UART Implementation in FPGA

EN2111 - Electronic Circuit Design

Group Members

220004M Aashir M.R.M
220017F Ahilakumaran.T
220036L Anuradha D.M.B.P

University of Moratuwa
Faculty of Engineering
Department of Electronic & Telecommunication Engineering

Contents

1 Abstract

This project focuses on the implementation of a Universal Asynchronous Receiver Transmitter (UART) transceiver using Verilog on an FPGA platform. UART is a fundamental communication protocol widely used in embedded systems for serial data transmission. The objective was to design a UART transceiver at the Register Transfer Level (RTL), verify its functionality through simulation, and demonstrate real-time communication using FPGA hardware. A custom testbench was developed to simulate UART transmission and reception under different test scenarios. The final design was deployed on an FPGA, where it was verified using 7-segment displays in coordination with another group. The output was also validated using an oscilloscope to visualize signal timing and waveform integrity. This report presents the RTL code, testbench, simulation results, hardware snapshots, and timing diagrams, along with a discussion of observed behavior.

2 Introduction

Serial communication plays a critical role in modern digital systems, enabling reliable data exchange between devices using minimal wiring. UART is one of the simplest and most commonly used asynchronous communication protocols. In this assignment, we explore the complete design flow of implementing a UART transceiver in FPGA—from RTL coding and simulation to hardware verification.

In our UART implementation, we designed the system to transmit data using four on-board switches, eliminating the need to modify the code for each new input. For example, to transmit the number 5, we simply set the switches to the binary pattern 0101. This value is then sent via the UART transmitter and displayed on a 7-segment display using the receiver module. Initially, we tested the UART communication between our FPGA board and a laptop to verify basic functionality. Once successful, we conducted a cross-test with another group's FPGA board. In this setup, we transmitted a value from our board, which was received and displayed on their 7-segment display. In response, their system was configured to transmit back double the received value. For instance, if we sent the value 5, their board would send back 10, demonstrating successful bidirectional communication and functional logic on both sides.

3 Block Diagram

The block diagram illustrates the complete UART transceiver system implemented on an FPGA. It consists of separate transmitter and receiver modules (`uart_txtransmitter` and `uart_rxreceiver`), coordinated by a shared `baud_tick` signal generated through a baud rate counter. The transmitter section includes data registers (`tx_data`) and a control signal (`tx_start`) to initiate transmission. On the receiver side, incoming serial data (`rxd`) is synchronized and decoded into parallel form (`rx_data[7:0]`). The received data is displayed using a 7-segment display driver (`seg7_decoderseg_driver`) and also output to `leds[7:0]` for debugging. Control inputs include switches (`sw[3:0]`) for baud rate selection, keys (`key1_n`) for manual triggering, and a reset signal (`rst_n`). The entire system operates based on the main clock (`clk`), ensuring synchronized communication and data visualization.

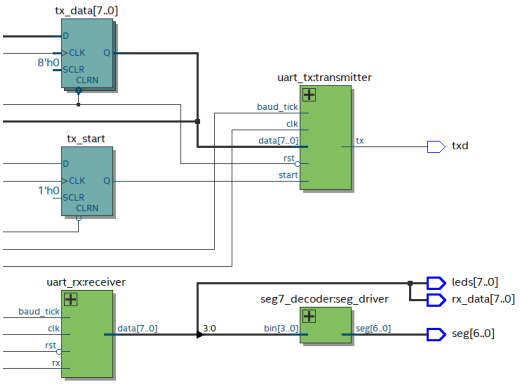

Figure 1: Block Diagram(Zoomed In)

