Table of Contents

1. Abstract

The project "Design and Verification of Universal Asynchronous Receiver and Transmitter (UART)" is a comprehensive exploration of a fundamental component of modern digital communication systems. It is divided into two critical phases: "Verilog-Based Verification of Each IP Block" and "System Verilog-Based Testbench Verification."

In the first phase, the project delved into the intricacies of UART, dissecting it into its constituent IP blocks. Each block, from the transmitter to the receiver and clock management units, underwent meticulous design and rigorous verification to ensure seamless operation, compliance with UART standards, and robust performance.

Transitioning into the second phase, the project constructed an extensive testing environment to evaluate the complete UART system. System Verilog-based testbenches were developed to simulate real-world scenarios and stress-test the UART's functionality, error-handling capabilities, and data throughput under various conditions.

The project signifies a commitment to excellence and a pursuit of knowledge in digital communication systems, contributing to the advancement of reliable and efficient data transmission in modern electronic devices.

2. Introduction

2.1 Introduction to UART

A UART (Universal Asynchronous Receiver/Transmitter) is the microchip with programming that controls a computer's interface to its attached serial devices. Specifically, it provides the computer with the RS-232C Data Terminal Equipment (DTE) interface so that it can "talk" to and exchange data with modems and other serial devices. As part of this interface, the UART also:

- Converts the bytes it receives from the computer along parallel circuits into a single serial bit stream for outbound transmission.
- On inbound transmission, converts the serial bit stream into the bytes that the computer handles.
- Adds a parity bit (if it's been selected) on outbound transmissions and checks the parity of incoming bytes (if selected) and discards the parity bit.
- Adds start and stop delineators on outbound and strips them from inbound transmissions.
- Handles interrupts from the keyboard and mouse (which are serial devices with special ports).
- May handle other kinds of interrupt and device management that require coordinating the computer's speed of operation with device speeds.

2.2 Significance of UART

The significance of UART lies in its ability to provide asynchronous, full-duplex communication. In full-duplex mode, UART devices can both transmit and receive data simultaneously, allowing for bidirectional communication. Furthermore, UART operates asynchronously, meaning that data is sent without the need for a shared clock signal between the transmitter and receiver. This makes UART a reliable choice for various applications, especially when precise timing is not critical.

2.3 Applications of UART

UART's versatility makes it a fundamental component in numerous electronic systems. Some common applications include:

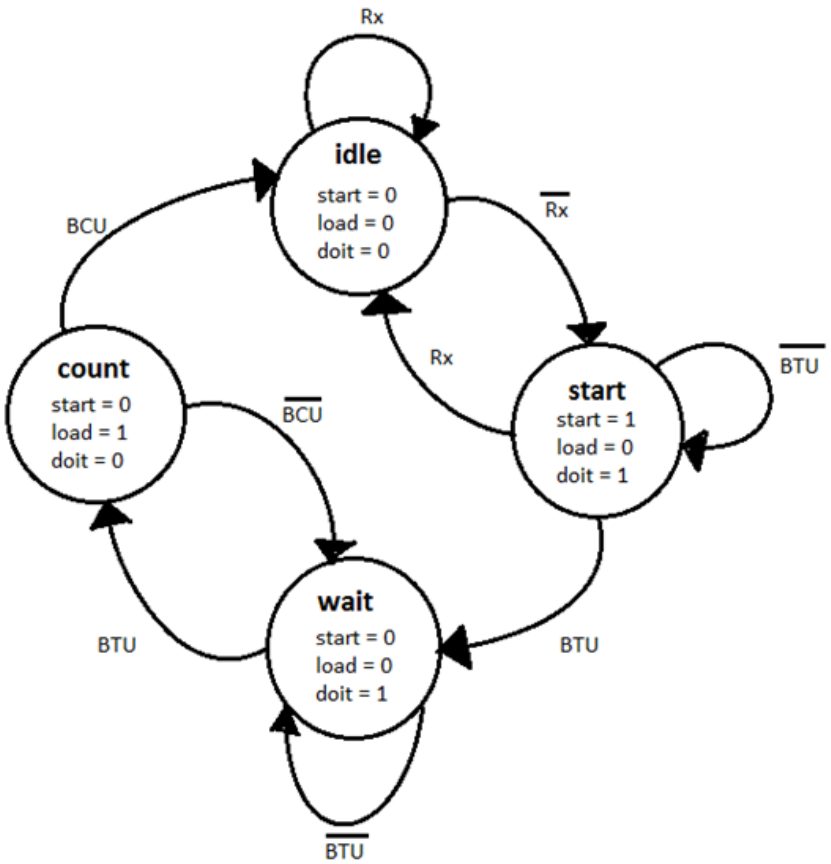
i. **Serial Communication:** UART is widely used for serial communication between microcontrollers, sensors, and various peripheral devices.
ii. **Wireless Communication:** UART interfaces are often used to connect devices to wireless modules like Bluetooth and Wi-Fi.

iii. **Data Logging:** UART is employed in data logging systems to capture and transmit data to external storage devices or remote servers.
iv. **Industrial Control:** In industrial automation, UART facilitates communication between programmable logic controllers (PLCs) and other control devices.
v. **GPS Navigation:** UART interfaces are found in GPS receivers for the exchange of location data.
vi. **Interfacing with Legacy Systems:** UART is also utilized to connect modern systems with legacy equipment that use serial communication.

2.4 Project Objective

For this project, our objective is to design and verify a UART system, focusing on both the design of individual IP blocks and the development of a System Verilog-based testbench to thoroughly validate its functionality. By delving into the core of UART technology and applying rigorous verification, we aim to contribute to the field of digital communication and further its reliability and efficiency in real-world applications.

3. FSM Diagram



4. UART Transmitter

The UART (Universal Asynchronous Receiver and Transmitter) transmitter is a critical component of the UART communication system. Its primary function is to convert parallel data from a digital source into serial data for transmission over a communication link. Here's an overview of the key elements and operations of a UART transmitter:

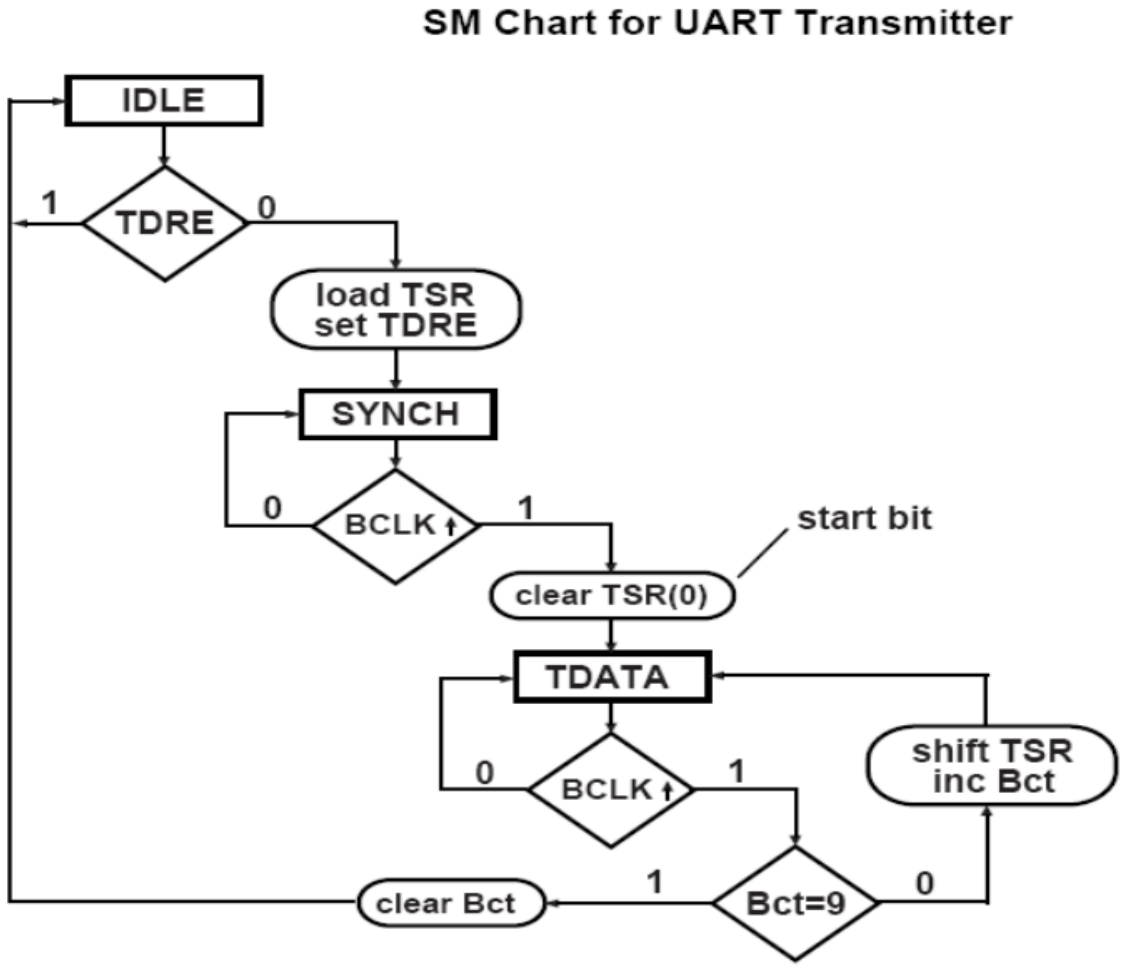
1. **Data Input (Data Buffer):** The transmitter begins with a data buffer where the parallel data to be transmitted is stored. This data buffer typically consists of a shift register or a data register, which holds the data temporarily.
2. **Baud Rate Generator:** The transmitter relies on a baud rate generator to establish the transmission speed. The baud rate generator generates a clock signal that determines the rate at which bits are sent. It is essential for both the transmitter and receiver to operate at the same baud rate for successful communication.
3. **Framing Generator:** The transmitter is responsible for adding start and stop bits to the data to frame it properly. Typically, a UART uses a start bit (logical 0) to indicate the beginning of a data frame and one or more stop bits (logical 1) to signify the end of the frame.
4. **Transmitter Control Logic:** This component controls the data transmission process. It coordinates the flow of data from the data buffer, ensures that the data is framed correctly with start and stop bits, and manages the clocking of data.
5. **Transmitter Shift Register:** The heart of the UART transmitter is the shift register. It takes the parallel data from the data buffer and shifts it out serially, starting with the start bit and ending with the stop bit(s). The transmitter control logic coordinates the shifting of bits and synchronization with the baud rate.
6. **Serial Data Output (TX Pin):** The serial data, which includes the start bit, data bits, optional parity bit, and stop bit(s), is output on the TX (transmit) pin. The TX pin connects to the communication link or the external device receiving the data.
7. **Parity (Optional):** Some UART configurations include a parity bit for error checking. If used, the transmitter inserts the parity bit based on the selected parity mode (even, odd, or none) before the stop bit(s).

The general process of a UART transmitter can be summarized as follows:

i. The transmitter awaits a signal to start transmitting data.

ii. When data is ready for transmission, it loads the data buffer with the parallel data.
iii. The transmitter control logic coordinates the framing of data, including adding the start bit and stop bit(s).
iv. The transmitter shift register serializes the data, sending it bit by bit, starting with the start bit.
v. The serial data, along with the start bit and optional parity bit, is transmitted serially on the TX pin.
vi. Once the entire data frame is sent, the transmitter is ready to send the next frame.

The UART transmitter operates asynchronously, which means it doesn't rely on a shared clock signal with the receiver. Instead, it uses the baud rate to ensure proper synchronization between transmitter and receiver.



5. UART Receiver

All operations of the UART hardware are controlled by an internal clock signal which runs at a multiple of the data rate, typically 8 or 16 times the bit rate. The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit. If the apparent start bit lasts at least one-half of the bit time, it is valid and signals the start of a new character. If not, it is considered a spurious pulse and is ignored. After waiting a further bit time, the state of the line is again sampled and the resulting level clocked into a shift register. After the required number of bit periods for the character length (5 to 8 bits, typically) have elapsed, the contents of the shift register are made available (in parallel fashion) to the receiving system. The UART will set a flag indicating new data is available, and may also generate a processor interrupt to request that the host processor transfers the received data.

Communicating UARTs have no shared timing system apart from the communication signal. Typically, UARTs resynchronize their internal clocks on each change of the data line that is not considered a spurious pulse. Obtaining timing information in this manner, they reliably receive when the transmitter is sending at a slightly different speed than it should. Simplistic UARTs do not do this; instead they resynchronize on the falling edge of the start bit only, and then read the center of each expected data bit, and this system works if the broadcast data rate is accurate enough to allow the stop bits to be sampled reliably.

It is a standard feature for a UART to store the most recent character while receiving the next. This "double buffering" gives a receiving computer an entire character transmission time to fetch a received character. Many UARTs have a small first-in, first-out (FIFO) buffer memory between the receiver shift register and the host system interface. This allows the host processor even more time to handle an interrupt from the UART and prevents loss of received data at high rates.

