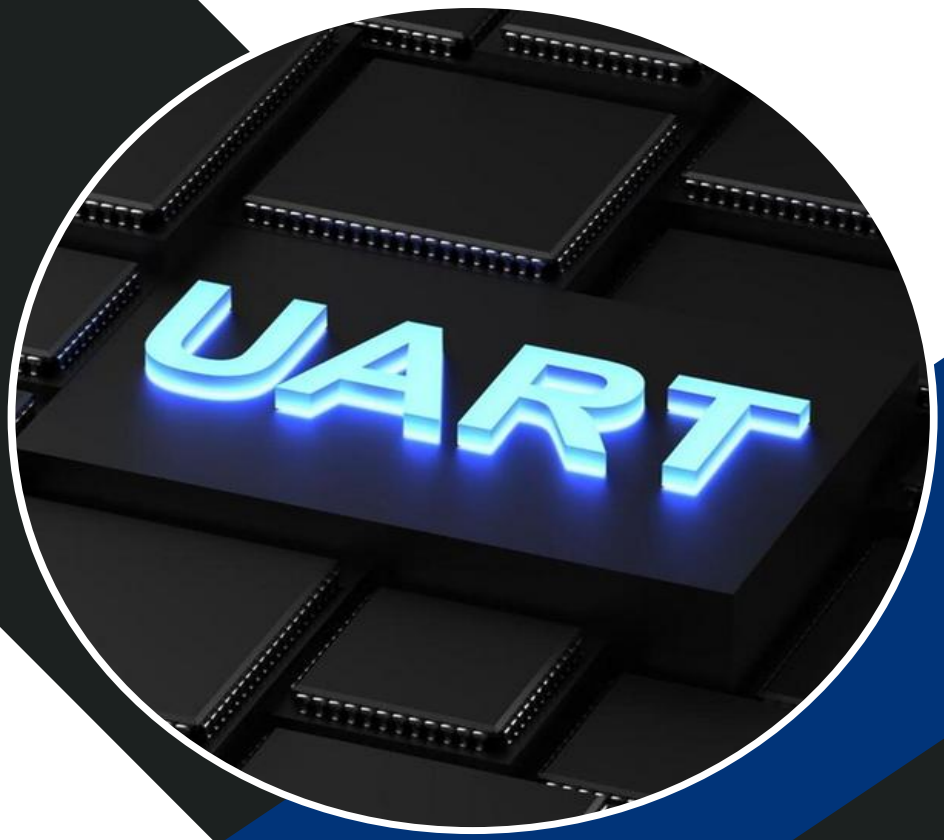



# UART Report

**PREPARED BY :**

Khaled Gamal

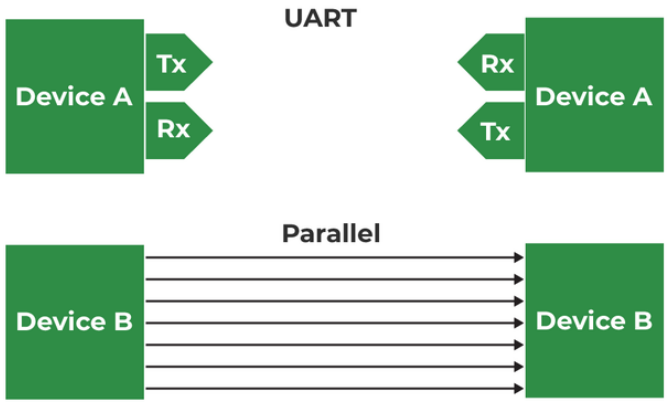
# Table of Contents

# Abstract:

UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols. This report shows how to use UART as a hardware communication protocol by following the standard procedure. When properly configured, 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. Depending on the application and system requirements, serial communications needs less circuitry and wires, which reduces the cost of implementation.

---

# Introduction:




Serial communication is one of the most fundamental techniques for data exchange between digital systems. The Universal Asynchronous Receiver and Transmitter (UART) protocol is widely used because of its simplicity and low hardware cost. Unlike synchronous protocols, UART does not require a shared clock. Instead, both devices agree on parameters such as baud rate, number of data bits, parity, and stop bits.

