Department of Electronic & Telecommunication Engineering
University of Moratuwa, Sri Lanka.

UART Implementation

210608J Silva L.J.J.P.
210609M Silva M.K.Y.U.N.
210610H Sirimanna N.T.W.

Submitted in partial fulfillment of the requirements for the module
EN 2111 Electronic Circuit Design

05/08/2024

Contents

1 What is UART?

UART, or universal asynchronous receiver-transmitter, is one of the most widely used device-to-device communication protocols. UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Asynchronous means that there is no clock signal to synchronise the transmitting device's output bits as they travel to the receiving end.

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 need less circuitry and wires, which reduces the cost of implementation.

1. **Data Framing**: In UART communication, data is sent in packets called frames. Each frame typically consists of a start bit, data bits (usually 8 bits), an optional parity bit for error checking, and one or more stop bits. The start bit signals the beginning of a frame, while the stop bit(s) indicate the end, providing synchronization for the data transmission.
2. **Synchronization**: UART is asynchronous, meaning there is no separate clock signal shared between the transmitter and receiver to synchronize their communication. Instead, both devices must agree on a specific baud rate, which determines the speed at which bits are transmitted and received. The receiver uses the start bit to synchronize with the incoming data stream and then samples the bits at the correct intervals based on the baud rate.
3. **Baud Rate**: The baud rate is the rate at which data is transferred in bits per second (bps). It represents the number of times the signal on the communication line changes state per second. Both the transmitter and receiver must be configured to use the same baud rate for successful communication. Common baud rates include 9600, 19200, 38400, and 115200 bps, among others.

2 Phase 1 – Find a Verilog RTL for UART transceiver

2.1 Baudrate Generator

Following Verilog code defines a baud rate generator module that converts a 50MHz clock into a pair of clocks for transmitting and receiving data at a baud rate of 115200. The receiver clock oversamples by 16x. The module maintains two accumulators, `rx_acc` and `tx_acc`, to track the number of clock cycles for the receiver and transmitter clocks, respectively. These accumulators are incremented on each positive edge of the 50MHz clock. When either accumulator reaches its maximum value, it resets to zero, ensuring the clocks are synchronized to the desired baud rate. The `Rxclk_en` and Txclk_en signals indicate when the clocks are enabled for receiving and transmitting data, respectively.

```
// Baudrate generator to divide 50MHz clock to 115200 baud
// RX clk oversamples by 16

module baudTick(CLK, RX_TICK, TX_TICK);
  input  wire CLK;
  output wire RX_TICK;
  output wire TX_TICK;

  // 50,000,000/115,200 = 435 CLK pulses 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 RX_TICK = (rx_acc == 5'd0);
  assign TX_TICK = (tx_acc == 9'd0);

  always @(posedge CLK)
    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)
    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
```

2.2 Selected Transmitter Design

The transmitter is deigned to convert the parallel data to serial form using a shift right register and transmit the data over to the receiver maintaining a fixed baud rate. It also consists of a Tx Enable input which is used as a trigger to begin transmission, a reset input to reset the transmission line and bit counter to keep track of the number of bits transferred.

```
module uartTx(DIN, WR_EN, CLK, CLK_EN, TX, TX_BUSY);
  input  wire [7:0] DIN;  // Input Register
  input  wire WR_EN;    // Enable to start
  input  wire CLK;
  input  wire CLK_EN;   // CLK for the transmitter
  output reg  TX;     // Register to hold the tranmitting bit
  output wire TX_BUSY;    // Busy signal

  initial begin
    TX = 1'b1;           // Initialize TX to 1 to begin transmit
  end

  // TX States
  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;         // Data Register
  reg [2:0] pos   = 3'h0;          // Bit position
  reg [1:0] state = TX_STATE_IDLE; // TX State


  always @(posedge CLK)
    begin
      case (state)
        TX_STATE_IDLE:
          begin
            if (~WR_EN)
              begin
                state <= TX_STATE_START;
                data  <= DIN;  // Get current data in
                pos   <= 3'h0; // Assign bit position to 0
              end
          end