The UART (Universal Asynchronous Receiver and Transmitter) receiver is a critical component in UART communication systems. Its primary function is to convert received serial data into parallel data for further processing by the connected device. Here's an overview of the key elements and operations of a UART receiver:

1. **Received Data Buffer:** The receiver begins with a received data buffer where the incoming serial data is temporarily stored. This data buffer is typically implemented using a shift register or data register.

2. **Baud Rate Generator:** Similar to the transmitter, the receiver also uses a baud rate generator to generate a clock signal, which determines the rate at which bits are received. It is crucial for both the transmitter and receiver to operate at the same baud rate for successful communication.
3. **Framing Detector:** The framing detector is responsible for identifying the start and stop bits within the incoming serial data. It identifies the start bit and locates the data frame within the serial stream, allowing the receiver to extract the data.
4. **Receiver Control Logic:** The receiver control logic manages the reception process. It coordinates the flow of data from the serial input, ensures that the received data is correctly framed, and manages the clocking of data.
5. **Receiver Shift Register:** The receiver shift register takes in the incoming serial data bit by bit, starting with the start bit and ending with the stop bit(s). The receiver control logic ensures proper synchronization with the baud rate.
6. **Data Output (Parallel Data):** The extracted data, which includes the data bits, and optional parity bit (if used), is available in parallel form for the connected device or microcontroller to process. The parallel data is typically stored in a data register.
7. **Parity (Optional):** If the UART configuration includes a parity bit for error checking, the receiver checks the received data for parity errors based on the selected parity mode (even, odd, or none).

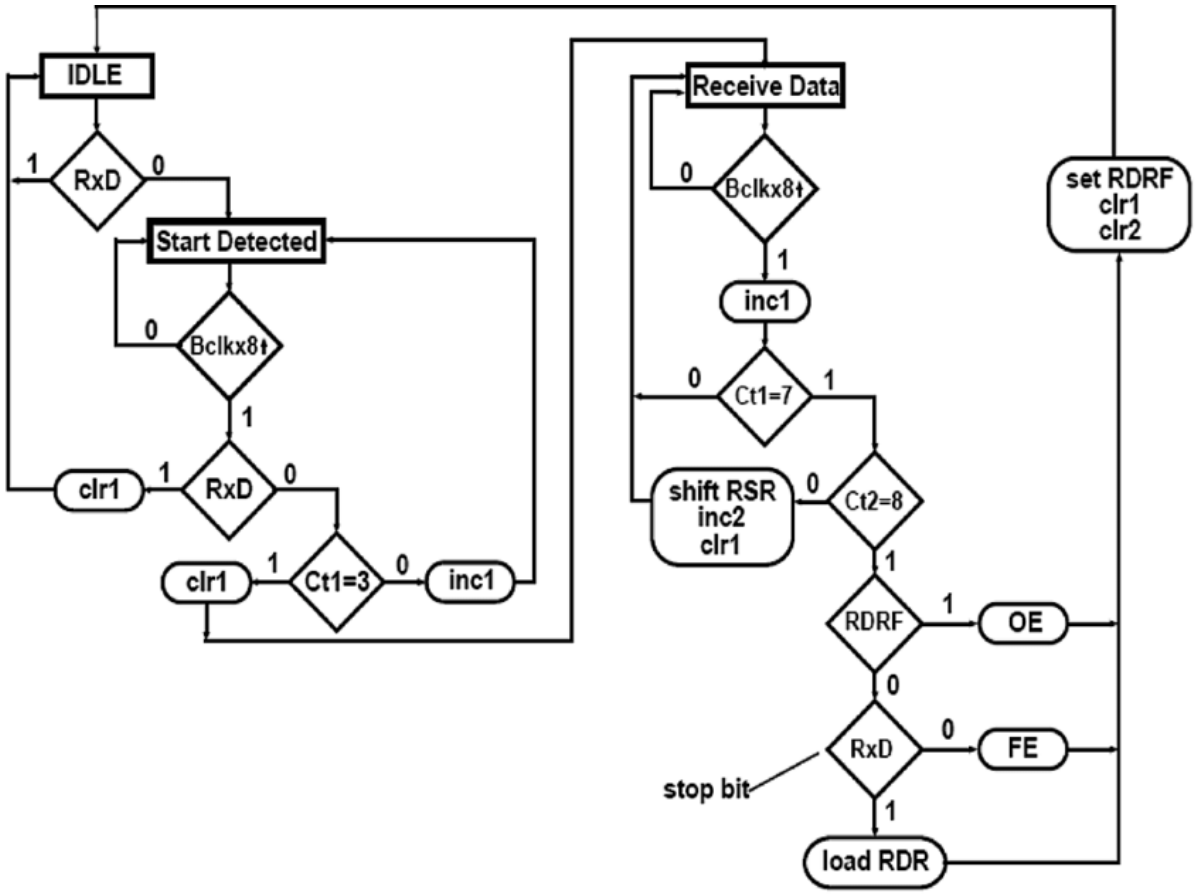
The general process of a UART receiver can be summarized as follows:

i. The receiver continuously monitors the incoming serial data stream on its RX (receive) pin.
ii. When it detects a start bit (usually a logical 0), it begins the reception process.
iii. The framing detector identifies the start bit and locates the data frame within the serial stream.
iv. The receiver control logic coordinates the reception process, ensuring that the received data is correctly framed and that it is clocked correctly.
v. The receiver shift register collects the incoming serial data bit by bit, including the data bits, optional parity bit, and stop bit(s).
vi. Once the entire data frame is received and checked for errors (if parity is used), the parallel data is available for the connected device to process.

The UART receiver operates asynchronously, meaning it does not rely on a shared clock signal with the transmitter. Instead, it uses the baud rate to ensure

proper synchronization between the transmitter and receiver. This makes UART a versatile choice for serial communication between digital devices, including microcontrollers, computers, and other embedded systems.

SM Chart for UART Receiver



6. Baud Rate Generator

Baud rate generator determines transmission speed in asynchronous communication. It is number of symbols per second transferred. Each bit is 1/(baud rate) wide.

The baud generator is responsible for generating a periodic baud pulse based on the divisor latch value which determines the baud rate for the serial transmission. This periodic pulse is used by transmitter and receiver to generate sampling pulses for sampling both received data and data to be transmitted. One baud out occurs for sixteen clock cycles. For sixteen clock cycles one bit data will be sent. There are two debug registers they work in32-bit data bus mode. It has 5-bit address mode. It is read only and is provided for debugging purpose for chip testing. Each has a 256-byte FIFO to buffer data flow [3]. The use of FIFO buffers increases the overall transmission rate by allowing slower processors to respond, and reducing the amount of time wasted context switching. Besides data transfer, they also facilitate start/stop framing bits, check various parity options, as well as detect transmission errors.

Baud Rate: It is the speed of transferring data from the transmitter to a receiver in the form of bits per second or we can say that number of signal units transferred per second.

