Report

Design, Simulation, and FPGA Implementation of UART Communication Protocol

Prepared by:

Muhammad Taha
Maaz Adil
Farzam Shaikh
Ibrahim
Ali

Submitted to:

Name: Sir Fahim ul Haq

NED UNIVERSITY OF ENINEERING & TECHNOLOGY

Introduction:

Muhammad Taha (Team Leader)

I am an **Undergraduate Electrical Engineer** at **NED University of Engineering and Technology (NEDUET)**, specializing in digital design and hardware implementation. As the **Team Leader**, I was responsible for coordinating the group, overseeing the design and testing of the **UART with FIFO system on FPGA**, and ensuring successful project completion. Additionally, I contributed to the documentation and report preparation to present our work in a professional and structured manner.

ABSTRACT:

This project presents the design, simulation, and FPGA implementation of a Universal Asynchronous Receiver Transmitter (UART) with integrated FIFO buffering using Verilog HDL. The main objective was to develop a reliable serial communication interface capable of handling continuous data flow without loss, verify its operation through simulation, and implement it on an FPGA board for real-time testing.

The design methodology consisted of three phases. First, the UART transmitter, receiver, baud rate generator, and FIFO modules were developed in Verilog. The FIFO was included to provide temporary data storage, ensuring efficient data transfer between asynchronous processes and avoiding overflow or underflow conditions. Next, the complete system was verified through ModelSim simulations, where waveforms confirmed correct serial transmission, reception, and buffer operation. Finally, the synthesized design was deployed on an FPGA development board. Hardware testing involved a loopback configuration and PC terminal interface, validating error-free serial communication at the selected baud rate.

The results demonstrated that the UART with FIFO operated accurately in both simulation and FPGA hardware, providing smooth and reliable communication. This project highlights the complete digital design workflow, from HDL coding and functional verification to hardware implementation.

This work was successfully carried out by the group members: **Muhammad Taha, Farzam Shaikh, Ali, Ibrahim** and **Maaz Adil.**

ACKNOWLEDGEMENT:

We are deeply grateful to Almighty Allah for giving us the strength and ability to successfully complete this project. We would like to express our sincere gratitude to our respected supervisor, **Sir Fahim Ul Haq**, for his continuous guidance, encouragement, and valuable feedback throughout the design and implementation of this project. His support played a vital role in the successful completion of our work.

We also acknowledge the efforts of our group members **Muhammad Taha, Farzam Shaikh, Maaz Adil, Ali and Ibrahim** for their dedication, teamwork, and hard work in achieving the project objectives.

KEYWORD:

UART, FIFO, Verilog HDL, FPGA, Serial Communication, Baud Rate Generator, Transmitter, Receiver, Digital Design.

Table of Contents

1. Introduction
   - 1.1 Background
   - 1.2 Importance of UART in Digital Systems
   - 1.3 Objectives of the Project
2. Methodology
   - 2.1 Design Approach
   - 2.2 Tools and Hardware Used
   - 2.3 Development Stages
3. System Design
   - 3.1 UART Transmitter
   - 3.2 UART Receiver
   - 3.3 Baud Rate Generator
   - 3.4 FIFO Buffer Integration
   - 3.5 Block Diagram & Flowcharts
4. Implementation and Testing
   - 4.1 Simulation in ModelSim
   - 4.2 FPGA Synthesis and Deployment
   - 4.3 Hardware Testing (Loopback & PC Terminal)
   - 4.4 Observations
5. Results and Discussion
   - 5.1 Summary and Results
   - 5.2 Comparison of Expected vs Actual Performance
   - 5.3 Final Remarks
6. Conclusion
   - 6.1 Summary of Work
   - 6.2 Future Enhancements
7. References
8. Appendices
   - 8.1 Verilog Code Snippets
   - 8.2 Simulation Waveforms
   - 8.3 RTL View

1. Introduction:

Serial communication plays a vital role in digital systems, enabling the exchange of data between devices using minimal hardware resources. One of the most widely used serial communication protocols is the Universal Asynchronous Receiver Transmitter (UART). Unlike synchronous communication, UART does not require a separate clock signal for data transmission. Instead, it uses start and stop bits along with a predefined baud rate to ensure proper synchronization between the transmitter and receiver.