Although modern communication standards like USB and Ethernet are faster, UART is still used in debugging, embedded systems, FPGA development, and legacy device communication.

This project presents a UART implementation in Verilog with:

- A baud rate generator based on oversampling.

- A transmitter and receiver FSM.
- FIFO buffers for handling data rate mismatches.

---

# Background & Theory:

## 2.1 UART Overview

UART (Universal Asynchronous Receiver and Transmitter) is a form of serial communication protocol that transmits data **bit by bit** over a single line without requiring a shared clock.

## 2.2 Serial Communication Concept

If we have 8-bit data, it can be sent in parallel using 8 wires. However, for larger data widths (e.g., 64 bits), using 64 wires is costly. Instead, the data is serialized and sent **one bit at a time** using fewer wires.

Steps of transmission:

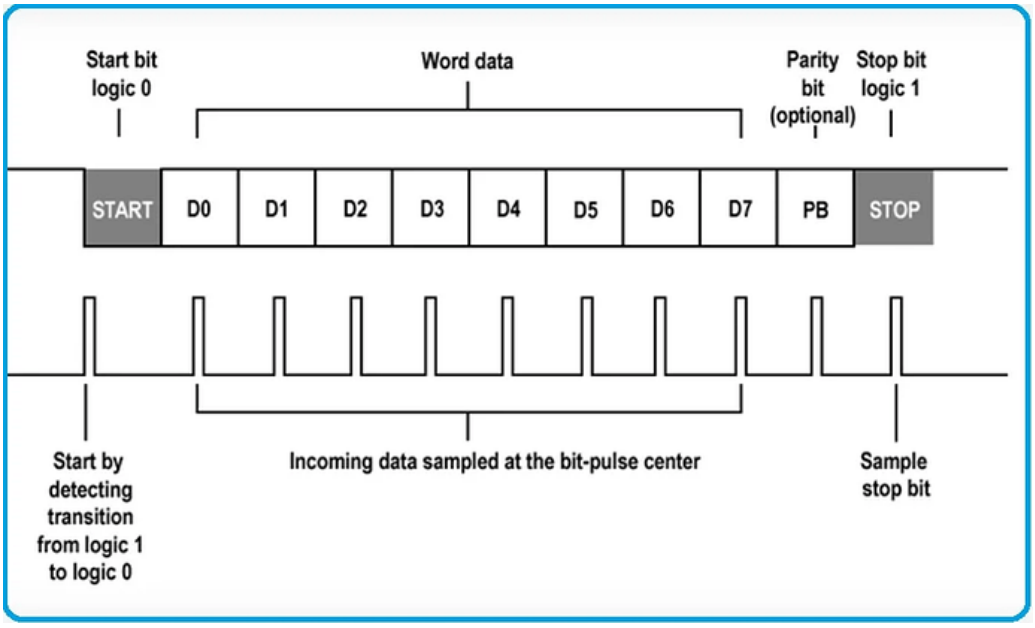
1. Data enters the transmitter in parallel.
2. A **shift register** serializes and transmits the data.
3. The receiver shifts in the bits serially.
4. The receiver outputs the reconstructed data in parallel.

This method is cost-effective and scalable, though slower than parallel transmission.

## 2.3 UART Frame Structure

- **Idle state:** Line held HIGH.
- **Start bit:** A LOW pulse marking the beginning of transmission.
- **Data bits:** Typically 8 bits (e.g., 11001101).
- **Optional parity bit:** For error detection.
- **Stop bit(s):** HIGH level for 1, 1.5, or 2 bit durations.

The transmitter and receiver must agree on the frame format.




## 2.4 Baud Rate

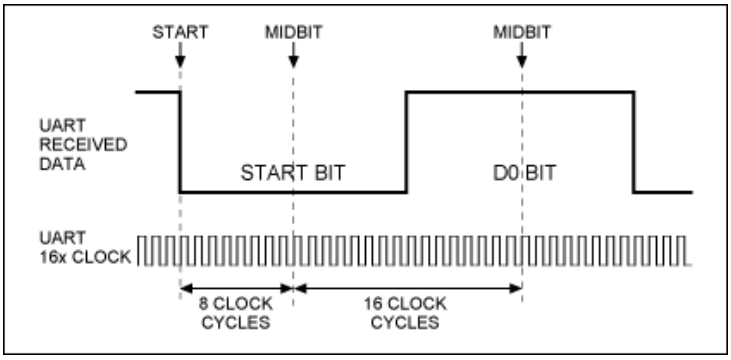
Since UART does not share a clock, both devices must agree on a transmission rate:

$$Baud\ Rate = bits\ per\ second\ (bps)$$

A common standard is **9600 bps**.

