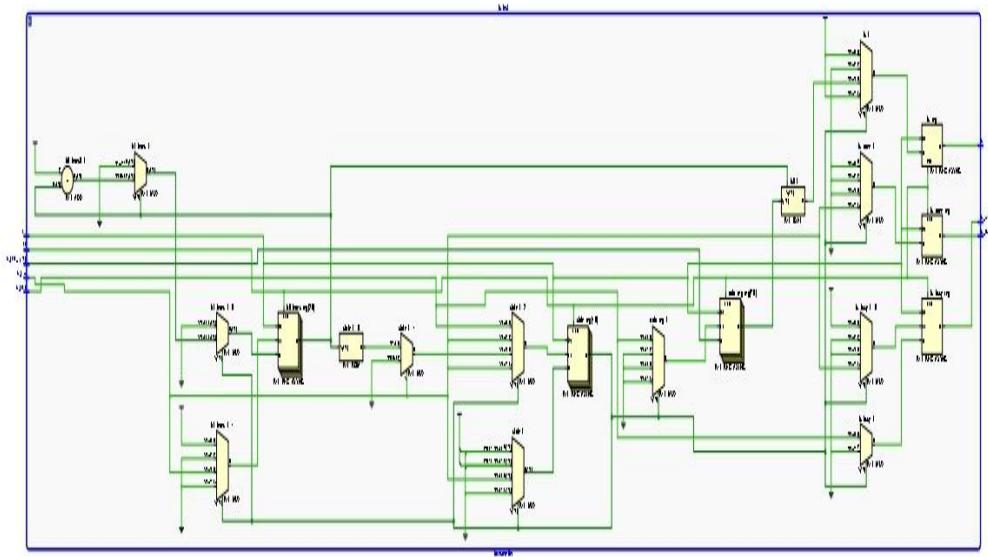


CODE

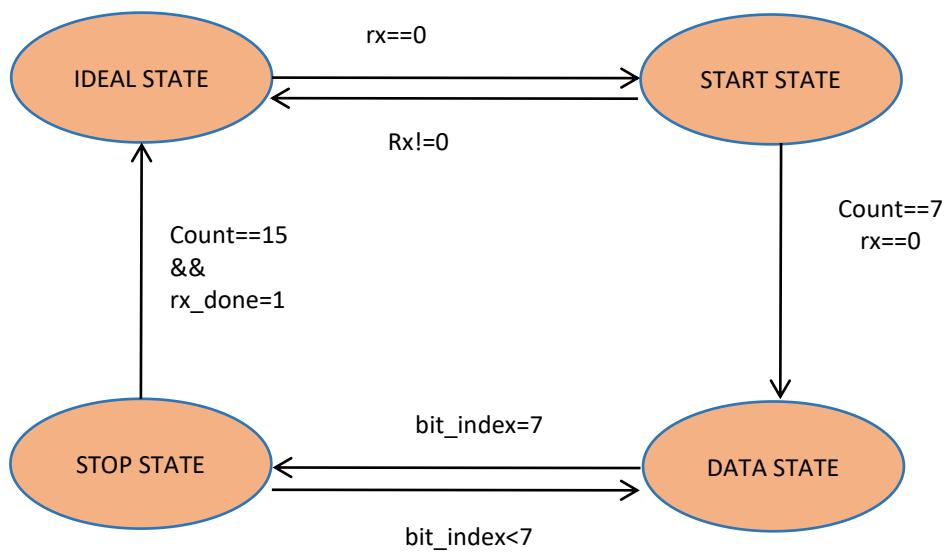
```
1  module transmitter
2  (
3      input  wire clk,
4      input  wire rst,
5      input  wire tx_start,
6      input  wire [7:0] tx_data_in,
7      input  wire tx_enb,    // Connected to your generator
8      output reg  tx,
9      output reg  tx_busy,
10     output reg  tx_done
11 );
12
13 parameter IDLE  = 2'b00;
14 parameter START = 2'b01;
15 parameter DATA  = 2'b10;
16 parameter STOP  = 2'b11;
17
18 reg [1:0] state;
19 reg [2:0] bit_index;
20 reg [7:0] data_reg;
21
22 always @(posedge clk or posedge rst) begin
23     if (rst) begin
24         state      <= IDLE;
25         tx        <= 1; // Idle High
26         tx_busy   <= 0;
27         tx_done   <= 0;
28         bit_index <= 0;
29         data_reg  <= 0;
30     end
31     else begin
32
33         tx_done <= 0; // Default low
```

```
      |
35 ⌂      case (state)
36 ⌂          IDLE: begin
37          tx <= 1;
38          bit_index <= 0;
39 ⌂          if (tx_start) begin
40              data_reg <= tx_data_in;
41              tx_busy <= 1;
42              state <= START;
43          end
44 ⌂          else begin
45              tx_busy <= 0;
46          end
47      end
48
49 ⌂      START: begin
50          tx <= 0; // Start bit = 0
51 ⌂          if (tx_enb)
52              state <= DATA;
53      end
54
55 ⌂      DATA: begin
56          tx <= data_reg[bit_index];
57 ⌂          if (tx_enb) begin
58              if (bit_index == 7) begin
59                  bit_index <= 0;
60                  state <= STOP;
61              end
62 ⌂          else begin
63              bit_index <= bit_index + 1;
64          end
65      end
66  end
67
68 ⌂      STOP: begin
69          tx <= 1; // Stop bit = 1
70 ⌂          if (tx_enb) begin
71              tx_done <= 1;
72              state <= IDLE;
73              tx_busy <= 0;
74      end
```