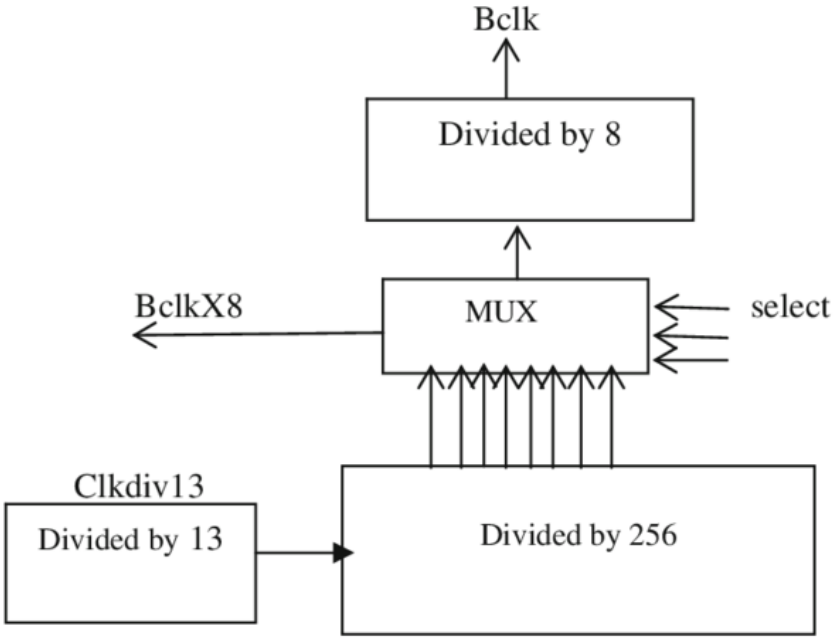
Baud rate = clock frequency/(16*divisor)

Some standard baud rate:

i. 2400
ii. 9600 (Mostly used)
iii. 19200

This is designed using Verilog HDL language, as this part is to provide the speed of data transmission it uses a clock(clk) , reset and a multiplexer dor selecting the speed of data transfer. clock out (clkout) is the output signal which is then provided as the input to further modules.

Block diagram of Baud Rate Generator



The UART (Universal Asynchronous Receiver and Transmitter) baud rate generator is a crucial component in UART communication systems. It generates a clock signal that determines the data rate at which bits are transmitted and received over the communication link. The baud rate (expressed in bits per second or bps) is essential for ensuring that both the transmitter and receiver operate at the same speed to maintain proper synchronization. Here's how the UART baud rate generator typically works:

1. Input Clock Source: The baud rate generator relies on an input clock source, which can come from an external crystal oscillator, a system clock, or any other source of known frequency. The frequency of this input clock source, often denoted as F_clk, serves as the reference for generating the baud rate.

2. Divisor Calculation: To generate the desired baud rate, a divisor (often denoted as "divisor value" or "divider") is calculated based on the input clock frequency and the target baud rate. The formula to calculate the divisor is as follows:

Divisor = F_clk / (Baud Rate * 16)

The factor of 16 is commonly used for standard UART communication. In some cases, where there are additional options for oversampling, the factor may vary, but 16 is the most widely used.

3. Divisor Configuration: The calculated divisor value is then configured in the baud rate generator, typically using specific control registers or configuration settings within the UART module.

4. **Clock Division:** The configured divisor divides the input clock frequency to generate the baud rate clock signal, which is used to determine the timing of bit transmission. This clock signal is often referred to as the "baud rate clock" and is typically in the range of a few megahertz.

5. **Synchronization:** The transmitter and receiver in the UART system use this common baud rate clock signal to ensure proper synchronization when transmitting and receiving data. This synchronization allows both devices to correctly interpret the start and stop bits and the data bits between them.

6. **Matching Baud Rate:** To ensure successful communication, it's crucial that both the transmitter and receiver are configured with the same baud rate settings. Mismatched baud rates can lead to errors and data corruption.

It's important to note that UART communication relies on the assumption that both the transmitter and receiver use the same baud rate and operate with accurate and stable clocks. Inaccurate baud rate settings can lead to communication errors and data loss.

The UART baud rate generator plays a central role in establishing the communication speed and ensuring that data is transmitted and received correctly between devices. Configuring and calibrating the baud rate generator is a critical part of setting up UART communication in embedded systems.

7. Verification of each IP Block

The verification of each IP (Intellectual Property) block of a UART (Universal Asynchronous Receiver and Transmitter) is a crucial step in the design and development process of the UART system. Each block within the UART, including the transmitter and receiver, must be thoroughly verified to ensure that they perform their functions correctly and reliably. The verification process typically involves a combination of simulation, testing, and analysis. Below, I'll provide an overview of how to verify each IP block of a UART:

1. Transmitter Verification:

Functional Verification:

- Verify that the transmitter correctly sends data from the data buffer.
- Ensure that the transmitter inserts the correct number of start and stop bits.
- Check that the data is framed correctly and in the desired format.
- Verify that the data rate (baud rate) is as configured and consistent with the receiver's settings.

Simulation:

- Use RTL (Register Transfer Level) simulation to verify the functionality of the transmitter.
- Generate test vectors to simulate data transmission under various conditions, including different data patterns, baud rates, and framing options.
- Ensure that the transmitter responds correctly to control signals (e.g., enable/disable signals).

Timing Analysis:

- Analyze the timing of the transmitter to ensure that the data is sent at the correct rate and in the correct format.
- Verify that there are no setup or hold time violations for critical signals.

2. Receiver Verification:

Functional Verification:

- Verify that the receiver correctly identifies and interprets the start and stop bits.

- Check that the received data is framed correctly and in the desired format.
- Ensure that the receiver detects and handles errors such as framing errors, parity errors, and overrun errors.
- Verify that the data rate (baud rate) is as configured and matches the transmitter's settings.

Simulation:

- Use RTL simulation to verify the functionality of the receiver.
- Generate test vectors to simulate data reception under various conditions, including different data patterns, baud rates, and framing options.
- Simulate error conditions to ensure that the receiver can detect and report errors.

Timing Analysis:

Analyze the timing of the receiver to ensure that it correctly samples incoming data and that there are no timing violations.

3. Baud Rate Generator Verification:

Functional Verification:

- Verify that the baud rate generator calculates the divisor correctly based on the input clock frequency and the desired baud rate.
- Ensure that the generated baud rate clock is stable and matches the desired rate.
- Check that the baud rate generator can be reconfigured to support different baud rates.

Simulation and Testing:

- Simulate the baud rate generation process to verify the correctness of the calculated divisor.
- Measure the frequency of the generated baud rate clock and compare it to the expected value.