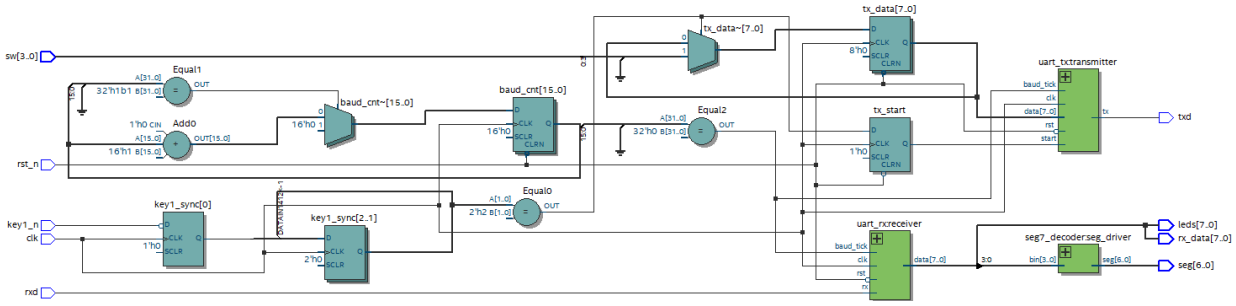

Figure 2: Block Diagram

4 UART Codes

4.1 Code Explanation

The UART (Universal Asynchronous Receiver Transmitter) receiver module is implemented as a simple finite state machine (FSM) with two states: **IDLE** and **READ**.

IDLE State

In the **IDLE** state, the receiver line is held at a high logic level (3.3 V), indicating an idle bus. UART communication begins with the transmission of a start bit, represented by a low-level signal (logic 0). Upon detecting this falling edge (transition from high to low), the state transitions from **IDLE** to **READ**.

READ State

Once the start bit is detected, the FSM enters the **READ** state to begin the data acquisition process. The data reception is synchronized using a separate timing signal known as the *tick*, which operates at 16 times the frequency of the UART baud rate. This oversampling technique allows accurate bit detection at the midpoint of each bit period, improving reliability against noise and signal jitter.

The data reception sequence is as follows:

- **Start Bit:** After detecting the falling edge, the tick counter waits for 8 ticks (half the bit period) to sample the center of the start bit.
- **Data Bits (8 bits):** Each of the 8 data bits is sampled in the middle by counting 16 ticks per bit. This ensures that sampling occurs at the most stable point of the bit.
- **Stop Bit:** After all data bits are read, the receiver waits another 16 ticks to sample the stop bit, which should be a high logic level (logic 1).
- **Return to IDLE:** Once the stop bit is successfully received, the FSM returns to the **IDLE** state, ready to detect the next start bit.

This UART receiver implementation is designed for reliability and synchronization accuracy using oversampling and state-based control, making it suitable for FPGA-based serial communication systems.

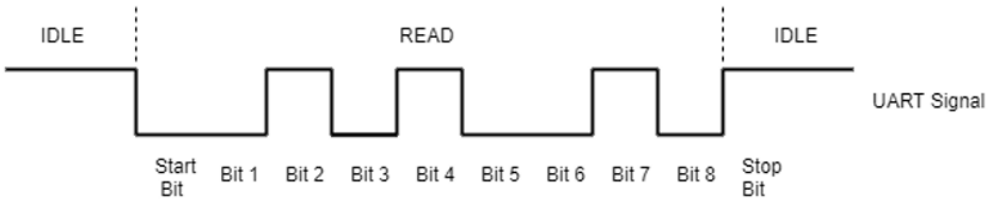

Figure 3: State Diagram(UART Data Frame)

