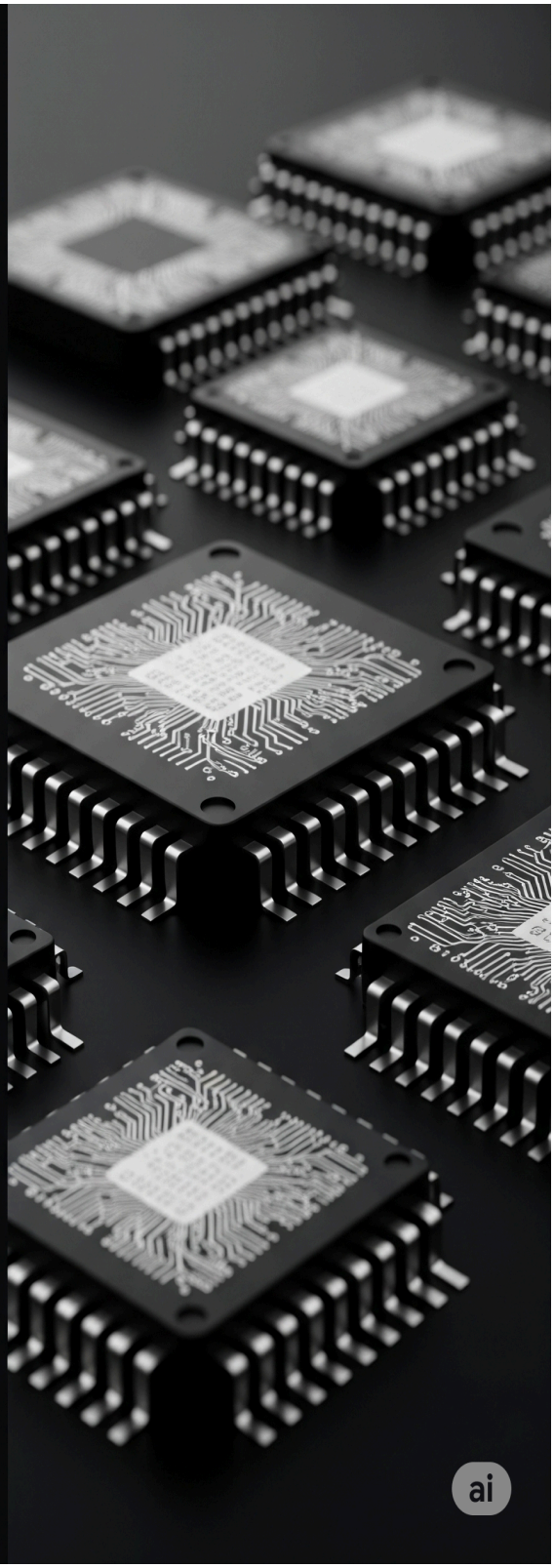
# Dual-FPGA UART System:

## A Full Duplex Hardware Communication Design

1. Hemanth S.
1. Naveen Kumar B
2. Sabarish Mohan JS.



# Dual-FPGA UART System: A Full Duplex Hardware Communication Design

**Contributors:**
Hemanth S
Naveen Kumar B
Sabarish Mohan JS

## 📘 Introduction:

This project demonstrates a bidirectional UART (Universal Asynchronous Receiver-Transmitter) communication system implemented on two Nexys A7 FPGA development boards. The system is designed with full duplex capability, allowing each board to function dynamically as both transmitter and receiver.

By leveraging the programmable logic of the Artix-7 FPGA on the Nexys A7 platform, the implementation showcases fundamental digital communication principles including asynchronous serial data transmission, baud rate generation, and frame formatting. To simulate the code and debugging, we used the Xilinx Vivado 2022 version.

---

## 🛠️ Key Steps:

- Gained deep understanding of UART protocol specifics.
- Implemented robust Verilog RTL for both transmit and receive paths.
- Successfully interfaced two Nexys A7-100T boards.
- Conducted rigorous simulation and hardware verification.
- Leveraged Xilinx Vivado tools.
- Maintained detailed project logs and source control on GitHub

