Digital IC Design using FPGA

Universal Asynchronous Receiver-Transmitter (UART) RTL Implementation

-Implemented with Verilog-

Table of Contents

List of Figures

Introduction

UART, or Universal Asynchronous Receiver-Transmitter, is a fundamental hardware communication protocol that facilitates asynchronous serial data exchange between two digital devices. It plays a vital role in a wide range of electronic systems, including microcontrollers, embedded systems, sensors, and computer peripherals. Unlike parallel communication—where multiple data bits are transmitted simultaneously across multiple lines—UART operates over just two wires: one for transmitting data (TX) and another for receiving data (RX), making it a cost-effective, low-pin-count, and widely adopted solution for short-distance, low- to medium-speed communication.

At its core, a UART is either a dedicated integrated circuit or a built-in module within a microcontroller that performs the conversion between parallel and serial formats. During transmission, UART captures parallel data from the processor's data bus, serializes it, and adds protocol-specific framing (start bit, data bits, optional parity bit, and stop bit) before sending it sequentially over the TX line. During reception, the process is reversed: the UART samples the incoming serial data stream, detects the start bit, reconstructs the original parallel data, and verifies it based on the agreed communication parameters. The asynchronous nature of UART means there is no shared clock between the transmitter and receiver; instead, both must operate at the same predefined baud rate and configuration to interpret data correctly.

This report focuses on the design and RTL implementation of a UART system using Verilog HDL, a hardware description language commonly used in digital circuit design. The UART system includes key components such as a baud rate generator, a transmitter (TX) module, and a receiver (RX) module. The receiver design incorporates 16x oversampling to improve bit detection reliability, especially under timing mismatch or jitter conditions. Each module is described using Register Transfer Level (RTL) abstraction, which models hardware in terms of the flow of data between registers and the logical operations performed on that data.

The design process follows a modular approach, with each component developed, simulated, and verified independently through testbenches written in Verilog. This ensures that individual modules behave correctly before integrating them into the complete system. Key design challenges addressed include synchronization of asynchronous inputs, timing accuracy, and robust handling of framing and data integrity. A solid understanding of UART protocol specifications—such as frame structure, baud rate calculation, parity, and stop bit handling—is essential for guiding architectural decisions and implementing reliable communication logic.

Through this project, the RTL design of UART provides not only practical applications of digital design principles but also reinforces core concepts such as state machine design, sampling theory, serial data framing, and testbench verification—all of which are critical in real-world hardware design workflows.

Basic Concepts of UART

Asynchronous Communication

The most defining characteristic of UART is its asynchronous operation. This implies that the sender and receiver do not share a common clock signal. Instead, each device uses its own internal clock, and synchronization is achieved through start and stop bits. A start bit, typically a logic low, signals the beginning of a new data frame, while one or more stop bits, usually a logic high, mark the end. This allows the receiving UART to synchronize its internal clock with the incoming data stream for the duration of each frame.

Baud Rate

The baud rate defines the speed at which data is transmitted, measured in bits per second (bps). Both the transmitting and receiving UARTs must be configured to the same baud rate for successful communication. Common baud rates include 9600, 19200, 38400, 57600, and 115200 bps. A mismatch in baud rates can lead to misinterpretation of data, resulting in communication errors.

Data Bits

UART communication typically involves sending data in frames, where each frame contains a certain number of data bits. The most common configurations are 7 or 8 data bits, though other values are possible. This parameter determines the size of the data unit being transmitted in each frame.

Parity Bit

An optional parity bit can be included in the data frame for error detection. Parity checking is a simple method to detect single-bit errors during transmission. There are two types of parties: even and odd. In even parity, the parity bit is set or cleared to ensure that the total number of '1's in the data bits and parity bit is even. In odd parity, the parity bit ensures an odd number of '1's. If the receiver calculates a different parity, it indicates a transmission error.

Stop Bits

Stop bits are used to signal the end of a data frame and provide a brief idle period, allowing the receiving UART to process the received data and prepare for the next frame. Typically, 1 or 2 stop bits are used. The stop bits are always at logic high level.

Design Methodology

UART RECEIVING SUBSYSTEM

In UART communication, the transmitted signal does not include a clock, meaning the receiver has no explicit timing reference to know exactly when each bit begins and ends. Instead, both the transmitter and receiver must rely on predefined parameters—such as the baud rate—to stay synchronized in time.

To accurately recover the transmitted data, the receiver uses an oversampling technique. In this approach, the incoming serial signal is sampled multiple times per bit period—commonly 16 times—to precisely estimate the timing of each bit. Once the start bit is detected, the receiver waits for a calculated number of sampling ticks to reach the center of each data bit, where the signal is most stable and less prone to noise or jitter. Data bits are then sampled at these midpoint positions to reconstruct the original byte accurately.

This oversampling method improves robustness and allows the receiver to reliably extract data even when there are slight mismatches between the transmitter and receiver clocks.

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 8 data bits and 1 stop bits. The oversampling scheme works as follows:

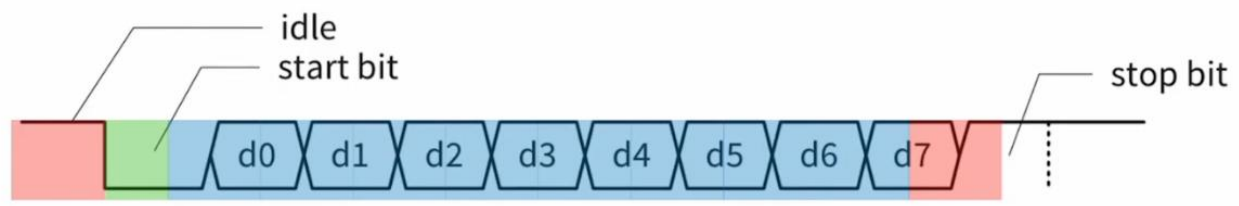

Figure 1. UART Receiving Subsystem Design Flow

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, 7 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, 1 more time to obtain the stop bits.

The oversampling scheme basically performs the function of a clock signal. Instead of using the rising edge to indicate when the input signal is valid, it utilizes sampling ticks to estimate the middle point of each bit. While the receiver has no information about the exact onset time of the start bit, the estimation can be off by at most 1/16. The subsequent data bit retrievals are off by at most 1/16 from the middle point as well. Because of the oversampling, the baud rate can be only a small fraction of the system clock rate, and thus this scheme is not appropriate for a high data rate.