4.2 RTL Code for UART (Top level module)

This is the primary RTL file for the UART transceiver. It instantiates both the (transmitter) `uart_tx` and `uart_rx` (receiver) modules.

```
‘timescale 1ns / 1ps

// ------------------------------
// Top Module: UART Transceiver Test with 7-Segment Display
// ------------------------------
module invert_uart_transceiver_test #(
    parameter CLK_FREQ  = 50000000,
    parameter BAUD_RATE = 115200
)(
    input  wire        clk,
    input  wire        rst_n,       // Active LOW KEY0
    input  wire        key1_n,      // Active LOW KEY1 (transmit
    trigger)
    input  wire [3:0]  sw,          // 4 input switches (high = 1)
    output wire        txd,
    input  wire        rxd,
    output wire [7:0]  rx_data,
    output wire [7:0]  leds,
    output wire [6:0]  seg          // 7-segment output (a-g)
);

    // Invert active-low inputs
    wire rst = ~rst_n;
    wire key1 = ~key1_n;

    // Baud rate tick
    wire baud_tick;
    reg [15:0] baud_cnt = 0;

    // Transmit logic
```

```
    reg tx_start;
    reg [7:0] tx_data;

    // Debounce and edge detect key1
    reg [2:0] key1_sync;
    wire key1_pressed;

    always @(posedge clk) begin
        key1_sync <= {key1_sync[1:0], key1};
    end
    assign key1_pressed = (key1_sync[2:1] == 2'b10);

    // Baud tick generator
    always @(posedge clk or posedge rst) begin
        if (rst)
            baud_cnt <= 0;
        else if (baud_cnt == (CLK_FREQ / BAUD_RATE - 1))
            baud_cnt <= 0;
        else
            baud_cnt <= baud_cnt + 1;
    end
    assign baud_tick = (baud_cnt == 0);

    // Transmit start logic
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            tx_start <= 0;
            tx_data  <= 8'h00;
        end else begin
            tx_start <= key1_pressed;
            if (key1_pressed)
                tx_data <= {4'b0000, sw}; // Upper nibble = 0000,
    lower nibble = switch input
        end
    end

    // UART transmitter
    uart_tx transmitter (
        .clk(clk),
        .rst(rst),
        .start(tx_start),
        .data(tx_data),
        .baud_tick(baud_tick),
```

```
        .tx(txd)
    );

    // UART receiver
    uart_rx receiver (
        .clk(clk),
        .rst(rst),
        .rx(rxd),
        .baud_tick(baud_tick),
        .data(rx_data)
    );

    // Output to LEDs
    assign leds = rx_data;

    // Show rx_data[3:0] on 7-segment display
    seg7_decoder seg_driver (
        .bin(rx_data[3:0]),
        .seg(seg)
    );

endmodule

// ----------------------------
// 7-Segment Display Decoder
// ----------------------------
module seg7_decoder (
    input  wire [3:0] bin,
    output reg  [6:0] seg
);
    always @(*) begin
        case (bin)
            4'h0: seg = 7'b1111110;
            4'h1: seg = 7'b0110000;
            4'h2: seg = 7'b1101101;
            4'h3: seg = 7'b1111001;
            4'h4: seg = 7'b0110011;
            4'h5: seg = 7'b1011011;
            4'h6: seg = 7'b1011111;
            4'h7: seg = 7'b1110000;
            4'h8: seg = 7'b1111111;
            4'h9: seg = 7'b1111011;
            4'hA: seg = 7'b1110111;
```

```
            4'hB: seg = 7'b0011111;
            4'hC: seg = 7'b1001110;
            4'hD: seg = 7'b0111110;
            4'hE: seg = 7'b1001111;
            4'hF: seg = 7'b1000111;
            default: seg = 7'b0000000;
        endcase
    end
endmodule
```

4.2.1 Transmitter Module

Implements the UART transmitter, responsible for serializing data with start and stop bits.

```
`timescale 1ns / 1ps

module uart_tx (
    input  wire clk,
    input  wire rst,
    input  wire start,
    input  wire [7:0] data,
    input  wire baud_tick,
    output reg  tx
);

    reg [3:0] bit_index = 0;
    reg [9:0] shift_reg = 10'b1111111111;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            tx <= 1'b1;                         // Idle state of TX
    line is high
            bit_index <= 0;
            shift_reg <= 10'b1111111111;
        end else begin
            // Start a new transmission only if idle (bit_index ==
    0)
            if (start && bit_index == 0) begin
                shift_reg <= {1'b1, data, 1'b0}; // Stop bit, data,
     start bit
                bit_index <= 1;                  // Start counting
    from 1 (transmitting)
```

```
            end else if (baud_tick && bit_index != 0) begin
                tx <= shift_reg[0];                     // Send
    LSB
                shift_reg <= {1'b1, shift_reg[9:1]};    // Shift
    right
                if (bit_index == 10)                    // All 10
    bits sent
                    bit_index <= 0;                     // Reset
    to idle
                else
                    bit_index <= bit_index + 1;
            end
        end
    end
endmodule
```