---

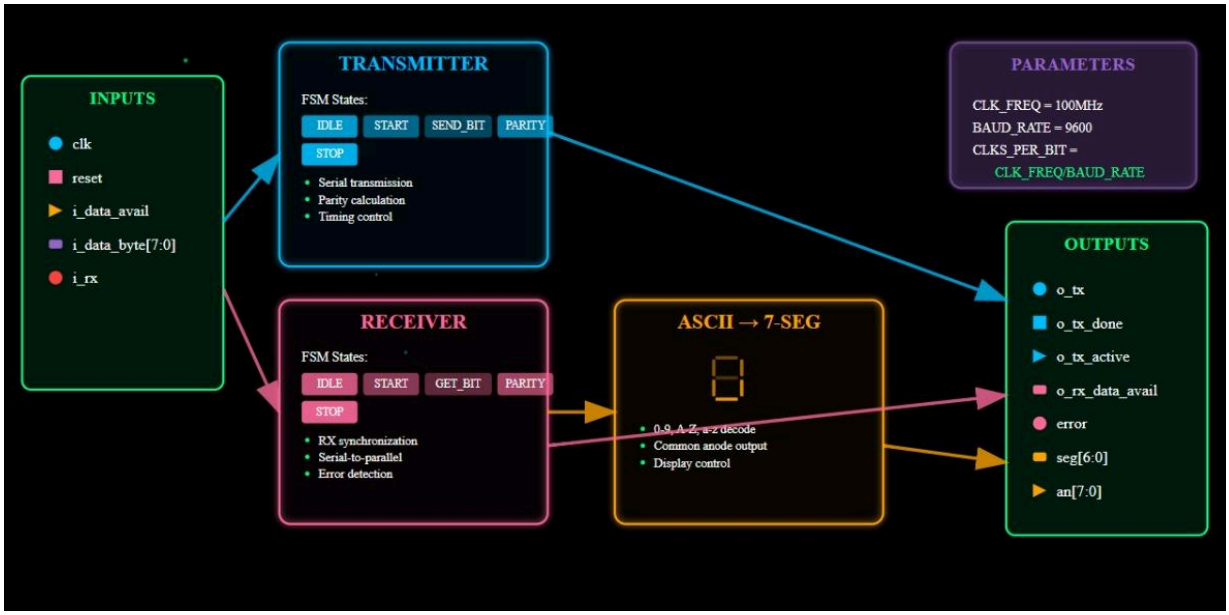
## 🔁 Workflow:

- The transmitter remains in idle until data is available.
- Once data is ready, it loads the 8-bit data and computes the parity bit.
- It begins the transmission by sending a start bit (logic 0).
- The 8 data bits are transmitted one by one, starting from the least significant bit.
- After the data bits, the parity bit is sent to assist in error detection.
- A stop bit (logic 1) is transmitted to indicate the end of the frame.
- The transmitter resets its status and signals the completion of transmission, returning to idle.
- Meanwhile, the receiver constantly monitors the line for a falling edge indicating a start bit.
- On detecting a potential start bit, it samples mid-bit to confirm validity.
- It then receives the 8 data bits in sequence with proper timing.

- The parity bit is received next for validation.
- The stop bit is expected and confirms the end of the frame.

---

## 📐 UART Architecture:




---

## 📡 UART Transmitter Module: Core Functionality

This transmitter module efficiently converts parallel 8-bit data into a serial bitstream, adding necessary framing bits for robust UART communication. It operates synchronously with the system clock, using the **CLKS_PER_BIT** parameter to precisely time each serial bit.

**Finite State Machine (FSM) States:**

The module's 5-state FSM orchestrates the precise UART transmission sequence:

1. **IDLE:**
    - **Purpose:** Waits for data.
    - **Cruciality:** Ensures the transmitter is ready and prevents accidental transmissions.
    - **Actions:** o_tx is high. Upon i_data_avail, loads i_data_byte, calculates odd t_parity, asserts o_active, and transitions to START.

```
IDLE: begin
                    counter <= 0; // Reset counter
                    index <= 0;   // Reset bit index
                    o_tx <= 1'b1; // Ensure TX is idle high
                    if (i_data_avail == 1'b1) begin
                        o_active <= 1'b1;     // Indicate transmission is active
                        data_byte <= i_data_byte; // Load data byte
                        t_parity = ^i_data_byte;  // Calculate parity for the loaded byte
                        state <= START;       // Move to START state
                    end else begin
                        state <= IDLE;        // Stay in IDLE
                    end
                end
```

2. **START:**
    - **Purpose:** Send's the **Start Bit** (logic '0').
    - **Cruciality:** This is the vital handshake; it signals the receiver to begin sampling, preventing data misalignment.
    - **Actions:** o_tx goes low for CLKS_PER_BIT cycles, then transitions to SEND_BIT.

```
START: begin
                    o_tx <= 0; // Send start bit (low)
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        state <= SEND_BIT;      // Move to SEND_BIT state
                    end
                end
```

3. **SEND_BIT:**
    - **Purpose:** Transmits the 8 data bits (LSB first).
    - **Cruciality:** Delivers the actual payload. Accurate timing (CLKS_PER_BIT per bit) is paramount for correct reception.

