Asynchronous UART Design with FIFO and Baud Rate Generator

“Implemented with Verilog”

Mahmoud Ghamry

Feb 2025

Table of Contents

Table of Figures

Introduction

In the realm of digital communication systems, asynchronous serial interfaces remain a cornerstone for enabling reliable data exchange between heterogeneous devices operating at disparate clock domains. Among these, the Universal Asynchronous Receiver-Transmitter (UART) stands out as a ubiquitous and versatile protocol, prized for its simplicity, low hardware footprint, and adaptability to diverse applications—from embedded systems to industrial automation. However, the inherent challenges of asynchronous communication, such as timing synchronization, data integrity, and throughput optimization, demand sophisticated architectural enhancements to meet the rigorous demands of modern electronics. This report delves into the intricacies of a UART system augmented with First-In-First-Out (FIFO) buffers and dedicated baud rate generators for its receiver (Rx) and transmitter (Tx) modules, unraveling the interplay of these components in achieving robust, high-performance serial communication.

At its core, UART relies on precise timing coordination governed by a baud rate—the frequency at which data bits are transmitted and received. Deviations in timing, even by marginal degrees, can precipitate catastrophic errors in data interpretation. To mitigate this, a dedicated baud rate generator is employed, synthesizing clock signals from a system's master clock to establish synchronized, independent timing domains for Rx and Tx operations. This segregation ensures that sampling and transmission occur at harmonized yet isolated rates, eliminating skew-induced errors and enabling full-duplex communication. Meanwhile, FIFO buffers act as critical intermediaries, decoupling data production from consumption. By temporarily storing incoming and outgoing data streams, FIFOs absorb timing mismatches, prevent overflows, and reduce CPU overhead, thereby enhancing system efficiency and scalability.

This report explores the symbiotic relationship between these elements: the baud rate generator's role in maintaining temporal fidelity, the FIFO's function in buffering data bursts, and the UART's inherent asynchrony that liberates systems from rigid clock dependencies. Through a detailed examination of their design principles, operational workflows, and performance trade-offs, this study illuminates how the integration of FIFOs and precision baud rate generation elevates UART from a rudimentary serial interface to a resilient, high-throughput communication backbone. By dissecting the challenges of jitter tolerance, buffer depth optimization, and clock division accuracy, this analysis aims to provide a comprehensive understanding of modern UART architectures, underscoring their enduring relevance in an era dominated by high-speed, low-latency connectivity demands.

Design Methodology

Transmission with 8 data bits, no parity, and 1 stop bit is shown in Figure 1. Note that the LSB of the data word is transmitted first.

No clock information is conveyed through the serial line. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits and stop bits, and use of the parity bit. The commonly used baud rates are 2400,4800, 9600, and 19,200 bauds.

We illustrate the design of the receiving and transmitting subsystems in the following sections. The design is customized for a UART with a 19,200 baud rate, 8 data bits, 1 stop bit, and no parity bit.

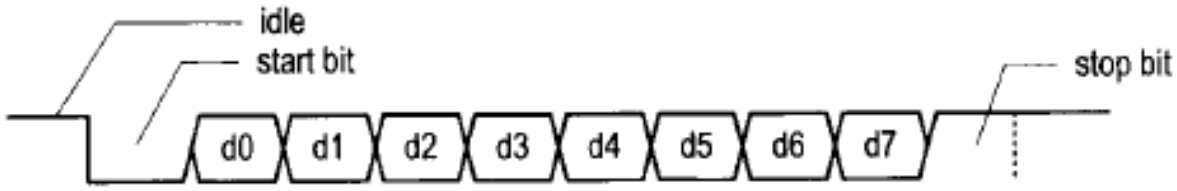

Figure 1

UART Receiving Subsystem

Since no clock information is conveyed from the transmitted signal, the receiver can retrieve the data bits only by using the predetermined parameters. We use an ***oversampling scheme*** to estimate the middle points of transmitted bits and then retrieve them at these points accordingly.

UART Transmitting Subsystem

The organization of a UART transmitting subsystem is similar to that of the receiving subsystem. It consists of a UART transmitter, baud rate generator, and interface circuit.

The interface circuit is like that of the receiving subsystem except that the main system sets the flag FF or writes the FIFO buffer, and the UART transmitter clears the flag FF or reads the FIFO buffer.

The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. The rate can be controlled by one-clock-cycle enabled ticks generated by the baud rate generator. Because no oversampling is involved, the frequency of the ticks is 16 times slower than that of the UART receiver. Instead of introducing a new counter, UART Transmitter usually shares the baud rate generator of the UART receiver and uses an internal counter to keep track of the number of enable ticks. A bit is shifted out every 16 enable ticks.

The ASMD chart of the UART transmitter is like that of the UART receiver.

After assertion of the tx-start signal, the FSMD loads the data and then gradually progresses through the start, data, and stop states to shift out the corresponding bits.

It signals completion by asserting the tx-done-tick signal for a one clock cycle. A 1-bit buffer, tx-reg, is used to filter out any potential glitch.

Oversampling Procedure

The most used sampling rate is 16 times the baud rate, which means that each serial bit is sampled 16 times. Assume that the communication uses N data bits and *M* stop bits. The oversampling scheme works as follows:

1. Wait until the incoming signal becomes 0, the beginning of the start bit, and then start the sampling tick counter.
2. When the counter reaches 7, the incoming signal reaches the middle point of the start bit. Clear the counter to 0 and restart.
3. When the counter reaches 15, the incoming signal progresses for one bit and reaches the middle of the first data bit. Retrieve its value, shift it into a register, and restart the counter.
4. Repeat step 3 N-1 more times to retrieve the remaining data bits.
5. If the optional parity bit is used, repeat step 3 one time to obtain the parity bit.
6. Repeat step **3** *hf* more times to obtain the stop bits.