## 2.5 Oversampling Technique

Because there is no clock line, the receiver must sample the incoming data at a higher frequency. In this design, **16× oversampling** is used:



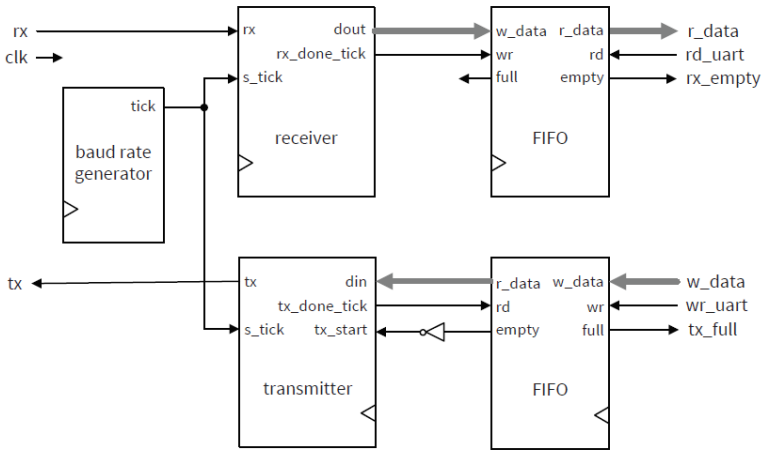

- The baud rate generator produces ticks at **16 × baud rate**.
- The receiver waits for the **falling edge** of the start bit.
- It samples each data bit at the **midpoint** (8th tick) to avoid errors from jitter or noise.
- For the stop bit(s), the receiver counts the appropriate ticks (16 for 1 stop bit, 24 for 1.5, 32 for 2).

This ensures accurate data recovery even with small baud rate mismatches.

---

# System Design Overview:

The UART system is composed of four main modules: Baud Rate Generator, Receiver, Transmitter, and FIFO Buffers. These components interact to ensure reliable asynchronous data transmission and reception. The block diagram below illustrates the overall architecture.



## Baud Rate Generator

The baud rate generator is implemented as a counter-based timer. It divides the high-speed system clock into smaller periods corresponding to the baud rate ticks. Since the design employs 16× oversampling, the baud generator produces ticks at a frequency equal to $16 \times baud\ rate$. These ticks (s_tick) are distributed to both the receiver and transmitter FSMs for timing control.

## Transmitter and Receiver

Both transmitter and receiver are designed as finite state machines (FSMs).

- Receiver FSM samples the incoming serial data at the correct instants, reconstructs the byte, and asserts (rx_done_tick) once a complete frame is received.
- Transmitter FSM shifts out the start bit, data bits, and stop bit(s) in sequence. It asserts (tx_done_tick) after completing a frame.

## FIFO Buffers

To handle data rate mismatches, the design uses two FIFOs:

- RX FIFO stores received bytes before the system consumes them.
- TX FIFO buffers outgoing bytes before transmission.

---

# Data Flow

## Receiving Data

1. A byte (e.g., 11001101) is transmitted.
2. The baud generator begins ticking at $16 \times baud\ rate$.
3. The receiver FSM samples the serial input bit by bit and shifts them into a register.
4. Once the entire byte is received, it appears at (rx_dout).
5. The signal (rx_done_tick) asserts HIGH, enabling a write into the RX FIFO.
6. With a FWFT (First Word Fall Through) FIFO, the received byte is immediately available at (r_data).

The FIFO ensures reliable buffering:

- If data arrives too fast, multiple bytes can be queued.
- (rx_empty )indicates no data is available.
- If (rd_uart) is asserted when the FIFO is empty, the output is invalid but flagged by (rx_empty) = HIGH.

