122001049 Arun Kumar V

UART

UART, which stands for Universal Asynchronous Receiver/Transmitter, is a widely used serial communication protocol for transmitting and receiving data between two devices. It is a straightforward and versatile method for communicating data over a serial link, making it suitable for a wide range of applications.

CODE

```
// Code your design here
module uart_transmitter_receiver(
  input clk,       // Clock input
  input reset,     // Reset input
  input [7:0] data_in,  // Data to be transmitted from the sender side
  input rx,        // Receiver input (RX)
  output reg tx,   // Transmitter output (TX)
  output reg txd,  // Transmitter data output (TXD)
  output reg rts,  // Request to Send (RTS) output
  input cts        // Clear to Send (CTS) input
);

// Parameters
parameter BAUD_RATE = 9600;
parameter CLOCK_FREQ = 50000000; // Replace with your clock frequency
(e.g., 50MHz)

// States for transmitter and receiver state machines
  parameter [3:0] IDLE       = 4'b0000;
  parameter [3:0] START_BIT  = 4'b0001;
  parameter [3:0] DATA_BITS  = 4'b0010;
  parameter [3:0] PARITY_BIT = 4'b0011;
  parameter [3:0] STOP_BIT   = 4'b0100;
// Transmitter variables
reg [3:0] tx_state;
reg [3:0] tx_bit_counter;
reg [7:0] tx_data;
reg [10:0] tx_baud_timer;
reg tx_transmitting;
```

```verilog
reg [2:0] tx_parity;
reg [2:0] tx_parity_counter;
reg tx_rts;
reg tx_xoff_sent;

// Receiver variables
reg [3:0] rx_state;
reg [3:0] rx_bit_counter;
reg [7:0] rx_data;
reg [10:0] rx_baud_timer;
reg rx_receiving;
reg [2:0] rx_parity;
reg [2:0] rx_parity_counter;
reg rx_xoff_received;
reg rx_rts;

// Baud rate generator
always @(posedge clk) begin
  if (rx_state == IDLE || tx_state == IDLE) begin
    rx_baud_timer <= 0;
    tx_baud_timer <= 0;
  end else begin
    if (rx_baud_timer == (CLOCK_FREQ / BAUD_RATE) - 1) begin
      rx_baud_timer <= 0;
    end else begin
      rx_baud_timer <= rx_baud_timer + 1;
    end

    if (tx_baud_timer == (CLOCK_FREQ / BAUD_RATE) - 1) begin
      tx_baud_timer <= 0;
    end else begin
      tx_baud_timer <= tx_baud_timer + 1;
    end
  end
end

// Transmitter state machine
always @(posedge clk) begin
  if (reset) begin
    tx_state <= IDLE;
    tx_bit_counter <= 0;
    tx_data <= 8'b0;
    tx_transmitting <= 0;
    tx_parity <= 0;
    tx_parity_counter <= 0;
    tx_rts <= 0;
    tx_xoff_sent <= 0;
  end else begin
    if (tx_state == IDLE && tx_transmitting) begin
      tx_state <= START_BIT;
      tx_bit_counter <= 0;
      tx_data <= data_in;
      tx_parity <= 0;
      tx_parity_counter <= 0;
      tx_rts <= cts;
      tx_xoff_sent <= 0;
    end else begin
      case (tx_state)
        IDLE: begin
          tx_state <= tx_transmitting ? START_BIT : IDLE;
        end
        START_BIT: begin
          tx_state <= DATA_BITS;
          tx_bit_counter <= 0;
          tx_parity_counter <= 0;
        end
        DATA_BITS: begin
          if (tx_bit_counter == 7) begin
            tx_state <= PARITY_BIT;
          end else begin
            tx_bit_counter <= tx_bit_counter + 1;
          end
        end
        PARITY_BIT: begin
          if (tx_parity_counter == 2) begin
            tx_state <= STOP_BIT;
          end else begin
            tx_parity_counter <= tx_parity_counter + 1;
          end
        end
        STOP_BIT: begin
          tx_state <= IDLE;
        end
        default: begin
          tx_state <= IDLE;
        end
      endcase
    end
  end
end

// Transmitter data output
always @(posedge clk) begin
  if (reset) begin
    txd <= 1;
  end else begin
    if (tx_state == DATA_BITS) begin
      txd <= tx_data[tx_bit_counter];
    end else if (tx_state == PARITY_BIT) begin
      txd <= tx_parity;
    end else if (tx_state == STOP_BIT) begin
      txd <= 1;
    end else begin
      txd <= 1; // Idle state or Start bit
    end
  end
end

// Transmitter logic
always @(posedge clk) begin
  if (reset) begin
    tx <= 1;
  end else begin
    if (tx_state == START_BIT || tx_state == DATA_BITS ||
        tx_state == PARITY_BIT || tx_state == STOP_BIT) begin
      tx <= 0;
    end else begin
      tx <= 1;
    end
  end
end

// Start transmission on the rising edge of data_in
always @(posedge clk) begin
  if (reset) begin
    tx_transmitting <= 0;
  end else begin
    if (!tx_transmitting && data_in != 8'b0) begin
      tx_transmitting <= 1;
    end
  end
end

// Parity calculation for transmitter
always @(posedge clk) begin
  if (reset) begin
    tx_parity <= 0;
  end else begin
    if (tx_state == DATA_BITS) begin
      tx_parity <= tx_data[0] ^ tx_data[1] ^ tx_data[2] ^
                   tx_data[3] ^ tx_data[4] ^ tx_data[5] ^
                   tx_data[6] ^ tx_data[7];
    end else begin
      tx_parity <= 0;
    end
  end
end

// Receiver state machine
always @(posedge clk) begin
  if (reset) begin
    rx_state <= IDLE;
    rx_bit_counter <= 0;
    rx_data <= 8'b0;
    rx_receiving <= 0;
    rx_parity <= 0;
    rx_parity_counter <= 0;
    rx_xoff_received <= 0;
    rx_rts <= 0;
  end else begin
    if (rx_state == IDLE && rx_receiving) begin
      rx_state <= START_BIT;
      rx_bit_counter <= 0;
      rx_parity <= 0;
      rx_parity_counter <= 0;
    end else begin
      case (rx_state)
        IDLE: begin
          rx_state <= rx_receiving ? START_BIT : IDLE;
        end
        START_BIT: begin
          rx_state <= DATA_BITS;
          rx_bit_counter <= 0;
          rx_parity_counter <= 0;
        end
        DATA_BITS: begin
          if (rx_bit_counter == 7) begin
            rx_state <= PARITY_BIT;
          end else begin
            rx_bit_counter <= rx_bit_counter + 1;
          end
        end
        PARITY_BIT: begin
          if (rx_parity_counter == 2) begin
            rx_state <= STOP_BIT;
          end else begin
            rx_parity_counter <= rx_parity_counter + 1;
          end
        end
        STOP_BIT: begin
          rx_state <= IDLE;
        end
        default: begin
          rx_state <= IDLE;
        end
      endcase
    end
  end
end

// Receiver data input
always @(posedge clk) begin
  if (reset) begin
    rx_data <= 8'b0;
  end else begin
    if (rx_state == DATA_BITS) begin
      rx_data[rx_bit_counter] <= rx;
    end
  end
end

// Receiver logic
always @(posedge clk) begin
  if (reset) begin
    rts <= 0;
  end else begin
    if (rx_state == PARITY_BIT) begin
      rts <= 1;
    end else begin
      rts <= 0;
    end
  end
end

// Start reception on the rising edge of RX
always @(posedge clk) begin
  if (reset) begin
    rx_receiving <= 0;
  end else begin
    if (!rx_receiving && rx == 0) begin
      rx_receiving <= 1;
      rx_xoff_received <= 0;
    end
  end
end

// Parity calculation for receiver
always @(posedge clk) begin
  if (reset) begin
    rx_parity <= 0;
  end else begin
    if (rx_state == DATA_BITS) begin
      rx_parity <= rx_data[0] ^ rx_data[1] ^ rx_data[2] ^
                   rx_data[3] ^ rx_data[4] ^ rx_data[5] ^
                   rx_data[6] ^ rx_data[7] ^ rx;
    end else begin
      rx_parity <= 0;
    end
  end
end

// XON/XOFF flow control
always @(posedge clk) begin
  if (reset) begin
    tx_xoff_sent <= 0;
  end else begin
    if (rx_xoff_received && !tx_xoff_sent) begin
      // Implement logic to handle XOFF
```

```
        tx_transmitting <= 0;
      end else begin
        // Implement logic to handle XON
        if (!tx_transmitting && data_in != 8'b0) begin
          tx_transmitting <= 1;
        end
      end
    end
  end

  // XOFF reception detection
  always @(posedge clk) begin
    if (reset) begin
      rx_xoff_received <= 0;
    end else begin
      if (rx_state == DATA_BITS && rx_data == 8'h13) begin
        rx_xoff_received <= 1;
      end else begin
        rx_xoff_received <= 0;
      end
    end
  end

  endmodule
```