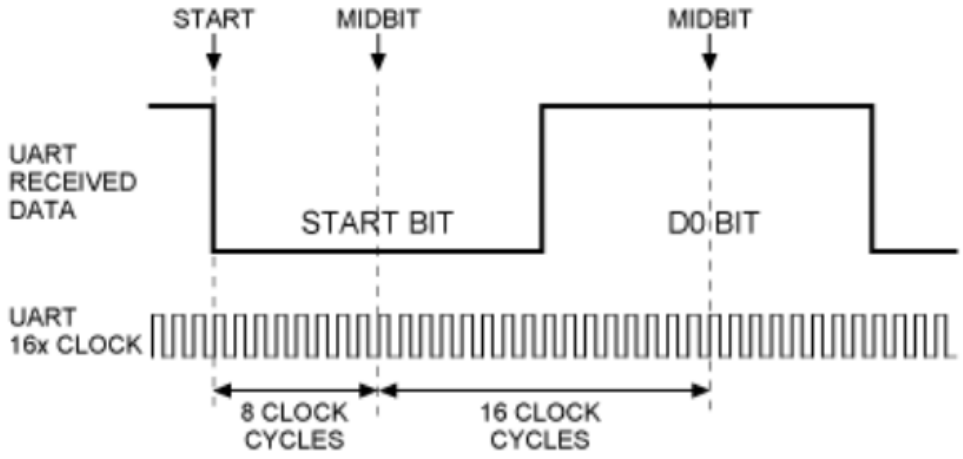

Figure 2

For more information visit repo

Baud Rate Generator

The baud rate generator generates a sampling signal whose frequency is exactly 16 times the UART's designated baud rate. To avoid creating a new clock domain and violating the synchronous design principle, the sampling signal should function as enable ticks rather than the clock signal to the UART receiver.

UART RX & TX

With an understanding of the oversampling procedure, we can derive the FSM chart accordingly, as shown in **Figure 3**. To accommodate future modification, two constants are used in the description. The DBIT constant indicates the number of data bits, and the SB-TICK constant indicates the number of ticks needed for the stop bits, which is 16, 24, and 32 for 1, 1.5, and 2 stop bits, respectively. DBIT and SB-TICK are assigned to 8 and 16 in this design.

