UART Implementation In Verilog

1. Introduction

The project focuses on the design and implementation of a **Universal Asynchronous Receiver Transmitter (UART)** using Verilog HDL. UART is a widely used hardware communication protocol that enables asynchronous serial communication between two devices. Unlike parallel communication, UART transmits data bit by bit over a single wire for transmission (TX) and another for reception (RX), making it simple, low-cost, and efficient for short- to medium-distance data transfer.

The objective of this project is to design a synthesizable and functional UART module that can reliably transmit and receive serial data while meeting timing and protocol requirements. By implementing the design in Verilog, we can simulate, verify, and synthesize the module for FPGA or ASIC deployment.

Motivation:
UART is one of the most fundamental communication protocols used in embedded systems, microcontrollers, FPGAs, and industrial equipment. A hands-on implementation helps strengthen understanding of digital communication principles, HDL coding, and FPGA-based design flow.

---

2. Applications

UART is used extensively in systems where reliable, low-complexity serial communication is needed. Typical applications include:

1. **Embedded Systems** - Communication between microcontrollers, sensors, and modules.

2. **FPGA–Microcontroller Interfaces** - Sending debug or control data between FPGA designs and external processors.

3. **Industrial Automation** - Machine-to-machine (M2M) communication for control and monitoring.

4. **PC Communication Ports** - Legacy RS-232/RS-485 serial ports still rely on UART for interfacing with devices.

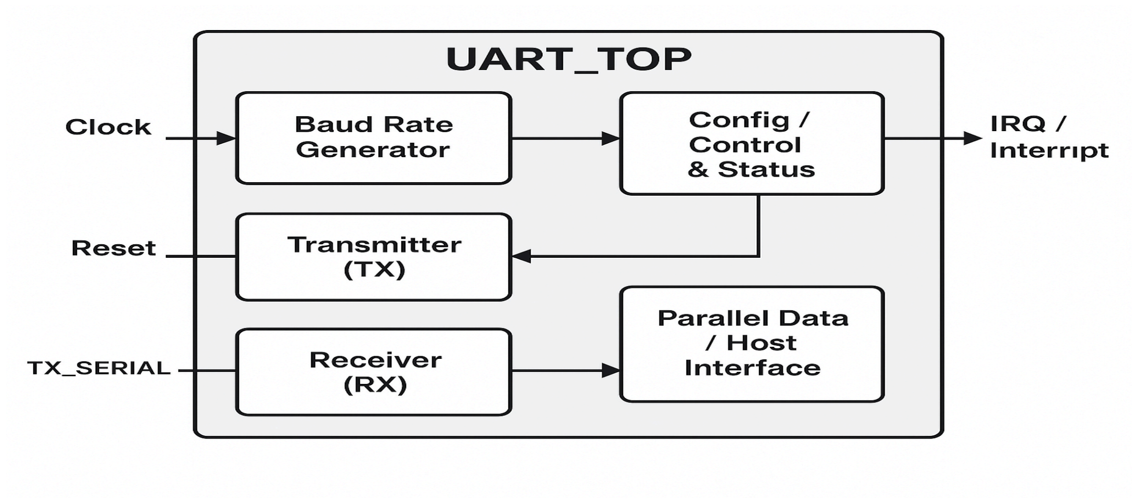
5. **IoT Devices** - Transferring configuration and sensor data between nodes and gateways.

6. **Bootloaders & Firmware Updates** - Many systems use UART as a simple channel for updating firmware.

In this project, the UART module can be integrated into larger FPGA-based designs to enable external communication with sensors, actuators, or a host computer, making it a key building block in many digital systems.

---

3. Block Diagram



4. RTL Code

Receiver -

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////
// Set Parameter CLKS_PER_BIT as follows:
// CLKS_PER_BIT = (Frequency of i_Clock)/(Frequency of UART)
// Example: 10 MHz Clock, 115200 baud UART
// (10000000)/(115200) = 87

module receiver (
   input        i_Clock,
   input        i_Rx_Serial,
   output       o_Rx_DV,
   output [7:0] o_Rx_Byte
   );

  parameter CLKS_PER_BIT   =87;
  parameter s_IDLE         = 3'b000;
  parameter s_RX_START_BIT = 3'b001;
  parameter s_RX_DATA_BITS = 3'b010;
  parameter s_RX_STOP_BIT  = 3'b011;
  parameter s_CLEANUP      = 3'b100;

  reg           r_Rx_Data_R = 1'b1;
  reg           r_Rx_Data   = 1'b1;

  reg [7:0]     r_Clock_Count = 0;
  reg [2:0]     r_Bit_Index   = 0; //8 bits total
  reg [7:0]     r_Rx_Byte     = 0;
  reg           r_Rx_DV       = 0;
  reg [2:0]     r_SM_Main     = 0;