## Transmitting Data

1. System writes data into TX FIFO using (w_data) and (wr_uart).
2. When FIFO is not empty, (tx_start) is asserted (via the inverted empty flag).
3. The transmitter FSM loads (tx_din) from FIFO output and starts transmission.
4. Data is shifted out bit by bit at each tick from the baud generator.
5. When a frame completes, (tx_done_tick) asserts and triggers FIFO read (rd_en).
6. If FIFO becomes empty, (tx_start) is deasserted, and the transmitter returns to idle.

# Detailed Module Design

## 1- Baud Rate Generator:

The baud generator produces sampling ticks required for oversampling.

$$\text{Tick Period} = \frac{1}{16 \cdot b}$$

Given system clock frequency $f$, the timer counts up to a final value:

$$(FINAL\_VALUE + 1) \cdot \frac{1}{f} = \frac{1}{16 \cdot b}$$

$$FINAL\_VALUE = \frac{f}{16 \cdot b} - 1$$

**Example**

$For f = 100\,MHz, b = 9600\,bps$:

$$FINAL\_VALUE = \frac{100 \times 10^6}{16 \times 9600} - 1 \approx 650$$

Thus, the counter runs from 0 to 650, producing ticks at

$$16 \times 9600 = 153.6\,\text{kHz}.$$

Timer Design code:

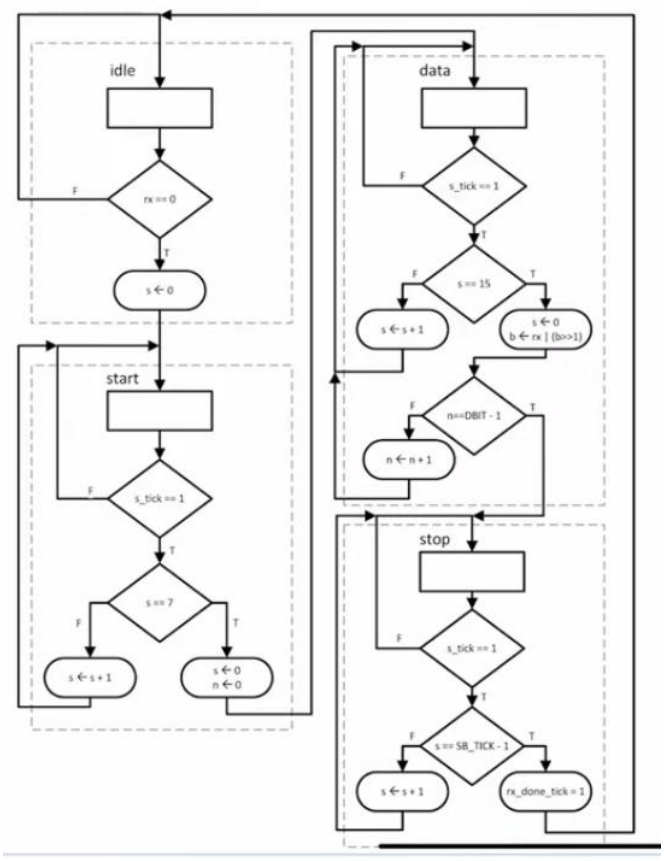
```
module timer_input
    #(parameter N = 4)
(
    input clk , rstn , enable,
    input [N-1 : 0] FINAL_VALUE,
    output done
);

reg [N-1 : 0] Q_reg , Q_next;

always @(posedge clk or negedge rstn) begin
    if(~rstn)
        Q_reg <= 0;
```

```
    else if (enable)         //only works with enable
        Q_reg <= Q_next;
    else
        Q_reg <= Q_reg;     //stay at the same value if enable is 0 and rstn is 1 and this
line isn't necessary
end

//Next state logic
assign done = Q_reg == FINAL_VALUE;    //done signal will be 1 when the counter reaches the
final_value input

always @(*) begin
    Q_next = done ? 'b0 : Q_reg + 1;            //if done signal is 1 the counter will reset
to zero but if it's still 0 the counter will continue counting up
end

endmodule //timer_input
```

---

## 2- Receiver FSM:

The receiver is designed using an **ASMD chart** with four states:




1. **Idle** – Line is HIGH. Waits for start bit (rx = 0).
2. **Start** – Counts 7 ticks to sample at the middle of start bit.
3. **Data** – Shifts in each bit. For every 16 ticks, one bit is sampled. Continues until all data bits (DBIT) are received.
4. **Stop** – Waits for the required stop ticks (SB_TICK). Asserts (rx_done_tick) and returns to idle.

