Experiment - 8: RS232 Transmitter and Receiver Implementation Using Quartus Prime

NAME: Jayakrishnan Menon

REG NO: 22BEC1205

DATE: 12/03/2025

Aim:
Write a Verilog RTL code for Simulation and Implementation of RS232 protocol. Also, perform the simulation of the circuit using Quartus Prime and Model Sim. Finally, implement these circuits on the FPGA kit, "5CSXFC6D6F31C6N"

Software Required: Quartus Prime**,** ModelSim

Haraware Required: Altera Cyclone V 5CSXFC6D6F31C6N

Procedure:

Jayakrishnan Menon 22BEC1205

- Open Quartus Prime 21.1.
- Go to File -> New Project wizard -> select Source folder & type file name -> Next.
- Select Empty Project in Project type -> Next.
- Click on Next in Add Files dialog box.
- Select "Cyclone V SX Extended Features" in Device Family.
- In available devices, select the one ending with "31C6" -> Next.
- Select the tool name as "ModelSim" and Format as "VerilogHDL" in simulation.
- Click on Finish. The project is now created.
- In the Task window, select RTL simulation and run, this would open the ModelSim window.
- Simulate the full adder as you would do using ModelSim by forcing the input values.
- In Compilation -> select compile design.
- Go to Assignments tab -> Pin planner -> Give the location for each i/o pin.
- Go to Hardware Setup -> Select USB.
- Change the file to Fulladder.v in the program/configure option and select Start.

Theory

• UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols. This article 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.

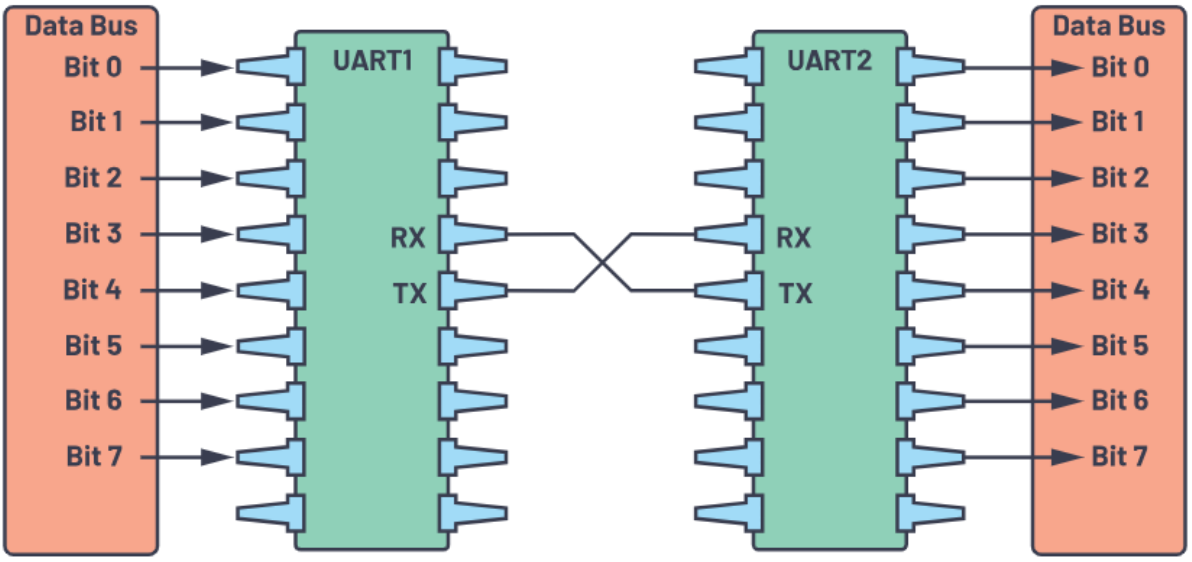
• By definition, UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.

• The two signals of each UART device are named:

Jayakrishnan Menon 22BEC1205

- Transmitter (Tx)
- Receiver (Rx)

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.



Source code and Outputs:

Transmitter:

```
module uart_tx
  #(parameter CLKS_PER_BIT)
  (
   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 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;
```

Jayakrishnan Menon 22BEC1205

```
  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
```

Jayakrishnan Menon 22BEC1205

```
        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;
```

Jayakrishnan Menon 22BEC1205

```
            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
```

Jayakrishnan Menon 22BEC1205

Receiver:

```
module uart_rx
  #(parameter CLKS_PER_BIT)
  (
   input        i_Clock,
   input        i_Rx_Serial,
   output       o_Rx_DV,
   output [7:0] o_Rx_Byte
   );

  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;
```

Jayakrishnan Menon 22BEC1205

```
    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;
```

Jayakrishnan Menon 22BEC1205

```
        // 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;
```

Jayakrishnan Menon 22BEC1205

```
      endcase
    end

  assign o_Rx_DV   = r_Rx_DV;
  assign o_Rx_Byte = r_Rx_Byte;

endmodule // uart_rx
```