Overall Design

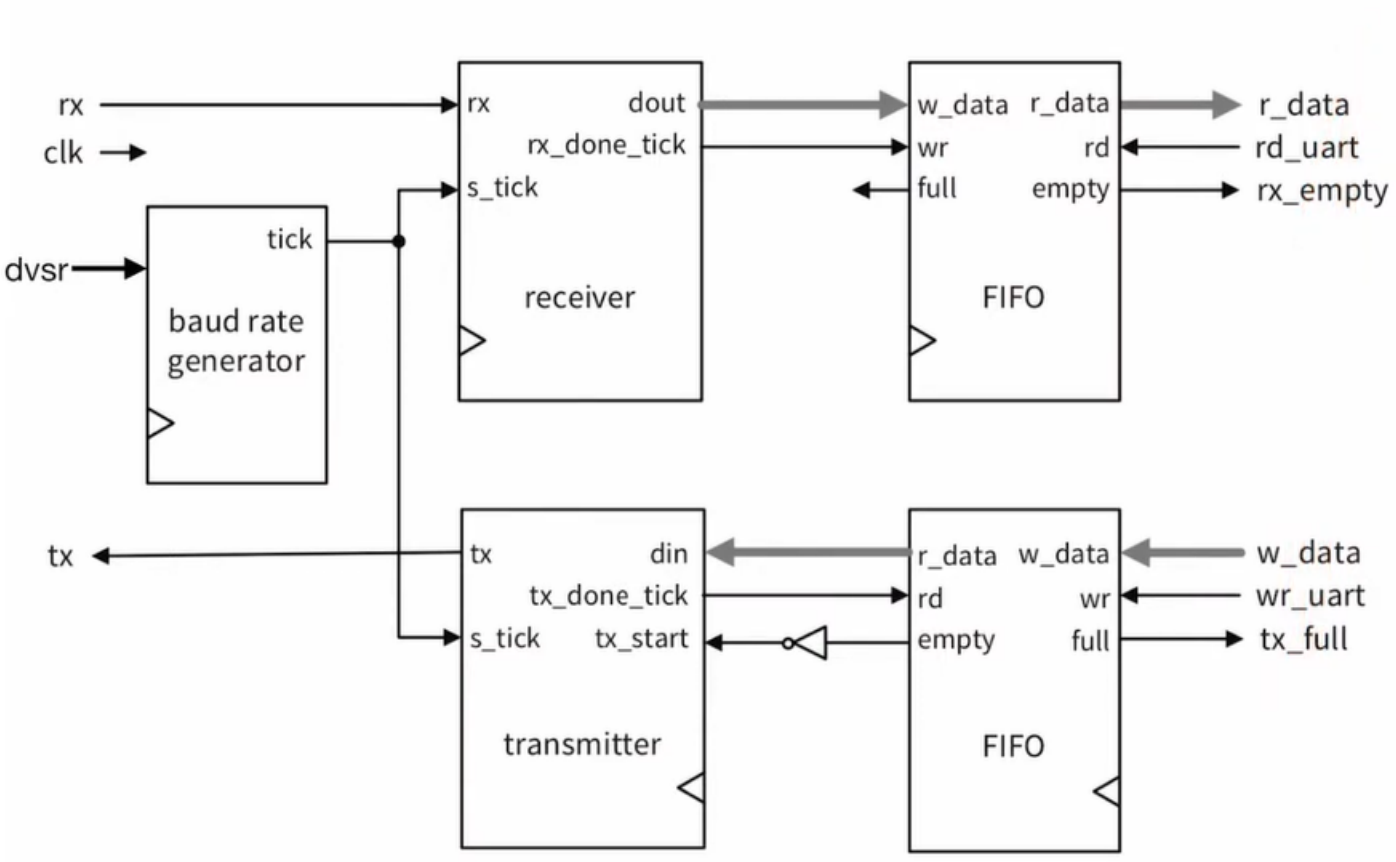

Figure 4

FSM Chart

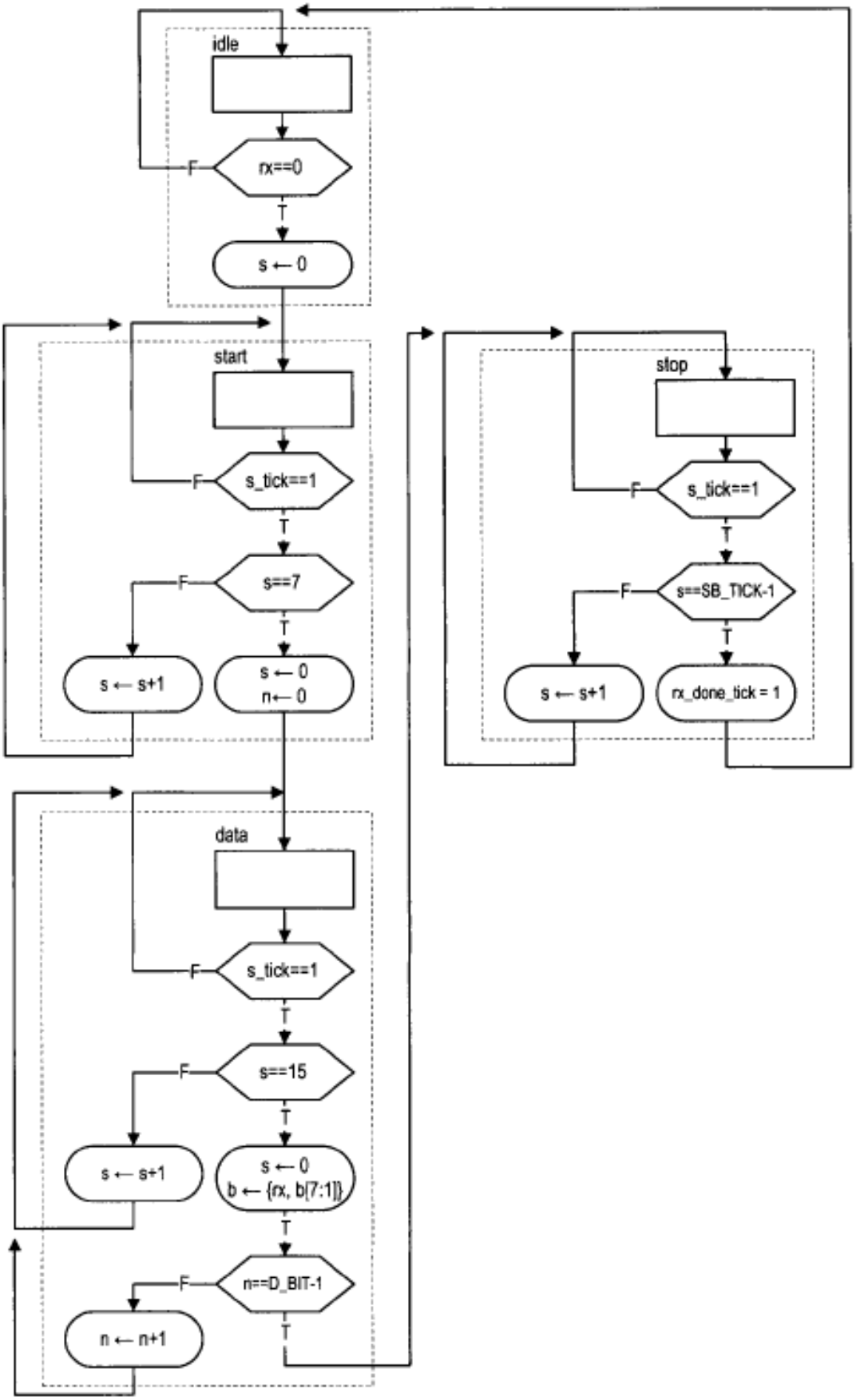

Figure 3

UART Module I/O Table

Signals	Direction	Width	Description
Inputs			
clk	Input	1-bit	System clock.
reset	Input	1-bit	Global reset signal (active-high or active-low, depending on implementation).
rx	Input	1-bit	Serial data input (received bitstream).
rd_uart	Input	1-bit	Read enable signal for the receive FIFO buffer.
wr_uart	Input	1-bit	Write an enable signal for the transmit FIFO buffer.
w_data[7:0]	Input	8-bit	Parallel data to be transmitted (written to the transmit FIFO).
dysr[31:0]	Input	32-bit	Divisors register for baud rate generation (configures clock division ratio).
Outputs			
tx	Output	1-bit	Serial data output (transmitted bitstream).
rx_empty	Output	1-bit	Flag indicating the receive FIFO is empty.
tx_full	Output	1-bit	Flag indicating the transmit FIFO is full.
r_data[7:0]	Output	8-bit	Parallel data read from the receive FIFO.
full	Output	1-bit	General FIFO full flag (may apply to either RX or TX FIFO).
almostempty1	Output	1-bit	Flag indicating the receive FIFO is *almost empty* (e.g., ≤ 1 word remaining).
almostempty2	Output	1-bit	Flag indicating the transmit FIFO is *almost empty*.
almostfull1	Output	1-bit	Flag indicating the receive FIFO is *almost full* (e.g., 1 slot remaining).
almostfull2	Output	1-bit	Flag indicating the transmit FIFO is *almost full*.
wr_ack1	Output	1-bit	Write acknowledgment for the receive FIFO (successful write).
wr_ack2	Output	1-bit	Write acknowledgment for the transmit FIFO.
overflow1	Output	1-bit	Overflow flag for the receive FIFO (write attempted when full).
overflow2	Output	1-bit	Overflow flag for the transmit FIFO.
underflow1	Output	1-bit	Underflow flag for the receive FIFO (read attempted when empty).
underflow2	Output	1-bit	Underflow flag for the transmit FIFO.