This FSM ensures stable sampling by always reading at the middle of each bit period.

[Design Code:]

```
module uart_rx
    #(parameter DBIT = 8,           //Data bits
                SB_TICK = 16        //Stop bit ticks
    )
(
    input clk , rstn ,
    input rx , s_tick,
    output [DBIT-1:0] rx_dout ,
    output reg rx_done_tick
);

//----------------- States encoding -----------------------------//
localparam  idle = 0 , start = 1 , data = 2 , stop = 3;

//------------ Registers for the current and next state----------//
reg [1:0] state_reg , state_next;
reg [3:0] s_reg , s_next ;                  // keep track of the baud rate ticks (16 total)
```

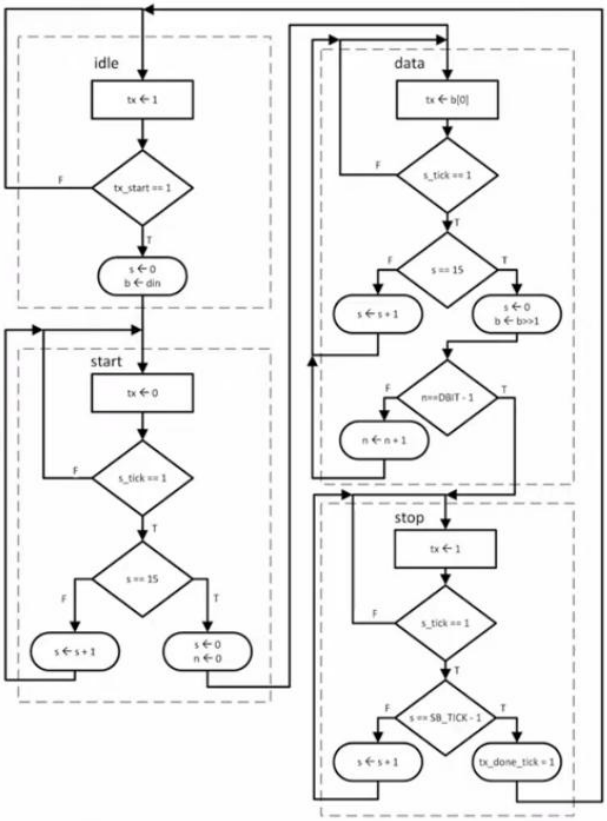
```
reg [$clog2(DBIT) -1 :0] n_reg , n_next;    // keep track of the number of data bits received
reg [DBIT -1 : 0] b_reg , b_next;           // stores the received data bits


/*============================================================
                        Sequential logic
============================================================ */
always @(posedge clk or negedge rstn) begin
    if(~rstn)begin
        state_reg <= idle;
        s_reg <= 0;
        n_reg <= 0;
        b_reg <= 0;
    end
    else begin
        state_reg <= state_next;
        s_reg <= s_next;
        n_reg <= n_next;
        b_reg <= b_next;
    end
end


/*============================================================
                        Next state logic
============================================================*/
always @( *) begin
    // Default values
    state_next = state_reg;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;
    rx_done_tick = 1'b0;

    case (state_reg)

//----------------------Idle State-----------------------------
        idle :begin
            if (~rx) begin
                s_next = 0;
                state_next = start;
            end                         // else it'll stay at the same state
        end


//----------------------Start State----------------------------
        start : begin
            if (s_tick)begin
                if(s_reg == 7)begin
                    s_next = 0 ;
```

```
                    n_next = 0;
                    state_next = data;
                end
                else
                    s_next = s_reg +1 ;
            end
        end

//-----------------------Data State-----------------------------
        data : begin
            if(s_tick)begin
                if (s_reg == 15 )begin
                    s_next = 0;
                    b_next = {rx , b_reg[DBIT-1 : 1]};       //Right shift
                    if (n_reg == (DBIT -1 ))
                        state_next = stop;
                    else
                        n_next = n_reg +1;
                end
                else
                    s_next = s_reg +1;
            end
        end

//------------------------Stop State-----------------------------
        stop : begin
            if (s_tick)begin
                if(s_reg == (SB_TICK -1))begin
                    rx_done_tick = 1'b1;
                    state_next = idle;
                end
                else
                    s_next = s_reg +1;
            end
        end

        default: state_next = idle;
    endcase
end

/*=============================================================
                    Output Logic
=============================================================*/
assign rx_dout = b_reg;

endmodule //uart_rx
```