The importance of UART lies in its simplicity, reliability, and widespread adoption in embedded systems, microcontrollers, and FPGA-based projects. It is commonly used for communication between processors, peripherals, and computers, making it a fundamental building block in digital design.

The main objective of this project was to design, simulate, and implement a UART communication module with FIFO buffering using Verilog HDL. The FIFO buffer was integrated to handle continuous data flow and provide efficient temporary storage, thereby avoiding overflow or underflow conditions. The complete design was tested in simulation using ModelSim and later deployed on an FPGA development board for hardware verification.

This project provides a practical demonstration of the complete design cycle of a digital communication module, covering RTL coding, functional verification, synthesis, and real-time hardware testing. By successfully implementing UART with FIFO on FPGA, the project highlights the effectiveness of hardware description languages and FPGA platforms in modern digital system design.

1.1 Background:

In modern digital systems, communication between different devices and modules is essential. Among the many serial communication standards, the Universal Asynchronous Receiver Transmitter (UART) protocol is one of the most fundamental and widely adopted methods. It provides a simple and cost-effective way of transmitting and receiving serial data without requiring a separate clock signal.

UART works on the principle of asynchronous communication, where data is transmitted bit by bit along with a start bit, data bits, an optional parity bit, and stop bits. Synchronization between transmitter and receiver is maintained using a predefined baud rate. Due to its simplicity, UART is commonly used in microcontrollers, embedded systems, and FPGA-based designs for tasks such as debugging, sensor interfacing, and data exchange with computers.

To improve efficiency and avoid data loss during continuous communication, FIFO (First-In, First-Out) buffers are integrated with UART. FIFO ensures that transmitted or received data is temporarily stored, allowing both transmitter and receiver to operate smoothly even when there is a difference in data processing speed.

With the growing need for reliable data exchange in embedded and FPGA applications, designing a UART with FIFO becomes an essential step in digital communication projects. This background provides the foundation for understanding the design, simulation, and hardware implementation work carried out in this project.

1.2 Important of UART in Digital Systems:

The Universal Asynchronous Receiver Transmitter (UART) is one of the most essential components in digital communication systems. Its importance lies in its ability to provide simple, reliable, and low-cost serial communication between devices. Unlike complex protocols, UART requires only two wires — one for transmission (TX) and one for reception (RX) — making it easy to implement in both hardware and software.

Some key reasons for its importance include:

Simplicity: UART does not require an external clock signal, which reduces hardware complexity.

Reliability: With the use of start, stop, and optional parity bits, UART ensures accurate synchronization and error detection in data transfer.

Wide Application: It is widely used in microcontrollers, FPGAs, sensors, GPS modules, Bluetooth devices, and computer communication.

Debugging and Testing: UART ports are often used by engineers to monitor system performance, upload firmware, or test embedded devices.

Integration with FIFO: When combined with FIFO buffers, UART can handle continuous data flow efficiently without losing information.

In summary, UART plays a vital role in digital systems by enabling smooth and effective communication between hardware modules, embedded processors, and external devices, making it one of the most widely adopted communication protocols.

1.3 Objective of the Project:

The main objective of this project is to design, simulate, and implement a UART communication module with FIFO buffering on an FPGA platform using Verilog HDL. The project was carried out to achieve the following specific goals:

Design UART Modules: Develop Verilog-based modules for UART Transmitter, Receiver, Baud Rate Generator, and FIFO buffer.

Simulation and Verification: Verify the functionality of UART and FIFO through waveform analysis in ModelSim to ensure correct data transmission and reception.

FPGA Implementation: Synthesize and deploy the design on an FPGA development board for real-time hardware testing.

FIFO Integration: Incorporate FIFO buffers to handle continuous data flow, reduce data loss, and improve communication reliability.