The verification process for each IP block of a UART is essential to ensure the overall reliability and correctness of the UART communication system. It's important to follow established best practices for verification, and thorough testing should be conducted to catch any potential issues before the UART is deployed in a real-world application.

8. SV Testbench Verification

The System Verilog (SV) testbench verification phase represents a crucial step in our project, focusing on ensuring the reliability and robustness of our Universal Asynchronous Receiver and Transmitter (UART) design. This phase is dedicated to the thorough testing and validation of our UART system under a variety of conditions, simulating real-world scenarios.

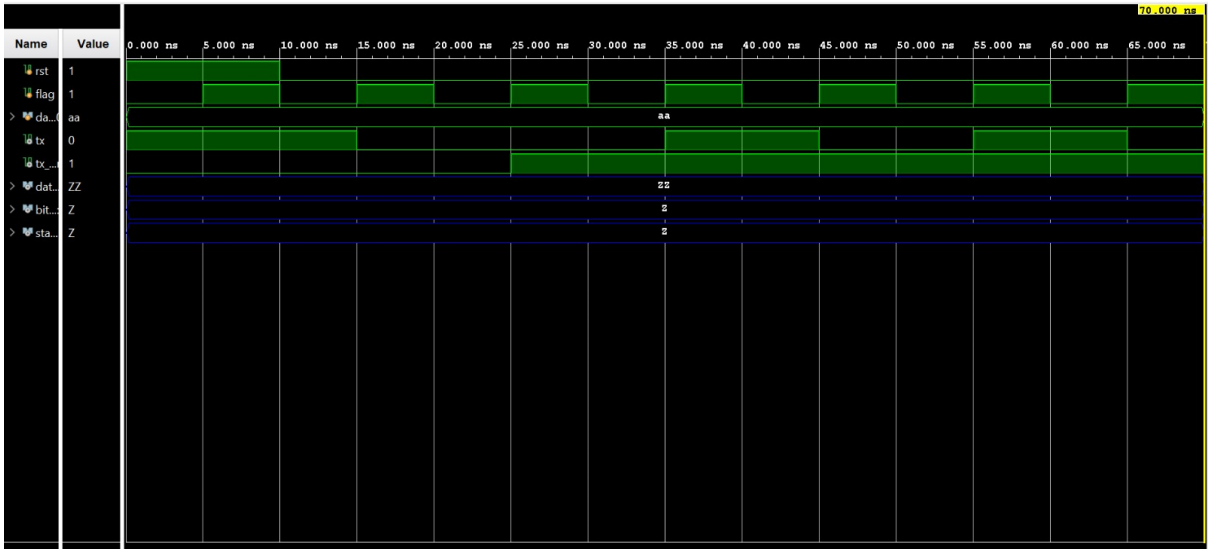
Our SV testbench has been meticulously developed to achieve the following objectives:

1. **Functional Testing:** The SV testbench rigorously evaluates the functionality of our UART system. This verification step aims to guarantee that the UART operates as intended, effectively transmitting and receiving data according to the defined specifications.
2. **Error Handling:** We place a particular emphasis on error handling in this phase. Our testbenches are designed to assess the UART's ability to handle different error conditions, such as noise in the communication channel and data corruption. This ensures the implementation of correct and reliable error-handling mechanisms.
3. **Data Throughput Analysis:** We analyze the data throughput of our UART system using the SV testbenches. This evaluation helps us understand how efficiently the UART transmits data at various speeds and under different load conditions.
4. **Compliance with Standards:** It is of utmost importance that our UART design complies with industry standards and protocols. The SV testbench is used to verify that our UART design aligns with these standards, ensuring compatibility with a wide range of devices and systems.
5. **Performance Under Stress:** Stress testing is another critical aspect of our SV testbench verification. We assess how our UART system performs under extreme conditions, identifying potential performance bottlenecks or vulnerabilities.

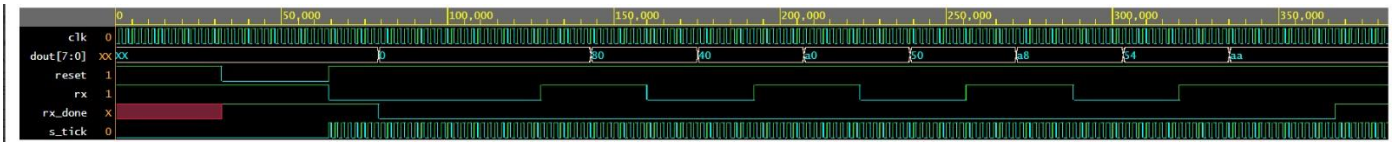
In conclusion, the SV testbench verification phase serves as the final evaluation before our UART design is considered ready for deployment in various digital communication systems. It ensures that our UART system is not only functionally correct but also robust and reliable, meeting the high standards demanded by real-world applications.

9. Results

Part-1: Verilog Based Verification of Each IP Block



Waveform of Transmitter IP Block



Waveform of Receiver IP Block

Waveform of Baud Rate Generator IP Block

Part 2: System Verilog Based Testbench Verification

```
# DRIVER RESET INITIATED
# DRIVER RESET TERMINATED
#        Transaction no. = 0
#        start = 1,      data_in = 9d,   done_tx = 1
# TX PASS
#        Expected data = 24,     Obtained data = 48
# RX FAIL
#        Transaction no. = 1
#        start = 1,      data_in = 3e,   done_tx = 1
# TX PASS
#        Expected data = 81,     Obtained data = 81
# RX PASS
#        Transaction no. = 2
#        start = 1,      data_in = ce,   done_tx = 1
# TX PASS
#        Expected data = 9,      Obtained data = 9
# RX PASS
#        Transaction no. = 3
#        start = 1,      data_in = a9,   done_tx = 1
# TX PASS
#        Expected data = 63,     Obtained data = 63
# RX PASS
#        Transaction no. = 4
#        start = 1,      data_in = ea,   done_tx = 1
# TX PASS
#        Expected data = d,      Obtained data = d

# TX PASS
#        Expected data = d,      Obtained data = d
# RX PASS
#        Transaction no. = 5
#        start = 1,      data_in = 75,   done_tx = 1
# TX PASS
#        Expected data = 8d,     Obtained data = 8d
# RX PASS
#        Transaction no. = 6
#        start = 1,      data_in = 69,   done_tx = 1
# TX PASS
#        Expected data = 65,     Obtained data = 65
# RX PASS
#        Transaction no. = 7
#        start = 1,      data_in = 5e,   done_tx = 1
# TX PASS
#        Expected data = 12,     Obtained data = 12
# RX PASS
#        Transaction no. = 8
#        start = 1,      data_in = 3e,   done_tx = 1
# TX PASS
#        Expected data = 1,      Obtained data = 1
# RX PASS
#        Transaction no. = 9
#        start = 1,      data_in = ad,   done_tx = 1
# TX PASS
#        Expected data = d,      Obtained data = d
# RX PASS
# ** Note: $finish    : uart_env.sv(42)
```