Design Files

Baud Rate Generator

```
module baud_rate_generator(
    input clk,
    input reset,
    input [31:0] dvsr,
    output reg tick
);

reg [31:0] count;

always @(posedge clk or posedge reset ) begin

    if(reset) begin
        count <= 0;
        tick <= 0;
    end
    else begin
        if(count == dvsr) begin
            count <= 0;
            tick <= ~tick;
        end
        else begin
            count <= count + 1;
        end
    end


end
endmodule
```

Figure 5

TX

```
module tx(
    input s_tick, clk, reset, tx_start,
    input [7:0] din,
    output reg tx,
    output reg tx_done_tick
);


localparam IDLE  = 2'b00;
localparam START = 2'b01;
localparam DATA  = 2'b10;
localparam STOP  = 2'b11;
parameter MIDBIT = 8; // Middle of the bit (for 16x oversampling)

reg [1:0] state;

reg [3:0] count; // Counts up to 15 for 16x oversampling
reg [3:0] bit_count;

always @(posedge clk or posedge reset) begin

        if (reset) begin
            state <= IDLE;
            count <= 0;
            bit_count <= 0;
            tx_done_tick <= 0;
            tx <= 1;

        end else begin
            tx_done_tick <= 0; // Default to 0, pulse only when done
            if (s_tick) begin // Proceed only on baud rate tick
                case(state)
                    IDLE: begin
                        if (!tx_start) begin // Start bit detected
                            state <= START;
                            count <= 0;
                            bit_count <= 0;
                        end
                    end

                    START: begin
                        if (count == MIDBIT - 1) begin // Sample mid-start bit
                            if (!tx_start) begin // Confirm start bit
                                state <= DATA;
                                count <= 0;
                                tx <= 0;
                            end else begin // False start, return to IDLE
                                state <= IDLE;
                            end
                        end else begin
                            count <= count + 1;
                        end
                    end
```

```
                DATA: begin
                    if (count == 15) begin // Wait 16 ticks per bit
                        tx <= din[bit_count]; // Capture bit (LSB first)
                        count <= 0;
                        if (bit_count == 8) begin
                            state <= STOP;
                            tx<=1;
                            count <= 0;
                        end else begin
                            bit_count <= bit_count + 1;
                        end
                    end else begin
                        count <= count + 1;
                    end
                end

                STOP: begin
                    if (count == 15) begin // Wait 16 ticks per bit
                        state <= IDLE;
                        tx_done_tick <= 1; // Signal end of transmission
                    end else begin
                        count <= count + 1;
                    end
                end
            endcase
        end

    end


end
```

Figure 6

RX

```
module rx(      You, 19 hours ago • Push
    input s_tick, clk, reset, rx,
    output reg [7:0] dout,
    output reg rx_done_tick
);

localparam IDLE  = 2'b00;
localparam START = 2'b01;
localparam DATA  = 2'b10;
localparam STOP  = 2'b11;
parameter MIDBIT = 8; // Middle of the bit (for 16x oversampling)

reg [1:0] state;
reg [7:0] data_reg;
reg [3:0] count; // Counts up to 15 for 16x oversampling
reg [2:0] bit_count;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        state <= IDLE;
        count <= 0;
        bit_count <= 0;
        rx_done_tick <= 0;
        dout <= 8'b0;
        data_reg <= 8'b0;
    end else begin
        rx_done_tick <= 0; // Default to 0, pulse only when done
        if (s_tick) begin // Proceed only on baud rate tick
            case(state)
                IDLE: begin
                    if (!rx) begin // Start bit detected
                        state <= START;
                        count <= 0;
                        bit_count <= 0;
                    end
                end

                START: begin
                    if (count == MIDBIT - 1) begin // Sample mid-start bit
                        if (!rx) begin // Confirm start bit
                            state <= DATA;
                            count <= 0;
                        end else begin // False start, return to IDLE
                            state <= IDLE;
                        end
                    end else begin
                        count <= count + 1;
                    end
                end
```

```
                DATA: begin
                    if (count == 15) begin // Wait 16 ticks per bit
                        data_reg[bit_count] <= rx; // Capture bit (LSB first)
                        count <= 0;
                        if (bit_count == 7) begin
                            state <= STOP;
                        end else begin
                            bit_count <= bit_count + 1;
                        end
                    end else begin
                        count <= count + 1;
                    end
                end

                STOP: begin
                    if (count == 15) begin // Wait for stop bit duration
                        state <= IDLE;
                        dout <= data_reg;
                        rx_done_tick <= 1; // Pulse done signal
                    end else begin
                        count <= count + 1;
                    end
                end
            endcase
        end
    end
end

endmodule
```

Figure 7

FIFO

```
module FIFO(data_in, wr_en, rd_en, clk, rst_n, full, empty, almostfull, almostempty, wr_ack, overflow, underflow, data_out);
parameter FIFO_WIDTH = 8;
parameter FIFO_DEPTH = 8;
input [FIFO_WIDTH-1:0] data_in;
input clk, rst_n, wr_en, rd_en;
output reg [FIFO_WIDTH-1:0] data_out;
output reg wr_ack, overflow, underflow;
output full, empty, almostfull, almostempty;

localparam max_fifo_addr = $clog2(FIFO_DEPTH);

reg [FIFO_WIDTH-1:0] mem [FIFO_DEPTH-1:0];

reg [max_fifo_addr-1:0] wr_ptr, rd_ptr;
reg [max_fifo_addr:0] count;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        wr_ptr <= 0;
        wr_ack <= 0;
        overflow <= 0;

    end

    else if (wr_en && count < FIFO_DEPTH) begin
        mem[wr_ptr] <= data_in;
        wr_ack <= 1;
        wr_ptr <= wr_ptr + 1;
    end
    else if ({wr_en, rd_en} == 2'b00) begin

    end
    else begin
        wr_ack <= 0;
        if (full && wr_en&&!rd_en)
            overflow <= 1;
        else
            overflow <= 0;
    end
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_ptr <= 0;
        underflow <= 0 ;
    end
    else if (rd_en && count != 0) begin
        data_out <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
    end
    else if ({wr_en, rd_en} == 2'b00) begin

    end
    else begin
      if (rd_en&&!wr_en&&empty) begin
        underflow <= 1 ;
      end
      else begin
        underflow <= 0 ;
      end
    end
end
```