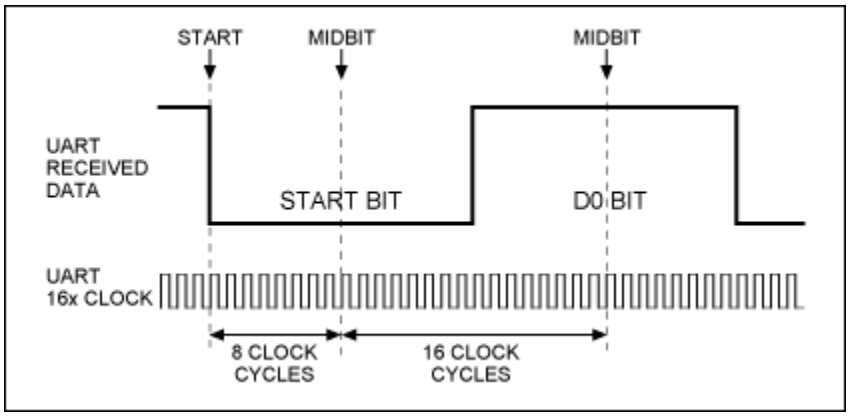

Figure 2. Oversampling

Top Level Design Diagram

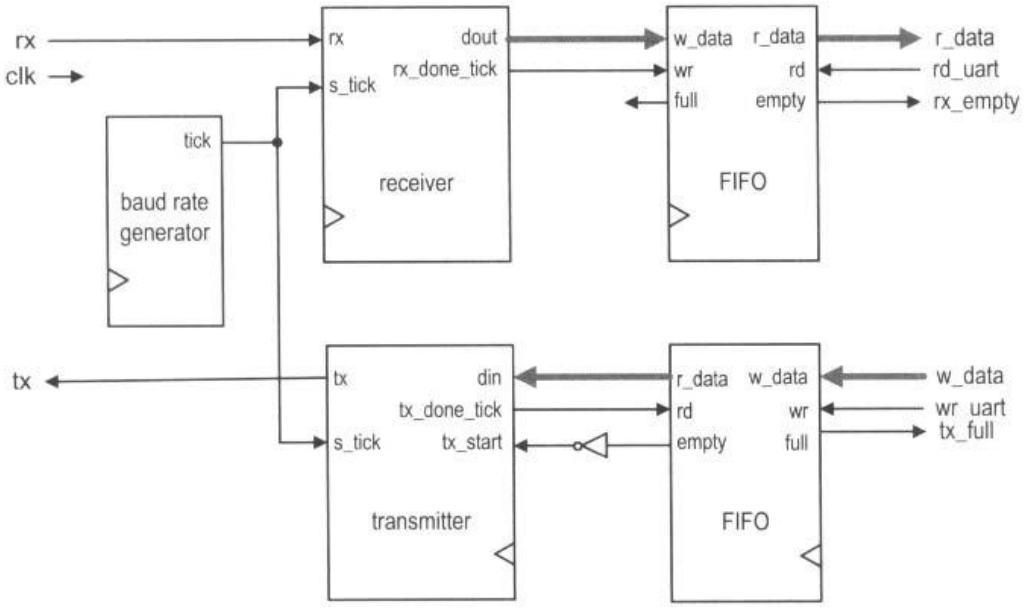

Figure 3. Top-Level UART Design Diagram

Component Breakdown

The system is composed of four primary blocks: the Baud Rate Generator, the Receiver, the Transmitter, and two FIFO buffers.

Baud Rate Generator

The Baud Rate Generator is the timing heart of the UART. Its primary purpose is to generate precise timing ticks for the receiver and transmitter.

- **Function:** It generates a high-frequency sampling tick (s_tick) that is a multiple of the agreed-upon baud rate (bits per second).
- **Inputs:** It takes the main system clk (clock) as input.
- **Outputs:** It produces the s_tick signal.
- **Operation:** The core of this block is a modulo-M counter that divides the high-frequency system clk. To reliably sample the incoming serial line, a technique called oversampling is used. A common oversampling rate is 16 times the baud rate. For example, to achieve a 19,200 baud rate with a 50 MHz system clock, the s_tick frequency must be 307,200 Hz (19,200 * 16). The counter's modulus would be 163 (50,000,000 / 307,200 ≈ 163). The s_tick is asserted for one clock cycle every 163 system clock cycles, providing 16 ticks per bit period.

Receiver Module

The Receiver module's function is to detect an incoming transmission, sample the serial data at the correct moments, and reassemble the parallel data byte.

- **Function:** The receiver implements an Algorithmic State Machine with Datapath (ASMD) to manage the reception process. It operates as follows:
- **Idle State:** Continuously monitors the rx line for a falling edge, which indicates a start bit.
- **Start State:** Upon detecting a start bit, it uses the s_tick to count to the middle of the start bit (e.g., to the 8th tick in a 16x oversampling scheme) to confirm it's a valid start bit.
- **Data State:** It then repeatedly counts 16 s_ticks to find the middle of each subsequent data bit, shifting the sampled value of rx into a data register. This is repeated for all data bits.
- **Stop State:** After receiving all data bits, it waits for another bit period and checks if the rx line is high ('1'), corresponding to the stop bit.
- **Inputs:**
    - rx: The serial data input line.
    - clk: The main system clock.
    - s_tick: The 16x oversampling tick from the Baud Rate Generator.
- **Outputs:**
    - dout: The reassembled 8-bit parallel data output.

- rx_done_tick: A pulse asserted for one clock cycle after a valid stop bit is received, indicating the data on dout is valid.

Transmitter Module

The Transmitter module performs the reverse operation of the receiver. It takes a parallel byte and transmits it serially according to the UART frame format.

- **Function:** The transmitter is essentially a specialized parallel-in, serial-out shift register. When triggered by tx_start, it loads the parallel data from din and proceeds through a state machine:
  - **Start State:** It drives the tx line low for one full bit period (e.g., 16 s_ticks) to create the start bit.
  - **Data State:** It shifts out the data byte one bit at a time, from LSB to MSB, holding each bit value on the tx line for one bit period.
  - **Stop State:** It drives the tx line high for one or more bit periods to create the stop bit(s).
- **Inputs:**
  - din: The 8-bit parallel data input from the transmit FIFO.
  - tx_start: A control signal to initiate transmission.
  - s_tick: The timing tick from the Baud Rate Generator.
  - clk: The main system clock.
- **Outputs:**
  - tx: The serial data output line.
  - tx_done_tick: A pulse asserted for one clock cycle after the stop bit has been fully transmitted, indicating the transmitter is ready for new data.

FIFO (First-In, First-Out) Buffers

The FIFO buffers are critical for decoupling the real-time UART operations from the host system. They provide elasticity, preventing data loss when the host system cannot service the UART instantaneously.

Receive FIFO:

- **Purpose:** To store incoming bytes from the Receiver, preventing **data overrun errors**. An overrun occurs if a new byte is fully received before the host system has read the previous one. The FIFO provides a multi-word buffer, giving the host more time to retrieve data.

- **Interaction:** The rx_done_tick from the Receiver acts as the FIFO's write enable (wr), automatically storing the received byte from dout. The host reads from the FIFO when the rx_empty flag is not asserted.

Transmit FIFO:

- **Purpose:** To store outgoing bytes from the host system. This allows the host to write a burst of data to the UART and continue with other tasks, while the FIFO ensures the transmitter is continuously supplied with data, preventing **data underrun** (an idle transmitter).
- **Interaction:** The host writes to the FIFO when the tx_full flag is not asserted. The FIFO's empty signal is inverted to create the tx_start signal, automatically triggering a new transmission as long as there is data in the buffer.

Overall Data Flow

Receive Path

1. Serial data arrives on the rx pin.
2. The Receiver module uses the oversampling s_tick to detect the frame and assemble a parallel byte.
3. Upon completion, the Receiver pulses rx_done_tick, which writes the byte from dout into the Receive FIFO.
4. The host system monitors the rx_empty flag. When low, the host asserts rd_uart to read the oldest byte from the FIFO.

Transmit Path

1. The host checks the tx_full flag. If low, it writes a byte to the Transmit FIFO by asserting wr_uart.
2. The Transmit FIFO's empty flag goes low. This is inverted to assert tx_start, triggering the Transmitter.
3. The Transmitter reads the byte from the FIFO, constructs the serial frame (start, data, stop bits), and shifts it out on the tx pin.
4. Upon completion, the transmitter pulses tx_done_tick, which acts as the read signal for the Transmit FIFO, removing the byte that was just sent. The process repeats if the FIFO is still not empty.

Fully Configurable UART RTL Implementation

Timer Input (timer_input.v)

The timer_input module is a generic, parameterized counter designed to generate a single-cycle tick output when its internal count reaches a specified FINAL_VALUE. This module serves as a fundamental building block for various timing-related functionalities within the larger system, such as the baud_generator.

Parameters

Parameter Name	Description
BITS	Specifies the width of the internal counter and the FINAL_VALUE input.

Ports

Port Name	Direction	Description
clk	Input	System clock.
reset_n	Input	Active-low asynchronous reset.
enable	Input	When high, the counter increments. When low, the counter holds its value.
FINAL_VALUE	Input	The target value for the counter. When the internal count reaches this value, tick is asserted.
tick	Output	High for one clock cycle when the internal counter equals FINAL_VALUE.

Internal Logic

The timer_input module consists of a synchronous counter (Q_reg) and combinational logic to determine the next state (Q_next) and the tick output.