10. Appendix

UART Transmitter Verilog Code:

```
timescale 1ns / 1ps
module transmitter(
    input [7:0] data_in,
    input rst,
    input flag,
    output reg tx,
    output reg tx_done
);

reg [7:0] data_reg; // To hold the data to be transmitted
reg [3:0] bit_count; // Counter to keep track of the number of bits transmitted
reg [2:0] state; // For FSM

parameter idle = 3'b000, start = 3'b001, stop = 3'b010;

always @(posedge flag or posedge rst) // Asynchronous
begin
    if (rst) begin
        data_reg <= 8'b00000000; // Data reg is set to 0 initially
        bit_count <= 4'b0000;   // Initial bit count is 0
        state <= idle;          // Initial state is idle
        tx <= 1'b1;
        tx_done <= 1'b0;
    end
    else begin
        case (state)
            idle:
            begin
                if (tx_done == 1'b0) // If the transmitter is not busy
                begin
                    state <= start; // Go to the start state
```

```
            bit_count <= 4'b0000; // Initialize the counter to 0
            data_reg <= data_in; // Incoming data is stored in data_reg
            tx <= 1'b0; // Transmission line is not in the idle state
        end
    end

    start:
    begin
        tx <= data_reg[0]; // UART transmitter sends 1 data bit at a time starting from the least significant bit
(LSB)
        data_reg <= data_reg >> 1; // Next bit to be transmitted on the next clock cycle
        bit_count <= bit_count + 1; // Keeps track of the number of transmitted bits
        if (bit_count == 4'b1000) // If all 8 bits have been transmitted
        begin
            state <= stop; // Move to the stop state
            bit_count <= 4'b0000; // Reset the counter to 0
        end
    end

    stop:
    begin
        tx <= 1'b1; // Transmit line returns to the idle state
        tx_done <= 1'b0; // Transmitter is not busy
        state <= idle; // Return to the idle state
    end
  endcase
 end
end

// Setting the busy and idle conditions for the transmitter
always @(posedge flag or posedge rst) // Asynchronous
begin
 if (rst) begin
  tx_done <= 1'b0; // When reset, the transmitter is not busy
 end else begin
```

```
        if (state == start || state == stop) begin
            tx_done <= 1'b1; // Transmitter is busy when in the start or stop state
        end
    end
end
endmodule
```

UART Receiver Verilog Code:

```
module uart_rx(
    rx,
    s_tick,
    dout,
    rx_done,
    reset
);
input rx;
input s_tick;
input reset;
output reg [7:0] dout;
output reg rx_done;
reg [3:0] counter;
reg [2:0] state;
reg [3:0] bit_count;
  parameter [1:0]
    IDLE   = 2'd0,
    DATA  = 2'd1,
    STOP  = 2'd2;

always@(posedge s_tick) begin

if(s_tick == 1)
    case(state)
        IDLE:
            if(rx == 0 && counter == 4'd7) begin
```

```
            rx_done <= 0;
            state <= DATA;
            counter <= 0;
            bit_count <= 0;
            dout <= 0;
          end else
           begin
            counter <= counter + 1;
          end
        DATA:
          if(counter == 4'd15) begin
            state <= DATA;
            dout <= {rx, dout[7:1]};
            bit_count <= bit_count + 1;
            counter <= counter + 1;
            if(bit_count == 3'd7) begin
              state <= STOP;
              bit_count <= 0;
              rx_done <= 0;
            end
          end else begin
            counter <= counter + 1;
          end
        STOP:
          if(counter == 4'd15) begin
            rx_done <= 1;
            state <= IDLE;
            counter <= 0;
          end else begin
            counter <= counter + 1;
          end
      endcase
  end
  always@(negedge reset) begin
```

```
            state <= IDLE;
            counter <= 4'b0;
            rx_done <= 1;
end
endmodule
```

Baud Rate Generator Verilog Code:

```
module BaudRateGenerator  #(
    parameter CLOCK_RATE = 100000000, // board internal clock (def == 100MHz)
    parameter BAUD_RATE = 9600
)(
    input wire clk, // board clock
    output reg rxClk, // baud rate for rx
    output reg txClk // baud rate for tx
);
parameter MAX_RATE_RX = CLOCK_RATE / (2 * BAUD_RATE * 16); // 16x oversample
parameter MAX_RATE_TX = CLOCK_RATE / (2 * BAUD_RATE);
parameter RX_CNT_WIDTH = $clog2(MAX_RATE_RX);
parameter TX_CNT_WIDTH = $clog2(MAX_RATE_TX);

reg [RX_CNT_WIDTH - 1:0] rxCounter = 0;
reg [TX_CNT_WIDTH - 1:0] txCounter = 0;

initial begin
    rxClk = 1'b0;
    txClk = 1'b0;
end

always @(posedge clk) begin
    // rx clock
    if (rxCounter == MAX_RATE_RX[RX_CNT_WIDTH-1:0]) begin
        rxCounter <= 0;
        rxClk <= ~rxClk;
    end else begin
```

```
        rxCounter <= rxCounter + 1'b1;
    end
    // tx clock
    if (txCounter == MAX_RATE_TX[TX_CNT_WIDTH-1:0]) begin
        txCounter <= 0;
        txClk <= ~txClk;
    end else begin
        txCounter <= txCounter + 1'b1;
    end
end
endmodule
```

(i) Verilog Based Verification of Each IP Block

UART Transmitter Testbench Code:

```
`timescale 1ns / 1ps
module tb_transmitter;
reg rst,flag;
reg [7:0] data_in;
wire tx, tx_done;
wire [7:0] data_reg;
wire [3:0] bit_count;
wire [2:0] state;

//instatiation of transmitter module
transmitter dut(.data_in(data_in),.rst(rst),.flag(flag),.tx(tx),.tx_done(tx_done));

//generate clk
always
begin
#5 flag=~flag;
end

//test cases
initial begin
```

```
flag=0; rst=1; data_in=8'b10101010; //DA
#10 rst=0;
#60 rst=1;
$stop;
end
endmodule
```