## 3- Transmitter FSM:

The transmitter FSM is symmetric to the receiver but focuses on shifting data out:

1. **Idle** – TX line held HIGH. Waits for (tx_start).
2. **Start** – TX line driven LOW for 16 ticks.
3. **Data** – Shifts out bits from (tx_din) LSB first, one every 16 ticks.
4. **Stop** – TX line held HIGH for SB_TICK ticks. Asserts (tx_done_tick), then returns to idle.




Design code:

```
module uart_tx
    #(parameter DBIT = 8,           //Data bits
                SB_TICK = 16        //Stop bit ticks
    )
(
    input clk , rstn ,
    input tx_start , s_tick,
    input [DBIT-1:0] tx_din ,
    output tx,
    output reg tx_done_tick
);

//----------------- States encoding -----------------------------//
localparam  idle = 0 , start = 1 , data = 2 , stop = 3;

//------------ Registers for the current and next state----------//
reg [1:0] state_reg , state_next;
reg [3:0] s_reg , s_next ;                  // keep track of the baud rate ticks (16 total)
reg [$clog2(DBIT) -1 :0] n_reg , n_next;    // keep track of the number of data bits
transmitted
reg [DBIT -1 : 0] b_reg , b_next;           // shift the transmitted data bits
reg tx_reg , tx_next;                      // track the transmitted bits
```

```
/*=============================================================
                        Sequential logic
============================================================= */
always @(posedge clk or negedge rstn) begin
    if(~rstn)begin
        state_reg <= idle;
        s_reg <= 0;
        n_reg <= 0;
        b_reg <= 0;
        tx_reg <= 1'b1;           //tx should be kept HIGH at the Idle state
    end
    else begin
        state_reg <= state_next;
        s_reg <= s_next;
        n_reg <= n_next;
        b_reg <= b_next;
        tx_reg <= tx_next;
    end
end


/*=============================================================
                        Next state logic
=============================================================*/
always @( *) begin
    // Default values
    state_next = state_reg;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;
    tx_done_tick = 1'b0;

    case (state_reg)

//----------------------Idle State-----------------------------
        idle :begin
            tx_next = 1'b1;
            if (tx_start) begin
                s_next = 0;
                b_next = tx_din;
                state_next = start;
            end                         // else it'll stay at the same state
        end


//----------------------Start State----------------------------
        start : begin
            tx_next = 1'b0;
            if (s_tick)begin
                if(s_reg == 15)begin
```

```
                    s_next = 0 ;
                    n_next = 0;
                    state_next = data;
                end
                else
                    s_next = s_reg +1 ;
            end
        end

//-----------------------Data State-----------------------------
        data : begin
            tx_next = b_reg[0];
            if(s_tick)begin
                if (s_reg == 15 )begin
                    s_next = 0;
                    b_next = {1'b0 , b_reg[DBIT-1 : 1]};       //Right shift
                    if (n_reg == (DBIT -1 ))
                        state_next = stop;
                    else
                        n_next = n_reg +1;
                end
                else
                    s_next = s_reg +1;
            end
        end

//-----------------------Stop State-----------------------------
        stop : begin
            tx_next = 1'b1;
            if (s_tick)begin
                if(s_reg == (SB_TICK -1))begin
                    tx_done_tick = 1'b1;
                    state_next = idle;
                end
                else
                    s_next = s_reg +1;
            end
        end

        default: state_next = idle;
    endcase
end


/*=============================================================
                    Output Logic
=============================================================*/
assign tx = tx_reg;
endmodule //uart_tx
```