- **Counter (Q_reg)**: This register holds the current count. It is updated on the positive edge of clk or asynchronous negative edge of reset_n. If reset_n is low, Q_reg is cleared. If enable is high, Q_reg is updated with Q_next; otherwise, it retains its current value.
- **Next State Logic (Q_next)**: The Q_next signal is determined combinatorially. If Q_reg is greater than or equal to FINAL_VALUE, Q_next is reset to 0. Otherwise, if tick is asserted (meaning Q_reg just reached FINAL_VALUE), Q_next is also reset to 0. In all other cases, Q_next increments Q_reg by 1. This logic ensures that the counter rolls over to 0 once FINAL_VALUE is reached.
- **Tick Generation**: The tick output is asserted (1'b1) when Q_reg is equal to FINAL_VALUE. This provides a single-cycle pulse at the desired interval.

This modular design allows the timer_input to be reused across different parts of an RTL design where precise timing and counting are required, simply by adjusting the BITS parameter and FINAL_VALUE input.

Baud Rate Generator (baud_generator.v)

The baud_generator module is a critical component responsible for generating the fundamental timing signal (tick) required for UART communication. This signal operates at 16 times the desired baud rate, enabling accurate oversampling for the receiver and precise bit timing for the transmitter. The module's primary function is to convert a baud_rate_sel input into a specific counter FINAL_VALUE that is then used by an instantiated timer_input module.

Ports

The baud_generator module has the following ports:

Port Name	Direction	Description
clk	Input	System clock.
reset_n	Input	Active-low asynchronous reset.
baud_rate_sel	Input	4-bit input to select the desired baud rate.
tick	Output	Output pulse, high for one clock cycle when the baud rate period elapses.

Baud Rate Selection

The baud_rate_sel input (4 bits) allows the user to select from a predefined set of common baud rates. The module internally maps these selection values to a FINAL_VALUE for the timer_input module. The FINAL_VALUE is calculated based on a 100MHz system clock and the formula:

$$FINAL\_VALUE = \frac{System\_Clock\_Frequency}{16 \times Baud\_Rate} - 1$$

The mapping is as follows:

baud_rate_sel	Baud Rate (bps)	FINAL_VALUE (for 100MHz clock)
4'd0	600	10416
4'd1	1200	5208
4'd2	2400	2604
4'd3	4800	1302
4'd4	9600	651
4'd5	19200	325

4'd6	38400	162
4'd7	57600	108
4'd8	115200	54
Default	9600	651

Internal Logic

The baud_generator module contains a combinational always @(baud_rate_sel) block that sets the FINAL_VALUE based on the baud_rate_sel input. This FINAL_VALUE is then passed as a parameter to the timer_input instance. The timer_input module is instantiated with BITS=14 to accommodate the largest FINAL_VALUE (10416, which requires 14 bits to represent, as $2^{13} = 8192$ and $2^{14} = 16384$). The enable input of the timer_input is tied high (1'b1), ensuring that the baud rate generation is continuously active when the system is not in reset.

UART Receiver (uart_rx.v)

The uart_rx module implements the receiver portion of the UART protocol, responsible for converting incoming serial data into parallel data. It is a state machine-driven design that precisely samples the serial input (rx) based on the s_tick signal (generated at 16x the baud rate) to accurately capture each bit.

Ports

Port Name	Direction	Description
clk	Input	System clock.
reset_n	Input	Active-low asynchronous reset.
rx	Input	Serial data input line.
s_tick	Input	Baud rate tick signal (16x baud rate).
dbit_select_i	Input	3-bit input to select the number of data bits (000: 5 bits to 011: 8 bits).
sbit_select_i	Input	2-bit input to select the number of stop bits (00: 1 stop, 01: 1.5 stop, 10: 2 stop bits).
parity_select_i	Input	2-bit input to select parity type (00: None, 01: Even, 10: Odd).
rx_done_tick	Output	High for one clock cycle when a complete byte has been received.
rx_dout	Output	8-bit parallel output for the received data byte.
parity_error	Output	High if a parity error is detected.
frame_error	Output	High if a framing error (incorrect stop bit) is detected.

State Machine

The uart_rx module operates using a finite state machine (FSM) with the following states:

- **idle (3'd0)**: The initial state, waiting for a falling edge on the rx line, which signifies the start bit.
- **start (3'd1)**: After detecting a falling edge, this state samples the rx line in the middle of the start bit (at s_reg == 7) to confirm it's a valid start bit. If rx is still low, it proceeds to the data state; otherwise, it returns to idle (false start).
- **data (3'd2)**: In this state, the module samples the rx line at the middle of each data bit (at s_reg == 15) and shifts the received bit into the b_reg (data shift register). It continues until all data bits (configured by dbit_select_i) have been received.
- **parity (3'd3)**: If parity is enabled (parity_en is high), the FSM transitions to this state to receive the parity bit. It samples the rx line at the middle of the parity bit period.
- **stop (3'd4)**: This state samples the rx line to verify the stop bit(s). The duration of this state depends on the sbit_select_i input. If the rx line is not high during the stop bit period, a frame_error is asserted. Upon successful reception of the stop bit(s), rx_done_tick is asserted, rx_dout is updated, and the FSM returns to idle.

Data Bit, Stop Bit, and Parity Configuration

- **Data Bits**: The dbit_select_i input determines the number of data bits to receive. data_bits_to_rx is derived from this input (dbit_select_i + 3'd4), indicating the index of the last data bit (e.g., for 5 data bits, data_bits_to_rx is 4).
- **Stop Bits**: The sbit_select_i input configures the stop bit duration. stop_ticks is calculated to represent the total s_tick counts for 1, 1.5, or 2 stop bits.
- **Parity**: The parity_select_i input enables and selects the parity type (None, Even, or Odd). The module calculates the expected parity (parity_calc) by XORing the received data bits. For Even parity, parity_expected is parity_calc; for odd parity, it's ~parity_calc. If the received parity bit (p_reg) does not match parity_expected, parity_error is asserted.

Oversampling

The uart_rx module employs a 16x oversampling technique, meaning it samples the rx line 16 times per bit period. This is achieved by using the s_tick signal, which pulses at 16 times the baud rate. Sampling is typically performed in the middle of each bit period (e.g., at s_reg == 7 for the start bit, and s_reg == 15 for data and parity bits) to maximize noise immunity and accurately capture the bit value.

UART Transmitter (uart_tx.v)

The uart_tx module is responsible for transmitting parallel data serially over the UART interface. It takes 8-bit data, along with configuration for data bits, stop bits, and parity, and converts it into a serial bitstream, including start and stop bits.

Ports

Port Name	Direction	Description
clk	Input	System clock.
reset_n	Input	Active-low asynchronous reset.
tx_start	Input	Active high pulse to initiate a transmission.
s_tick	Input	Baud rate tick signal (16x baud rate).
tx_din	Input	8-bit data input to be transmitted.
dbit_select_i	Input	3-bit input to select the number of data bits (000: 5 bits to 011: 8 bits).
sbit_select_i	Input	2-bit input to select the number of stop bits (00: 1 stop, 01: 1.5 stop, 10: 2 stop bits).
parity_select_i	Input	2-bit input to select parity type (00: None, 01: Even, 10: Odd).
tx_done_tick	Output	High for one clock cycle when the transmission is complete.
tx	Output	Serial data output line.

State Machine

The uart_tx module operates using a finite state machine (FSM) with the following states:

- **idle (0)**: The initial state. The tx line is held high. The module waits for tx_start to be asserted to begin a transmission.
- **start (1)**: Upon tx_start assertion, the tx line is driven low to transmit the start bit. It remains in this state for 16 s_tick cycles (one bit period).
- **data (2)**: In this state, the module transmits the data bits, LSB first. For each data bit, the tx line is driven with the current data bit for 16 s_tick cycles. The b_reg (data shift register) is shifted right after each bit transmission. The number of data bits transmitted is determined by dbit_select_i.
- **parity (3)**: If parity is enabled, the FSM transitions to this state to transmit the calculated parity bit. The tx line is driven with the parity bit (p_reg) for 16 s_tick cycles.
- **stop (4)**: This state transmits the stop bit(s). The tx line is driven high for the duration specified by sbit_select_i (1, 1.5, or 2 bit periods). After the stop bit(s) are transmitted, tx_done_tick is asserted for one clock cycle, and the FSM returns to the idle state.

