Contents

1. Introduction

UART stands for ***U***niversal ***S***ynchronous ***R***eceiver ***T***ransmitter. Data can be sent by itself or along with a clock as in synchronous data transmission. Here we do not send a clock, data travels by itself and therefore it is known as Asynchronous.

UART is also commonly known by some other names like:

- Universal Asynchronous Transmitter
- Serial port
- COM port
- RS-232 Interface

UART is one of the simplest ways of connecting the FPGA to a computer using a transmitting and receiving module to send commands to the FPGA and vice-versa.

In this assignment we have implemented UART on an FPGA to transfer between a computer and an FPGA.

Software and Hardware used in this assignment:

1. **'Quartus' 2 web edition version 13.0** – for compiling the RTL code and implementation of the data Transmission

2. Altera DE2-115 Education and Development board with Cyclone IV FPGA device used.

FPGA

FPGA stands for ***F***ield ***P***rogrammable ***G***ate ***A***rray.

It is a very powerful Device used to manipulate digital electronic circuits.

It has millions of logic ICs arranged in a grid pattern, where the connections between them can be changed according to the requirements very easily using programming. We can program them to do almost any digital function.

The FPGA board that was used has a chip manufactured by Altera along with other peripheral devices.

To design the connections between the ICs instructions must be given from the computer, so UART interface is used for the communication purpose.

UART

UART send out ***8*** bits of data at a time over a single wire. Since it is Asynchronous it does not forward a clock along with the data.

UART can be implemented in the following ways:

1. Half duplex – the two transmitters share the same line
2. Full duplex - the two transmitters have dedicated transmission lines

The following parameters can be set by the user and need to the same for the transmitter and the receiver.

- **Baud rate** – the rate at which serial data is sent(e.g. **9600** bits/**s**ec ,19200 bits/**s**ec,115200 bits/**s**ec, others)
- **No of Data bits** – usually **8**
- **Parity bit** – can be on/off (if On, the parity bit is appended to the 8 data bits)
- **Stop Bits(0,1,2) –** usually set to 1
- **Flow Control** (None, On, Hardware) - Not used much, mostly set to 'none'.

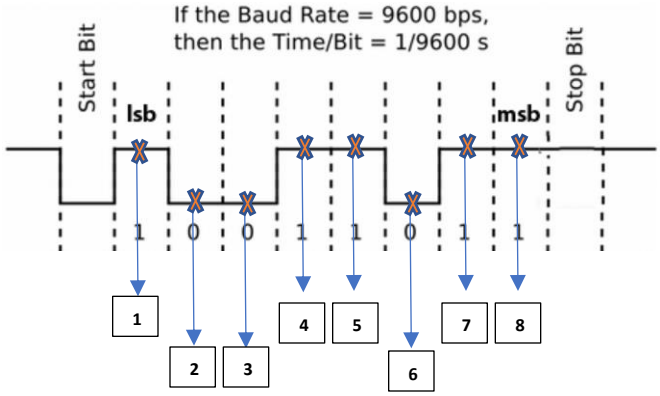
To recover data correctly it must be sampled, because a clock is not sent along with the data of asynchronous interfaces. Moreover, data sampling rate must be eight times faster than the rate of the data bits. Hence faster sampling clock can be used.

2. Method

To receive data correctly, the transmitter and receiver must agree on the Baud Rate. The code given in the appendix a parameter in Verilog determines how many clock cycles per bit which implies the Baud rate.

Receiving part of FPGA

The FPGA samples the line continuously. When it sees a line transition from high to low, it recognizes that a UART data word is coming. This first transition corresponds to the start bit. After this bit is found, the FPGA waits for one **half of a bit period** to ensure that the middle of the data bit gets sampled. Then the FPGA only needs to wait **one-bit period** (as specified by the baud rate) and sample the rest of the data. The UART receiver's operation inside of the FPGA is shown below.