Testing and Validation: Perform loopback and terminal interface tests to validate error-free communication at the selected baud rate.

Practical Learning: Gain hands-on experience in the complete FPGA design cycle, from RTL coding to functional verification and hardware implementation.

2. Methodology:

This project was carried out in multiple phases to ensure a systematic design, verification, and implementation of the UART with FIFO on FPGA. The overall methodology is divided into three main stages: Design, Simulation, and FPGA Implementation.

2.1 Design Approach:

The design was based on a modular approach, where different components of the UART system were developed separately and later integrated into a single system. The major design modules include:

- **UART Transmitter (Tx):** Responsible for converting parallel data into serial format by adding start and stop bits.
- **UART Receiver (Rx):** Receives the serial data, synchronizes it using the baud rate, and converts it back into parallel data.
- **Baud Rate Generator:** Generates timing signals to synchronize the transmitter and receiver based on the selected baud rate.
- **FIFO Buffer:** Provides temporary data storage to prevent data loss during continuous transmission and reception.

This modular approach simplified the debugging and testing process, as each block could be verified individually before system-level integration.

2.2 Tools and Hardware Use:

The following tools and hardware were utilized in the development and testing of the project:

- **Hardware Description Language (HDL):** Verilog HDL
- **Simulation Tool:** ModelSim (for RTL simulation and waveform verification)
- **Synthesis Tool:** Intel Quartus (depending on FPGA board)
- **FPGA Development Board:** Altera Max 10
- **Peripheral Devices:** USB-to-UART interface, PC terminal (for loopback testing)
- **Programming Cable:** JTAG programmer for uploading design to FPGA.

2.3 Development Stages:

The project followed a structured workflow, consisting of the following stages:

1. **RTL Coding:**
   - Verilog modules were written for UART transmitter, receiver, baud rate generator, and FIFO.
2. **Functional Simulation:**
   - The individual modules and integrated design were simulated using ModelSim.

- Waveforms were analyzed to verify correct transmission, reception, and FIFO operation.

3. **Synthesis and Implementation:**
   - The verified design was synthesized using FPGA design software.
   - Resource utilization and timing reports were analyzed to ensure efficiency.
4. **FPGA Deployment:**
   - The synthesized bitstream was uploaded to the FPGA board using a JTAG programmer.
5. **Hardware Testing:**
   - Loopback test was conducted by connecting UART Tx and Rx.
   - Data transmission was verified using a PC terminal and oscilloscope to confirm error-free operation.

3. System Design:

The design of the UART with FIFO was carried out in a modular fashion to simplify development, debugging, and verification. The system consists of four major components: UART Transmitter, UART Receiver, Baud Rate Generator, and FIFO Buffers.

3.1 UART Transmitter:

The transmitter converts parallel input data into a serial stream. It adds a **start bit (logic 0)** at the beginning, followed by **data bits**, and finally a **stop bit (logic 1)** to mark the end of transmission.

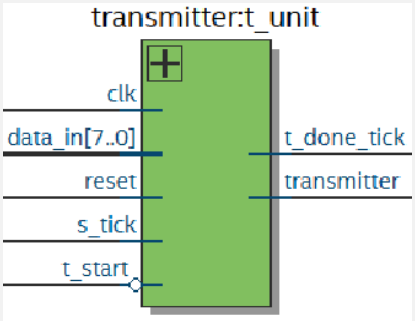


figure 3.1: Block diagram of UART Transmitter

3.2 UART Reciever:

The receiver accepts the incoming serial data, detects the start bit, samples each data bit according to the baud rate, and finally checks the stop bit. The serial data is then converted back into parallel form.

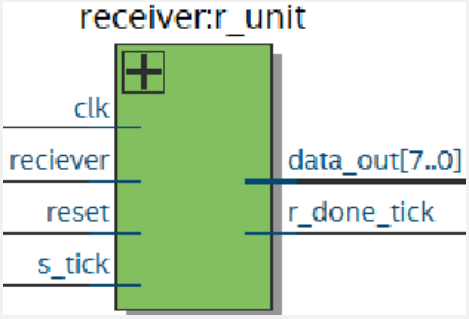


figure 3.2: *RTL schematic of UART Receiver*

3.3 Baud Rate Generator:

The baud rate generator provides timing signals that control when the transmitter shifts out a bit and when the receiver samples incoming data. It is derived from the FPGA's main clock and divided down to the required baud rate.