Timing

The uart_tx module relies on the s_tick signal, which pulses at 16 times the baud rate, to precisely time the transmission of each bit. Each bit (start, data, parity, stop) is held on the tx line for 16 s_tick cycles, ensuring proper bit duration according to the UART protocol.

Top-Level Transceiver (uart_transceiver.v)

The uart_transceiver module is the top-level entity that integrates all the sub-modules to form a complete full-duplex UART communication system. It orchestrates the data flow between the external system and the UART core, providing configurable options for baud rate, data format, and error handling.

Ports

Port Name	Direction	Description
clk	Input	System clock.
reset_n	Input	Active-low asynchronous reset.
r_data	Output	8-bit received data output from the RX FIFO.
rd_uart	Input	Read enable for the RX FIFO. Assert high to read data.
rx_empty	Output	High when the RX FIFO is empty.
rx	Input	Serial data input from the external world.
parity_error	Output	High if a parity error is detected during reception.
baud_rate_sel	Input	4-bit input to select the desired baud rate.
dbit_select_i	Input	3-bit input to select the number of data bits (000: 5 bits to 011: 8 bits).
sbit_select_i	Input	2-bit input to select the number of stop bits (00: 1 stop, 01: 1.5 stop, 10: 2 stop bits).
parity_select_i	Input	2-bit input to select parity type (00: None, 01: Even, 10: Odd).
w_data	Input	8-bit data input to be transmitted, written to the TX FIFO.
wr_uart	Input	Write enable for the TX FIFO. Assert high to write data.
tx_full	Output	High when the TX FIFO is full.
tx	Output	Serial data output to the external world.

Internal Connections and Instantiations

The uart_transceiver module instantiates and connects the following sub-modules:

- **baud_generator**: Generates the s_tick signal based on baud_rate_sel. This s_tick is then distributed to both the uart_rx and uart_tx modules to synchronize their operations.
- **uart_rx**: Receives serial data from the rx input. Its rx_done_tick and rx_dout outputs are connected to the RX FIFO. The configuration inputs (dbit_select_i, sbit_select_i, parity_select_i) are passed directly from the uart_transceiver inputs.

- **fifo_generator_0 (RX FIFO)**: This FIFO (assuming it's a generic FIFO IP core) buffers the received data from uart_rx. Data is written into the FIFO when rx_done_tick is asserted and read out when rd_uart is asserted by the external system. The rx_empty output indicates the FIFO's status.
- **fifo_generator_0 (TX FIFO)**: This FIFO buffers data to be transmitted. Data is written into the FIFO from w_data when wr_uart is asserted by the external system. Data is read from the FIFO by uart_tx when tx_done_tick is asserted. The tx_full output indicates the FIFO's status.
- **uart_tx**: Transmits serial data to the tx output. It reads data from the TX FIFO (tx_fifo_out) and is triggered to start transmission when the TX FIFO is not empty (~tx_fifo_empty). The configuration inputs (dbit_select_i, sbit_select_i, parity_select_i) are passed directly from the uart_transceiver inputs.

Data Flow and Control

1. **Baud Rate Synchronization**: The baud_generator creates the s_tick signal, which is the fundamental timing reference for both reception and transmission.
2. **Reception Path**: Incoming serial data on rx is processed by uart_rx. Once a byte is successfully received, uart_rx asserts rx_done_tick and provides the data on rx_dout. This triggers a write operation to the RX FIFO. The external system can read data from the RX FIFO by asserting rd_uart, and monitor its status via rx_empty.
3. **Transmission Path**: The external system writes data to be transmitted into the TX FIFO via w_data and wr_uart. When the TX FIFO is not empty, uart_tx is signaled to start transmission (tx_start is tied to ~tx_fifo_empty). uart_tx reads data from the FIFO, serializes it, and outputs it on tx. Upon completion of a transmission, uart_tx asserts tx_done_tick, which triggers a read from the TX FIFO.
4. **Error Handling**: The parity_error output from uart_rx is directly exposed at the uart_transceiver level, allowing the external system to detect data integrity issues. The frame_error output of uart_rx is currently unconnected but could be exposed if needed.

This integrated design provides a robust and flexible UART solution, abstracting the complexities of serial communication behind a clear and manageable interface.