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        count <= 0;
    end
    else begin
        if  ( ({wr_en, rd_en} == 2'b10) && !full)
            count <= count + 1;
        else if ( ({wr_en, rd_en} == 2'b01) && !empty)
            count <= count - 1;
        else if (({wr_en, rd_en} == 2'b11) && full  ) begin
            count <= count - 1;
        end
        else if (({wr_en, rd_en} == 2'b11) && empty ) begin
            count <= count + 1;
        end
        else begin

        end
    end
end

assign full = (count == FIFO_DEPTH)? 1 : 0;
assign empty = (count == 0)? 1 : 0;
assign almostfull = (count == FIFO_DEPTH-1)? 1 : 0;
assign almostempty = (count == 1)? 1 : 0;

endmodule
```

Figure 8

UART Top

```
module UART(      You, 19 hours ago • Push
    input clk, reset, rx, rd_uart, wr_uart,
    input [7:0] w_data,
    input [31:0] dvsr,
    output tx, rx_empty, tx_full,
    output [7:0] r_data,
    output full, almostempty1, almostempty2,
    output wr_ack1, wr_ack2, overflow1, overflow2,
    output underflow1, underflow2, almostfull1, almostfull2
);

    // Baud rate generator instantiation
    wire s_tick;
    baud_rate_generator baud_gen(
        .clk(clk),
        .reset(reset),
        .dvsr(dvsr),
        .tick(s_tick)
    );

    // Receiver section
    wire [7:0] rx_data_out;
    wire rx_done;
    rx receiver(
        .s_tick(s_tick),
        .clk(clk),
        .reset(reset),
        .rx(rx),
        .dout(rx_data_out),
        .rx_done_tick(rx_done)
    );

    // Transmitter section
    wire tx_empty;
    wire [7:0] tx_data_in;
    wire tx_done, tx_start;
    assign tx_start = ~tx_empty;  // Start when FIFO has data

   tx transmitter(
        .s_tick(s_tick),
        .clk(clk),
        .reset(reset),
        .tx_start(tx_start),
        .din(tx_data_in),
        .tx(tx),
        .tx_done_tick(tx_done)
    );
```

```
    // FIFO instantiation for receive path
    FIFO rx_fifo(
        .data_in(rx_data_out),
        .wr_en(rx_done),
        .rd_en(rd_uart),
        .clk(clk),
        .rst_n(~reset),
        .data_out(r_data),
        .full(full),
        .empty(rx_empty),
        .almostfull(almostfull1),
        .almostempty(almostempty1),
        .wr_ack(wr_ack1),
        .overflow(overflow1),
        .underflow(underflow1)
    );

    // FIFO instantiation for transmit path
    FIFO tx_fifo(
        .data_in(w_data),
        .wr_en(wr_uart),
        .rd_en(~tx_start),
        .clk(clk),
        .rst_n(~reset),
        .data_out(tx_data_in),
        .full(tx_full),
        .empty(tx_empty),
        .almostfull(almostfull2),
        .almostempty(almostempty2),
        .wr_ack(wr_ack2),
        .overflow(overflow2),
        .underflow(underflow2)
    );

endmodule
```

Figure 9

Testbenches

Testbench for test RX, baud rate generator and FIFO

```
module rx_tb;
    reg clk;
    reg reset;
    reg [31:0] dvsr;
    wire tick;
    reg rx;
    wire [7:0] dout;
    wire rx_done_tick;
    logic rd_uart;
    logic [7:0]r_data;

    // Instantiate the baud rate generator
    baud_rate_generator brg (
        .clk(clk),
        .reset(reset),
        .dvsr(dvsr),
        .tick(tick)
    );

    // Instantiate the UART receiver
    rx u_rx (
        .s_tick(tick),
        .clk(clk),
        .reset(reset),
        .rx(rx),
        .dout(dout),
        .rx_done_tick(rx_done_tick)
    );

      FIFO fifo(
        .data_in(dout),
        .wr_en(rx_done_tick),
        .rd_en(rd_uart), // Read when transmission is done
        .clk(clk),
        .rst_n(~reset),
        .full(full),
        .empty(empty),
        .data_out(r_data)       You, 20 hours ago • Push
    );
    // Generate 100 MHz clock
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10ns period
    end

    // Apply reset
    initial begin
        reset = 1;
        #20 reset = 0;
    end
```

```
// Stimulus: Send data 0xA5 via UART
initial begin
    dvsr = 0; // Set for 115200 baud with 100 MHz clock
    rx = 1; // Idle state
     rd_uart =  0 ;

    // Wait for reset to complete
    #30;

    // Send start bit (0)
    rx = 0;
    #320; // Wait for 1 bit duration (16 * 540ns)

    // Send data bits (LSB first: 1,0,1,0,0,1,0,1)
    rx = 1; // Bit 0
    #320;
    rx = 0; // Bit 1
    #320;
    rx = 1; // Bit 2
    #320;
    rx = 0; // Bit 3
    #320;
    rx = 0; // Bit 4
    #320;
    rx = 1; // Bit 5
    #320;
    rx = 0; // Bit 6
    #320;
    rx = 1; // Bit 7
    #320;

    // Send stop bit (1)
    rx = 1;
    #320;

     rd_uart = 1;
     #320;
     $display("Received Data: 0x%h", r_data);
     if (dout === 8'hA5)
         $display("Test Passed!");
     else
         $display("Test Failed!");
     $finish;
end

endmodule       You, 20 hours ago • Push
```

Figure 10

Testbench for test TX, baud rate generator and FIFO

```
module tx_tb;

    // Parameters
    parameter CLK_PERIOD = 10;  // 100 MHz clock
    parameter DVSR = 1;        // Baud rate divisor for 115200 baud (assuming oversampling of 16)