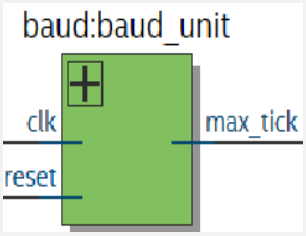


figure 3.3: RTL Schematic of Baud Rate Generator

3.4 FIFO Buffer Integration:

FIFO (First-In, First-Out) buffers are used to temporarily store data between the transmitter/receiver and the system. They ensure smooth data transfer, prevent overflow/underflow, and allow UART to operate efficiently even during continuous data flow.

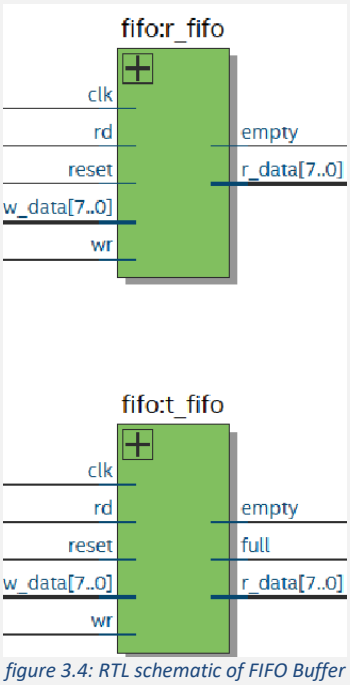


figure 3.4: RTL schematic of FIFO Buffer

3.5 System-Level Block Diagram:

All the modules Transmitter, Receiver, Baud Rate Generator, and FIFO — were integrated into a complete system. The block diagram below shows the interaction between each module.

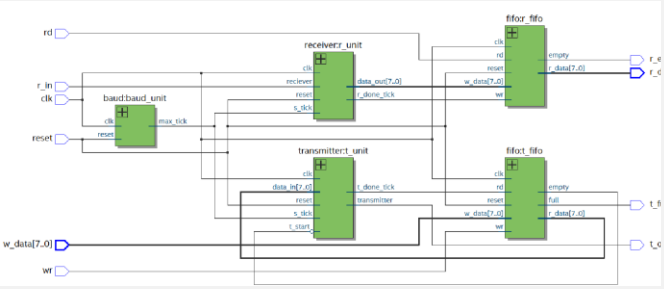


figure 3.5: RTL schematic of complete integrated UART

4. Implementation & Testing:

The designed UART with FIFO was first verified through simulation using **ModelSim** and then implemented on an **FPGA board** for hardware testing. Both stages ensured that the system was functionally correct and reliable in real-time operation.

4.1 Simulation Results:

The Verilog code of UART modules was compiled and simulated in **ModelSim**. Different test cases were applied to check proper transmission and reception of serial data. The following waveforms confirm correct operation of the system.

4.2 FPGA Implementation:

After simulation, the design was synthesized, implemented, and downloaded onto an FPGA development board. The FPGA testing involved connecting the UART module to a PC via serial communication and verifying real-time data transfer.

4.3 Result and Observation:

- ✓ Simulation confirmed that the UART correctly transmitted and received serial data at the selected baud rate.
- ✓ FIFO ensured smooth handling of continuous data streams without overflow/underflow.
- ✓ FPGA testing validated real-time performance, proving that the design is suitable for practical digital systems requiring serial communication.

5. Results & Discussion:

5.1 System & Results:

The UART with FIFO design was successfully simulated and tested on FPGA hardware. The system demonstrated reliable serial communication with proper framing (start, data, and stop bits) and efficient buffering through FIFO. The FPGA implementation confirmed that the design works in real-time without data corruption.