Basic ASMD Charts

ASMD chart of a UART receiver

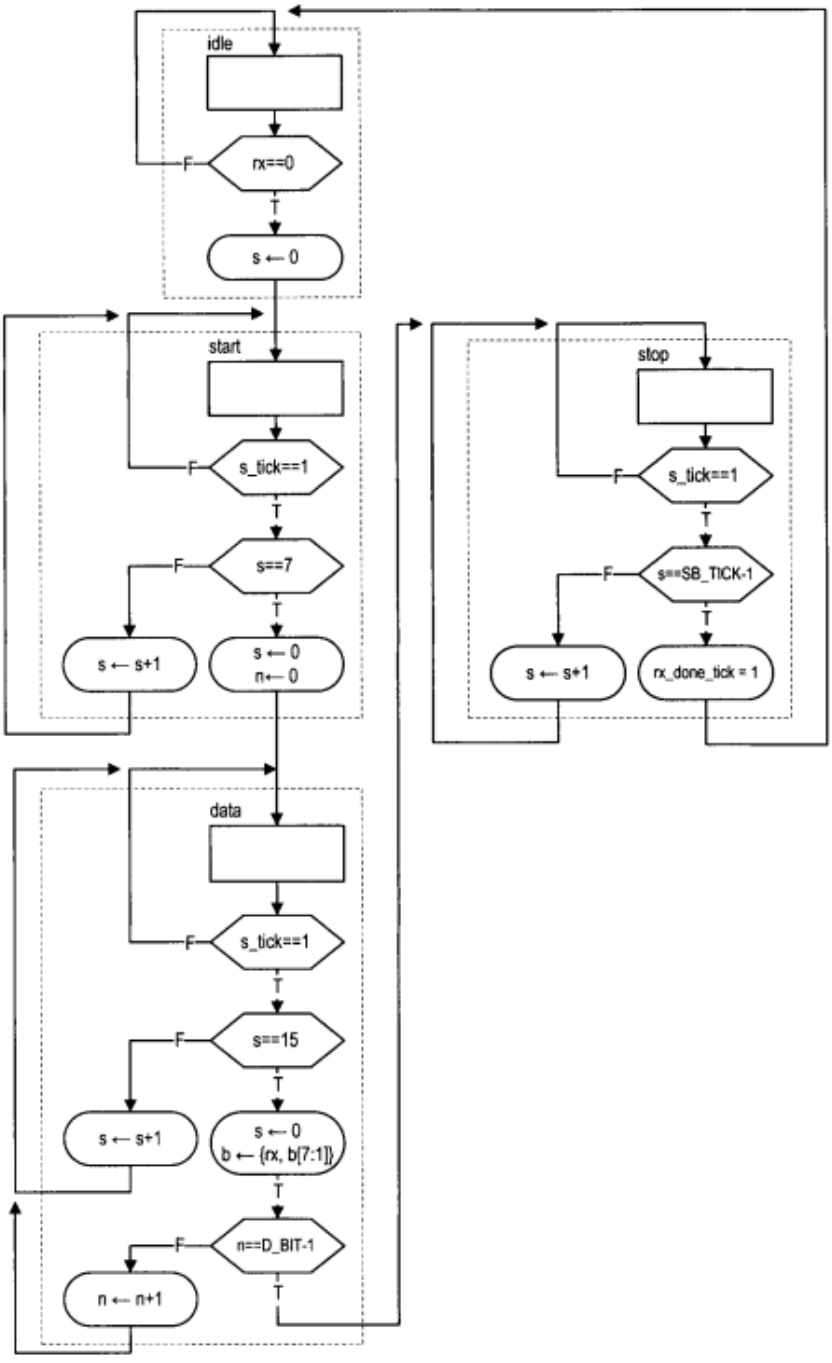

Figure 4. Reciever ADSM

ASMD chart of a UART transmitter

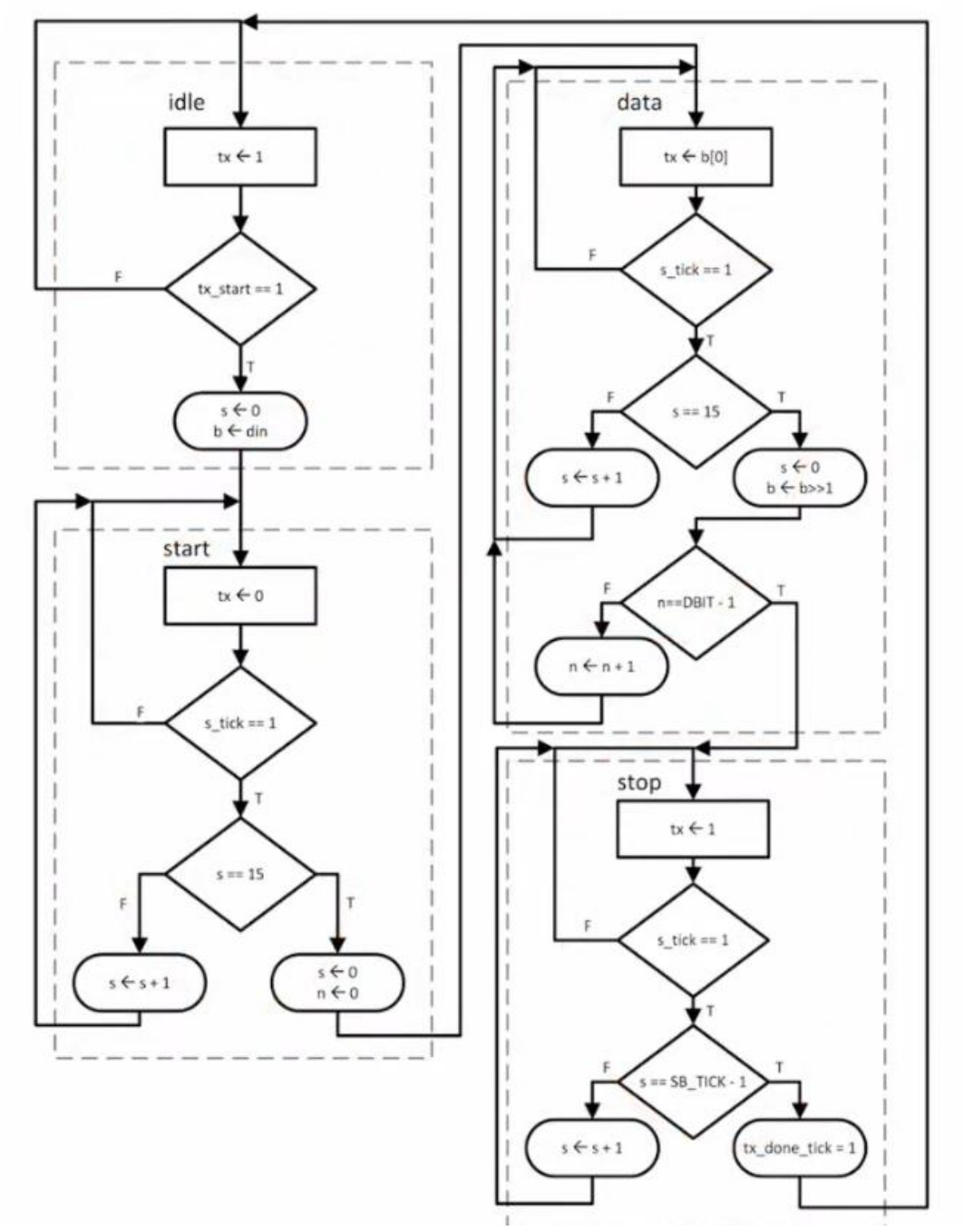

Figure 5. Transmitter ADSM

Testbenches & Simulation

Receiver

The UART receiver testbench is designed to verify the correctness of the receiver module under various standard frame formats. It systematically transmits serial data frames with different configurations to ensure that the receiver can accurately decode them.

Each frame includes:

- A start bit (always 0)
- A configurable number of data bits (5 to 8)
- An optional parity bit (even or odd)
- One or two stop bits (always 1)

The testbench uses a task ***send_frame(data, data_bits, parity_type, stop_bits)*** to serialize a full UART frame according to the specified format. The receiver samples the incoming serial data using 16× oversampling.

#	Data	Format	Description
1	0xAC	8N1	8 data, no parity, 1 stop
2	0x15	7E2	7 data, even parity, 2 stop
3	0x1E	6O1	6 data, odd parity, 1 stop
4	0x1B	5N1	5 data, no parity, 1 stop
5	0xA5	8E1	8 data, even parity, 1 stop
6	0x55	8O2	8 data, odd parity, 2 stop

Each frame is verified against expected output, checking:

- Correct reception of data bits
- Correct handling of parity
- Detection of parity errors

The simulation confirms that the receiver successfully decodes each valid frame. For example, the first frame (0xAC) is correctly interpreted in 8N1 format (8 data bits, no parity, 1 stop bit), while another frame (0x1B) is accurately processed in 5N1 format. Upon successful reception, the rx_done_tick signal asserts for one clock cycle, and the received data is output through rx_dout.

Parity logic was verified using both even (e.g., 0x15, 0xA5) and odd (e.g., 0x1E, 0x55) parity configurations. The third frame (0x1E) was specifically included to validate odd parity handling, which the receiver passed correctly. In scenarios with incorrect parity (if injected), the receiver is capable of identifying the mismatch and sets the parity_error signal high, confirming the functionality of the error detection mechanism.

Furthermore, the receiver handled both one and two stop bit formats correctly, as demonstrated by frames configured with sbit_select = 0 and sbit_select = 2. This validates proper frame delimitation and ensures reliable operation under different stop bit conditions.