4.2.2 Receiver Module

Implements the UART receiver, responsible for deserializing incoming data and detecting start/stop bits.

```
`timescale 1ns / 1ps

module uart_rx (
    input  wire clk,
    input  wire rst,
    input  wire rx,
    input  wire baud_tick,
    output reg [7:0] data
);

    reg [3:0] bit_index = 0;
    reg [7:0] shift_reg = 0;
    reg [1:0] state = 0;
    reg rx_sync = 1;

    localparam IDLE  = 0,
               START = 1,
               DATA  = 2,
               STOP  = 3;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
```

```
            state <= IDLE;
            bit_index <= 0;
            shift_reg <= 0;
            data <= 0;
            rx_sync <= 1;
        end else begin
            rx_sync <= rx;

            case (state)
                IDLE: begin
                    if (!rx_sync)  // start bit detected
                        state <= START;
                end

                START: begin
                    if (baud_tick) begin
                        if (!rx_sync) begin
                            state <= DATA;
                            bit_index <= 0;
                        end else begin
                            state <= IDLE; // false start
                        end
                    end
                end

                DATA: begin
                    if (baud_tick) begin
                        shift_reg[bit_index] <= rx_sync;
                        if (bit_index == 7)
                            state <= STOP;
                        bit_index <= bit_index + 1;
                    end
                end

                STOP: begin
                    if (baud_tick) begin
                        if (rx_sync) begin
                            data <= shift_reg;  // accept byte
                        end
                        state <= IDLE;
                    end
                end
            endcase
```

```
        end
    end
endmodule
```

4.3 Test Bench of Top Level Module

This is a testbench for the UART transmitter (`uart_tx`) and receiver (`uart_rx`) modules.

```
`timescale 1ns / 1ps

module tb_invert_uart_transceiver_test;

    // Parameters
    parameter CLK_FREQ  = 50000000;
    parameter BAUD_RATE = 115200;
    parameter CLK_PERIOD = 20; // 50MHz

    // DUT I/O
    reg clk;
    reg rst_n;
    reg key1_n;
    reg [3:0] sw;
    wire txd;
    wire rxd;
    wire [7:0] rx_data;
    wire [7:0] leds;
    wire [6:0] seg;

    // Internal loopback wire
    assign rxd = txd;  // Loopback mode for testing

    // Instantiate DUT
    invert_uart_transceiver_test #(
        .CLK_FREQ(CLK_FREQ),
        .BAUD_RATE(BAUD_RATE)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .key1_n(key1_n),
        .sw(sw),
        .txd(txd),
        .rxd(rxd),
```

```
        .rx_data(rx_data),
        .leds(leds),
        .seg(seg)
    );

    // Clock generation
    always #(CLK_PERIOD/2) clk = ~clk;

    initial begin
        $display("Starting UART Transceiver Testbench...");

        // Initialize
        clk     = 0;
        rst_n   = 0;
        key1_n  = 1;
        sw      = 4'b0000;

        // Reset the system
        #100;
        rst_n = 1;

        // Wait for a while
        #200;

        // Set the switch to 4'b0011 (decimal 3)
        sw = 4'b0011;

        // Trigger transmission (falling edge of key1_n)
        key1_n = 0;
        #40;          // Keep low for a short time
        key1_n = 1;

        // Wait for transmission and reception
        #100000;      // Enough time for 10 UART bits x bit time
    (~87us)

        // Display received data and 7-segment value
        $display("Received Data = %h", rx_data);
        $display("7-Segment Output = %b", seg);

        // End simulation
        #1000;
        $finish;
```

```
    end

endmodule
```