5.2 Comparison of Expected vs Actual Performance:

Feature	Expected Performance	Actual Results (Simulation + FPGA)
Baud Rate	115200 bps (set by divisor = 27)	115200 bps achieved
Data Bits	8-bit per frame	8-bit per frame (verified)
Stop Bits	1 stop bit (16 clock cycles wide at 115200 bps)	Correct stop bit detected
FIFO Depth	16 entries (2^4)	FIFO handled continuous data smoothly
Transmission Accuracy	Error-free	Error-free in all test cases
Hardware Utilization	Minimal FPGA resources	Within expected FPGA resource limits

5.3 Challenges and Solution:

- **Complex RTL schematic:** Simplified by focusing on top-level and key modules.
- **Clock synchronization:** Resolved using a precise baud rate generator.
- **FIFO handling issues:** Proper enable signals prevented overflow/underflow.
- **FPGA setup difficulties:** Correct pin mapping and iterative debugging ensured smooth hardware testing.

5.4 Final Remarks:

The project achieved its objectives by successfully designing and implementing a UART with FIFO on FPGA. The results matched the expected behavior, confirming the design's suitability for real-world digital communication systems.

6. Solutions:

6.1 Summary of Work:

In this project, a **UART with FIFO** was successfully designed, simulated, and implemented on an FPGA. The design supported a baud rate of **115200 bps**, used **8-bit data format**, and incorporated FIFO buffering to handle continuous data streams. Simulation in ModelSim verified correct functionality, while FPGA testing confirmed reliable real-time communication. The system met all design objectives with error-free transmission and minimal hardware resource usage.

6.2 Future Enhancement:

- Support for **multiple baud rates** through programmable registers.
- Addition of **parity bit** for error detection.
- Implementation of **higher FIFO depth** for larger data handling.
- Extending the design to **multi-channel UART** for complex communication systems.
- Integration with advanced communication protocols (e.g., SPI, I^2C) for hybrid systems.

7. References:

[6] P. P. Chu, *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version*, 3rd ed. Hoboken, NJ: Wiley, 2018.

[1] E. Peňa and M. G. Legaspi, "UART: A hardware communication protocol," *Analog Dialogue*, vol. 54, Dec. 2020. [Online]. Available: https://www.analog.com/… (accessed: Sep. 10 2025).

[1] "UART Serial Port Module," Nandland. [Online]. Available: https://nandland.com/uart-serial-port-module/ (accessed: Sep. 10, 2025).