Step1 – Look for a falling edge to identify the start of the word

Step2 – Wait for one-bit period, then sample the first bit after half bit period

Step3 – Sample the data, every 1-bit period(1/9600 in this case)

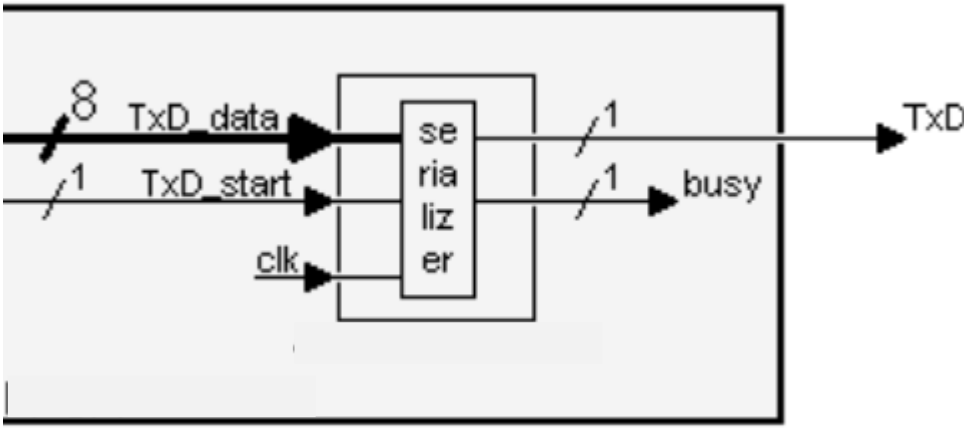
Once 8 sampling is done look for a stop bit(high) in the next bit period and then loop again for the next word

The code given below in the appendix is structured with one start bit, 8 data bits, one stop bit and no parity bit. 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. Selection of the duplex mode depends on the application.

What is a testbench?

A testbench is needed to simulate the code.
In the Appendix is given a testbench for the transmitter and the receiver working at 115,200 baudrate. A test bench is only used for simulation purpose only, not to synthesize into the functional FPGA.

Transmitter



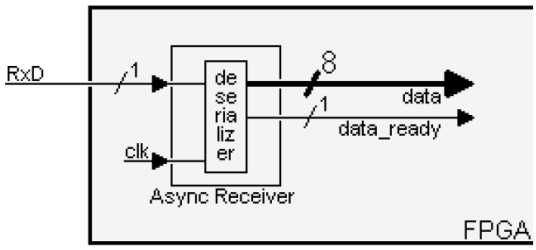
- When the "TxD_start " signal is asserted, the transmitter takes in an 8-bit data and serializes it.
- The "busy" signal is asserted while a transmission occurs (the "TxD_start" signal is ignored during that time).

Serializer

To go through the start bit, the 8 data bits, and the stop bits, a state machine seems appropriate.

Async Receiver

It takes a signal "RxD" from outside the FPGA and "de-serializes" it for easy use inside the FPGA.



- The module assembles data from the RxD line as it comes.
- As a byte is being received, it appears on the "data" bus. Once a complete byte has been received, "data_ready" is asserted for one clock.

Note that "data" is valid only when "data_ready" is asserted. The rest of the time don't use it

- ***Oversampling***

An asynchronous receiver must somehow get in-sync with the incoming signal.
To determine when a new data byte is coming, we look for the "start" bit by oversampling the signal at a multiple of the baud rate frequency.

Once the "start" bit is detected, we sample the line at the known baud rate to acquire the data bits.
The used multiplying factor is 8, therefore for 115200 Bauds, the sampling rate is 921600Hz. Assume that a "Baud8Tick" signal is available, asserted 921600 times a second.

The Design

The incoming "RxD" signal has no relationship with the clock. Two D flip-flops are used to oversample it and synchronize it to the clock domain.