4.3.1 Operation Varification Crosscheck

It enables one FPGA board to act as a transmitter and another as a receiver, ensuring that the UART protocol implementation works correctly across hardware setups.

```
// ----------------------------
// Top Module: UART Transceiver Test with 7-Segment Display
// ----------------------------
module invert_uart_transceiver_test_crosscheck #(
    parameter CLK_FREQ  = 50000000,
    parameter BAUD_RATE = 115200
)(
    input  wire        clk,
    input  wire        rst_n,       // Active LOW KEY0
    input  wire        key1_n,      // Active LOW KEY1 (not used
   now)
    input  wire [3:0]  sw,          // 4 input switches (not used
   now)
    output wire        txd,
    input  wire        rxd,
    output wire [7:0]  rx_data,
    output wire [7:0]  leds,
    output wire [6:0]  seg          // 7-segment output (a-g)
);

    wire rst = ~rst_n;

    // Baud rate tick
    wire baud_tick;
    reg [15:0] baud_cnt = 0;

    always @(posedge clk or posedge rst) begin
        if (rst)
            baud_cnt <= 0;
        else if (baud_cnt == (CLK_FREQ / BAUD_RATE - 1))
            baud_cnt <= 0;
        else
```

```
        baud_cnt <= baud_cnt + 1;
end
assign baud_tick = (baud_cnt == 0);

// UART receiver signals
wire [7:0] uart_rx_data;
reg [7:0] rx_data_reg;
assign rx_data = rx_data_reg;

// UART receiver
uart_rx receiver (
    .clk(clk),
    .rst(rst),
    .rx(rxd),
    .baud_tick(baud_tick),
    .data(uart_rx_data)
);

// Edge detector for new data (basic method)
reg [7:0] prev_uart_rx_data;
wire new_data_received;
assign new_data_received = (uart_rx_data != prev_uart_rx_data);

// Transmit logic
reg tx_start = 0;
reg [7:0] tx_data;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        prev_uart_rx_data <= 8'h00;
        rx_data_reg       <= 8'h00;
        tx_data           <= 8'h00;
        tx_start          <= 1'b0;
    end else begin
        tx_start <= 1'b0; // default

        if (new_data_received) begin
            prev_uart_rx_data <= uart_rx_data;
            rx_data_reg       <= uart_rx_data;

            // Process: extract lower 4 bits, multiply by 2
            tx_data  <= {4'b0000, uart_rx_data[3:0]} << 1;
            tx_start <= 1'b1; // trigger transmit
```

```
            end
        end
    end

    // UART transmitter
    uart_tx transmitter (
        .clk(clk),
        .rst(rst),
        .start(tx_start),
        .data(tx_data),
        .baud_tick(baud_tick),
        .tx(txd)
    );

    // Output to LEDs and 7-segment display
    assign leds = rx_data_reg;
    seg7_decoder seg_driver (
        .bin(rx_data_reg[3:0]),
        .seg(seg)
    );

endmodule
```

