Department of Electronic & Telecommunication Engineering University of Moratuwa

EN 2111 – Electronics Circuit Design

UART TRANSCEIVER USING FPGA

Index number	Name
200010J	ABSAR M.I.A.
200014B	AHAMED M.B.S.A.
200022X	AMANA M.A.N.

This report is submitted in partial fulfillment of the requirements for the module - EN 2111

July 2023

Abstract

UART (Universal Asynchronous Receiver Transmitter) is a communication protocol used for serial communication between two devices. It is commonly used to transmit and receive data between an FPGA and other devices such as microcontrollers, computers, and sensors. This report provides an overview of UART, its functionality, and code examples for implementing UART in an FPGA using Verilog.

1. Introduction to UART

UART is referred to as Universal Asynchronous Serial Transmitter. It is also referred to as Serial Port, COM port and RS-232 Interface. UART operates in an asynchronous manner, meaning it does not rely on a clock signal to synchronize data transmission. Instead, it sends data one bit at a time over a single wire. UART parameters that need to be set for proper communication include:

- Baud Rate: Determines the rate at which data is transmitted.

- Number of Data Bits: Typically set to 7 or 8.

- Parity Bit: Optional error-checking bit.

- Stop Bits: Number of bits at the end of each data frame.

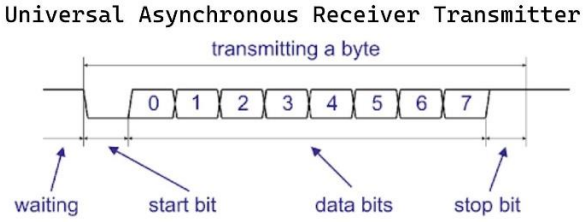
- Flow Control: Optional mechanism for managing data flow.

To recover data correctly it must be sampled, because a clock is not sent along with the data of asynchronous interfaces. Moreover, data sampling rate must be eight times faster than the rate of the data bits. Hence faster sampling clock can be used.

2. Method

Sampling Data in an FPGA:

To receive data correctly in an FPGA, the receiver must sample the data at the right time. In UART, the FPGA continuously samples the line and looks for the start bit (a transition from high to low). After detecting the start bit, the FPGA waits for one-half of a bit period and then samples the data at regular bit intervals based on the baud rate.



The code given below is structured as the shown data stream. This code is for one Start Bit, one Stop Bit, eight Data Bits, and no parity. The transmitter modules below both have a signal called o_tx_active which is used to infer a tri-state buffer for half-duplex communication. The selection of the duplex mode depends on the application.

It needs to be sampled at least eight times faster than the rate of the data bits. This means that for an 115200 baud UART, the data needs to be sampled at at least 921.6 KHz (115200 baud * 8). A faster sampling clock can be used.

Transmitter:

The transmitter converts the 8-bit parallel data into serial data and adds start bit at start of the data frame and stop bit at the end of the data frame.

Serializing Data:

Serializing data refers to the process of converting parallel data into a serial bit stream. In a UART implementation, this is achieved by sending each bit one at a time in a sequential manner. The data is usually transmitted LSB (Least Significant Bit) first, followed by the subsequent bits until all bits are transmitted.

Receiver:

The receiver module in a UART implementation is responsible for converting the received serial data back into parallel data. It performs the reverse process of the transmitter.

3. UART Implementation in Verilog:

The following sections provide codes for implementing UART in Verilog.

Testbench:

A testbench is used to simulate the functionality of the UART transmitter and receiver. The testbench in the appendix simulates both the Transmitter and Receiver code. It works at 115200 baud. A test bench is only used for simulation only and cannot be used to synthesize into functional FPGA code.

RTL Code of UART

```
module uart(input wire [7:0] data_in, //input data
                                input wire wr_en,
                                input wire clear,
                                input wire clk_50m,
                                output wire Tx,
                                output wire Tx_busy,
                                input wire Rx,
                                output wire ready,
                                input wire ready_clr,
                                output wire [7:0] data_out,
                                output [7:0] LEDR,
                                output wire Tx2//output data
                                );
assign LEDR = data_in;
assign Tx2 = Tx;
wire Txclk_en, Rxclk_en;
baudrate uart_baud(     .clk_50m(clk_50m),
                                                        .Rxclk_en(Rxclk_en),
                                                        .Txclk_en(Txclk_en)
                                                        );
transmitter uart_Tx(    .data_in(data_in),
                                                        .wr_en(wr_en),
                                                        .clk_50m(clk_50m),
                                                        .clken(Txclk_en), //We assign Tx clock to enable clock
                                                        .Tx(Tx),
                                                        .Tx_busy(Tx_busy)
                                                        );
receiver uart_Rx(       .Rx(Rx),
                                                .ready(ready),
                                                .ready_clr(ready_clr),
                                                .clk_50m(clk_50m),
                                                .clken(Rxclk_en), //We assign Tx clock to enable clock
                                                .data(data_out)
                                                );

endmodule
```

```
//This is a baud rate generator to divide a 50MHz clock into a 115200 baud Tx/Rx pair.
//The Rx clock oversamples by 16x.

module baudrate  (input wire clk_50m,
                                                output wire Rxclk_en,
                                                output wire Txclk_en
                                                );
//Our Testbench uses a 50 MHz clock.
//Want to interface to 115200 baud UART for Tx/Rx pair
//Hence, 50000000 / 115200 = 435 Clocks Per Bit.
parameter RX_ACC_MAX = 50000000 / (115200 * 16);
parameter TX_ACC_MAX = 50000000 / 115200;
parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;

assign Rxclk_en = (rx_acc == 5'd0);
assign Txclk_en = (tx_acc == 9'd0);

always @(posedge clk_50m) begin
        if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
                rx_acc <= 0;
        else
                rx_acc <= rx_acc + 5'b1; //increment by 00001
end

always @(posedge clk_50m) begin
        if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
                tx_acc <= 0;
        else
                tx_acc <= tx_acc + 9'b1; //increment by 000000001
end

endmodule
```

UART Transmitter

```
module transmitter(     input wire [7:0] data_in, //input data as an 8-bit regsiter/vector
                                                  input wire wr_en, //enable wire to start
                                                  input wire clk_50m,
                                                  input wire clken, //clock signal for the transmitter
                                                  output reg Tx, //a single 1-bit register variable to hold transmitting bit
                                                  output wire Tx_busy //transmitter is busy signal
                                                  );

initial begin
         Tx = 1'b1; //initialize Tx = 1 to begin the transmission
end
//Define the 4 states using 00,01,10,11 signals
parameter TX_STATE_IDLE = 2'b00;
parameter TX_STATE_START        = 2'b01;
parameter TX_STATE_DATA = 2'b10;
parameter TX_STATE_STOP = 2'b11;

reg [7:0] data = 8'h00; //set an 8-bit register/vector as data,initially equal to 00000000
reg [2:0] bit_pos = 3'h0; //bit position is a 3-bit register/vector, initially equal to 000
reg [1:0] state = TX_STATE_IDLE; //state is a 2 bit register/vector,initially equal to 00

always @(posedge clk_50m) begin
        case (state) //Let us consider the 4 states of the transmitter
        TX_STATE_IDLE: begin //We define the conditions for idle  or NOT-BUSY state
                if (~wr_en) begin
                        state <= TX_STATE_START; //assign the start signal to state
                        data <= data_in; //we assign input data vector to the current data
                        bit_pos <= 3'h0; //we assign the bit position to zero
                end
        end
        TX_STATE_START: begin //We define the conditions for the transmission start state
                if (clken) begin
                        Tx <= 1'b0; //set Tx = 0 after transmission has started
                        state <= TX_STATE_DATA;
                end
        end
        TX_STATE_DATA: begin
                if (clken) begin
                        if (bit_pos == 3'h7) //we keep assigning Tx with the data until all bits have been transmitted from 0 to 7
                                state <= TX_STATE_STOP; // when bit position has finally reached 7, assign state to stop transmission
                        else
                                bit_pos <= bit_pos + 3'h1; //increment the bit position by 001
                        Tx <= data[bit_pos]; //Set Tx to the data value of the bit position ranging from 0-7
                end
```

```
         end
         TX_STATE_STOP: begin
                 if (clken) begin
                         Tx <= 1'b1; //set Tx = 1 after transmission has ended
                         state <= TX_STATE_IDLE; //Move to IDLE state once a transmission has been completed
                 end
         end
         default: begin
                 Tx <= 1'b1; // always begin with Tx = 1 and state assigned to IDLE
                 state <= TX_STATE_IDLE;
         end
         endcase
end

assign Tx_busy = (state != TX_STATE_IDLE); //We assign the BUSY signal when the transmitter is not idle

endmodule
```

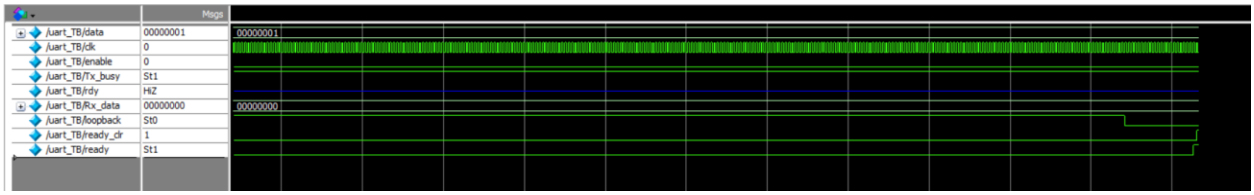
UART Receiver

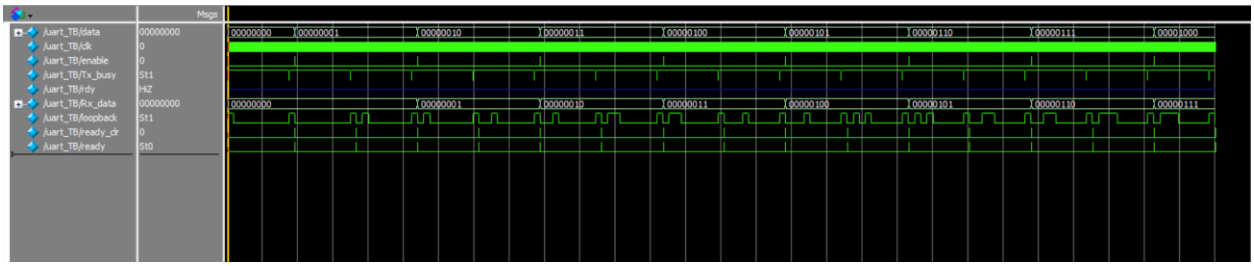
```
module receiver  (input wire Rx,
                                                output reg ready,
                                                input wire ready_clr,
                                                input wire clk_50m,
                                                input wire clken,
                                                output reg [7:0] data
                                                );
initial begin
        ready = 1'b0; // initialize ready = 0
        data = 8'b0; // initialize data as 00000000
end
// Define the 4 states using 00,01,10 signals
parameter RX_STATE_START        = 2'b00;
parameter RX_STATE_DATA         = 2'b01;
parameter RX_STATE_STOP         = 2'b10;

reg [1:0] state = RX_STATE_START; // state is a 2-bit register/vector,initially equal to 00
reg [3:0] sample = 0; // This is a 4-bit register
reg [3:0] bit_pos = 0; // bit position is a 4-bit register/vector, initially equal to 000
reg [7:0] scratch = 8'b0; // An 8-bit register assigned to 00000000

always @(posedge clk_50m) begin
        if (ready_clr)
                ready <= 1'b0; // This resets ready to 0

        if (clken) begin
                case (state) // Let us consider the 3 states of the receiver
                RX_STATE_START: begin // We define condtions for starting the receiver
                        if (!Rx || sample != 0) // start counting from the first low sample
                                sample <= sample + 4'b1; // increment by 0001
                        if (sample == 15) begin // once a full bit has been sampled
                                state <= RX_STATE_DATA; //      start collecting data bits
                                bit_pos <= 0;
                                sample <= 0;
                                scratch <= 0;
                        end
                end
                RX_STATE_DATA: begin // We define conditions for starting the data colleting
                        sample <= sample + 4'b1; // increment by 0001
                        if (sample == 4'h8) begin // we keep assigning Rx data until all bits have 01 to 7
                                scratch[bit_pos[2:0]] <= Rx;
                                bit_pos <= bit_pos + 4'b1; // increment by 0001
                        end
                        if (bit_pos == 8 && sample == 15) // when a full bit has been sampled and
                                state <= RX_STATE_STOP; // bit position has finally reached 7, assign state to stop
                end
                RX_STATE_STOP: begin
                        /*
                         * Our baud clock may not be running at exactly the
                         * same rate as the transmitter.  If we thing that
                         * we're at least half way into the stop bit, allow
                         * transition into handling the next start bit.
                         */
                        if (sample == 15 || (sample >= 8 && !Rx)) begin
                                state <= RX_STATE_START;
                                data <= scratch;
                                ready <= 1'b1;
                                sample <= 0;
                        end
                        else begin
                                sample <= sample + 4'b1;
                        end
                end
                default: begin
                        state <= RX_STATE_START; // always begin with state assigned to START
                end
                endcase
        end
end

endmodule
```