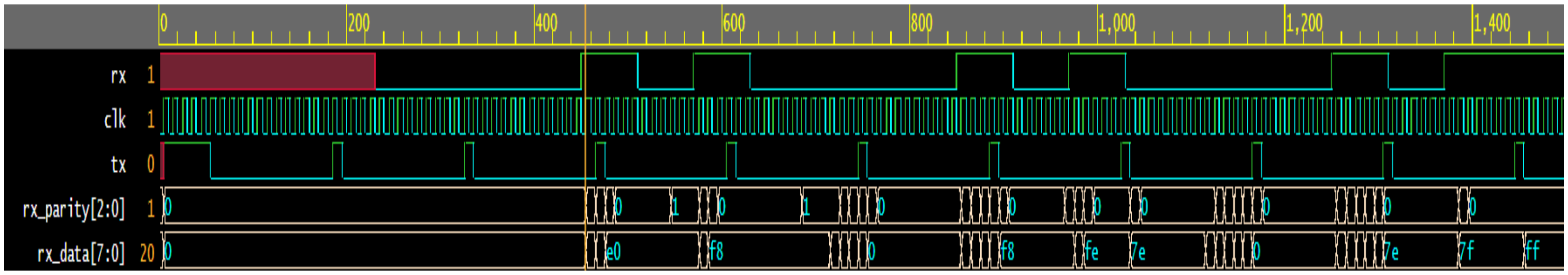
TESTBENCH

```
module tb_uart;

  // Parameters
  parameter BAUD_RATE = 9600;
  parameter CLOCK_PERIOD = 20; // Time in ns for each clock cycle

  // Inputs
  reg clk = 1'b0;
  reg reset;
  reg [7:0] data_in;
  reg rx;
  reg cts;

  // Outputs
```

```
wire tx;
wire txd;
wire rts;

// Instantiate the UART transmitter/receiver
uart_transmitter_receiver uut (
  .clk(clk),
  .reset(reset),
  .data_in(data_in),
  .rx(rx),
  .tx(tx),
  .txd(txd),
  .rts(rts),
  .cts(cts)
);

// Clock generator
always #5 clk = ~clk;

// Test data
reg [7:0] test_data [0:1] = '{8'h45, 8'h6C}; // Send "E" and "l"

initial
  begin
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end

// Testbench behavior
initial begin
  // Reset
  reset = 1;
  cts = 1;
  data_in = 8'b0;
  #10;
  reset = 0;

  // Test transmission
  #20;
  data_in = test_data[0];
  #200; // Wait for the data to be transmitted

  // Test reception
```

```
    rx = 0;
    #220; // Start bit
    rx = 1;
    #60; // Bit 0 - "E"
    rx = 0;
    #60;
    rx = 1;
    #60; // Bit 1
    // ... Continue for all bits (2 to 9) of the character

    // Test flow control (XOFF)
    rx = 0;
    #220; // Start bit
    rx = 1;
    #60; // Bit 0 - XOFF character
    rx = 0;
    #60;
    rx = 1;
    #60; // Bit 1
    // ... Continue for all bits (2 to 9) of the XOFF character

    // Test flow control (XON)
    rx = 0;
    #220; // Start bit
    rx = 1;
    #60; // Bit 0 - XON character
    rx = 0;
    #60;
    rx = 1;
    #60; // Bit 1
    // ... Continue for all bits (2 to 9) of the XON character

    #100;
    $finish;
  end

endmodule
```

OUTPUT


rx 1
clk 1
tx 0
rx_parity[2:0] 1
rx_data[7:0] 20
e0
f8
fe
7e
7f
ff