5 Testing

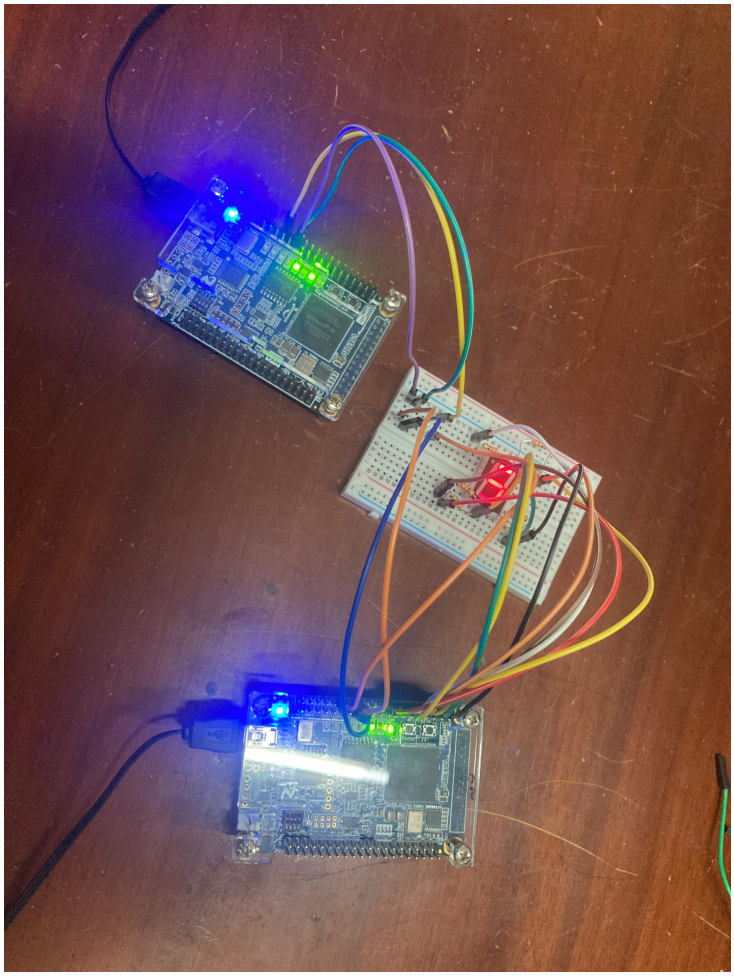

Figure 4: Cross-verification of UART communication between two FPGA boards: one board transmits the value 5, and the second board receives and processes it to transmit back double the value (10), which is displayed on the 7-segment display.