    // Signals
    reg clk, reset;
    reg wr_en;
    reg [7:0] data_in;
    wire tx, tx_done_tick;
    wire full, empty;

    // FIFO-TX connections
    wire [7:0] fifo_to_tx;
    wire tx_start = ~empty;  // Start when FIFO has data

    // Instantiate FIFO
    FIFO fifo(
        .data_in(data_in),
        .wr_en(wr_en),
        .rd_en(~tx_done_tick), // Read when transmission is done
        .clk(clk),
        .rst_n(~reset),
        .full(full),
        .empty(empty),
        .data_out(fifo_to_tx)
    );

    // Baud rate generator
    wire s_tick;
    baud_rate_generator brg(
        .clk(clk),
        .reset(reset),
        .tick(s_tick),
        .dvsr(DVSR)
    );
   // UART Transmitter
   tx uart_tx(
        .s_tick(s_tick),
        .clk(clk),
        .reset(reset),
        .tx_start(tx_start),
        .din(fifo_to_tx),
        .tx(tx),
        .tx_done_tick(tx_done_tick)
    );
```

```
    always begin
        clk = 0;
        #(CLK_PERIOD/2);
        clk = 1;
        #(CLK_PERIOD/2);
    end

    // Test sequence
    initial begin
        // Initialize signals
        reset = 1;
        wr_en = 0;
        data_in = 8'h00;
        #100;

        // Release reset
        reset = 0;
        #100;

        // Send a single byte (0xA5)
        $display("Sending 0xA5");
        write_to_fifo(8'hA5);

        #100;


        #5000;
        $display("Test complete");
        $finish;
    end
    task write_to_fifo(input [7:0] data);
        begin
            @(negedge clk);
            wr_en = 1;
            data_in = data;
            @(negedge clk);
            wr_en = 0;
        end
    endtask

    // Task to wait for transmission
    task wait_for_transfer;
        begin
            wait(tx_done_tick);
            #100;
        end
    endtask

    // Monitoring
    initial begin
        $monitor("Time: %t | TX: %b | Data Sent: 0x%h | FIFO: %s",
                 $time, tx, fifo_to_tx, empty ? "Empty" : "Active");
    end

endmodule
```

Testbench for test UART AS RX

```
module uart_as_rx_tb;
    reg clk;
    reg reset;
    reg [31:0] dvsr;
    reg rx;

    logic rd_uart,full,rx_empty;
    logic [7:0]r_data;
    UART DUT(

        .clk(clk),
        .reset(reset),
        .rx(rx),
        .tx(),
        .rd_uart(rd_uart),
        .r_data(r_data),
        .wr_uart(),
        .dvsr(dvsr),
        .rx_empty(rx_empty),
        .tx_full(),
        .full(full),
        .w_data()

    );
    // Generate 100 MHz clock
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10ns period
    end

    // Apply reset
    initial begin
        reset = 1;
        #20 reset = 0;
    end
```

```
    // Stimulus: Send data 0xA5 via UART
    initial begin
        dvsr = 0; // Set for 115200 baud with 100 MHz clock
        rx = 1; // Idle state
        rd_uart =  0 ;              You, 1 second ago • Uncommitted changes
        // Wait for reset to complete
        #30;
        // Send start bit (0)
        rx = 0;
        #320; // Wait for 1 bit duration (16 * 540ns)
        // Send data bits (LSB first: 1,0,1,0,0,1,0,1)
        rx = 1; // Bit 0
        #320;
        rx = 0; // Bit 1
        #320;
        rx = 1; // Bit 2
        #320;
        rx = 0; // Bit 3
        #320;
        rx = 0; // Bit 4
        #320;
        rx = 1; // Bit 5
        #320;
        rx = 0; // Bit 6
        #320;
        rx = 1; // Bit 7
        #320;
        // Send stop bit (1)
        rx = 1;
        #320;
        rd_uart = 1;
        #320;
        $display("Received Data: 0x%h", r_data);
        if (r_data === 8'hA5)
            $display("Test Passed!");
        else
            $display("Test Failed!");
        $finish;
    end