  // Purpose: Double-register the incoming data.
  // This allows it to be used in the UART RX Clock Domain.
  // (It removes problems caused by metastability)
  always @(posedge i_Clock)
    begin
      r_Rx_Data_R <= i_Rx_Serial;
      r_Rx_Data   <= r_Rx_Data_R;
    end


  // Purpose: Control RX state machine
  always @(posedge i_Clock)
    begin

      case (r_SM_Main)
        s_IDLE :
          begin
            r_Rx_DV       <= 1'b0;
            r_Clock_Count <= 0;
            r_Bit_Index   <= 0;

            if (r_Rx_Data == 1'b0)          // Start bit detected
              r_SM_Main <= s_RX_START_BIT;
            else
              r_SM_Main <= s_IDLE;
          end

        // Check middle of start bit to make sure it's still low
        s_RX_START_BIT :
          begin
            if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
              begin
                if (r_Rx_Data == 1'b0)
                  begin
                    r_Clock_Count <= 0;  // reset counter, found the middle
                    r_SM_Main     <= s_RX_DATA_BITS;
                  end
                else
                  r_SM_Main <= s_IDLE;
              end
            else
              begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main     <= s_RX_START_BIT;
              end
          end // case: s_RX_START_BIT


        // Wait CLKS_PER_BIT-1 clock cycles to sample serial data
        s_RX_DATA_BITS :
          begin
            if (r_Clock_Count < CLKS_PER_BIT-1)
              begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main     <= s_RX_DATA_BITS;
              end
            else
              begin
                r_Clock_Count          <= 0;
                r_Rx_Byte[r_Bit_Index] <= r_Rx_Data;

                // Check if we have received all bits
                if (r_Bit_Index < 7)
                  begin
                    r_Bit_Index <= r_Bit_Index + 1;
                    r_SM_Main   <= s_RX_DATA_BITS;
                  end
                else
                  begin
                    r_Bit_Index <= 0;
                    r_SM_Main   <= s_RX_STOP_BIT;
                  end
              end
          end // case: s_RX_DATA_BITS


        // Receive Stop bit.  Stop bit = 1
        s_RX_STOP_BIT :
          begin
            // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
            if (r_Clock_Count < CLKS_PER_BIT-1)
              begin
                r_Clock_Count <= r_Clock_Count + 1;
                r_SM_Main     <= s_RX_STOP_BIT;
              end
            else
              begin
                r_Rx_DV       <= 1'b1;
                r_Clock_Count <= 0;
                r_SM_Main     <= s_CLEANUP;
              end
          end // case: s_RX_STOP_BIT


        // Stay here 1 clock
```

```
      s_CLEANUP :
        begin
          r_SM_Main <= s_IDLE;
          r_Rx_DV   <= 1'b0;
        end


      default :
        r_SM_Main <= s_IDLE;

    endcase
  end

  assign o_Rx_DV   = r_Rx_DV;
  assign o_Rx_Byte = r_Rx_Byte;