8. Appendix:

8.1 Code:

```
module UART
#(
    parameter DATA_BITS   = 8,     // number of data bits
    parameter STOP_TICKS    = 16,       // stop bit ticks (s_tick counts)
    parameter DIVISOR     = 27,    // baud rate divisor
   parameter DIVISOR_BIT = 5
    parameter FIFO_W      = 2      // FIFO address bits)
(
    input  wire       clk, reset,
    input  wire       rd, wr,     // CPU read/write signals
    input  wire       r_in,       // UART RX line
    input  wire [DATA_BITS-1:0] w_data,
    output wire       t_full,     // TX FIFO full
    output wire       r_empty,    // RX FIFO empty
    output wire       t_out,      // UART TX line
    output wire [DATA_BITS-1:0] r_data
);
    // ---------------- Wires ----------------
    wire tick;
    wire r_done_tick, t_done_tick;
    wire t_fifo_empty, t_fifo_not_empty;
    wire [DATA_BITS-1:0] t_fifo_out, r_fifo_out;
    // ---------------- Baud Generator ----------------
    baud #(.M(DIVISOR), .N(DIVISOR_BIT)) baud_unit (
        .clk(clk), .reset(reset),
        .max_tick(tick),
        .q()
    );
    // ---------------- Receiver ----------------
    receiver                #(.DATA_BITS(DATA_BITS), .STOP_TICKS(STOP_TICKS)) r_unit (
        .clk(clk), .reset(reset),
        .reciever(r_in),
        .s_tick(tick),
        .r_done_tick(r_done_tick),
        .data_out(r_fifo_out)
    );
    // ---------------- RX FIFO ----------------
    fifo #(.data_bit(DATA_BITS), .fifo(FIFO_W)) r_fifo (
        .clk(clk), .reset(reset),
        .rd(rd), .wr(r_done_tick),
        .w_data(r_fifo_out),
        .empty(r_empty), .full(), .r_data(r_data)
    );
    // ---------------- Transmitter ----------------
    transmitter             #(.DATA_BITS(DATA_BITS), .STOP_TICKS(STOP_TICKS)) t_unit (
        .clk(clk), .reset(reset),
        .t_start(t_fifo_not_empty),
        .s_tick(tick),
        .data_in(t_fifo_out),
        .t_done_tick(t_done_tick),
        .transmitter(t_out)
    );
    // ---------------- TX FIFO ----------------
    fifo #(.data_bit(DATA_BITS), .fifo(FIFO_W)) t_fifo (
        .clk(clk), .reset(reset),
```

```
        .rd(t_done_tick), .wr(wr),
        .w_data(w_data),
        .empty(t_fifo_empty),                    .full(t_full),
    .r_data(t_fifo_out)
    );
    assign t_fifo_not_empty = ~t_fifo_empty;
endmodule
```

```
module baud
#(parameter M = 27, N = 5) // @ 50 MHz
(
    input  wire          clk, reset,
    output wire          max_tick,
    output wire [N-1:0]   q
);
    reg [N-1:0] r_reg;
    wire [N-1:0] r_next;
    always @(posedge clk, posedge reset)
        if (reset)
            r_reg <= 0;
        else
            r_reg <= r_next;
    assign r_next   = (r_reg == (M-1)) ? 0 : r_reg + 1;
    assign q        = r_reg;
    assign max_tick = (r_reg == (M-1));
endmodule
```

```
module receiver #(
    parameter DATA_BITS = 8,
    parameter STOP_TICKS = 16
)(
    input  wire clk, reset,
    input  wire reciever, s_tick,       // serial input
    output reg  r_done_tick,
    output wire [7:0] data_out
);
    // State encoding
    localparam [1:0]
        IDLE  = 2'b00,
        START = 2'b01,
        DATA  = 2'b10,
        STOP  = 2'b11;
    // Registers
    reg [1:0] state_reg, state_next;
    reg [3:0] s_reg, s_next;        // oversample counter
    reg [2:0] n_reg, n_next;        // bit counter
    reg [7:0] b_reg, b_next;        // shift register
```

```
// Sequential part
always @(posedge clk, posedge reset) begin
   if (reset) begin
      state_reg <= IDLE;
      s_reg     <= 0;
      n_reg     <= 0;
      b_reg     <= 0;
   end else begin
      state_reg <= state_next;
      s_reg     <= s_next;
      n_reg     <= n_next;
      b_reg     <= b_next;
   end
end
// Next-state logic
always @* begin
   // defaults
   state_next  = state_reg;
   s_next      = s_reg;
   n_next      = n_reg;
   b_next      = b_reg;