UART Receiver Testbench Code:

```
`timescale 1ns/100ps // 1 ns time unit, 100 ps resolution
module tick_generator(
    input clk,
    input reset,
    output reg s_tick
);
parameter N = 1;
// reg s_tick;
reg [N -1: 0] counter;
always@(clk, negedge reset) begin
  if(~reset)
    counter <= 0;
  else
    counter <= counter + 1;
end
always@(counter) begin
  s_tick <= &counter;
end
always@(negedge clk) begin
    s_tick <= 0;
end
endmodule
module uart_rx_test;
parameter tick = 1;
parameter clock_cycle = tick*32;
reg rx;
```

```
reg clk;
reg reset;
wire s_tick;
wire [7:0] dout;
wire rx_done;
tick_generator
#(.N(1))
gen(
    clk,
    reset,
    s_tick
);
uart_rx uut(
    rx,
    clk,
    dout,
    rx_done,
    reset
);
always #1 clk = !clk;

 initial begin
  clk = 0;
  rx = 1;
  reset = 1;
  #clock_cycle ;
  reset = 0;
  #clock_cycle ;
  rx = 0;
  reset = 1;
  #clock_cycle rx = 1;
  #clock_cycle rx = 0;
  #clock_cycle rx = 1;
  #clock_cycle rx = 0;
```

```
  #clock_cycle rx = 1;
  #clock_cycle rx = 0;
  #clock_cycle rx = 1;
  #clock_cycle rx = 1;
  #clock_cycle rx = 1;
  #clock_cycle
  $display("Received data: %b",dout);
    $finish();
end
  initial begin
  $dumpfile("wave.vcd");
    $dumpvars(0,uart_rx_test);
  end
endmodule
```

Baud Rate Generator Testbench Code:

```
module BaudRateGenerator_tb;
    reg clk;
    wire rxClk;
    wire txClk;

    BaudRateGenerator dut (
        .clk(clk),
        .rxClk(rxClk),
        .txClk(txClk)
    );

    initial begin
        $dumpfile("dump.vcd"); // Specify the VCD file for waveform dumping
        $dumpvars(0, BaudRateGenerator_tb); // Dump all variables in the module
        clk = 0;
        #0.25; // Wait for 0.25 microseconds for half of the clock period
        repeat (20) begin // Simulate for 20 clock cycles
            #0.5; // Toggle clock every 0.5 microseconds for one full clock period
```

```
            clk = ~clk;
        end
        $finish; // End the simulation
    end

    always @(posedge rxClk) begin
        $display("RX Clock: %d", rxClk);
    end

    always @(posedge txClk) begin
        $display("TX Clock: %d", txClk);
    end
endmodule
```

Top Module Verilog Code:

```
module uart_top(
  input clk,
  input rst,
  input [7:0] data_in,
  output tx,
  output rx,
  output [7:0] dout,
  output rx_done
);

  // Instantiate transmitter module
  transmitter u_transmitter (
    .data_in(data_in),
    .rst(rst),
    .clk(clk),
    .tx(tx)
  );

  // Instantiate baud rate generator module for transmitter
```

```
 BaudRateGenerator #(
  .CLOCK_RATE(100000000), // Example clock rate
  .BAUD_RATE(9600)    // Example baud rate
 ) baud_gen_tx (
  .clk(clk),
  .rxClk(),
  .txClk(tx)
 );

 // Instantiate receiver module
 uart_rx u_receiver (
  .rx(rx),
  .s_tick(tx),
  .dout(dout),
  .rx_done(rx_done),
  .reset(rst)
 );

 // Instantiate baud rate generator module for receiver
 BaudRateGenerator #(
  .CLOCK_RATE(100000000), // Example clock rate
  .BAUD_RATE(9600)    // Example baud rate
 ) baud_gen_rx (
  .clk(clk),
  .rxClk(tx),
  .txClk()
 );

endmodule
```

(ii) System Verilog Based Testbench Verification

1. Transaction Class:

```
class uart_trans;
```

```
  bit rx;
  rand bit [7:0] data_in;
  bit tx;
  bit tx_done;
  bit rx_done;
  bit [7:0] dout;
     endclass
```

2. Generator Class:

```
class uart_gen;

  rand uart_trans trans;
  mailbox gen2driv;
  int repeat_count;
  event ended;

  function new(mailbox gen2driv,event ended);
    this.gen2driv = gen2driv;
    this.ended = ended;
  endfunction

  task main;
    repeat(repeat_count) begin
      trans = new();
      if(!trans.randomize())
        $fatal("Randomization failed");
        gen2driv.put(trans);
      end
      -> ended;
    endtask

  endclass
```

3. Driver Class:

```
class uart_driv;

parameter clk_freq = 100000000; //MHz
parameter baud_rate = 9600; //bits per second


  uart_trans trans;
  virtual uart_intf vif;
  mailbox gen2driv;
  int no_transactions;
  reg [7:0] data;

localparam clock_divide = (clk_freq/baud_rate);

  function new(virtual uart_intf vif,mailbox gen2driv);
    this.vif = vif;
    this.gen2driv = gen2driv;
  endfunction
```

```
task reset;
  $display("DRIVER RESET INITIATED, time in ns = %0d",$time);
  wait(vif.rst);
  vif.rx <= 0;
  vif.data_in <= 0;
  vif.flag <= 0;
  wait(!vif.rst);
  $display("DRIVER RESET TERMINATED, time in ns= %0d",$time);
endtask


task main;
  forever begin
    gen2driv.get(trans);

    $display("\t Transaction no. = %0d",no_transactions);
    //Test tx
    vif.start <= 1;
    @(posedge vif.clk);
    vif.data_in <= trans.data_in;
    @(posedge vif.clk);
    wait(vif.tx_done == 1);
    if(vif.tx_done == 1) begin
    $display("\t start = %0b, \t data_in = %0h,\t tx_done = %0b",vif.flag,trans.data_in,vif.tx_done);
    $display("TX PASS");
    end
    else begin
    $display("\t start = %0b, \t data_in = %0h,\t tx_done = %0b",vif.flag,trans.data_in,vif.tx_done);
    $display("TX FAIL");
    end
    repeat(100) @(posedge vif.clk);
    //Test rx
        @(posedge vif.clk);
        data = $random;
        vif.rx <= 1'b0;
        repeat(clock_divide) @(posedge vif.clk);
        for(int i=0;i<8;i++) begin
        vif.rx <= data[i];
        repeat(clock_divide) @(posedge vif.clk);
        end
        vif.rx <= 1'b1;
        repeat(clock_divide) @(posedge vif.clk);
        repeat(100) @(posedge vif.clk);
        $display("\t Expected data = %0h, \t Obtained data = %0h", data,vif.dout);
        begin
        if(vif.dout == data) begin
        $display("RX PASS");

        end
        else begin
        $display("RX FAIL");

      end
end
     no_transactions++;
    end
```

```
    endtask

    endclass
```