endmodule
```

Figure 12

DO File

```
vlib work       You, 20 hours ago • Push
vlog FIFO.v RX.v TX.v UART.v baud_rate_generator.v tb.sv
vsim -voptargs=+acc work.tb
add wave *
run -all
#quit -sim
```

Figure 13

Results

Waveform1

Receiving 10100101

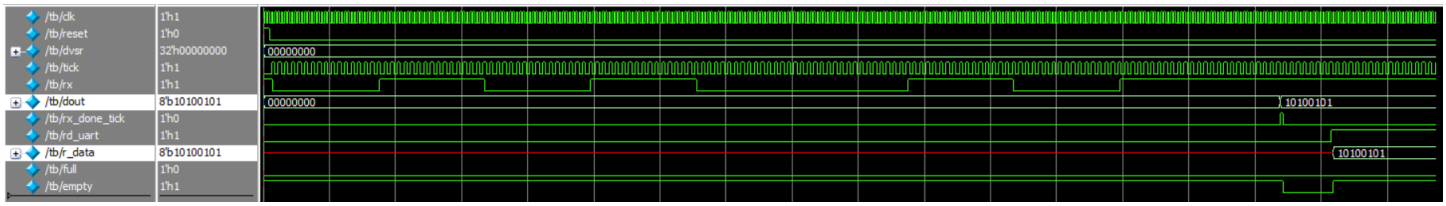

Figure14

Waveform2

Transmit 10100101

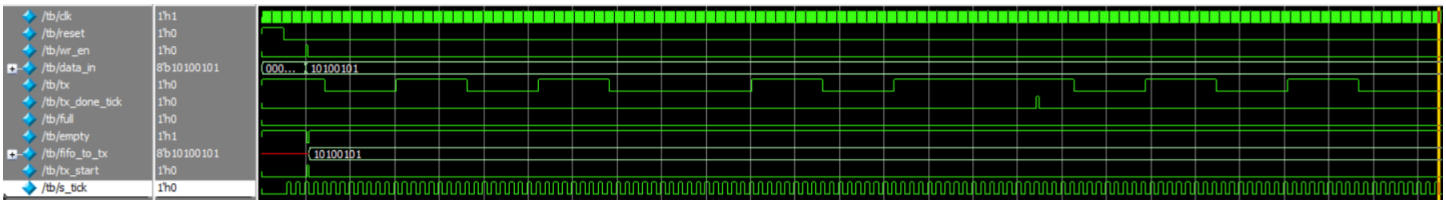

Figure15

Waveform3

Receiving 10100101

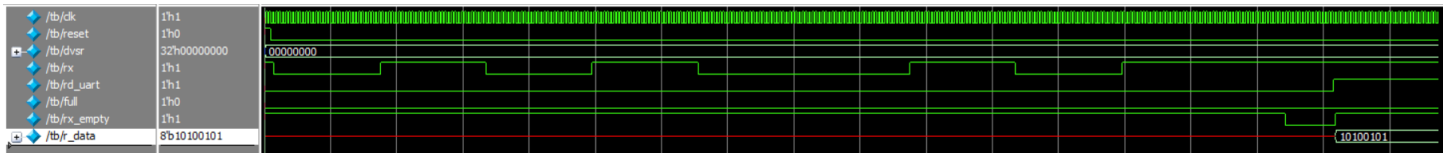

Figure16

VIVADO

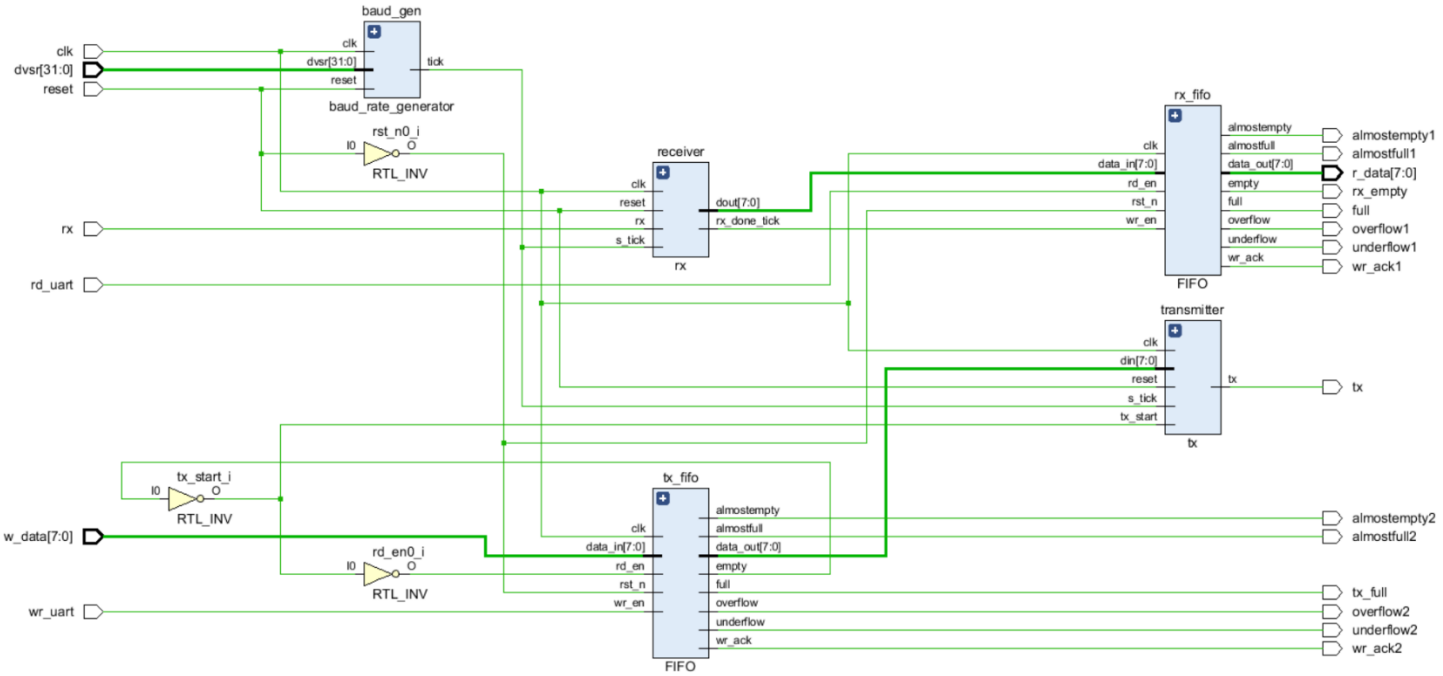

figure 17

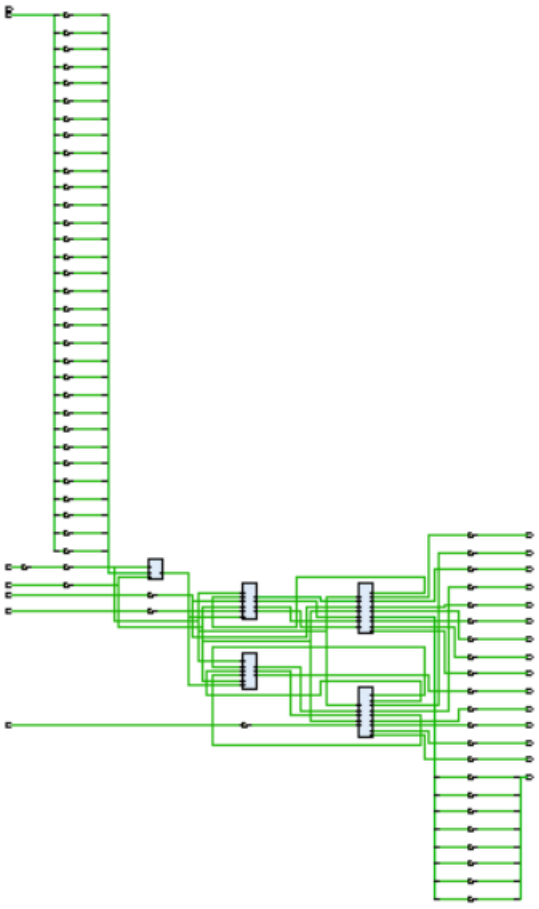

Figure 18

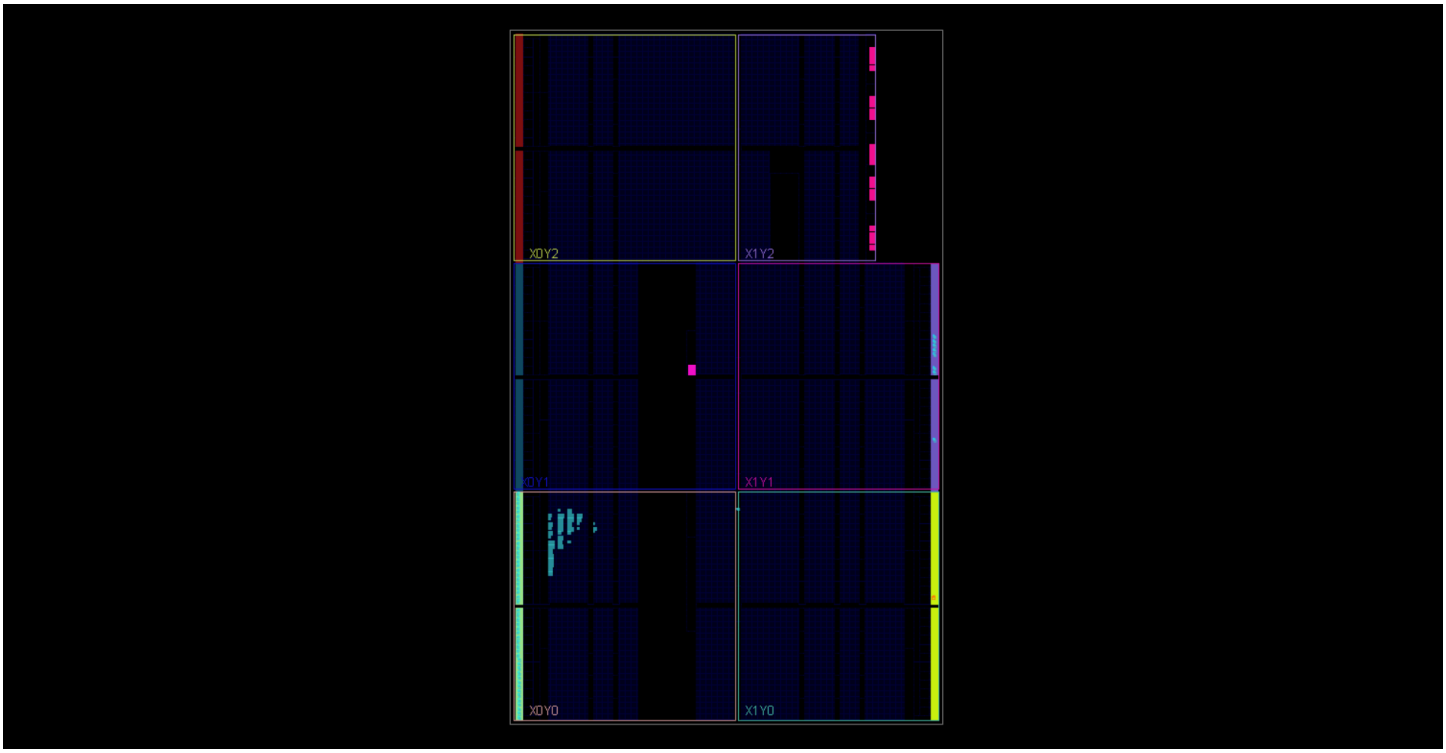

Figure 19

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.888 ns	Worst Hold Slack (WHS):	0.165 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	269	Total Number of Endpoints:	269	Total Number of Endpoints:	163