r_done_tick = 1'b0;
case (state_reg)
   // wait for start bit
   IDLE: begin
      if (~reciever) begin        // line low = start bit
         state_next = START;
         s_next = 0;
      end
   end
   // confirm start bit (sample at midpoint = 8th tick)
   START: begin
      if (s_tick) begin
         if (s_reg == 7) begin
            state_next = DATA;
            s_next = 0;
            n_next = 0;
         end else
            s_next = s_reg + 1;
      end
```

```
        end
        // receive data bits
        DATA: begin
          if (s_tick) begin
            if (s_reg == 15) begin
              s_next = 0;
              b_next = {reciever, b_reg[7:1]};  // shift in
LSB first
              if (n_reg == (DATA_BITS-1))
                state_next = STOP;
              else
                n_next = n_reg + 1;
            end else
              s_next = s_reg + 1;
          end
        end
        // check stop bit
        STOP: begin
          if (s_tick) begin
            if (s_reg == (STOP_TICKS-1)) begin
              state_next  = IDLE;
              r_done_tick = 1'b1;  // byte received
            end else
              s_next = s_reg + 1;
          end
        end
        default: state_next = IDLE;
      endcase
    end
    // Output
    assign data_out = b_reg;
  endmodule
```

```
  module transmitter #(
    parameter DATA_BITS = 8,
    parameter STOP_TICKS = 16
  )(
    input  wire clk, reset,
    input  wire t_start, s_tick,
    input  wire [7:0] data_in,
    output reg  t_done_tick,
    output wire transmitter
  );
    // State encoding
```

```
localparam [1:0]
   IDLE  = 2'b00,
   START = 2'b01,
   DATA  = 2'b10,
   STOP  = 2'b11;
// State registers
reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;         // sample tick counter
reg [2:0] n_reg, n_next;         // data bit counter
reg [7:0] b_reg, b_next;         // data shift register
reg t_reg, t_next;               // tx output buffer
// Sequential logic
always @(posedge clk, posedge reset) begin
   if (reset) begin
      state_reg <= IDLE;
      s_reg     <= 0;
      n_reg     <= 0;
      b_reg     <= 0;
      t_reg     <= 1'b1; // idle line = high
   end else begin
      state_reg <= state_next;
      s_reg     <= s_next;
      n_reg     <= n_next;
      b_reg     <= b_next;
      t_reg     <= t_next;
   end
end
// Next-state logic
always @* begin
// defaults
state_next   = state_reg;
s_next       = s_reg;
n_next       = n_reg;
b_next       = b_reg;
t_next       = t_reg;
t_done_tick  = 1'b0;
case (state_reg)
   IDLE: begin
      t_next = 1'b1; // idle line
      if (t_start) begin
         state_next = START;
         s_next = 0;
         b_next = data_in;
      end
   end
   START: begin
      t_next = 1'b0; // start bit
      if (s_tick) begin
         if (s_reg == 15) begin
            state_next = DATA;
            s_next = 0;
            n_next = 0;
         end else
            s_next = s_reg + 1;
      end
   end
   DATA: begin
      t_next = b_reg[0]; // send LSB
```

```
            if (s_tick) begin
               if (s_reg == 15) begin
                  s_next = 0;
                  b_next = b_reg >> 1;
                  if (n_reg == (DATA_BITS-1))
                     state_next = STOP;
                  else
                     n_next = n_reg + 1;
               end else
                  s_next = s_reg + 1;
            end
         end
         STOP: begin
            t_next = 1'b1; // stop bit = high
            if (s_tick) begin
               if (s_reg == (STOP_TICKS-1)) begin
                  state_next = IDLE;
                  t_done_tick = 1'b1;
               end else
                  s_next = s_reg + 1;
            end
         end
         default: state_next = IDLE;
      endcase
   end
   // output assignment
   assign transmitter = t_reg;
endmodule
```