Jayakrishnan Menon 22BEC1205

Testbench:

```
`timescale 1ns/10ps

module rsrx ();

  // Testbench uses a 10 MHz clock
  // Want to interface to 115200 baud UART
  // 10000000 / 115200 = 87 Clocks Per Bit.
  parameter c_CLOCK_PERIOD_NS = 100;
  parameter c_CLKS_PER_BIT    = 87;
  parameter c_BIT_PERIOD      = 8600;

  reg r_Clock = 0;
  reg r_Tx_DV = 0;
  wire w_Tx_Done;
  reg [7:0] r_Tx_Byte = 0;
  reg r_Rx_Serial = 1;
  wire [7:0] w_Rx_Byte;


  // Takes in input byte and serializes it
  task UART_WRITE_BYTE;
    input [7:0] i_Data;
    integer     ii;
    begin

      // Send Start Bit
      r_Rx_Serial <= 1'b0;
      #(c_BIT_PERIOD);
      #1000;


      // Send Data Byte
      for (ii=0; ii<8; ii=ii+1)
        begin
          r_Rx_Serial <= i_Data[ii];
          #(c_BIT_PERIOD);
        end

      // Send Stop Bit
      r_Rx_Serial <= 1'b1;
      #(c_BIT_PERIOD);
    end
  endtask // UART_WRITE_BYTE


  uart_rx #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_RX_INST
    (.i_Clock(r_Clock),
     .i_Rx_Serial(r_Rx_Serial),
```

Jayakrishnan Menon 22BEC1205

```
    .o_Rx_DV(),
    .o_Rx_Byte(w_Rx_Byte)
    );

  uart_tx #(.CLKS_PER_BIT(c_CLKS_PER_BIT)) UART_TX_INST
    (.i_Clock(r_Clock),
    .i_Tx_DV(r_Tx_DV),
    .i_Tx_Byte(r_Tx_Byte),
    .o_Tx_Active(),
    .o_Tx_Serial(),
    .o_Tx_Done(w_Tx_Done)
    );


  always
    #(c_CLOCK_PERIOD_NS/2) r_Clock <= !r_Clock;


  // Main Testing:
  initial
    begin

      // Tell UART to send a command (exercise Tx)
      @(posedge r_Clock);
      @(posedge r_Clock);
      r_Tx_DV <= 1'b1;
      r_Tx_Byte <= 8'hAB;   //Change and check
      @(posedge r_Clock);
      r_Tx_DV <= 1'b0;
      @(posedge w_Tx_Done);

      // Send a command to the UART (exercise Rx)
      @(posedge r_Clock);
      UART_WRITE_BYTE(r_Tx_Byte);
      @(posedge r_Clock);

      // Check that the correct command was received
      if (w_Rx_Byte == r_Tx_Byte)
        $display("Test Passed - Correct Byte Received");
      else
        $display("Test Failed - Incorrect Byte Received");
```

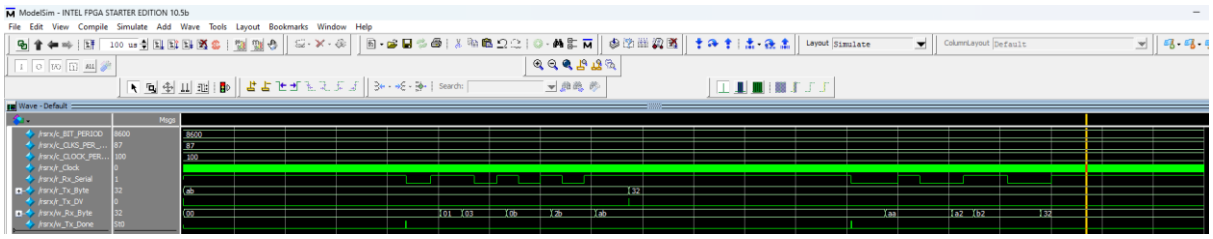
Jayakrishnan Menon 22BEC1205

```
// Tell UART to send a command (exercise Tx)
```

```
    @(posedge r_Clock);
    @(posedge r_Clock);
    r_Tx_DV <= 1'b1;
    r_Tx_Byte <= 8'h32;   //Change and check
    @(posedge r_Clock);
    r_Tx_DV <= 1'b0;
    @(posedge w_Tx_Done);

    // Send a command to the UART (exercise Rx)
    @(posedge r_Clock);
    UART_WRITE_BYTE(r_Tx_Byte);
    @(posedge r_Clock);

    // Check that the correct command was received
    if (w_Rx_Byte == r_Tx_Byte)
      $display("Test Passed - Correct Byte Received");
    else
      $display("Test Failed - Incorrect Byte Received");

  end

endmodule
```

Jayakrishnan Menon 22BEC1205

Output:



Result:

Hence, Verilog RTL codes were successfully verified and implemented using Quartus Prime. The given task was completed successfully

Jayakrishnan Menon 22BEC1205