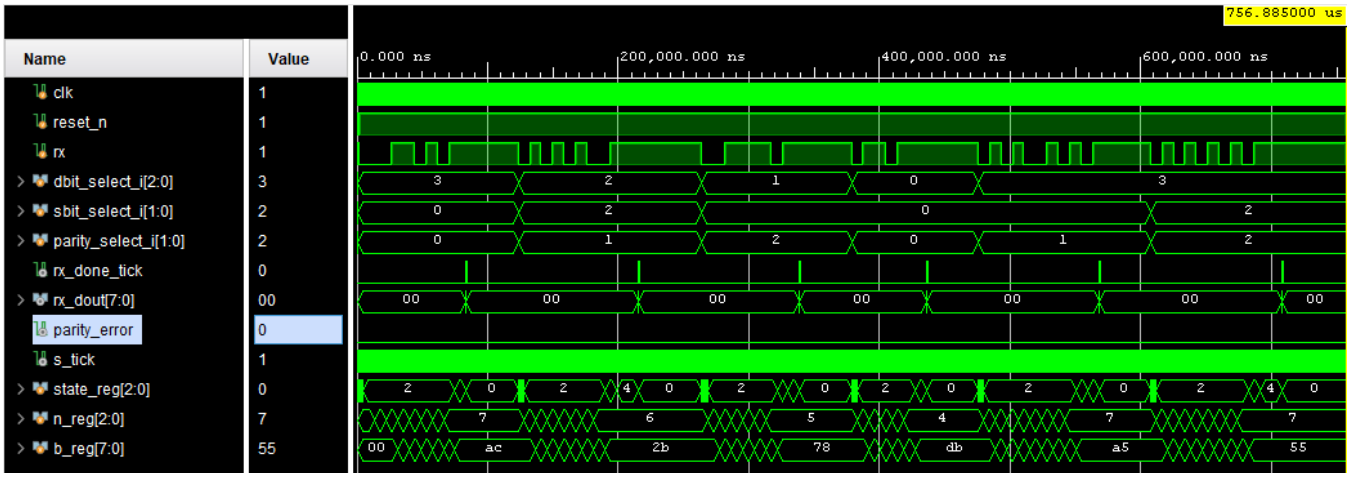

Figure 7. UART Receiver Simulation Waveform

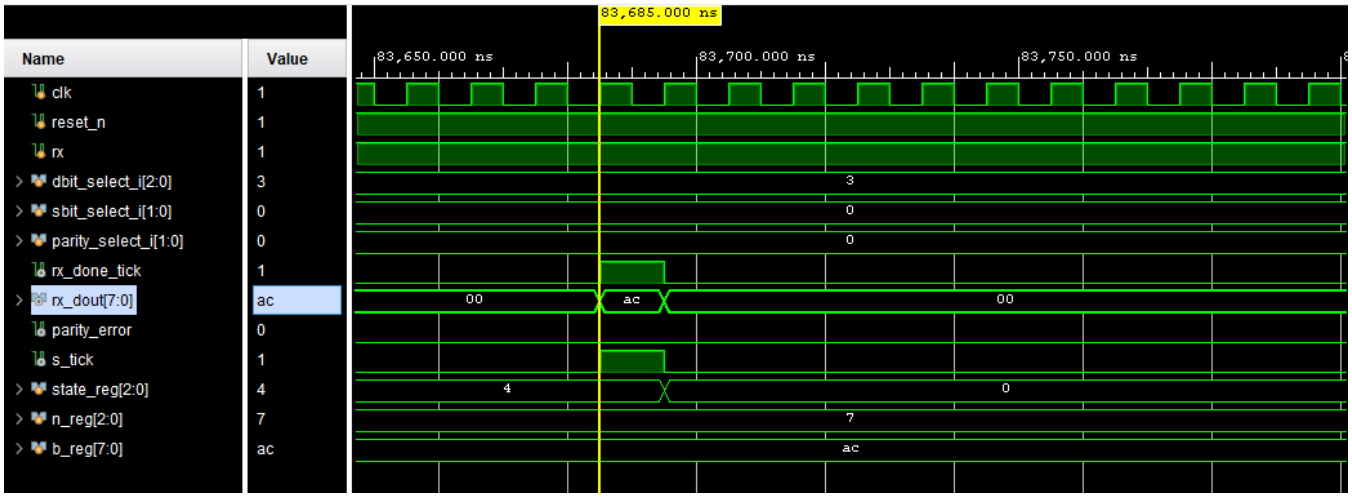

Figure 6. UART Receiver Simulation Waveform (Receiving)

Transmitter

The UART transmitter testbench is developed to validate the functionality and flexibility of the transmitter module across multiple UART frame formats. It ensures that the transmitter correctly serializes parallel input data into a properly framed UART bitstream according to configurable parameters.

Each frame includes:

A start bit (always 0)

A programmable number of data bits (5 to 8)

An optional parity bit (even or odd)

One, 1.5, or two stop bits (always 1s)

The testbench uses a task ***send_byte(data, data_bits, parity_type, stop_bits)*** that generates and transmits the UART frame based on the specified configuration. The transmitter logic appends the appropriate start, parity, and stop bits while maintaining proper bit order and timing.

#	Data	Format	Description
1	0xAC	8N1	8 data bits, no parity, 1 stop bit
2	0x55	7E2	7 data bits, even parity, 2 stop bits
3	0xF0	8O1.5	8 data bits, odd parity, 1.5 stop bits
4	0x15	5N1	5 data bits, no parity, 1 stop bit
5	0x2A	6E1	6 data bits, even parity, 1 stop bit
6	0xAA + 0x55	8N1 (x2)	Back-to-back transmission of 2 frames

Each transmission is observed on the serial output (tx) and compared against the expected waveform to ensure:

- **Correct framing** (start, data, optional parity, stop bits)
- **Accurate parity generation** based on the parity configuration
- **Proper timing** and adherence to UART protocol
- **Reliable continuous transmission**, including back-to-back frames

The simulation results confirm that the UART transmitter module is fully operational and adheres to its design specifications. The verification process successfully demonstrated the module's key features by sending a series of test frames with diverse configurations.

Specifically, the transmitter proved its ability to correctly construct serial frames with variable formats, handling data lengths of 5, 6, 7, and 8 bits, and stop bit configurations of 1, 1.5, and 2 bits. The internal parity generation logic was also validated; the simulation shows the calculated_parity signal generating the correct bit for both Even and Odd parity modes, which was then accurately inserted into the serial data stream on the tx line.