All user specified timing constraints are met.

Figure 20

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.066 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.3°C**
Thermal Margin:	74.7°C (14.9 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

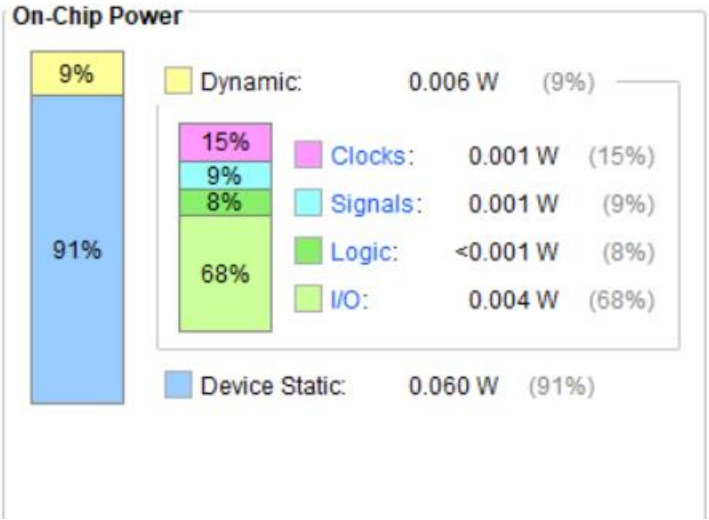

Figure 21

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
N UART	144	162	8	65	144	77	59	1
baud_gen (baud_rate_...	44	33	0	12	44	33	0	0
receiver (rx)	30	26	0	18	30	9	0	0
rx_fifo (FIFO)	41	85	8	27	41	18	0	0
transmitter (tx)	18	11	0	6	18	10	0	0
tx_fifo (FIFO_0)	11	7	0	6	11	5	0	0

Figure 22

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```

Figure 23

References

FPGA Prototyping by Verilog Examples. By Pong P. Chu 2008 John Wiley & Sons, Inc