```

```
        TX_STATE_START:
          begin
            if (CLK_EN)
              begin
                TX    <= 1'b0; // Transmission had started
                state <= TX_STATE_DATA;
              end
          end

        TX_STATE_DATA:
          begin
            if (CLK_EN)
              begin
                if (pos == 3'h7) // Assign data till all transmitted
                  state <= TX_STATE_STOP;
                else
                  pos   <= pos + 3'h1; // Increment by 001
                TX <= data[pos];
              end
          end

        TX_STATE_STOP:
          begin
            if (CLK_EN)
              begin
                TX    <= 1'b1; // TX=1, transmit ended
                state <= TX_STATE_IDLE;
              end
          end

        default:
          begin
            TX    <= 1'b1;      // Always begin with TX 1 after transmit end
            state <= TX_STATE_IDLE;
          end
      endcase
    end

    assign TX_BUSY = (state != TX_STATE_IDLE);
endmodule
```

2.3 Selected Receiver Design

The receiver converts the serially received data to a parallel format easy for the computer to process and store. Similar to the Transmitter, this also has a baud counter and a bit counter to synchronise the transmission.

```
module uartRx(RX, READY, READY_CLR, CLK, CLK_EN, DATA);
  input  wire RX;
  output reg  READY;
  input  wire READY_CLR;
  input  wire CLK;
  input  wire CLK_EN;
  output reg [7:0] DATA;

  initial begin
    READY = 1'b0; // initialize READY 0
    DATA  = 8'b0; // initalize DATA 00000000
  end

  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;
  reg [3:0] sample  = 0;
  reg [3:0] pos     = 0;
```

```
21  reg [7:0] scratch = 8'b0;
22
23  always @(posedge CLK)
24    begin
25      if (READY_CLR)
26        READY <= 1'b0;  // Resets ready to 0
27
28      if (CLK_EN)
29        begin
30          case (state)
31            RX_STATE_START:
32              begin
33                if (!RX || sample != 0)     // Start from 1st low sample
34                  sample <= sample + 4'b1; // Invc by 0001
35                if (sample ==15)
36                  begin
37                    state   <= RX_STATE_DATA;
38                    pos     <= 0;
39                    sample  <= 0;
40                    scratch <= 0;
41                  end
42              end
43
44            RX_STATE_DATA:  // Start data collection
45              begin
46                sample  <= sample + 4'b1; //++0001
47                if (sample == 4'h8)
48                  begin
49                    scratch[pos[2:0]] <= RX;
50                    pos               <= pos + 4'b1;
51                  end
52                if (pos == 8 && sample ==15)
53                  state <= RX_STATE_STOP;
54              end
55
56            RX_STATE_STOP:
57              begin
58                if (sample == 15 || (sample >= 8 && !RX))
59                  begin
60                    state  <= RX_STATE_START;
61                    DATA   <= scratch;
62                    READY  <= 1'b1;
63                    sample <= 0;
64                  end
65                else
66                  begin
67                    sample <= sample + 4'b1;
68                  end
69              end
70
71            default:
72              begin
73                state <= RX_STATE_START;
74              end
75          endcase
76        end
77    end
78  endmodule
```

2.4 Transceiver Design

This Verilog module implements a UART transceiver for serial communication. It features components for transmitting and receiving data, utilizing a 50MHz clock. Inputs include data to be transmitted (d`data_in`), write enable (`wr_en`), clear signal (`clear`), and receive signal (`Rx`). Outputs include transmitted data (`Tx`), busy signal (`Tx_busy`), ready signal (`ready`), received data (`data_out`),

LED indicators (`LEDR`), and an extra transmit signal (`Tx2`). The module interfaces with submodules for baud rate generation, transmitting, and receiving to manage UART functionality.

```
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
```

3 Phase 2 – Develop a Testbench

3.1 Testbench Design

The Testbench is created using Verilog and tests the transciever by connecting the transmitter and receiver to each other and simulate actual communication using random data over a fixed baud rate.

```
module testbench();

  reg [7:0] data = 0;
  reg clk = 0;
  reg enable = 0;

  wire Tx_busy;
  wire rdy;
  wire [7:0] Rx_data;

  wire loopback;
  reg ready_clr = 0;

  wire TX2;
  wire [7:0] LEDR;
  wire rxen, txen;
  wire ready;

  uart uart1(
```

```
    .DATA_IN(data),
    .WR_EN(enable),
    .CLK(clk),
    .TX(loopback),
    .TX_BUSY(Tx_busy),
    .RX(loopback),
    .READY(ready),
    .READY_CLR(ready_clr),
    .DATA_OUT(Rx_data),
    .LEDR(LEDR),
    .TX2(TX2),
    .rxen(rxen),
    .txen(txen)
  );

  initial begin
      $dumpfile("uart.vcd");
      $dumpvars(0, testbench);
      enable <= 1'b1;
      #2 enable <= 1'b0;
  end

  always begin
    #1 clk = ~clk;
  end

  always @(posedge ready)
    begin
      #2 ready_clr <= 1;
      #2 ready_clr <= 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'h2)
            begin //Check if received data is 11111111
              $display("SUCCESS: all bytes verified");
              $finish;
            end

          data      <= data + 1'b1;
          enable    <= 1'b1;
          #2 enable <= 1'b0;
        end
    end
endmodule
```