03. References

UART protocol

https://www.nandland.com/articles/what-is-a-uart-rs232-serial.html
https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html
https://www.youtube.com/watch?v=fMmcSpgOtJ4&app=desktop

Code

https://github.com/AntonZero/UART

Appendices

4.1 Appendix 1 - RTL Code

```
module uart(input wire [7:0] data_in, //input data
            input wire wr_en,
            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 data
            );
 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

```

Transmitter

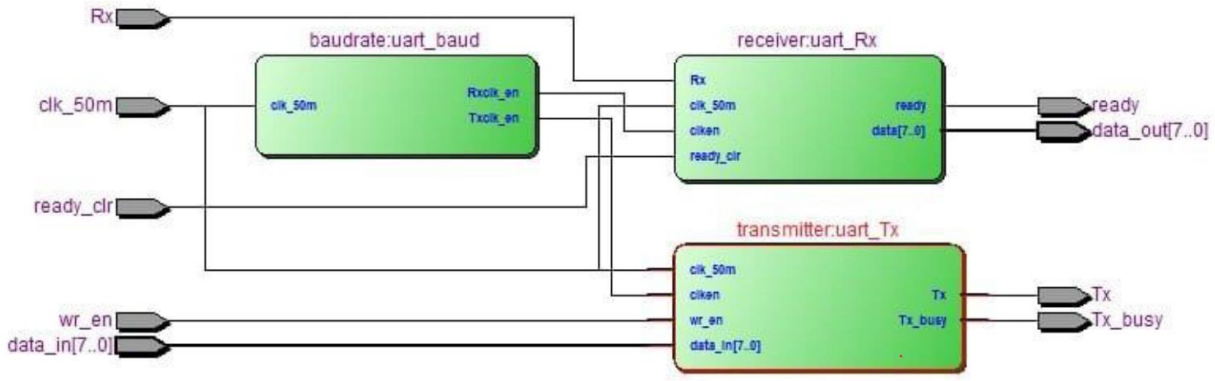
```
module transmitter( input wire [7:0] data_in, //input data as an 8-bit regsiter/vector
                    input wire wr_en, //enable wire to start
                    input wire clk_50m,
                    input wire clken, //clock signal for the transmitter
                    output reg Tx, //a 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_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); //assign the BUSY signal when transmitter is not idle

endmodule
```