Schematic:



Receiver:



CODE

```

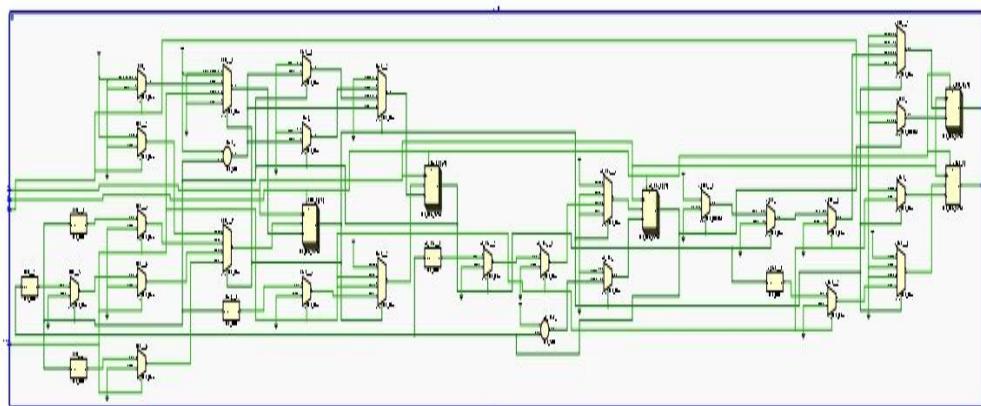
1  module receiver
2  (
3      input  wire clk,
4      input  wire rst,
5      input  wire rx,
6      input  wire rx_enb,      // Connected to baude rate generator
7      output reg  rx_done,
8      output reg [7:0] rx_data
9  );
10
11 parameter IDLE  = 2'b00;
12 parameter START = 2'b01;
13 parameter DATA  = 2'b10;
14 parameter STOP  = 2'b11;
15
16 reg [1:0] state;
17 reg [3:0] count;
18 reg [2:0] bit_index;
19
20 always @(posedge clk or posedge rst) begin
21 if (rst) begin
22     state      <= IDLE;
23     rx_done    <= 0;
24     count     <= 0;
25     bit_index <= 0;
26     rx_data   <= 0;
27 end
28 else begin
29
30 case (state)
31 IDLE: begin
32     rx_done    <= 0;
33     count     <= 0;
34     bit_index <= 0;
35 if (rx == 0) // Start bit
36     state <= START;
37 end

```

```
38 '
39     START: begin
40         if (rx_enb) begin
41             if (count == 7) begin
42                 count <= 0;
43                 if (rx == 0)
44                     state <= DATA;
45                 else
46                     state <= IDLE;
47             end
48             else begin
49                 count <= count + 1;
50             end
51         end
52     end
53 '
54     DATA: begin
55         if (rx_enb) begin
56             if (count == 15) begin
57                 count <= 0;
58                 rx_data[bit_index] <= rx;
59                 if (bit_index == 7)
60                     state <= STOP;
61                 else
62                     bit_index <= bit_index + 1;
63             end
64             else begin
65                 count <= count + 1;
66             end
67         end
68     end
```

```
69 ;
70     STOP: begin
71         if (rx_enb) begin
72             if (count == 15) begin
73                 rx_done <= 1;
74                 state    <= IDLE;
75             end
76         else begin
77             count <= count + 1;
78         end
79     end
80 end
81
82     default: state <= IDLE;
83 endcase
84 end
85 end
86 endmodule
```

Schematics:

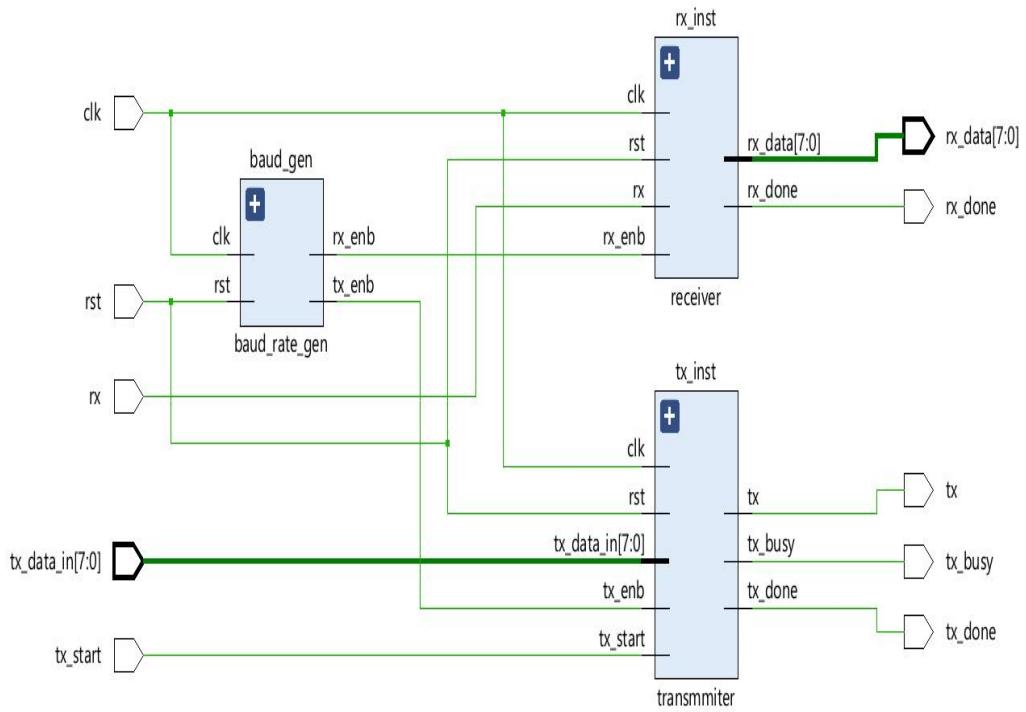


Top Module Code:

```

1  module uart_top
2    (
3      input wire clk,
4      input wire rst,
5
6      // RX Interface
7      input wire rx,
8      output wire [7:0] rx_data,
9      output wire rx_done,
10
11     // TX Interface
12     input wire tx_start,
13     input wire [7:0] tx_data_in,
14     output wire tx,
15     output wire tx_busy,
16     output wire tx_done
17   );
18
19   // Wires to connect Baud Generator
20   wire tx_enable;
21   wire rx_enable;
22
23   baud_rate_gen baud_gen (
24     .clk(clk),
25     .rst(rst),
26     .tx_enb(tx_enable),
27     .rx_enb(rx_enable)
28   );
29
30   transmitter tx_inst (
31     .clk(clk),
32     .rst(rst),
33     .tx_start(tx_start),
34     .tx_data_in(tx_data_in),
35     .tx_enb(tx_enable),
36     .tx(tx),
37     .tx_busy(tx_busy),
38     .tx_done(tx_done)
39   );
40
41   receiver rx_inst (
42     .clk(clk),
43     .rst(rst),
44     .rx(rx),
45     .rx_enb(rx_enable),
46     .rx_done(rx_done),
47     .rx_data(rx_data)
48   );
49
50 endmodule

```

Schematic:

TESTBENCH

```
1 `timescale 1ns / 1ps
2
3 module uart_test;
4
5     reg clk;
6     reg rst;
7     reg tx_start;
8     reg [7:0] tx_data_in;
9
10    wire tx_wire;
11    wire [7:0] rx_data;
12    wire rx_done;
13    wire tx_busy;
14    wire tx_done;
15
16    // Instantiate Top Module
17    uart_top uut (
18        .clk(clk),
19        .rst(rst),
20
21        .rx(tx_wire), // Input 'rx' gets signal from Output 'tx'
22        .tx(tx_wire),
23
24        .rx_data(rx_data),
25        .rx_done(rx_done),
26        .tx_start(tx_start),
27        .tx_data_in(tx_data_in),
28        .tx_busy(tx_busy),
29        .tx_done(tx_done)
30    );

```

```

31
32 // Clock Generation
33 initial begin
34   clk = 0;
35   forever #20 clk = ~clk;
36 end
37
38 initial begin
39   $dumpfile("uart_test.vcd");
40   $dumpvars(0, uart_top);
41
42   rst = 1;
43   tx_start = 0;
44   tx_data_in = 0;
45
46   #200;
47   rst = 0;
48   #200;
49
50   // Send 0xAB (10101011)
51   @(posedge clk);
52   tx_data_in = 8'hAB;
53   tx_start = 1;
54   @(posedge clk);
55   tx_start = 0;
56
57   // Wait for RX Done
58   @(posedge rx_done);
59
60   #100;
61
62 if (rx_data == 8'hAB)
63   $display("PASSED: Sent 0xAB, Received 0x%h", rx_data);
64 else
65   $display("FAILED: Sent 0xAB, Received 0x%h", rx_data);
66
67 #1000;
68 $finish;
69 end
70
71 endmodule

```

Simulation:

