National Telecommunication Institute

Custom UART(Universal Asynchronous Receiver Transmitte)

By: Ahmed Ebrahim Ahmed Ebrahim

Supervisors: Eng. Mohamed Salah & Eng. Mohamed Tareq

Project overview:

UART (Universal Asynchronous Receiver-Transmitter) is a widely used protocol for device-to-device serial communication. It enables reliable data transfer with minimal circuitry, typically requiring only two lines (TX and RX) for communication. Its simplicity and cost-effectiveness make it a standard interface in digital systems, embedded processors, and FPGA designs.

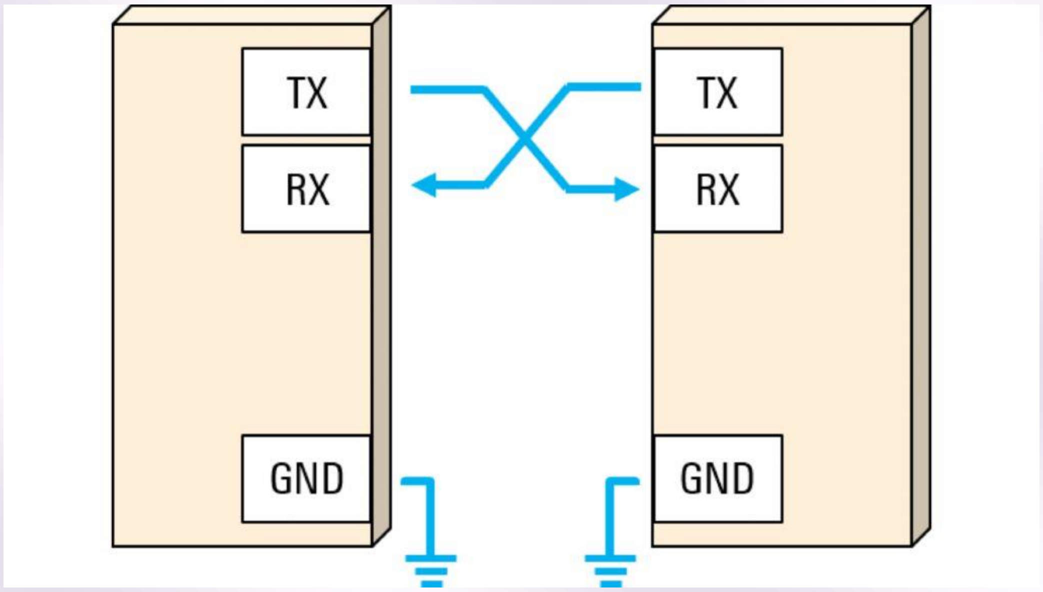


Figure 1(UART communication)

This project presents the design and implementation of a custom UART system in Verilog HDL, supporting both transmission (TX) and reception (RX). The design follows the standard 8 data bits, no parity, 1 stop bit (8N1) frame format, operating at a baud rate of 9600 bps with a 100 MHz system clock. The architecture includes key components such as a Baud Rate Generator**,** Edge Detector, Frame Generator, FSM Controllers, Bit Selector, MUX, and Serial-to-Parallel Shift Register.

The UART transmitter encapsulates data with start and stop bits and serially outputs each frame using a counter-driven MUX. The receiver detects the start bit via an edge detector, aligns sampling with the baud counter, and reconstructs the parallel data through a shift register under FSM control. Error detection mechanisms validate stop bits and flag invalid frames.

The system was verified using self-checking testbenches in Questa Sim, which applied diverse input patterns and confirmed correct operation through waveform analysis. Results validated accurate frame transmission, reception, and error detection.

This project demonstrates a modular, reusable UART design, showcasing practical skills in RTL design, FSM implementation, simulation, and verification. It provides a solid foundation for

FPGA/ASIC development and can be extended with features such as configurable baud rates, parity support, and FIFO buffering.

System and specification:

1. Protocol Requirements

- **Frame format:** 1 start bit, 8 data bits, no parity, 1 stop bit (8N1).
- **Baud rate:** 9600 bps.
- **System clock:** 100 MHz.
- **Communication type:** Full-duplex (separate TX and RX modules).

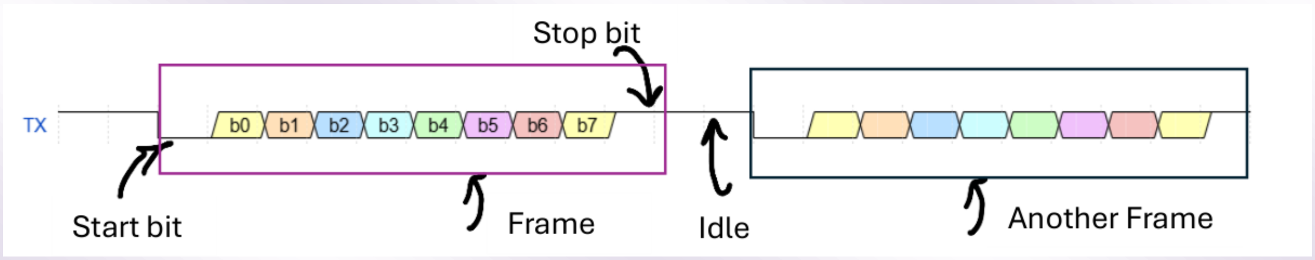


Figure 2(UART Frame Format)

2. Functional Requirements

- **Transmitter (UART_TX):**
  - Accepts 8-bit parallel input.
  - Generates start and stop bits.
  - Outputs serial data stream.
  - Provides status signals (`busy`, `done`).
- **Receiver (UART_RX):**
  - Detects start bit using edge detector.
  - Samples incoming data at correct baud intervals.
  - Converts serial data to 8-bit parallel output.
  - Validates stop bit; outputs `err` if invalid.
  - Provides status signals (`busy`, `done`).

3. Design Constraints

- Operates under a **100 MHz system clock**.
- Fixed **baud rate of 9600 bps** (can be extended later).
- Reset support: asynchronous reset (`arst_n`) and synchronous reset (`rst`).

4. I/O Port Definition (Top-Level Modules)

UART_TX

Signal	Direction	Width	Description
Clk	Input	1	System clock (100 MHz)
rst	Input	1	Synchronous reset (active high)
arst_n	Input	1	Asynchronous reset (active low)
tx_en	Input	1	Enable signal for transmission
data	Input	8	Parallel data input
tx	Output	1	Serial data output
busy	Output	1	High when transmission is ongoing
Done	Output	1	High when frame transmission ends

UART_RX

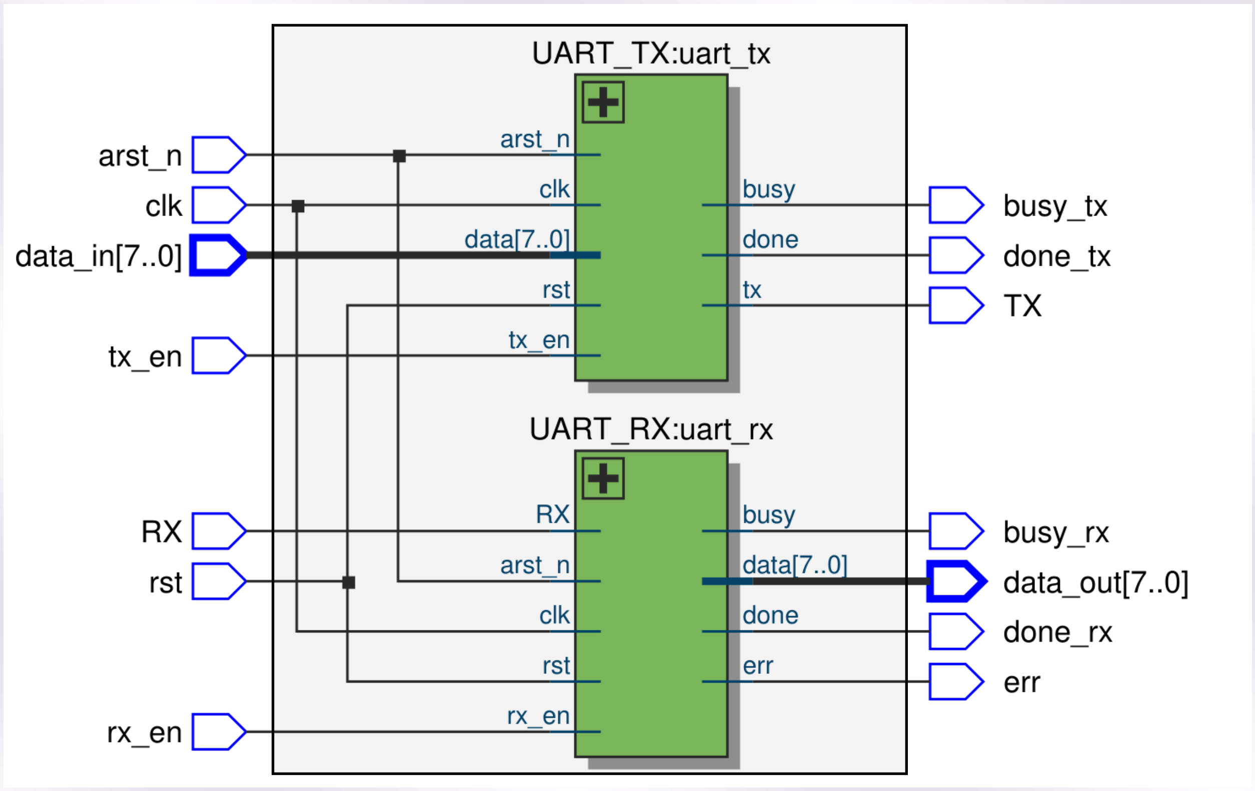
Signal	Direction	Width	Description
Clk	Input	1	System clock (100 MHz)
rst	Input	1	Synchronous reset (active high)
arst_n	Input	1	Asynchronous reset (active low)
rx_en	Input	1	Enable signal for reception

rx	intput	1	Serial data intput
data	Output	8	Parallel data output
busy	Output	1	High when reception is ongoing
Done	Output	1	High when frame reception ends
err	Output	1	High if stop-bit error detected

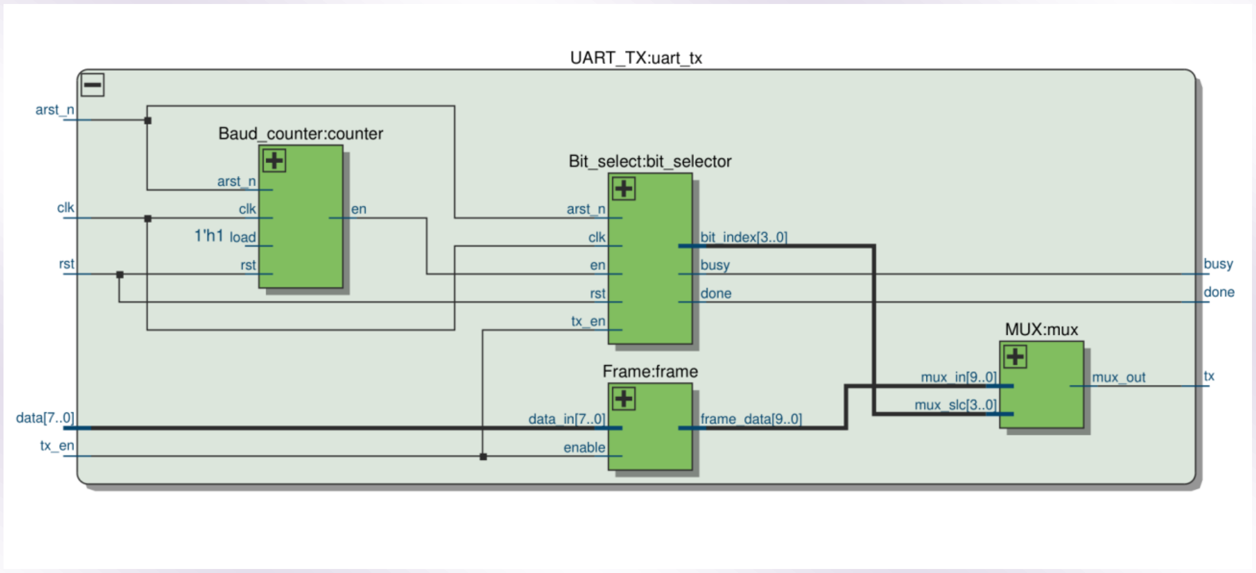
System Architecture / Design Methodology

Block Diagram

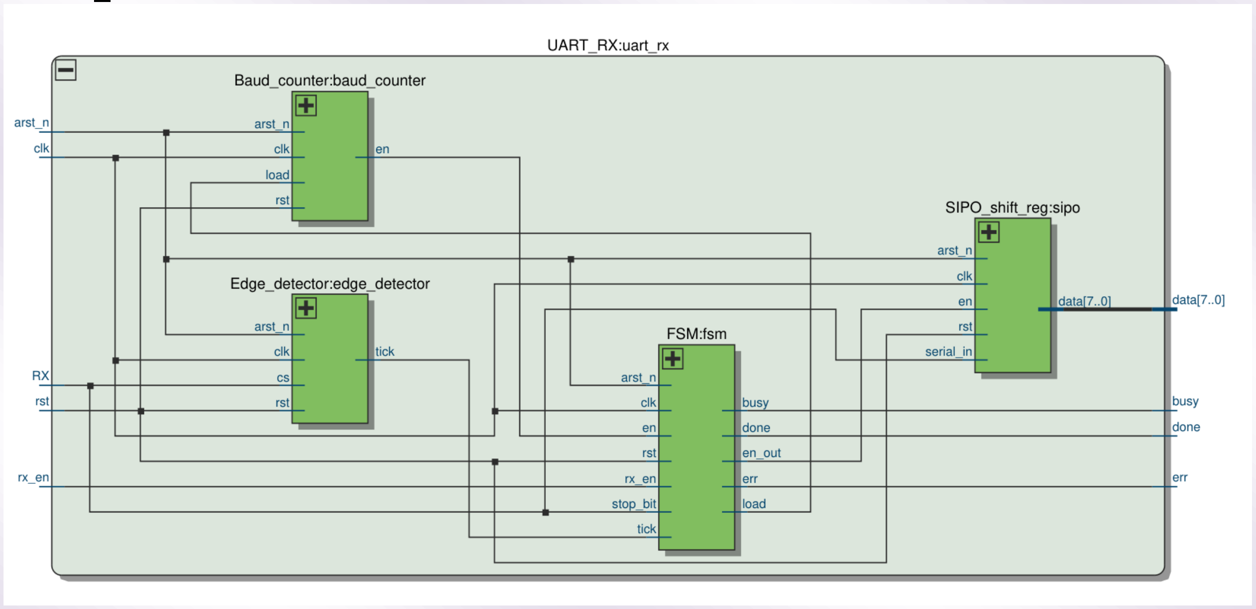
- **UART:**



- **UART_TX:**



- **UART_RX:**



Modules Description

Module Name: Baud counter

- **Purpose:** Generates timing signals at the baud rate (9600 bps) from the 100 MHz system clock.
- **Inputs:** `clk`, `rst`, `arst_n`, `load`, `en`.
- **Outputs:** Provides enable pulses for TX and RX modules.

- **Role in System:** Ensures both transmitter and receiver operate synchronously with the correct baud rate.
- **Code:**

```
module Baud_counter (
  input load,rst,clk,arst_n,
  output reg en
);

reg[13:0] count;

always @(posedge clk or negedge arst_n) begin
  if (!arst_n) begin
      count <= load ? 14'd10417 : 14'd5208;
      en <= 1'b0;
    end
    else if (rst) begin
      count <= load ? 14'd10417 : 14'd5208;
      en <= 1'b0;
    end
    else if (count == 0) begin
      en <= 1'b1;
      count <= load ? 14'd10417 : 14'd5208;
    end
    else begin
      en <= 1'b0;
      count <= count - 1;
    end
end
endmodule
```

Module Name: Frame

- **Purpose:** Prepares the parallel data by adding start and stop bits to form the UART frame.
- **Inputs:** `data_in`, `enable`.
- **Outputs:** `frame_data` (data + start + stop bits).
- **Role in System:** Converts user input (8-bit data) into a UART frame (10 bits: 1 start, 8 data, 1 stop).
- **Code:**

```
module Frame #(parameter data_width = 8)(
  input[data_width-1:0] data_in,
  input enable,
  output[data_width+1:0] frame_data
);

assign frame_data = enable ? {1'b1, data_in, 1'b0} :
{(data_width+2){1'b1}};

endmodule

```

Module Name: MUX

- **Purpose:** Selects the correct bit of the UART frame to send out.
- **Inputs:** `mux_in`, `mux_slc`.
- **Outputs:** `mux_out`.
- **Role in System:** Works with the `Bit_select` module to output serial data bit by bit during transmission.
- **Code:**

```
module MUX #(parameter bit_num = 10,parameter slc_num = 3)(
  input[bit_num-1:0] mux_in,
  input[slc_num:0] mux_slc,
  output reg mux_out
);

always @(*) begin
  case(mux_slc)
    4'b0000: mux_out = mux_in[mux_slc];
    4'b0001: mux_out = mux_in[mux_slc];
    4'b0010: mux_out = mux_in[mux_slc];
    4'b0011: mux_out = mux_in[mux_slc];
    4'b0100: mux_out = mux_in[mux_slc];
    4'b0101: mux_out = mux_in[mux_slc];
    4'b0110: mux_out = mux_in[mux_slc];
    4'b0111: mux_out = mux_in[mux_slc];
    4'b1000: mux_out = mux_in[mux_slc];
    4'b1001: mux_out = mux_in[mux_slc];
    default: mux_out = 1'b1;
  endcase
end
endmodule //MUX

```

Module Name: `Bit_select`

- **Purpose:** Controls which bit of the frame is currently being transmitted.
- **Inputs:** `clk`, `rst`, `arst_n`, `en`.
- **Outputs:** `bit_index`, `busy`, `done`.

- **Role in System:** Keeps track of transmission progress and signals when frame transmission is finished.
- Code:

```verilog
module Bit_select #(parameter bit_num = 10,parameter num_width = 3)(
  input en,clk,rst,arst_n,tx_en,
  output reg[num_width:0] bit_index,
  output reg busy,done
);

always @(posedge clk or negedge arst_n) begin
  if (~arst_n) begin
    bit_index <= {(num_width+1){1'b1}};
    busy <= 0;
    done <= 1;
  end
  else if (rst) begin
    bit_index <= {(num_width+1){1'b1}};
    busy <= 0;
    done <= 1;
  end
  else if (~tx_en) begin
    bit_index = {(num_width+1){1'b1}};
  end
  else begin
    if (bit_index == {(num_width+1){1'b1}}) begin
        bit_index <= {(num_width+1){1'b0}};
      end
    else begin
      if (en) begin
        if (bit_index == bit_num-1) begin
          busy <= 0;
          done <= 1;
          bit_index <= {(num_width+1){1'b0}};
        end
        else begin
          busy <= 1;
          done <= 0;
          bit_index <= bit_index + 1;
        end
      end
    end
  end
end
endmodule

```

Module Name: Edge_detector

- **Purpose:** Detects the start bit edge on RX line.
- **Inputs:** `cs` (RX line), `clk`, `rst`, `arst_n`.
- **Outputs:** `tick` (indicates a falling edge/start bit detected).
- **Role in System:** Synchronizes receiver with incoming data.
- **Code:**

```
module Edge_detector (
  input wire cs,clk,rst,arst_n,
  output reg tick
);

  reg prev;

  always @(posedge clk or negedge arst_n) begin
    if (~arst_n) begin
      prev <= 1'b1;
      tick <= 1'b0;
    end
    else if(rst) begin
      prev <= 1'b1;
      tick <= 1'b0;
    end
    else begin
      tick <= prev & ~cs;
      prev <= cs;
    end
  end

endmodule

```

Module Name: FSM (Receiver)

- **Purpose:** Controls RX operation using states (Idle, Start, Receive, Stop).
- **Inputs:** `clk`, `rst`, `tick`, `rx_en`, `stop_bit`.
- **Outputs:** `done`, `err`, `busy`, `sample_en`, `baud_load`.
- **Role in System:** Manages reception of serial bits, checks stop bit validity, and signals completion/error.
- **Code:**

```
module FSM (
  input rx_en,tick,stop_bit,en,clk,rst,arst_n,
  output load,busy,done,err,en_out
);

```

```
localparam [2:0]IDEL = 0;
localparam [2:0]START = 1;
localparam [2:0]DATA = 2;
localparam [2:0]STOP = 3;
localparam [2:0]ERROR = 4;
localparam [2:0]DONE = 5;

reg[2:0] cs,ns;
reg[3:0]count;

always @(*) begin
  if (~rx_en) begin
    ns = IDEL;
  end
  else begin
    case (cs)
      IDEL:
        if (tick) begin
          ns = START;
        end
        else
          ns = IDEL;
      START:
        if (en) begin
          ns = DATA;
        end
      DATA:
        if (en) begin
          if (count == 8)
            ns = STOP;
          else
            ns = DATA;
        end
      STOP:
        if (stop_bit) begin
          ns = DONE;
        end
        else
          ns = ERROR;
      ERROR:
        ns = ERROR;
      DONE:
        ns = DONE;
      default:
        ns = IDEL;
      endcase
    end
  end

  always @(posedge clk or negedge arst_n) begin
    if (~arst_n) begin
      cs <= IDEL;
      count <= 0;
    end
    else if (rst) begin
      cs <= IDEL;
      count <= 0;
    end
    else begin
      cs <= ns;
      if ((cs == START) && en)
        count <= 0;
      else if ((cs == DATA) && en)
        count <= count + 1;
    end
  end

  assign en_out = ((cs==DATA) && en && (count<8))?1:0;
  assign load =(cs==START)?0:1;
  assign busy = ((cs== START)|| (cs==DATA))?1:0;
  assign done = (cs== DONE)?1:0;
  assign err = (cs == ERROR)?1:0;

  endmodule //FSM

```

Module Name: SIPO_shift_reg

- **Purpose:** Converts received serial bits into parallel data.
- **Inputs:** `clk`, `rst`, `serial_in`, `en`.
- **Outputs:** `data` (8-bit parallel output).
- **Role in System:** Builds the received byte from serial input.
- **Code:**

```
  module SIPO_shift_reg (
    input serial_in,en,clk,rst,arst_n,
    output reg[7:0] data
  );

  always @(posedge clk or negedge arst_n) begin
    if (~arst_n) begin
```

```
      data <= {8{1'b0}};
    end
    else if(rst) begin
      data <= {8{1'b0}};
    end
    else if (en) begin
      data <= {serial_in,data[7:1]};
    end
  end

  endmodule //SIPO_shift_reg

```

Module Name: UART_TX

- **Purpose:** Handles transmission of data.
- **Submodules:** `Frame`, `Baud_counter`, `Bit_select`, `MUX`.
- **Role in System:** Takes parallel data, frames it, and shifts out serially at the correct baud rate.
- **Code:**

```
module UART_TX #(parameter data_width = 8)(
  input tx_en,rst,arst_n,clk,
  input[data_width-1:0] data,
  output done,busy,tx
);
localparam tx_load = 1;
wire[data_width+1:0] frame_data;
wire en;
wire[3:0] mux_slc;

Baud_counter
counter(.rst(rst),.clk(clk),.arst_n(arst_n),.en(en),.load(tx_load));
Frame
#(data_width)frame(.data_in(data),.enable(tx_en),.frame_data(frame_data));
Bit_select #(10,3)
bit_selector(.en(en),.clk(clk),.rst(rst),.arst_n(arst_n),.bit_index(mux_sl
c),.busy(busy),.done(done),.tx_en(tx_en));
MUX #(10,3)mux(.mux_in(frame_data),.mux_slc(mux_slc),.mux_out(tx));

endmodule //UART_TX

```

Module Name: UART_RX

- **Purpose:** Handles reception of data.
- **Submodules:** `Edge_detector`, `FSM`, `Baud_counter`, `SIPO_shift_reg`.
- **Role in System:** Detects start bit, samples incoming bits, and outputs received byte with error detection.
- **Code:**

```
module UART_RX (
input RX,rx_en,clk,rst,arst_n,
output done,err,busy,
output [7:0] data
);

wire baud_load,sample_en,tick,baud_en;

Edge_detector edge_detector(
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .cs(RX),
  .tick(tick)
  );
FSM fsm(
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .done(done),
  .err(err),
  .busy(busy),
  .tick(tick),
  .stop_bit(RX),
  .rx_en(rx_en),
  .en(baud_en),
  .load(baud_load)
  ,.en_out(sample_en)
  );
SIPO_shift_reg sipo(
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .serial_in(RX),
  .data(data),
  .en(sample_en)
  );
Baud_counter baud_counter(
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .load(baud_load),
  .en(baud_en)
  );

  endmodule

```

Module Name: UART (Top-Level)

Inputs

- `RX` : Serial input data line from external device.
- `clk` : System clock (100 MHz).
- `rst` : Synchronous reset.
- `arst_n` : Asynchronous active-low reset.
- `rx_en` : Enable signal for the receiver.
- `tx_en` : Enable signal for the transmitter.
- `data_in[7:0]` : 8-bit parallel data to be transmitted.

Outputs

- `data_out[7:0]` : 8-bit parallel data received.
- `TX` : Serial output data line to external device.
- `err` : Error flag (framing or invalid data in RX).
- `done_rx` : High when receiver successfully receives a frame.
- `busy_rx` : Indicates receiver is currently processing data.
- `done_tx` : High when transmission of data is completed.
- `busy_tx` : Indicates transmitter is currently busy.

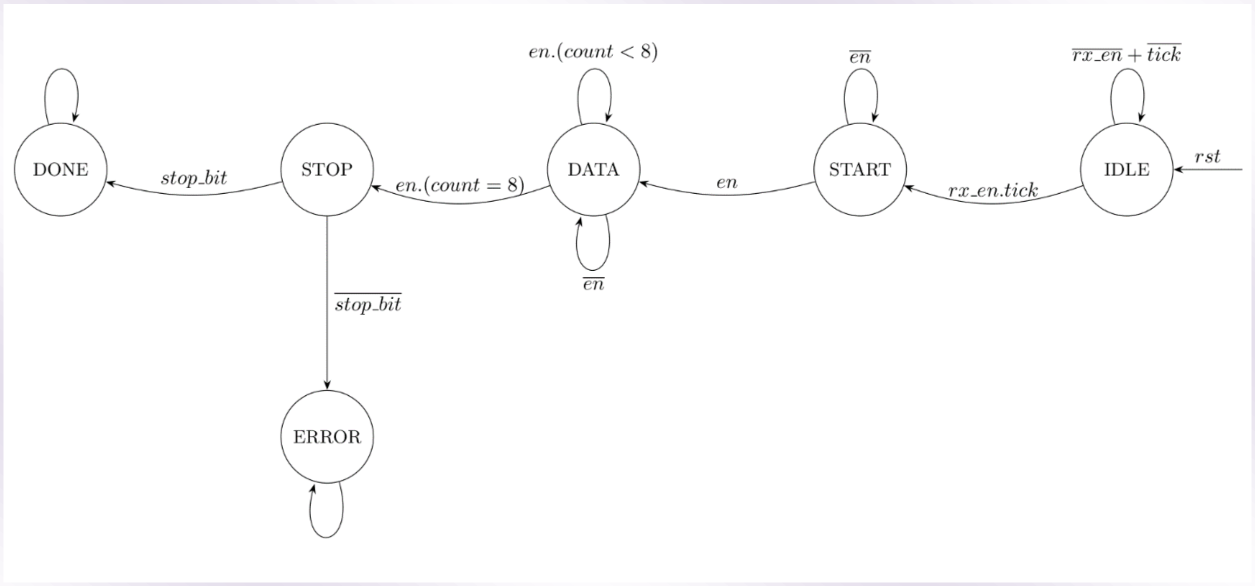
Function
The `UART` module acts as the **top-level wrapper** for the UART system. It integrates the transmitter (`UART_TX`) and receiver (`UART_RX`) modules, providing a unified interface for bidirectional serial communication. Based on enable signals (`tx_en` / `rx_en`), the module handles both sending and receiving operations simultaneously, ensuring proper synchronization with system resets.

Role in System
This module represents the **user-facing interface** of the UART design. It abstracts the internal details of transmission and reception, exposing only necessary control, status, and data signals. It enables the system to interact with external devices via the UART protocol in a simple plug-and-play fashion.

Design Methodology

The design of the custom UART follows a structured, modular approach to ensure clarity, reusability, and correctness. The main design principles adopted are:

1. **FSM-Based Design**
    - The **RX module** relies on a finite state machine with clearly defined states (IDLE, START, DATA, STOP, DONE, ERROR).
      This ensures predictable behavior, error detection, and synchronization with the baud tick.
    - **Diagram**:



2. **Clock Domain Synchronization**
    - The UART operates from a **100 MHz system clock**, while communication timing is derived from a **baud rate generator** that provides baud ticks (9600 bps).
    - This ensures accurate sampling of received bits and correct timing for transmitted frames.
    - Care was taken to synchronize control signals crossing between the system clock and baud tick domains to avoid metastability.
3. **Reset Strategy**
    - Both **asynchronous reset** (`arst_n`) and **synchronous reset** (`rst`) are supported.
    - Asynchronous reset guarantees a safe start-up condition immediately after power-on.
    - Synchronous reset ensures controlled initialization during simulation and runtime, improving reliability in FPGA implementation.
4. **Simulation-Driven Verification**
    - Each module (Baud Counter, FSMs, SIPO, Bit Select, TX/RX) was individually verified through **Questa Sim simulation**.

- Testbenches were written to validate correct data transmission, reception, error detection and transmission enabling.

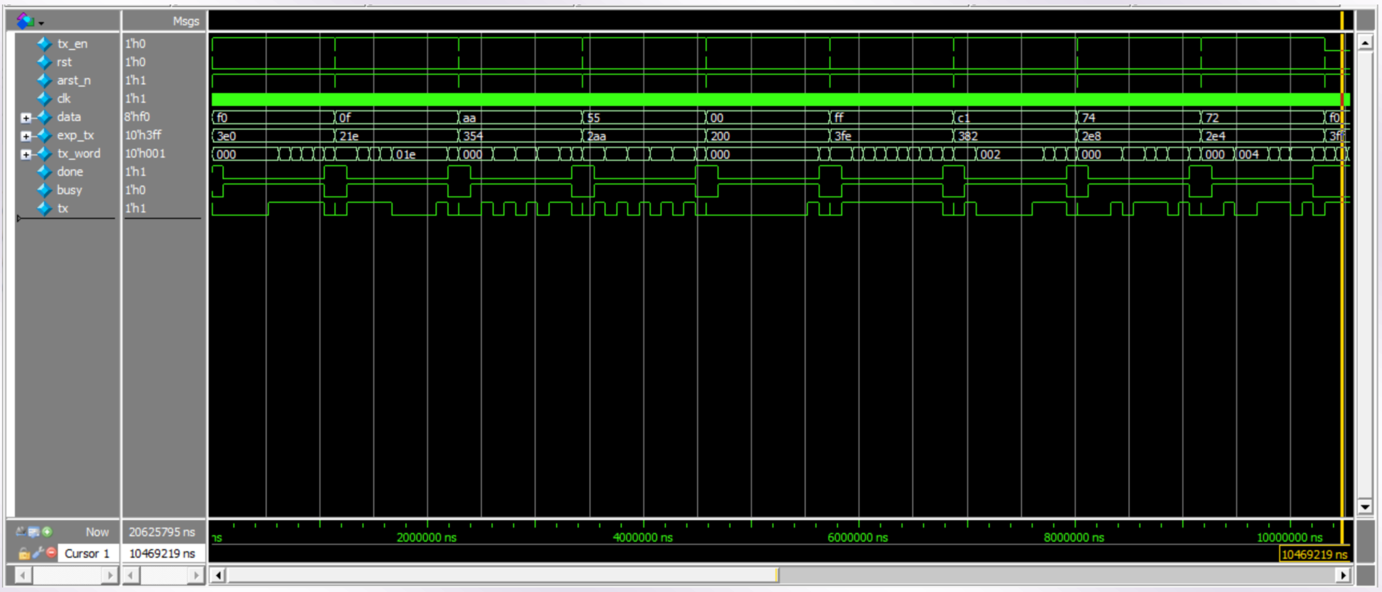
Testbenches:

Transmitting Data test:

Code:

```
module TX_tb ();

reg tx_en,rst,arst_n,clk;
reg[7:0] data;
reg [9:0] exp_tx,tx_word;
wire done,busy,tx;

UART_TX
uart_tx(.tx_en(tx_en),.rst(rst),.arst_n(arst_n),.clk(clk),.data(data
),.done(done),.busy(busy),.tx(tx));

initial begin
  clk = 0;
  forever begin
    #5 clk = ~clk;
  end
end

integer i;

initial begin
  testing(8'b11110000);
  testing(8'b00001111);
  testing(8'b10101010);
  testing(8'b01010101);
  testing(8'b00000000);
  testing(8'b11111111);
  testing(8'b11000001);
  testing(8'b01110100);
  testing(8'b01110010);

  $stop;
end

task testing(input [7:0]data_in);begin
  rst = 1;
  arst_n = 0;
  tx_en = 0;
  #5
  tx_word = {10{1'b0}};
  rst = 0;
  arst_n = 1;
  data = data_in;
  exp_tx = {1'b1,data_in,1'b0};
  tx_en = 1;
  for (i = 0 ; i < 10;i = i+1 ) begin
    #104170
    tx_word[i] = tx;
  end
  if (exp_tx != tx_word) begin
      $display("Error!");
      $stop;
    end
    else
      $display("based!");
  #104170;
end
endtask


endmodule //TX_tb
```

Wave form:

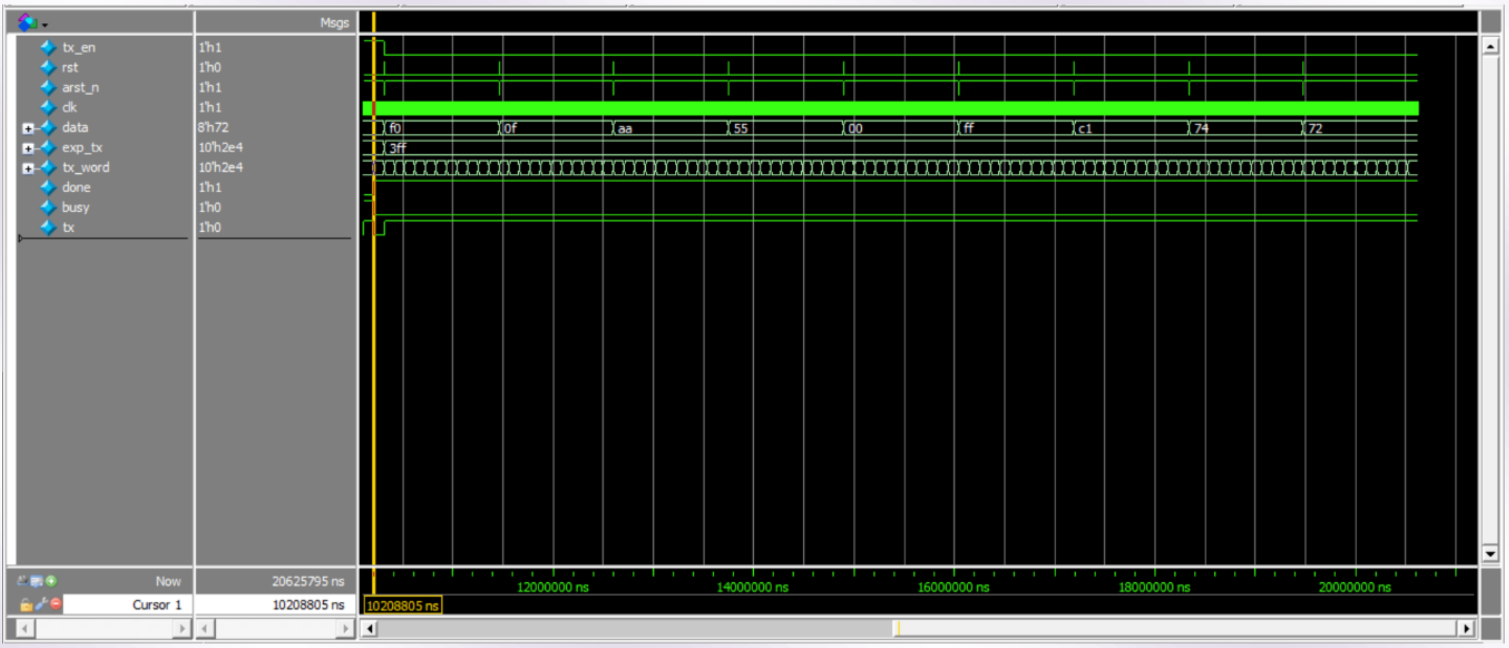


TX enable test:

Code:

```verilog
module TX_tb ();


reg tx_en,rst,arst_n,clk;
reg[7:0] data;
reg [9:0] exp_tx,tx_word;
wire done,busy,tx;

UART_TX
uart_tx(.tx_en(tx_en),.rst(rst),.arst_n(arst_n),.clk(clk),.data(data
),.done(done),.busy(busy),.tx(tx));

initial begin
  clk = 0;
  forever begin
    #5 clk = ~clk;
  end
end

integer i;

initial begin

  testing_en(8'b11110000);
  testing_en(8'b00001111);
  testing_en(8'b10101010);
  testing_en(8'b01010101);
  testing_en(8'b00000000);
  testing_en(8'b11111111);
  testing_en(8'b11000001);
  testing_en(8'b01110100);
  testing_en(8'b01110010);
  $stop;
end


task testing_en(input [7:0]data_in);begin
  rst = 1;
  arst_n = 0;
  tx_en = 0;
  #5
  tx_word = {10{1'b0}};
  rst = 0;
  arst_n = 1;
  #5
  data = data_in;
  exp_tx = {10{1'b1}};
  for (i = 0 ; i < 10;i = i+1 ) begin
    #104170
    tx_word[i] = tx;
  end
  if (exp_tx != tx_word) begin
      $display("Error!");
      $stop;
    end
    else
      $display("based!");
  #104170;
end
endtask
endmodule //TX_tb
```

Wave form:



	Msgs
tx_en	1'h1
rst	1'h0
arst_n	1'h1
clk	1'h1
data	8'h72
exp_tx	10'h2e4
tx_word	10'h2e4
done	1'h1
busy	1'h0
tx	1'h0
Now	20625795 ns
Cursor 1	10208805 ns