endmodule // uart_rx
```

Transmitter -

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////
// Set Parameter CLKS_PER_BIT as follows:
// CLKS_PER_BIT = (Frequency of i_Clock)/(Frequency of UART)
// Example: 10 MHz Clock, 115200 baud UART
// (10000000)/(115200) = 87

module transmitter (
   input       i_Clock,
   input       i_Tx_DV,
   input [7:0] i_Tx_Byte,
   output      o_Tx_Active,
   output    reg  o_Tx_Serial,
   output      o_Tx_Done
   );

  parameter CLKS_PER_BIT   = 2;
  parameter s_IDLE         = 3'b000;
  parameter s_TX_START_BIT = 3'b001;
  parameter s_TX_DATA_BITS = 3'b010;
  parameter s_TX_STOP_BIT  = 3'b011;
  parameter s_CLEANUP      = 3'b100;
```

```
reg [2:0]    r_SM_Main     = 0;
reg [7:0]    r_Clock_Count = 0;
reg [2:0]    r_Bit_Index   = 0;
reg [7:0]    r_Tx_Data     = 0;
reg          r_Tx_Done     = 0;
reg          r_Tx_Active   = 0;

always @(posedge i_Clock)
  begin

    case (r_SM_Main)
      s_IDLE :
        begin
          o_Tx_Serial   <= 1'b1;         // Drive Line High for Idle
          r_Tx_Done     <= 1'b0;
          r_Clock_Count <= 0;
          r_Bit_Index   <= 0;

          if (i_Tx_DV == 1'b1)
            begin
              r_Tx_Active <= 1'b1;
              r_Tx_Data   <= i_Tx_Byte;
              r_SM_Main   <= s_TX_START_BIT;
            end
          else
            r_SM_Main <= s_IDLE;
        end // case: s_IDLE


      // Send out Start Bit. Start bit = 0
      s_TX_START_BIT :
        begin
          o_Tx_Serial <= 1'b0;

          // Wait CLKS_PER_BIT-1 clock cycles for start bit to finish
          if (r_Clock_Count < CLKS_PER_BIT-1)
            begin
              r_Clock_Count <= r_Clock_Count + 1;
              r_SM_Main     <= s_TX_START_BIT;
            end
          else
            begin
              r_Clock_Count <= 0;
              r_SM_Main     <= s_TX_DATA_BITS;
            end
        end // case: s_TX_START_BIT


      // Wait CLKS_PER_BIT-1 clock cycles for data bits to finish
      s_TX_DATA_BITS :
        begin
          o_Tx_Serial <= r_Tx_Data[r_Bit_Index];

          if (r_Clock_Count < CLKS_PER_BIT-1)
            begin
              r_Clock_Count <= r_Clock_Count + 1;
              r_SM_Main     <= s_TX_DATA_BITS;
            end
          else
            begin
              r_Clock_Count <= 0;

              // Check if we have sent out all bits
              if (r_Bit_Index < 7)
                begin
                  r_Bit_Index <= r_Bit_Index + 1;
                  r_SM_Main   <= s_TX_DATA_BITS;
                end
              else
                begin
                  r_Bit_Index <= 0;
                  r_SM_Main   <= s_TX_STOP_BIT;
                end
            end
        end // case: s_TX_DATA_BITS


      // Send out Stop bit.  Stop bit = 1
      s_TX_STOP_BIT :
        begin
          o_Tx_Serial <= 1'b1;

          // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
          if (r_Clock_Count < CLKS_PER_BIT-1)
            begin
              r_Clock_Count <= r_Clock_Count + 1;
              r_SM_Main     <= s_TX_STOP_BIT;
            end
```

```
          else
            begin
              r_Tx_Done     <= 1'b1;
              r_Clock_Count <= 0;
              r_SM_Main     <= s_CLEANUP;
              r_Tx_Active   <= 1'b0;
            end
        end // case: s_Tx_STOP_BIT


      // Stay here 1 clock
      s_CLEANUP :
        begin
          r_Tx_Done <= 1'b1;
          r_SM_Main <= s_IDLE;
        end


      default :
        r_SM_Main <= s_IDLE;

    endcase
  end

  assign o_Tx_Active = r_Tx_Active;
  assign o_Tx_Done   = r_Tx_Done;

endmodule
```

---

5. Testbench

```
module tb_uart;

  // Declare signals to connect to the UART module
  reg reset;
  reg txclk;
  reg ld_tx_data;
  reg [7:0] tx_data;
  reg tx_enable;
  wire tx_out;
wire tx_empty;
reg rxclk;
reg uld_rx_data;
wire [7:0] rx_data;
reg rx_enable;
reg rx_in;
wire rx_empty;

// Instantiate the UART module
uart uut (
    .reset(reset),
    .txclk(txclk),
    .ld_tx_data(ld_tx_data),
    .tx_data(tx_data),
    .tx_enable(tx_enable),
    .tx_out(tx_out),
    .tx_empty(tx_empty),
    .rxclk(rxclk),
    .uld_rx_data(uld_rx_data),
    .rx_data(rx_data),
    .rx_enable(rx_enable),
    .rx_in(rx_in),
    .rx_empty(rx_empty)
);

// Initialize signals
initial begin
    reset = 1; // Reset active-high
    txclk = 0;
    ld_tx_data = 0;
    tx_data = 8'h00;
    tx_enable = 0;
    rxclk = 0;
    uld_rx_data = 0;
    rx_enable = 0;
    rx_in = 0;

    // Apply reset
    #10 reset = 0;

    // Test Case 1: Transmit and receive a single character
    reset = 1;
    #10 reset = 0;
    tx_enable = 1;
```