4. Interface Class:

```
interface uart_intf(input logic clk,rst);

  logic rx;
      logic [7:0] data_in;
      logic flag;
  logic tx;
      logic [7:0] dout;
      logic tx_done;

 clocking driver_cb @(posedge clk);
   default input #1 output #1;
   output rx;
   output data_in;
   output flag;
   input tx;
   input dout;
   input tx_done;
endclocking

modport DRIVER (clocking driver_cb,input clk,rst);


endinterface
```

5. Monitor Class:

```
class uart_monitor;

  // Monitor ports
  virtual uart_intf vif;
  mailbox mon2scb;

  // Constructor
  function new(virtual uart_intf vif, mailbox mon2scb);
    this.vif = vif;
    this.mon2scb = mon2scb;
  endfunction

  // Task to monitor the transaction
  task main;
    forever begin
      // Wait for the start flag to be asserted
      wait(vif.flag);

      // Create a new transaction
      transaction trans = new();

      // Record the transaction data
      trans.data_in = vif.data_in;
      trans.dout = vif.dout;
```

```
    trans.tx_done = vif.tx_done;
    trans.rx_done = vif.rx_done;

    // Put the transaction in the mailbox
    mon2scb.put(trans);

    // Print the transaction
    trans.display("Monitor");
  end
 endtask

 // Initial block
 initial begin
  // Start the monitor task
  fork
   main();
  join_none
 end

endclass
```

6. Scoreboard Class:

```
class uart_scoreboard;

 // Scoreboard ports
 mailbox mon2scb;
 int no_of_transaction;

 // Constructor
 function new(mailbox mon2scb);
  this.mon2scb = mon2scb;
 endfunction

 // Task to check the correctness of the transaction
 task main;
  forever begin
   // Get the next transaction from the mailbox
   transaction trans;
   mon2scb.get(trans);

   // Check if the transaction is correct
   if (trans.tx_done == 1 && trans.dout == trans.data_in) begin
    // Pass
    $display("Scoreboard: TX transaction passed");
   end else begin
    // Fail
    $display("Scoreboard: TX transaction failed");
   end

   if (trans.rx_done == 1 && trans.dout == trans.data_in) begin
    // Pass
    $display("Scoreboard: RX transaction passed");
   end else begin
    // Fail
    $display("Scoreboard: RX transaction failed");
   end
```

```
        // Increment the number of transactions
        no_of_transaction++;

        // Print the transaction
        trans.display("Scoreboard");
      end
    endtask

    // Initial block
    initial begin
      // Start the scoreboard task
      fork
        main();
      join_none
    end

  endclass
```

7. Environment Class:

```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "scoreboard.sv"

  class uart_env;

    uart_gen gen;
    uart_driv driv;
    uart_cov cov;

    mailbox gen2driv;
    virtual uart_intf vif;
    event ended;


    function new(virtual uart_intf vif);
      this.vif = vif;
      gen2driv = new();
      gen = new(gen2driv,ended);
      driv = new(vif,gen2driv);
      cov = new();
    endfunction

    task pre_test;
      driv.reset();
    endtask

    task test;
      fork
      gen.main();
      driv.main();
      cov.main();
      join_any
```

```
  endtask

  task post_test;
    wait(ended.triggered);
    wait(gen.repeat_count == driv.no_transactions);
  endtask

  task run;
    pre_test();
    test();
    post_test();
    $finish;
  endtask

endclass
```

8. Test Class:

```
program uart_test(uart_intf vif);

  uart_env env;


  initial begin
    env = new(vif);
    env.gen.repeat_count = 10;
    env.run();
  end


endprogram
```

9. Top Module SV tb:

```
    module tb_uart_top;

    parameter clk_freq = 100000000; //MHz
    parameter baud_rate = 9600; //bits per second


      bit clk;
      bit rst;

      uart_intf vif(clk,rst);
      uart_test t1(vif);

            uart #(.clk_freq(clk_freq),.baud_rate(baud_rate))
                dut( .clk(vif.clk),
                    .rst(vif.rst),
                    .rx(vif.rx),
                    .data_in(vif.data_in),
                    .flag(vif.flag),
                    .dout(vif.dout),
                    .tx(vif.tx),
                    .tx_done(vif.tx_done));
```

```
always #50 clk = ~clk;

initial begin
rst = 1;
repeat(2) @(posedge clk);
rst = 0;
end


endmodule
```

Testbench Code:

```
module tb_uart;
 reg clk;
 reg rst;
 wire [7:0] data_in;
 wire tx;
 wire tx_busy;
 wire rx;
 wire s_tick;
 wire [7:0] dout;
 wire rx_done;

 // Instantiate UART Transmitter
 transmitter u_transmitter (
  .data_in(data_in),
  .rst(rst),
  .clk(clk),
  .tx(tx),
  .tx_busy(tx_busy)
 );

 // Instantiate Baud Rate Generator for Transmitter
 BaudRateGenerator #(
  .CLOCK_RATE(100000000), // Example clock rate
  .BAUD_RATE(9600)    // Example baud rate
 ) baud_gen_tx (
```

```
  .clk(clk),
  .rxClk(s_tick),
  .txClk(tx)
);

// Instantiate UART Receiver
uart_rx u_receiver (
  .rx(rx),
  .s_tick(s_tick),
  .dout(dout),
  .rx_done(rx_done),
  .reset(rst)
);

// Instantiate Baud Rate Generator for Receiver
BaudRateGenerator #(
  .CLOCK_RATE(100000000), // Example clock rate
  .BAUD_RATE(9600)    // Example baud rate
) baud_gen_rx (
  .clk(clk),
  .rxClk(s_tick),
  .txClk(rx)
);

// Testbench behavior
initial begin
  // Initialize signals
  clk = 0;
  rst = 1;
  data_in = 8'h55; // Example data to transmit

  // Reset and release reset
  #10 rst = 0;
  #10 rst = 1;
```

```
    // Send data to transmitter
    data_in = 8'hAA; // Change data to another value
    #100; // Wait for some time

    // Check received data
    if (rx_done) begin
      $display("Received Data: %h", dout);
    end else begin
      $display("Data reception not complete.");
    end

    // Finish simulation
    $finish;
  end

  // Clock generation (1 ns period)
  always begin
    #0.5 clk = ~clk;
  end
endmodule
```

10. References

- https://www.ijeat.org/wp-content/uploads/papers/v9i5/E1135069520.pdf
- https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html
- https://www.academia.edu/52477499/Design_and_Verification_of_UART_IP_Core_Using_VMM
- https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf?ts=1696589804649&ref_url=https%253A%252F%252Fwww.google.com%252F
- https://www.geeksforgeeks.org/universal-asynchronous-receiver-transmitter-uart-protocol/
- https://www.linkedin.com/posts/ummidichandrika_uart-protocol-basics-activity-7061020286590087168-I1PZ