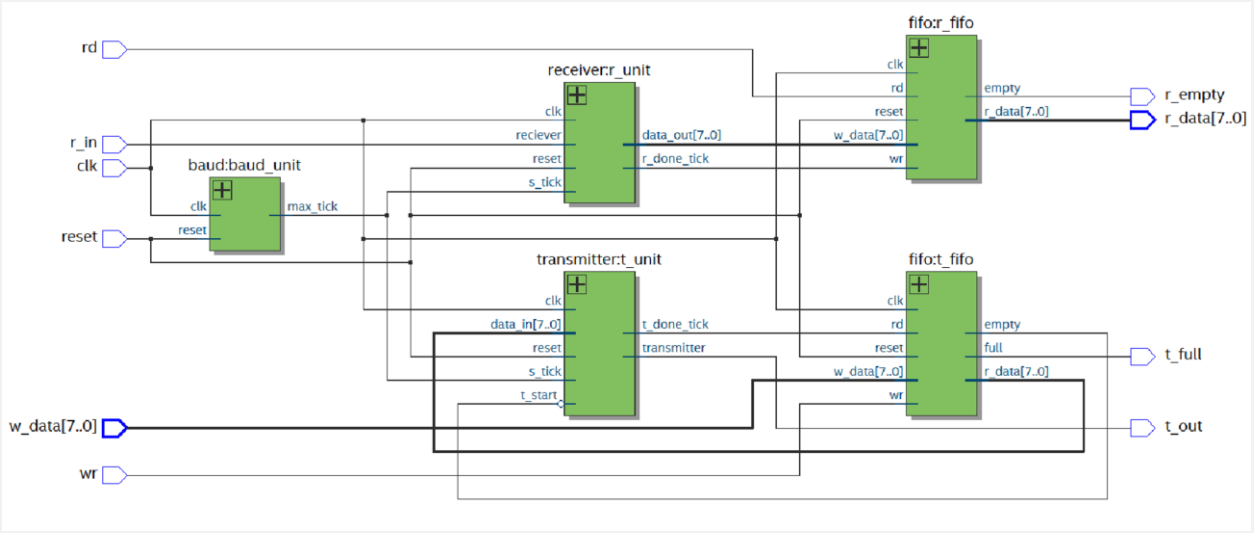
```
module fifo
#(
   parameter data_bit = 8,   // width of data (8-bit)
   parameter fifo     = 4    // address bits -> depth = 2^fifo
)
(
   input  wire               clk, reset,
   input  wire               rd, wr,
   input  wire [data_bit-1:0]   w_data,
   output wire               empty, full,
   output wire [data_bit-1:0]   r_data
);
   localparam DEPTH = (1 << fifo);   // FIFO depth (2^fifo)
   // memory array
   reg [data_bit-1:0] array_reg [0:DEPTH-1];
   // pointers and counter
   reg [fifo-1:0] w_ptr_reg, w_ptr_next;
   reg [fifo-1:0] r_ptr_reg, r_ptr_next;
   reg [fifo:0]    count_reg, count_next;    // need extra bit for full detection
   reg [data_bit-1:0] r_data_reg;
   // sequential part
   always @(posedge clk, posedge reset)
      if (reset) begin
         w_ptr_reg <= 0;
         r_ptr_reg <= 0;
         count_reg <= 0;
      end else begin
         w_ptr_reg <= w_ptr_next;
```

```
        r_ptr_reg <= r_ptr_next;
        count_reg <= count_next;
    end
// write memory (synchronous)
always @(posedge clk)
    if (wr & ~full)
        array_reg[w_ptr_reg] <= w_data;
// read memory (synchronous)
always @(posedge clk)
    if (rd & ~empty)
        r_data_reg <= array_reg[r_ptr_reg];


// next-state logic
always @* begin
    // defaults
    w_ptr_next = w_ptr_reg;
    r_ptr_next = r_ptr_reg;
    count_next = count_reg;
    // write
    if (wr & ~full)
        w_ptr_next = w_ptr_reg + 1;
    // read
    if (rd & ~empty)
        r_ptr_next = r_ptr_reg + 1;
    // count update
    case ({wr & ~full, rd & ~empty})
        2'b10: count_next = count_reg + 1; // write only
        2'b01: count_next = count_reg - 1; // read only
        default: count_next = count_reg;   // no change or both
        endcase
    end
    // outputs
    assign r_data = r_data_reg;
    assign full  = (count_reg == DEPTH);
    assign empty = (count_reg == 0);
endmodule
```

8.2 Simulation Wave:

8.3 RTL View:


rd
r_in
clk
reset
w_data[7..0]
wr
baud:baud_unit
clk
reset
max_tick
receiver:r_unit
clk
reciever
reset
s_tick
data_out[7..0]
r_done_tick
transmitter:t_unit
clk
data_in[7..0]
reset
s_tick
t_start
t_done_tick
transmitter
fifo:r_fifo
clk
rd
reset
w_data[7..0]
wr
empty
r_data[7..0]
r_empty
r_data[7..0]
fifo:t_fifo
clk
rd
reset
w_data[7..0]
wr
empty
full
r_data[7..0]
t_full
t_out