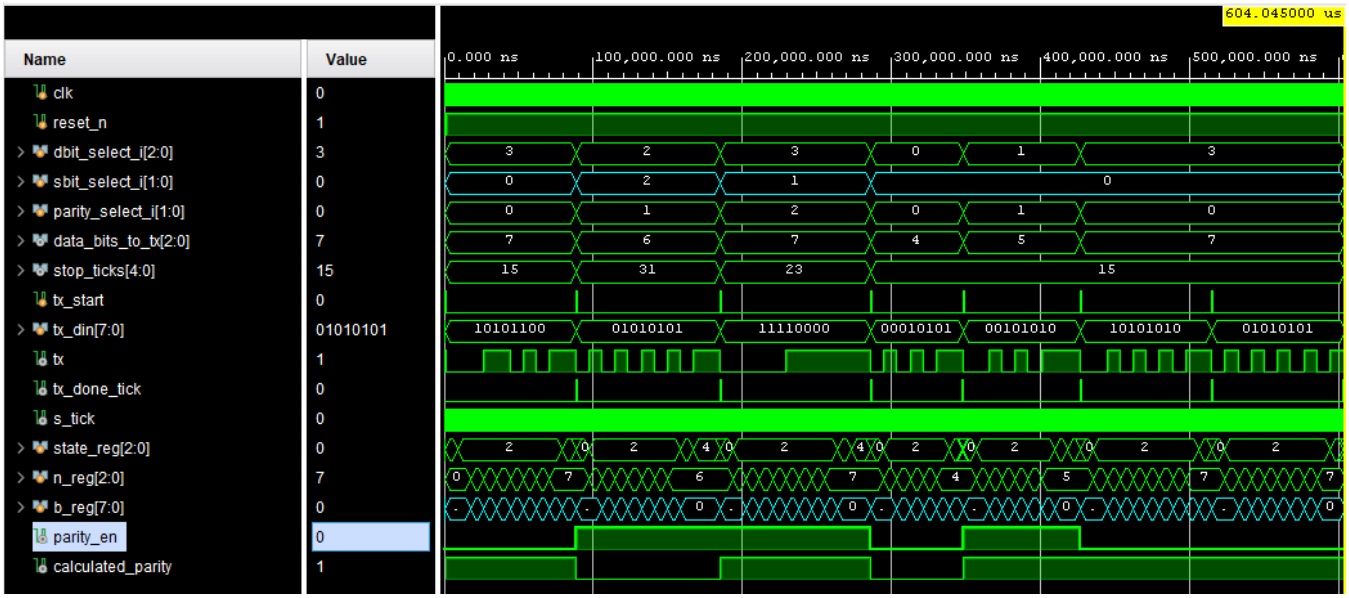

Figure 9. UART Transmitter Simulation Waveform

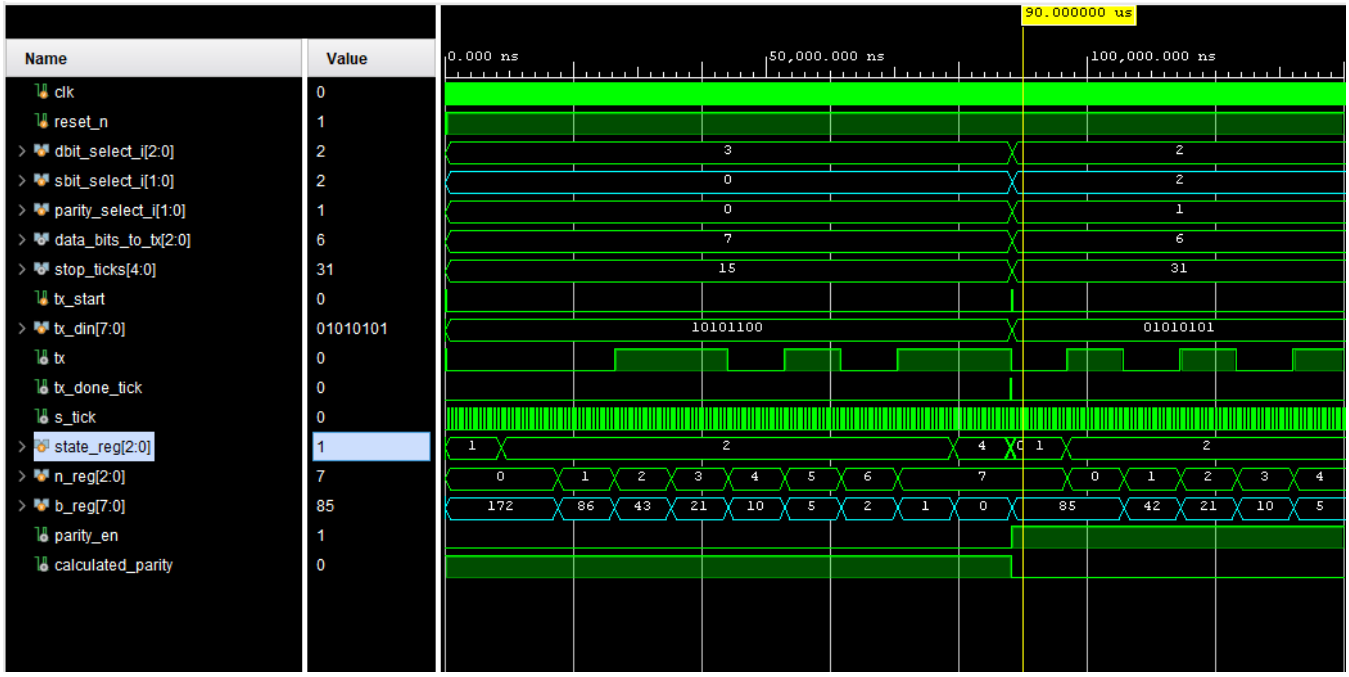

Figure 8. UART Transmitter Simulation Waveform (Single byte)

Full UART

The simulation demonstrates complete UART functionality by connecting a transmitter and receiver through a shared serial line, verifying the full data path from transmission to reception. The testbench sends a sequence of UART frames with varying configurations, validating the system's ability to adapt to different data lengths, parity modes, and stop bit settings in real-time.

Each transmission is initiated by writing data into the transmitter when tx_full_1 is deasserted, indicating the buffer is ready. The transmitter then serializes the data based on current configuration settings and outputs the frame on the tx line. The receiver captures the incoming signal, decodes it according to the same format, and asserts rx_empty_2 low once valid data is available. Data is then read from the receiver buffer using the rd_uart_2 signal.

#	Data	Format	Description
1	0x41	8N1	Transmits 'A' with 8 data bits, no parity, 1 stop bit
2	0x42	8N1	Transmits 'B' under the same configuration
3	0x55	8O1	Changes parity to odd; transmits 'U' with correct parity bit
4	0x66	7N1	Switches to 7 data bits; transmits 'f', verifying truncation
5	0x99	8N1	Returns to no parity mode and sends 0x99

Each case verifies:

- **Correct transmission and reception** of the data byte
- **Accurate response of handshaking signals** (tx_full_1, rx_empty_2)
- **Dynamic configurability** of parity and data length during runtime
- **Proper framing, synchronization, and data integrity** across the serial link

The system consistently demonstrated reliable operation under varying conditions, confirming the robustness of the full UART datapath from TX to RX.

This simulation demonstrates a loopback test of the complete UART system, where the transmitter's serial output (tx_1) is directly connected to the receiver's serial input (rx). The objective is to verify seamless end-to-end communication and the system's ability to adapt to on-the-fly configuration changes. The testbench writes a sequence of data bytes to the transmitter, and after each transmission, it reads the received data from the receiver to confirm data integrity.

Throughout all test cases, the data read from the receiver's output (r_data_2) perfectly matches the data written to the transmitter's input (w_data_1). The absence of any assertion on the parity_error_2 signal further confirms that the transmitter and receiver remained synchronized with matching configurations, ensuring data integrity across all tests. The FIFO status signals (tx_full_1 and rx_empty_2) correctly manage the data flow between the testbench and the UART modules.

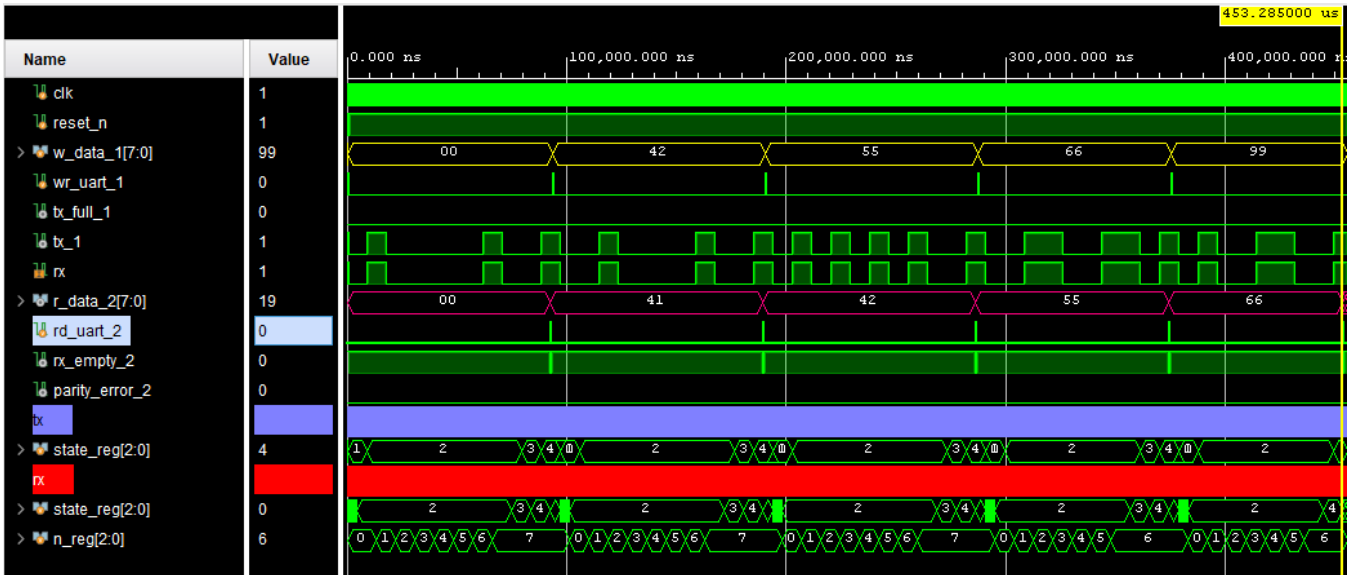

Figure 11. Full UART Transceiver Simulation Waveform

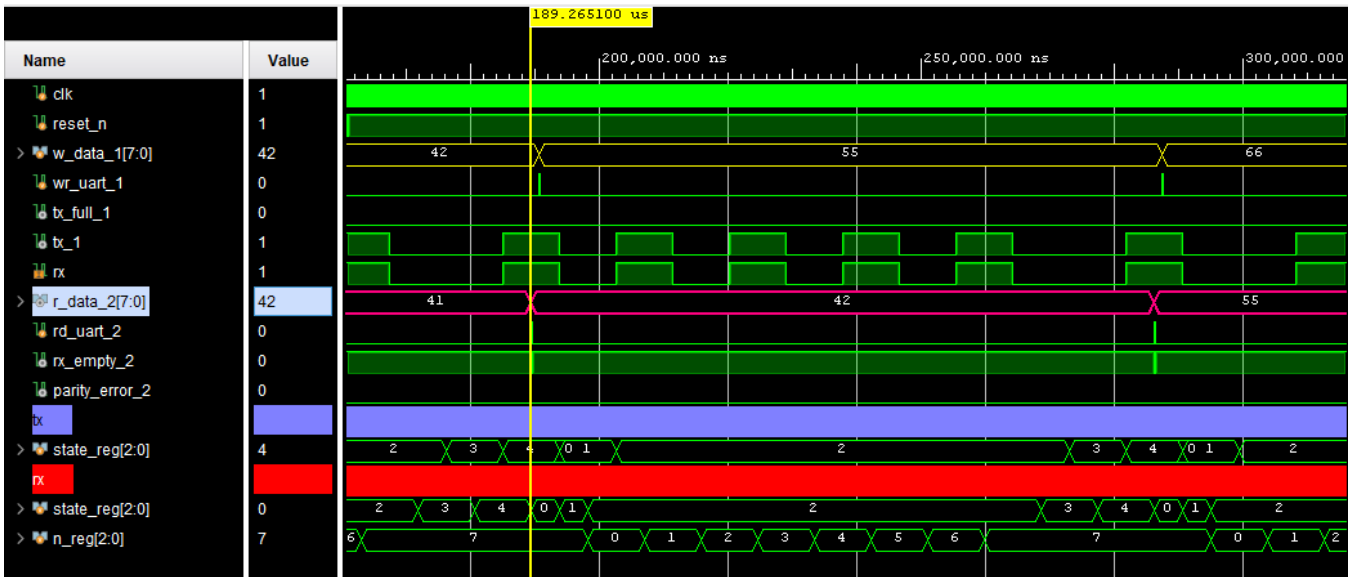

Figure 10. Full UART Transceiver Simulation Waveform (single Byte)