- **Actions:** o_tx outputs data_byte[index]. After CLKS_PER_BIT cycles, increments index. Continues until all 8 bits are sent, then moves to PARITY.

```
SEND_BIT: begin
                    o_tx <= data_byte[index]; // Send current data bit (LSB first)
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        if (index < 7) begin
                            index <= index + 1; // Move to next data bit
                            state <= SEND_BIT;  // Stay in SEND_BIT state
                        end else begin
                            state <= PARITY;    // All data bits sent, move to PARITY state
                        end
                    end
                end
```

4. **PARITY:**
   - **Purpose:** Sends the calculated parity bit.
   - **Cruciality:** Enables basic error detection at the receiver, enhancing data integrity against single-bit errors.
   - **Actions:** o_tx outputs t_parity for CLKS_PER_BIT cycles, then transitions to STOP.

```
PARITY: begin
                    o_tx <= t_parity; // Send parity bit
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        state <= STOP;          // Move to STOP state
                    end
                end
```

5. **STOP :**
   - **Purpose:** Sends the **Stop Bit(s)** (logic '1') and concludes transmission.
   - **Cruciality:** Signals the end of the frame, allowing the receiver to reset and ensuring a necessary idle period for robust asynchronous communication.

- **Actions:** o_tx goes high for CLKS_PER_BIT cycles. Resets o_active, pulses o_done, and returns to IDLE.

```
STOP: begin
                    o_tx <= 1'b1; // Send stop bit (high)
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        o_active <= 0;          // Deactivate transmission
                        o_done <= 1;            // Indicate transmission is done
                        state <= IDLE;          // Return to IDLE state
                    end
                end
```

This FSM guarantees a precise and reliable UART transmission sequence, from initiation with a start bit to conclusion with a stop bit

---

## 📥UART Receiver Module: Core Functionality

This receiver module is meticulously designed to receive serial data from a UART line, reconstruct the parallel 8-bit data, and perform parity checking. It operates synchronously with the system clock, using the CLKS_PER_BIT parameter to accurately sample incoming bits. Crucially, it incorporates a **two-flop synchronizer to safely handle the asynchronous input and mitigate metastability**, a critical aspect of robust digital design.

**Finite State Machine (FSM) States:**

The module's 5-state FSM orchestrates the precise UART reception sequence, operating on the *synchronized* rx signal:

- **IDLE :**
  - **Purpose:** Waits for the start of a new transmission.
  - **Cruciality:** Ensures the receiver is ready and actively monitors the line for a valid start bit, preventing misinterpretation of noise as data.

- **Actions:** counter and index are reset. Continuously monitors the *synchronized* rx line. Upon detecting a high-to-low transition (rx == 0), it transitions to the START state.

- **START :**

```
IDLE: begin
                    counter <= 0; // Reset counter
                    index <= 0;   // Reset bit index
                    if (rx == 0) begin // Detect start bit (low)
                        state <= START; // Move to START state
                    end else begin
                        state <= IDLE;  // Stay in IDLE
                    end
                end
```

  - **Purpose:** Validates the detected Start Bit (logic '0').
  - **Cruciality:** Prevents false starts due to noise or glitches. By performing **mid-bit sampling** – specifically, sampling the *synchronized* rx line in the middle of the expected start bit period (counter == (CLKS_PER_BIT - 1) / 2) – it confirms that the low signal is stable and represents a true start bit. This strategic sampling point helps in tolerating minor clock skew between the transmitter and receiver.
  - **Actions:** Increments counter. At (CLKS_PER_BIT - 1) / 2 (the nominal middle of the bit period), it checks rx. If rx is still low, it confirms the start bit, resets counter, and transitions to GET_BIT. Otherwise, it's a spurious start, and it returns to IDLE.

```
START: begin
                    // Sample in the middle of the start bit period to confirm it's still low
                    if (counter == (CLKS_PER_BIT - 1) / 2) begin
                        if (rx == 0) begin // Confirm start bit is still low
                            counter <= 0;   // Reset counter
                            state <= GET_BIT; // Move to GET_BIT state
                        end else begin
                            state <= IDLE;  // Spurious start bit, return to IDLE
                        end
                    end else begin
                        counter <= counter + 1; // Increment counter
                    end
                end
```

- **GET_BIT :**
    - **Purpose:** Receives the 8 data bits (LSB first).
    - **Cruciality:** This is where the actual data payload is captured. Accurate **mid-bit sampling** (implicitly at CLKS_PER_BIT - 1 when the counter rolls over, effectively hitting the middle of the *next* bit period after the start bit's mid-point) at the correct time within each bit period is paramount for reconstructing the original data without errors.
    - **Actions:** Increments counter. After CLKS_PER_BIT - 1 cycles (end of bit period for timing, which corresponds to the start of the next bit, and the sampling happens at this transition point in this FSM logic), it samples rx and stores it into data_byte[index]. Resets counter. If index < 7, increments index and stays in GET_BIT. Once all 8 bits are received (index == 7), it transitions to PARITY.

```
GET_BIT: begin
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        data_byte[index] <= rx; // Sample and store data bit (LSB first)
                        if (index < 7) begin
                            index <= index + 1; // Move to next data bit
                            state <= GET_BIT;   // Stay in GET_BIT state
                        end else begin
                            state <= PARITY;    // All data bits received, move to PARITY state
                        end
                    end
                end
```