6 ModelSim Simulation and Timing Diagram

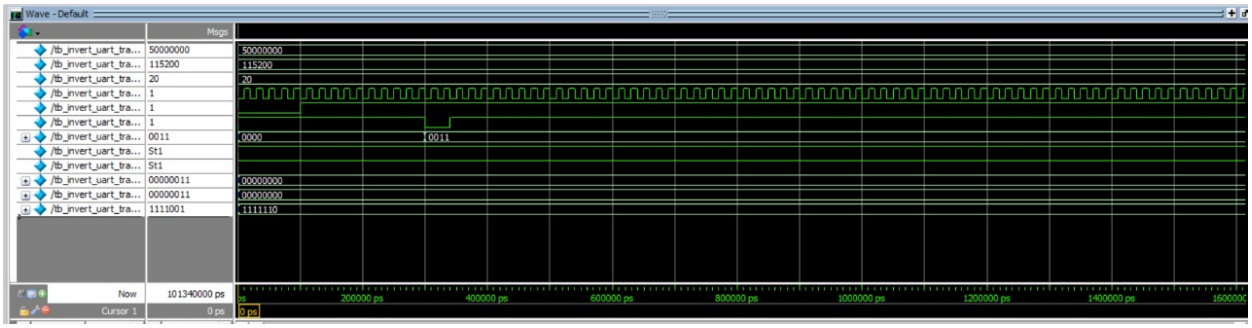

Figure 5: when initiating the transmission

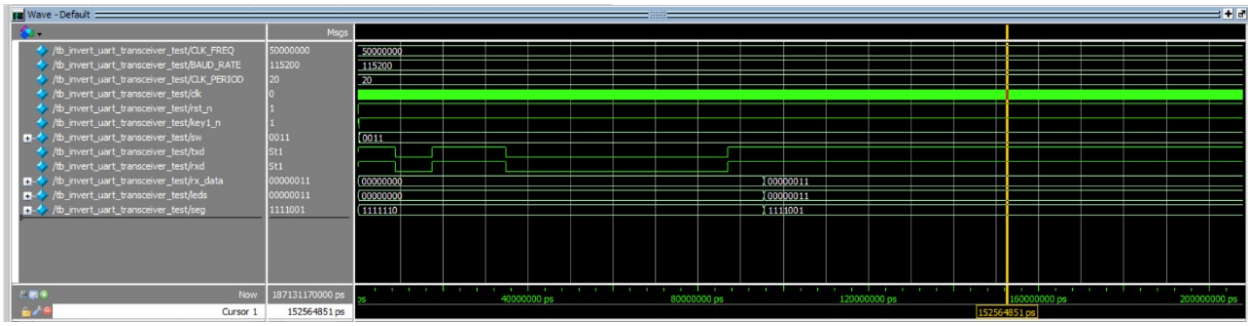

Figure 6: During the transmission

7 Pin Planner

Node Name	Direction	Location	I/O Bank	VREF Group	itter Location	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
clk	Input	PIN_R8	3	B3_N0	PIN_R8	2.5 V		8mA (default)			
key1_n	Input	PIN_E1	1	B1_N0	PIN_E1	2.5 V		8mA (default)			
leds[7]	Output	PIN_L3	2	B2_N0	PIN_L3	2.5 V		8mA (default)	2 (default)		
leds[6]	Output	PIN_B1	1	B1_N0	PIN_B1	2.5 V		8mA (default)	2 (default)		
leds[5]	Output	PIN_F3	1	B1_N0	PIN_F3	2.5 V		8mA (default)	2 (default)		
leds[4]	Output	PIN_D1	1	B1_N0	PIN_D1	2.5 V		8mA (default)	2 (default)		
leds[3]	Output	PIN_A11	7	B7_N0	PIN_A11	2.5 V		8mA (default)	2 (default)		
leds[2]	Output	PIN_B13	7	B7_N0	PIN_B13	2.5 V		8mA (default)	2 (default)		
leds[1]	Output	PIN_A13	7	B7_N0	PIN_A13	2.5 V		8mA (default)	2 (default)		
leds[0]	Output	PIN_A15	7	B7_N0	PIN_A15	2.5 V		8mA (default)	2 (default)		
rst_n	Input	PIN_J15	5	B5_N0	PIN_J15	2.5 V		8mA (default)			
rx_data[7]	Output				PIN_J1	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[6]	Output				PIN_F2	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[5]	Output				PIN_C2	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[4]	Output				PIN_G5	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[3]	Output				PIN_C11	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[2]	Output				PIN_A14	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[1]	Output				PIN_B14	2.5 V ...fault)		8mA (default)	2 (default)		
rx_data[0]	Output				PIN_C9	2.5 V ...fault)		8mA (default)	2 (default)		
rxd	Input	PIN_B6	8	B8_N0	PIN_B6	2.5 V		8mA (default)			
seg[6]	Output	PIN_D11	7	B7_N0	PIN_D11	2.5 V		8mA (default)	2 (default)		
seg[5]	Output	PIN_B11	7	B7_N0	PIN_B11	2.5 V		8mA (default)	2 (default)		
seg[4]	Output	PIN_E10	7	B7_N0	PIN_E10	2.5 V		8mA (default)	2 (default)		
seg[3]	Output	PIN_D9	7	B7_N0	PIN_D9	2.5 V		8mA (default)	2 (default)		
seg[2]	Output	PIN_E9	7	B7_N0	PIN_E9	2.5 V		8mA (default)	2 (default)		
seg[1]	Output	PIN_F8	8	B8_N0	PIN_F8	2.5 V		8mA (default)	2 (default)		
seg[0]	Output	PIN_D8	8	B8_N0	PIN_D8	2.5 V		8mA (default)	2 (default)		
seg[0]	Output	PIN_D8	8	B8_N0	PIN_D8	2.5 V		8mA (default)	2 (default)		
sw[3]	Input	PIN_M15	5	B5_N0	PIN_M15	2.5 V		8mA (default)			
sw[2]	Input	PIN_B9	7	B7_N0	PIN_B9	2.5 V		8mA (default)			
sw[1]	Input	PIN_T8	3	B3_N0	PIN_T8	2.5 V		8mA (default)			
sw[0]	Input	PIN_M1	2	B2_N0	PIN_M1	2.5 V		8mA (default)			
txd	Output	PIN_A6	8	B8_N0	PIN_A6	2.5 V		8mA (default)	2 (default)		
<<new node>>											

Figure 7: Pin Planner