## 4- FIFO Buffers:

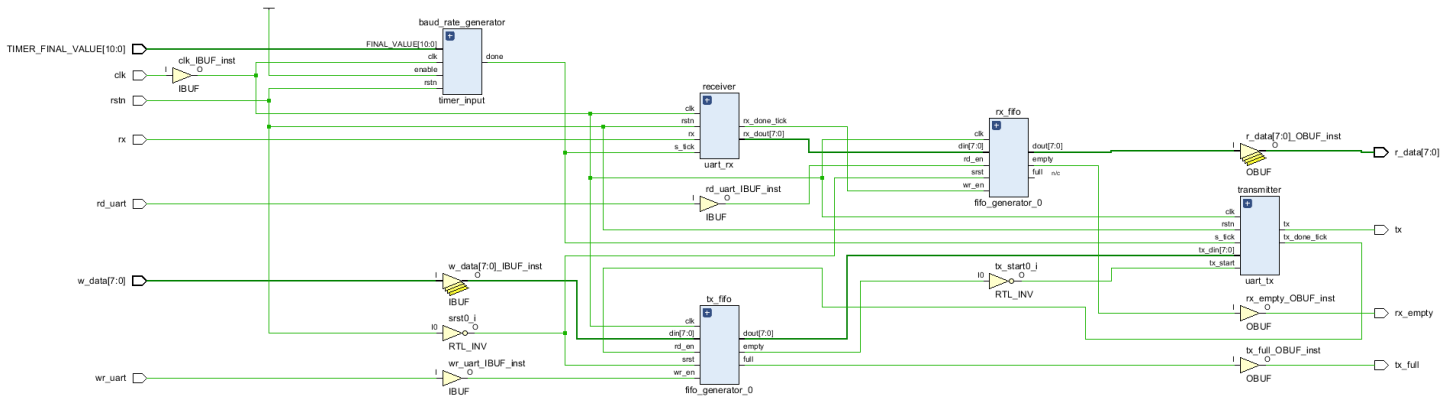
- **RX FIFO**: Prevents overflow when incoming data rate exceeds processing rate.
- **TX FIFO**: Decouples system write speed from UART transmission speed.
- **Flags**:
  - empty = no valid data.
  - full = no more data can be written.
- **FWFT**: Makes data immediately available at output without an extra read cycle.

**Note:** I used Vivado's FIFO IP source.

---

## 5- UART top module:

Elaborated Design:



## Design code:

```
module UART
    #(parameter DBIT = 8,           // no.of Data bits
                SB_TICK = 16        // no.of Stop bit ticks
    )
(
    input clk ,rstn,

    //receiver ports
    output [DBIT-1 : 0] r_data,
    input rd_uart,
    output rx_empty,
    input rx,

    //Transmitter ports
    input [DBIT-1 : 0] w_data,
    input wr_uart,
    output tx_full,
    output tx,

    //baud rate generator
    input [10:0] TIMER_FINAL_VALUE
);

/*=============================================================================================
                                        Instantiations
=============================================================================================*/

//------------------------------Timer as baud rate generator-----------------------------

wire tick;       //internal signal of the timer output

timer_input #(.N(11)) baud_rate_generator (
    .clk(clk),
    .rstn(rstn),
    .enable(1'b1),
    .FINAL_VALUE(TIMER_FINAL_VALUE),
    .done(tick)
    );


//-------------------------------------Receiver-----------------------------------------

wire [DBIT-1:0] rx_dout ;     //output data of the receiver
wire rx_done_tick;            //done counting signal of the receiver

uart_rx #( .DBIT(DBIT), .SB_TICK(SB_TICK)) receiver (
    .clk(clk),
```

```
    .rstn(rstn),
    .rx(rx),
    .s_tick(tick),
    .rx_dout(rx_dout),
    .rx_done_tick(rx_done_tick)
);


fifo_generator_0 rx_fifo (
  .clk(clk),      // input wire clk
  .srst(~rstn),    // input wire srst
  .din(rx_dout),      // input wire [7 : 0] din
  .wr_en(rx_done_tick),  // input wire wr_en
  .rd_en(rd_uart),  // input wire rd_en
  .dout(r_data),    // output wire [7 : 0] dout
  .full(),    // output wire full                    //assuming the receiving fifo isn't full
  .empty(rx_empty)  // output wire empty
);


//--------------------------------------Transmitter-----------------------------------------

wire tx_fifo_empty , tx_done_tick;  //internal signals between tx FIFO and transmitter
wire [DBIT-1 : 0] tx_din;          //output signal of tx FIFO

uart_tx #( .DBIT(DBIT), .SB_TICK(SB_TICK)) transmitter (
    .clk(clk),
    .rstn(rstn),
    .tx_start(~tx_fifo_empty),
    .s_tick(tick),
    .tx_din(tx_din),
    .tx(tx),
    .tx_done_tick(tx_done_tick)
);

fifo_generator_0 tx_fifo (
  .clk(clk),      // input wire clk
  .srst(~rstn),    // input wire srst
  .din(w_data),      // input wire [7 : 0] din
  .wr_en(wr_uart),  // input wire wr_en
  .rd_en(tx_done_tick),  // input wire rd_en
  .dout(tx_din),    // output wire [7 : 0] dout
  .full(tx_full),    // output wire full                    //assuming the receiving fifo
isn't full
  .empty(tx_fifo_empty)  // output wire empty
);

endmodule //UART
```