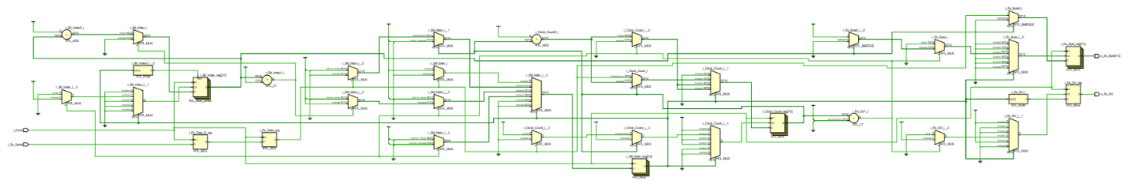
```
ld_tx_data = 1;
tx_data = 8'h41; // ASCII 'A'
#10 ld_tx_data = 0;
#100; // Wait for data reception
assert(rx_data == 8'h41) else $display("Test Case 1 Failed!");

// Test Case 2: Transmit and receive multiple characters
reset = 1;
#10 reset = 0;
tx_enable = 1;
ld_tx_data = 1;
tx_data = 8'h48; // ASCII 'H'
#10 ld_tx_data = 0;
#10 tx_data = 8'h65; // ASCII 'e'
#10 tx_data = 8'h6C; // ASCII 'l'
#10 tx_data = 8'h6C; // ASCII 'l'
#10 tx_data = 8'h6F; // ASCII 'o'
#10 tx_enable = 0;
#200; // Wait for data reception
assert(rx_data == 8'h48) else $display("Test Case 2 Failed!");
#10 assert(rx_data == 8'h65) else $display("Test Case 2 Failed!");
#10 assert(rx_data == 8'h6C) else $display("Test Case 2 Failed!");
#10 assert(rx_data == 8'h6C) else $display("Test Case 2 Failed!");
#10 assert(rx_data == 8'h6F) else $display("Test Case 2 Failed!");

// Test Case 3: Test frame error
reset = 1;
#10 reset = 0;
tx_enable = 1;
ld_tx_data = 1;
tx_data = 8'h55; // Arbitrary data
#10 ld_tx_data = 0;
rx_in = 1; // Inject a framing error
#100; // Wait for the error to propagate
rx_in = 0;
#100; // Wait for data reception
assert(rx_frame_err == 1) else $display("Test Case 3 Failed!");

// Test Case 4: Transmit and receive with frame error
reset = 1;
#10 reset = 0;
tx_enable = 1;
ld_tx_data = 1;
tx_data = 8'h41; // ASCII 'A'
```

```
        #10 ld_tx_data = 0;
        #10 tx_data = 8'h42; // ASCII 'B' (frame error)
        #100; // Wait for data reception
        assert(rx_data == 8'h41) else $display("Test Case 4 Failed!");

        // Test Case 5: Transmit and receive with overrun error
        reset = 1;
        #10 reset = 0;
        tx_enable = 1;
        ld_tx_data = 1;
        tx_data = 8'h41; // ASCII 'A'
        #10 ld_tx_data = 0;
        #10 tx_data = 8'h42; // ASCII 'B'
        #10 tx_data = 8'h43; // ASCII 'C'
        #10 tx_data = 8'h44; // ASCII 'D'
        #10 tx_data = 8'h45; // ASCII 'E'
        #10 tx_data = 8'h46; // ASCII 'F' (overrun error)
        #200; // Wait for data reception
        assert(rx_over_run == 1) else $display("Test Case 5 Failed!");

        // End simulation
        $finish;
    end

    // Clock generation
    always #5 txclk = ~txclk;
    always #7 rxclk = ~rxclk;

endmodule
```

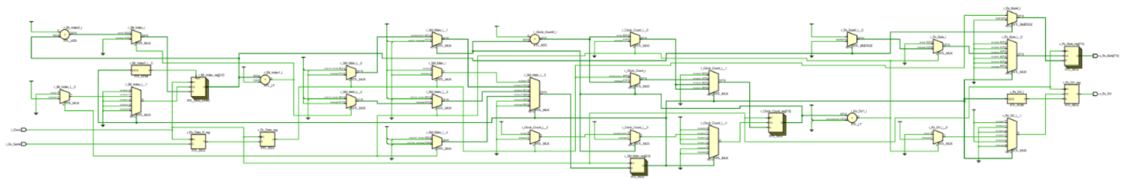
---

6. RTL Schematic

Rx Schematic



Tx Schematic


i_Clock
i_Rx_Serial
o_Rx_Byte[7:0]
o_Rx_DV

Contributions

Shubham - Overall system architecture, top-level `UART_TOP` integration, authored the RX module, coordinated team tasks, and prepared the project intro and block diagram.

Aman - Designed and implemented the Transmitter (TX) in Verilog (shift register, start/stop/parity handling, `tx_busy` logic) and wrote TX-focused test vectors.

Suyash - Implemented the Baud Rate Generator and RX sampling/timing logic; wrote the RX FSM and framing/parity/error detection code and helped debug timing issues.

Nikilesh - Developed the functional testbench (end-to-end verification), captured simulation waveforms, ran synthesis on Vivado, and produced the synthesis diagrams and final report assets.