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 = 0; // 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 <= 0; // 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

```

Baud rate

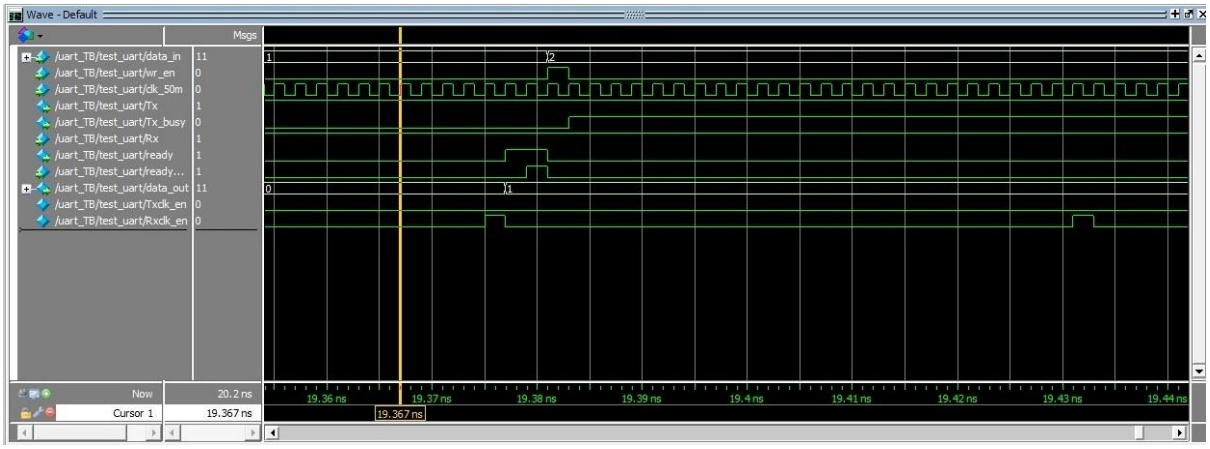
```
1  //This is a baud rate generator to divide a 50MHz clock into a 115200 baud Tx/Rx pair
2  //The Rx clock oversamples by 16x.
3
4  module baudrate(input wire clk_50m,
5                  output wire Rxclk_en,
6                  output wire Txclk_en
7                  );
8  //Our Testbench uses a 50 MHz clock.
9  //Want to interface to 115200 baud UART for Tx/Rx pair
10 //Hence, 50000000 / 115200 = 435 Clocks Per Bit.
11 parameter RX_ACC_MAX = 50000000 / (115200 * 16);
12 parameter TX_ACC_MAX = 50000000 / 115200;
13 parameter RX_ACC_WIDTH = $clog2(RX_ACC_MAX);
14 parameter TX_ACC_WIDTH = $clog2(TX_ACC_MAX);
15 reg [RX_ACC_WIDTH - 1:0] rx_acc = 0;
16 reg [TX_ACC_WIDTH - 1:0] tx_acc = 0;
17
18 assign Rxclk_en = (rx_acc == 5'd0);
19 assign Txclk_en = (tx_acc == 9'd0);
20
21 always @(posedge clk_50m) begin
22     if (rx_acc == RX_ACC_MAX[RX_ACC_WIDTH - 1:0])
23         rx_acc <= 0;
24     else
25         rx_acc <= rx_acc + 5'b1; //increment by 00001
26 end
27
28 always @(posedge clk_50m) begin
29     if (tx_acc == TX_ACC_MAX[TX_ACC_WIDTH - 1:0])
30         tx_acc <= 0;
31     else
32         tx_acc <= tx_acc + 9'b1; //increment by 000000001
33 end
34
35 endmodule
36
```



4.2 Appendix 2 - Testbench for UART

```
//This is a simple testbench for UART Tx and Rx.
//The Tx and Rx pins have been connected together creating a serial loopback.
//We check if we receive what we have transmitted by sending incremeting data bytes.
module uart_TB();

//assign the following parameter values initially
//Inputs are registers
reg [7:0] data = 0;
reg clk = 0;
reg enable = 0;
//Outputs are wires
wire Tx_busy;
wire rdy;
wire [7:0] Rx_data;
//We have connected Tx to Rx using the same wire
wire loopback;
reg ready_clr = 0;
// Instantiating a UART
uart test_uart(.data_in(data),
                    .wr_en(enable),
                    .clk_50m(clk),
                    .Tx(loopback),
                    .Tx_busy(Tx_busy),
                    .Rx(loopback),
                    .ready(ready),
                    .ready_clr(ready_clr),
                    .data_out(Rx_data)
                    );
initial begin //At start of the simulatinon
    $dumpfile("uart.vcd");
    $dumpvars(0, uart_TB);
    enable <= 1'b1; // assign b = 1
    #2 enable <= 1'b0; // 2ps later, assign b = 0
end
always begin //Always keep looping and looping
    #1;
    clk = ~clk; // implement the clock by inverting clock signal every 1ps
end
always @(posedge ready) begin
    #2 ready_clr <= 1; //2ps after positive edge of ready signal, assign ready clear = 1
    #2 ready_clr <= 0; // 2ps later, assign ready clear = 0 as well as ready = 0
    if (Rx_data != data) begin
        $display("FAIL: rx data %x does not match tx %x", Rx_data, data);
        $finish;
    end
    else begin
        if (Rx_data == 8'h14) begin //Check if received data is 10010
            $display("SUCCESS: all bytes verified");
            $finish;
        end
        data <= data + 1'b1; // increment data by 1
        enable <= 1'b1; // assign b = 1
        #2 enable <= 1'b0; // 2ps later, assign b = 0
    end
end
endmodule

```

4.3 Appendix 3 - Timing Diagram



a. Appendix 4 – Results