# Simulation & Verification:

## Loopback Test

A simple verification is connecting TX → RX.

- Data written into TX FIFO should be reconstructed at RX FIFO output.
- Waveform shows start, data, and stop bits aligned with baud ticks.

## Testbench Code:

```
module UART_tb ();

parameter DBIT = 8 ;            // no.of Data bits
parameter SB_TICK = 16 ;        // no.of Stop bit ticks

reg clk ,rstn;

//receiver ports
wire  [DBIT-1 : 0] r_data;
reg rd_uart;
wire rx_empty;
wire rx;

//Transmitter ports
reg [DBIT-1 : 0] w_data;
reg wr_uart;
wire tx_full;
wire tx;

//baud rate generator
reg [10:0] TIMER_FINAL_VALUE;

assign rx = tx;

//-----------------------------Clock Generation--------------------------
initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;
end
```

```
//---------------------------------Stimulus-----------------------------
initial begin

    //initializing inputs
    rstn = 1'b0;
    rd_uart = 1'b0;
    w_data = 0;
    wr_uart = 1'b0;
    TIMER_FINAL_VALUE = 650;         //baud rate = 9600 bps


    //Testing reset signal
    repeat (3) @(negedge clk);
    rstn = 1'b1;


    //Writing data into the FIFO
    w_data = 8'hA5;
    repeat (2) @(negedge clk);
    wr_uart = 1'b1;
    @(negedge clk);
    wr_uart = 1'b0;

    // Wait for transmission to complete
    repeat (110_000) @(negedge clk);

    //Reading the transmitted data
    rd_uart = 1'b1;
    @(negedge clk);
    rd_uart = 1'b0;
    repeat (10) @(negedge clk);

    // Check received data
    if (r_data == 8'hA5)
        $display("Test Passed: Received data = %h", r_data);
    else
        $display("Test Failed: Received data = %h", r_data);

    $stop;
end


initial begin
    $monitor("Time=%0t tx=%b rx=%b wr_uart=%b rd_uart=%b r_data=%h", $time, tx, rx, wr_uart,
rd_uart, r_data);
end

//-----------------------------Instantiation-----------------------------
UART #(.DBIT(DBIT) , .SB_TICK(SB_TICK)) dut (
    .clk(clk),
    .rstn(rstn),
```

```
    .r_data(r_data),
    .rd_uart(rd_uart),
    .rx_empty(rx_empty),
    .rx(rx),
    .w_data(w_data),
    .wr_uart(wr_uart),
    .tx_full(tx_full),
    .tx(tx),
    .TIMER_FINAL_VALUE(TIMER_FINAL_VALUE)
);


endmodule //UART_tb
```

## Simulation Waveform:

*detailed waveform with all the internal signals.

# References:


- YouTube – Digital Circuit Design Using Verilog Playlist. Available at: [link]
- Arduino Forum – Understanding the Frame Format of a UART Signal. Available at: [link]
- “GeeksforGeeks” – Universal Asynchronous Receiver Transmitter (UART) Protocol. Available at: [link]
- Analog Devices – UART: A Hardware Communication Protocol. Available at: [link]
- Electronics StackExchange – UART Receiver Sampling Rate. Available at: [link]