Testbench for UART

```
1  //This is a simple testbench for UART Tx and Rx.
2  //The Tx and Rx pins have been connected together creating a serial loopback.
3  //We check if we receive what we have transmitted by sending incremeting data bytes.
4
5  //It sends out byte 0xAB over the transmitter
6  //It then exercises the receive by receiving byte 0x3F
7  //`include "uart.v"
8
9  module uart_TB();
10
11 reg [7:0] data = 0;
12 reg clk = 0;
13 reg enable = 0;
14
15 wire Tx_busy;
16 wire rdy;
17 wire [7:0] Rx_data;
18
19 wire loopback;
20 reg ready_clr = 0;
21
22 uart test_uart(.data_in(data),
23                                        .wr_en(enable),
24                                        .clk_50m(clk),
25                                        .Tx(loopback),
26                                        .Tx_busy(Tx_busy),
27                                        .Rx(loopback),
28                                        .ready(ready),
29                                        .ready_clr(ready_clr),
30                                        .data_out(Rx_data)
31                                        );
32 initial begin
33         $dumpfile("uart.vcd");
34         $dumpvars(0, uart_TB);
35         enable <= 1'b1;
36         #2 enable <= 1'b0;
37 end
38 always begin
39         #1 clk = ~clk;
40 end
41 always @(posedge ready) begin
42         #2 ready_clr <= 1;
43         #2 ready_clr <= 0;
44         if (Rx_data != data) begin
45                 $display("FAIL: rx data %x does not match tx %x", Rx_data, data);
46                 $finish;
47         end
48         else begin
49                 if (Rx_data == 8'h2) begin //Check if received data is 11111111
50                         $display("SUCCESS: all bytes verified");
51                         $finish;
52                 end
53                 data <= data + 1'b1;
54                 enable <= 1'b1;
55                 #2 enable <= 1'b0;
56         end
57 end
58 endmodule
```

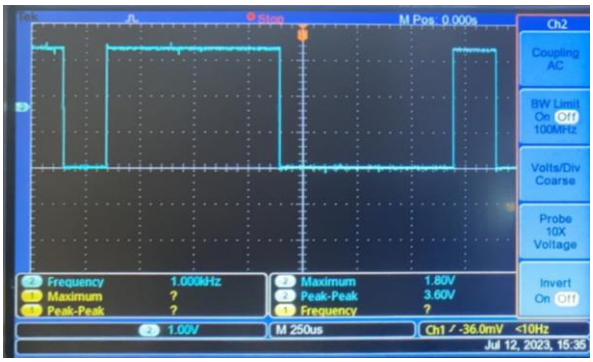
Timing Diagrams





Oscilloscope output

Bit stream: 00001111



Bit stream: 00001010

Bit Stream: 00000101