Elaborated Design

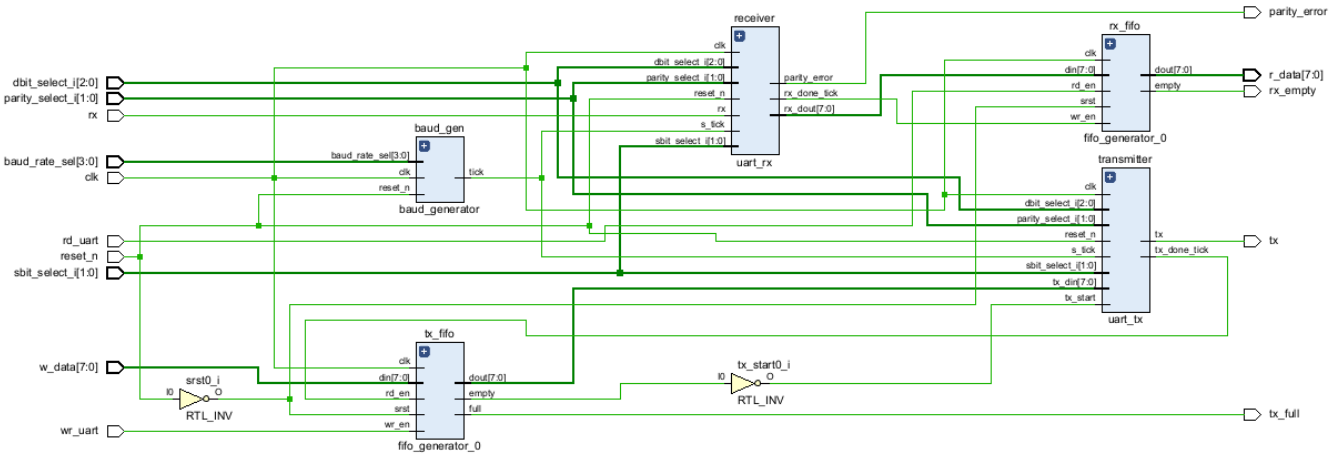

Figure 12. Elaborated Schematic of the UART Transceiver

Synthesized Schematic

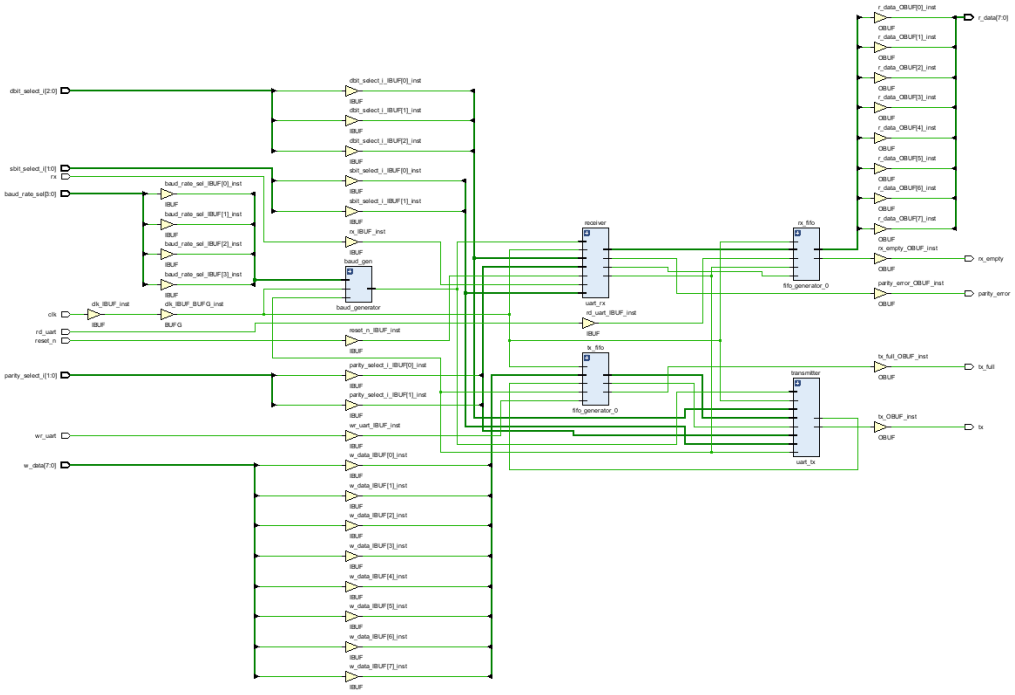

Figure 13. Synthesized Schematic of the UART Transceiver

Reports

Power

```
## Clock signal
#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];
```

Figure 14. Clock constraints

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.097 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (12.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

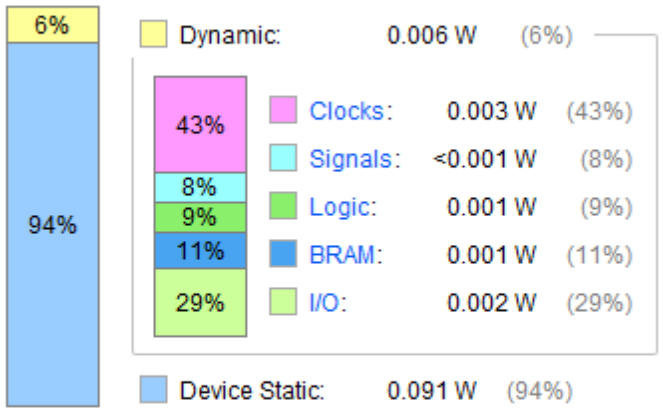

Figure 15. Power Report for UART Transceiver

Timing

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.782 ns	Worst Hold Slack (WHS):	0.132 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:	355	Total Number of Endpoints:	355	Total Number of Endpoints:	181

All user specified timing constraints are met.

Figure 16. Timing Report for UART Transceiver

Utilization

Name	Slice LUTs (63400)	Slice Registers (126800)	Block RAM Tile (135)	Bonded IOB (210)	BUFGCTRL (32)
uart_transceiver	209	176	1	36	1
baud_gen (baud_generator)	29	14	0	0	0
receiver (uart_rx)	49	21	0	0	0
rx_fifo (fifo_generator_0)	48	60	0.5	0	0
transmitter (uart_tx)	35	21	0	0	0
tx_fifo (fifo_generator_0)	48	60	0.5	0	0

Figure 17. Utilization Report

References

-Chu, P. P. (2008). FPGA prototyping by Verilog examples: Xilinx Spartan-3 version. Wiley-Interscience.

-Anas Salah Eddin. (n.d.). ECE 3300 - Digital Circuit Design Using Verilog. YouTube. Retrieved August 2, 2025, from https://www.youtube.com/playlist?list=PL-iIOnHwN7NXw01eBDR7wI8KzGK4mu8Sr

-Udemy. (n.d.). Verilog for an FPGA Engineer with Xilinx Vivado Design Suite. Retrieved August 2, 2025, from https://www.udemy.com/course/verilog-for-an-engineer-with-xilinx-vivado-design-suite/