- **PARITY :**
  - **Purpose:** Receives the parity bit.
  - **Cruciality:** Captures the bit needed for error detection. The value of this bit, combined with the received data, determines if a transmission error occurred.
  - **Actions:** Increments counter. After CLKS_PER_BIT - 1 cycles, it samples rx and stores it into r_parity. Resets counter, then transitions to STOP.

```
PARITY: begin
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        r_parity <= rx;         // Sample and store parity bit
                        state <= STOP;          // Move to STOP state
                    end
                end
```

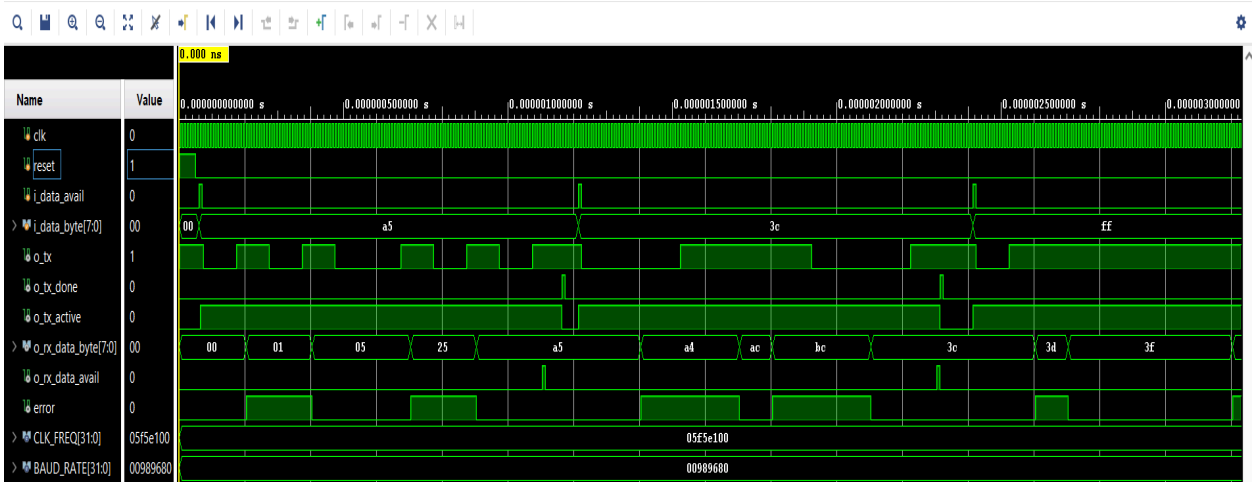
- **STOP :**
  - **Purpose:** Receives the Stop Bit(s) (logic '1') and concludes reception.
  - **Cruciality:** Confirms the end of the frame, allowing the receiver to finalize the data and reset its state. It also ensures the line returns to an idle high state, providing a crucial buffer before the next potential start bit.

- **Actions:** Increments counter. After CLKS_PER_BIT - 1 cycles, it checks the stop bit (implicitly expecting rx to be high). Resets counter, asserts o_data_avail for one clock cycle, and returns to IDLE.

```
STOP: begin
                    if (counter < CLKS_PER_BIT - 1) begin
                        counter <= counter + 1; // Increment counter
                    end else begin
                        counter <= 0;           // Reset counter
                        data_avail <= 1'b1;     // Indicate data is available
                        state <= IDLE;          // Return to IDLE state
                    end
                end
```

---

## ▶️ SIMULATION SNIPPETS :




---

## 💡 REFLECTIONS AND DEBUGGING:

- **Avoiding Metastability with RX Buffer and Double Flip-Flop**
  The RX signal is passed through a buffer and then two flip-flops to synchronize asynchronous input to the FPGA clock domain. This reduces metastability risks and ensures reliable data capture.
- **Improved Accuracy by Sampling at Mid-Baud Period**
  Data bits are sampled at the center of the baud interval, minimizing timing errors caused by clock mismatches or jitter. This improves the accuracy of received data.
- **Separate State for Parity Generation and Checking**
  Parity calculation and verification are handled in a dedicated state to avoid race conditions. This separation ensures correct parity handling after all data bits are received.
- **Debugging Using PuTTY SSH Terminal**
  PuTTY was used initially to test the UART communication via SSH, allowing quick verification and debugging of data transmission in a controlled environment.

---