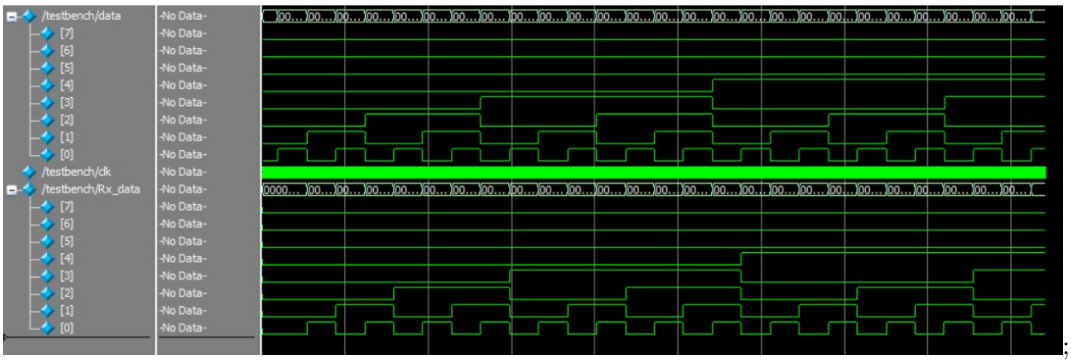

Figure 1: Timing Diagram

4 Phase 3 – Implement RTL Design in FPGA

For this design a DE0-Nano FPGA board was used. On this board, the FPGA chip EP4CE22F17C6N of the family Cyclone IV E has been used.

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strengtl	Slew Rate	Differential Pair
CLK	Input	PIN_R8	3	B3_N0	2.5 V (default)		8mA (default)		
DATA_IN[7]	Input	PIN_B4	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[6]	Input	PIN_B3	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[5]	Input	PIN_A3	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[4]	Input	PIN_A2	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[3]	Input	PIN_C3	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[2]	Input	PIN_B8	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[1]	Input	PIN_D3	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_IN[0]	Input	PIN_A8	8	B8_N0	2.5 V (default)		8mA (default)		
DATA_OUT[7]	Output	PIN_L3	2	B2_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[6]	Output	PIN_B1	1	B1_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[5]	Output	PIN_F3	1	B1_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[4]	Output	PIN_D1	1	B1_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[3]	Output	PIN_A11	7	B7_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[2]	Output	PIN_B13	7	B7_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[1]	Output	PIN_A13	7	B7_N0	2.5 V (default)		8mA (default)	2 (default)	
DATA_OUT[0]	Output	PIN_A15	7	B7_N0	2.5 V (default)		8mA (default)	2 (default)	
LEDR[7]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[6]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[5]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[4]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[3]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[2]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[1]	Output				2.5 V (default)		8mA (default)	2 (default)	
LEDR[0]	Output				2.5 V (default)		8mA (default)	2 (default)	
READY	Output				2.5 V (default)		8mA (default)	2 (default)	
READY_CLR	Input	PIN_J15	5	B5_N0	2.5 V (default)		8mA (default)		
RX	Input	PIN_D12	7	B7_N0	2.5 V (default)		8mA (default)		
TX	Output	PIN_B12	7	B7_N0	2.5 V (default)		8mA (default)	2 (default)	
TX2	Output				2.5 V (default)		8mA (default)	2 (default)	
TX_BUSY	Output				2.5 V (default)		8mA (default)	2 (default)	
WR_EN	Input	PIN_M1	2	B2_N0	2.5 V (default)		8mA (default)		
rxen	Output				2.5 V (default)		8mA (default)	2 (default)	
txen	Output				2.5 V (default)		8mA (default)	2 (default)	
<<new node>>									

;

Figure 2: Pin Planner pin assignment

The clock signal has been provided from the onboard 50MHz clock. As the data input, we have planned to use the pins 0-7 of the GPIO extension header. To display the data output, we have used the onboard LEDs. To connect with another FPGA board we have used pins 39 (`RX`) and 40 (`TX`) on the GPIO pins. For write enable (`WR_EN`) we have used one of the onboard DIP switches, and for the `READY_CLR` we have used the onboard pushbutton switch.