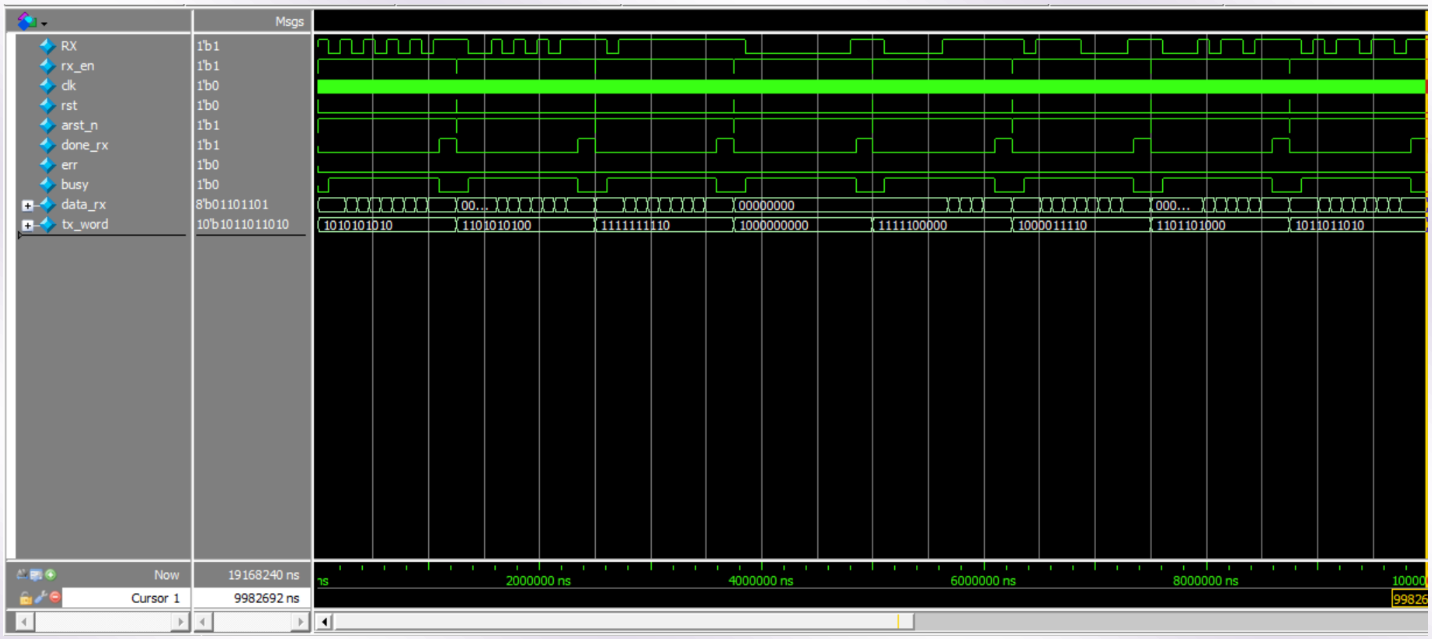
Receiving Data test:

Code:

```
module RX_tb ();

reg RX,rx_en,clk,rst,arst_n;
wire done_rx,err,busy;
wire [7:0]data_rx;
reg[9:0] tx_word;

UART_RX uart_rx(
  .RX(RX),
  .rx_en(rx_en),
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .done(done_rx),
  .err(err),
  .busy(busy),
  .data(data_rx)
  );

initial begin
  clk = 0;
  forever begin
    #5 clk = ~clk;
  end
end

integer i;

initial begin
  testing(8'b01010101);
  testing(8'b10101010);
  testing(8'b11111111);
  testing(8'b00000000);
  testing(8'b11110000);
  testing(8'b00001111);
  testing(8'b10110100);
  testing(8'b01101101);
  $stop;
end
```

```
task testing(input [7:0] data_in);begin
  RX = 1;
  tx_word = {1'b1,data_in,1'b0};
  rst = 1;
  arst_n = 0;
  rx_en = 0;
  RX = 1;
  #10
  rst = 0;
  arst_n = 1;
  rx_en = 1;
  #50
  for (i = 0 ; i < 10;i = i+1 ) begin
    #104170
    RX = tx_word[i];
  end
  #104170
  if (data_in!=data_rx) begin
    $display("Error detected!");
    $stop;
  end
  else
    $display("Test based!");

  #104170;
end
endtask


endmodule
```

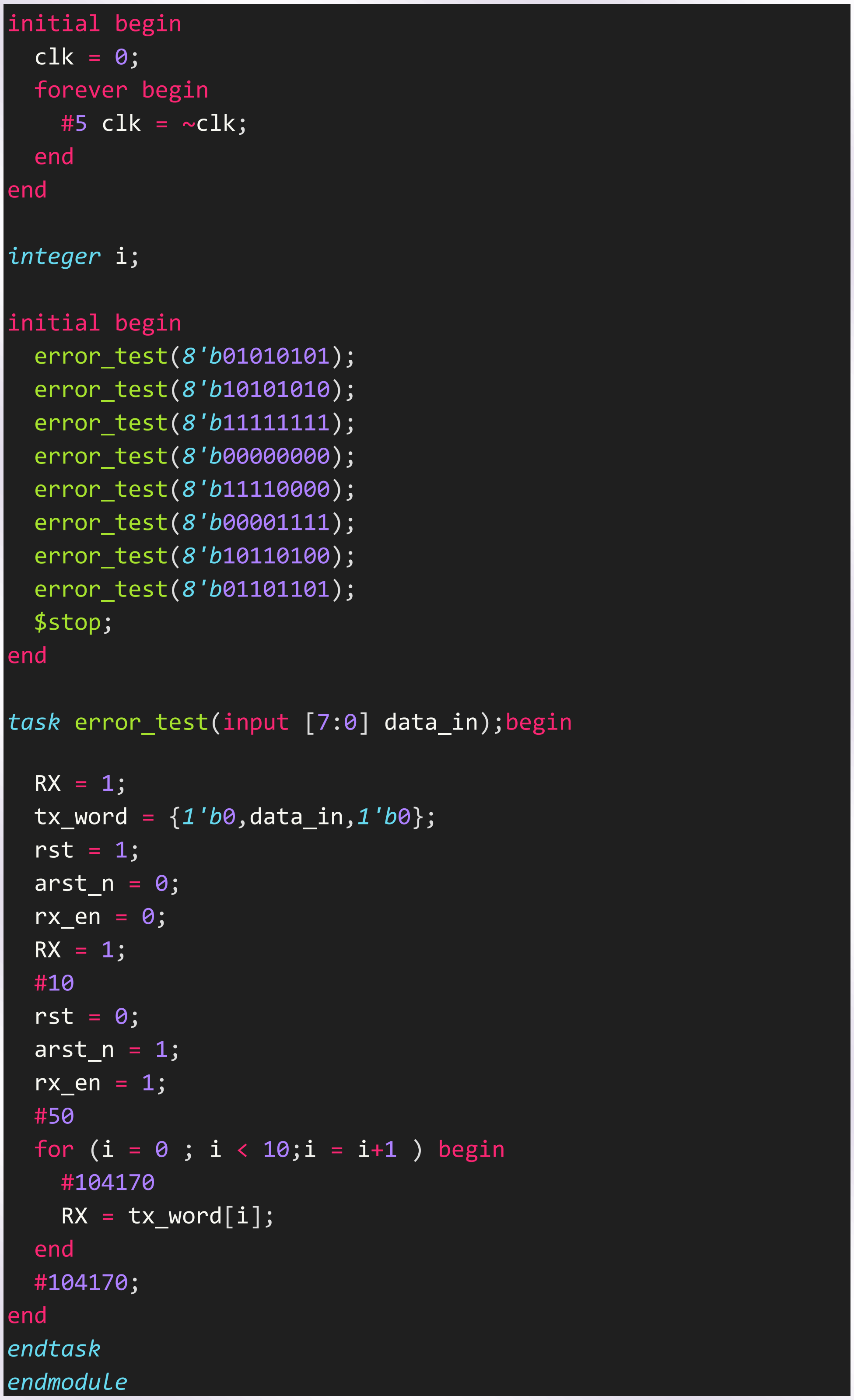
Wave fcorm:



Error detection test:

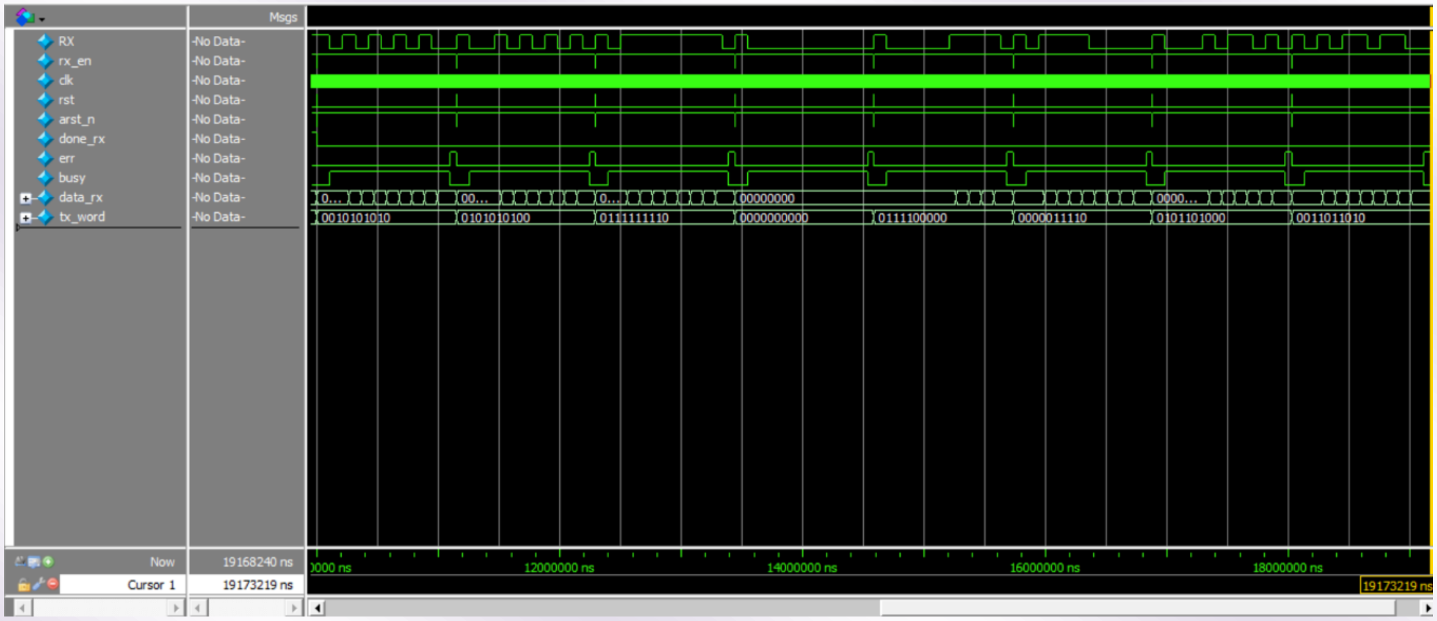
Code:

```
module RX_tb ();

reg RX,rx_en,clk,rst,arst_n;
wire done_rx,err,busy;
wire [7:0]data_rx;
reg[9:0] tx_word;

UART_RX uart_rx(
  .RX(RX),
  .rx_en(rx_en),
  .clk(clk),
  .rst(rst),
  .arst_n(arst_n),
  .done(done_rx),
  .err(err),
  .busy(busy),
  .data(data_rx)
  );
```



```verilog
initial begin
  clk = 0;
  forever begin
    #5 clk = ~clk;
  end
end

integer i;

initial begin
  error_test(8'b01010101);
  error_test(8'b10101010);
  error_test(8'b11111111);
  error_test(8'b00000000);
  error_test(8'b11110000);
  error_test(8'b00001111);
  error_test(8'b10110100);
  error_test(8'b01101101);
  $stop;
end

task error_test(input [7:0] data_in);begin

  RX = 1;
  tx_word = {1'b0,data_in,1'b0};
  rst = 1;
  arst_n = 0;
  rx_en = 0;
  RX = 1;
  #10
  rst = 0;
  arst_n = 1;
  rx_en = 1;
  #50
  for (i = 0 ; i < 10;i = i+1 ) begin
    #104170
    RX = tx_word[i];
  end
  #104170;
end
endtask
endmodule
```

Wave form:



Conclusion

This project successfully demonstrated the design and implementation of a UART (Universal Asynchronous Receiver-Transmitter) using Verilog HDL. The system was structured into modular components including the transmitter, receiver, baud rate generator, frame generator, FSM controller, and supporting logic, ensuring clarity and scalability of the design.

Simulation results verified correct data transmission and reception under the 8N1 format (8 data bits, no parity, 1 stop bit) at a baud rate of 9600 bps with a 100 MHz system clock. The RX and TX testbenches confirmed reliable operation by testing multiple data patterns and validating correct frame generation, synchronization, and error detection.

Beyond achieving the functional goals, this work provided valuable hands-on experience with FSM-based design, RTL coding, and verification methodologies. It also emphasized the importance of modularity, reset strategies, and simulation-driven validation in digital system design.

The developed UART can serve as a foundation for more advanced communication modules, and can be further extended with features such as configurable baud rates, parity bit options, FIFO buffering, and FPGA-based hardware testing for real-world applications.