[Custom APB UART IP]

[Project Report]

Prepared by: Manar Saber Abdelrahim
Instructor: Eng. Mohamed Tareq National Telecommunication Institute (NTI)

[September 09, 2025]

1.0 INTRODUCTION

The Universal Asynchronous Receiver/Transmitter (UART) is one of the most widely used serial communication protocols in embedded systems, enabling reliable point-to-point data transfer between devices. It provides a simple, low-cost, and efficient mechanism for transmitting and receiving data one bit at a time without requiring a shared clock line. Typical applications of UART include microcontroller-to-PC communication, sensor interfacing, and SoC peripheral integration.

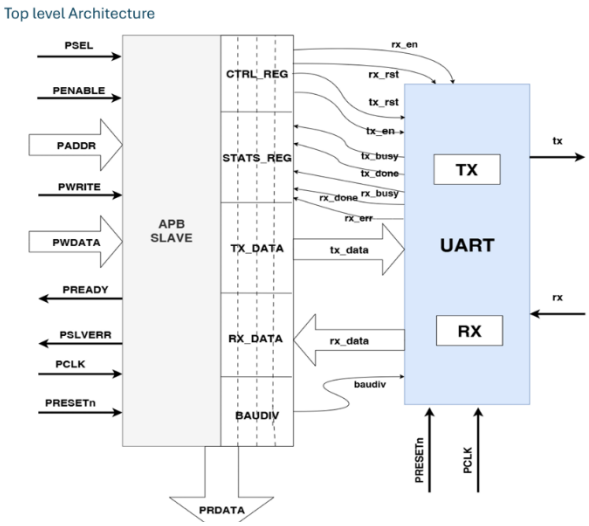


Figure 1: Custom APB UART IP

In modern digital systems, peripherals such as UARTs are commonly accessed through standardized bus protocols. The Advanced Microcontroller Bus Architecture (AMBA) introduced by ARM provides a family of buses for SoC integration. Among them, the Advanced Peripheral Bus (APB) is optimized for low-power, low-complexity peripheral connections. APB provides memory-mapped access to registers, simplifying the control of UART modules in system-on-chip designs.

This project focuses on designing and implementing a **custom UART peripheral wrapped with an APB slave interface**. The design includes both the UART Transmitter (TX) and Receiver (RX) modules, along with an APB wrapper that exposes memory-mapped registers for configuration, control, and data transfer. The integration of UART with APB allows seamless communication with processors and other master devices in SoC environments.

The primary objectives of this project are:

1. To design a synthesizable UART TX and RX core with fixed baud rate operation (9600 baud with 100 MHz system clock).
2. To implement an APB-compliant wrapper providing register-level access to UART functionalities.
3. To verify the correctness of the design using simulation and FPGA prototyping on the Basys3 board.

This work was carried out as part of the final training project at the National Telecommunication Institute **(NTI)**, under the supervision of **Eng. Mohamed Tareq**, aiming to strengthen practical skills in RTL design, SoC integration, and FPGA implementation.

2.0 Design Analysis

2.1 APB UART wrapper

The APB UART wrapper is responsible for connecting the UART transmitter (TX) and receiver (RX) modules to the AMBA APB bus. It acts as the bridge between software and hardware,

exposing UART functionality as memory-mapped registers.

Key Functions

1. APB Protocol Handling
    - Implements the three-state FSM (IDLE, SETUP, ACCESS) that ensures all APB transfers follow the required two-phase handshake.
    - IDLE waits for PSEL=1 and PENABLE=0.
    - SETUP samples the address and control signals.
    - ACCESS performs the read/write, asserts PREADY=1, then returns to IDLE.
2. Register Interface
    - CTRL_REG (0x00):
        - [0] tx_en → Enables transmitter.
        - [1] rx_en → Enables receiver.
        - [2] tx_rst → Soft reset for TX.
        - [3] rx_rst → Soft reset for RX.
    - STATS_REG (0x04):
        - [0] rx_busy
        - [1] tx_busy
        - [2] rx_done
        - [3] tx_done
    - TX_DATA (0x08): Holds data to transmit (8 bits).
    - RX_DATA (0x0C): Holds last received byte.
    - BAUDIV (0x10): Baud rate divider (read/write but unused in this fixed-baud design).

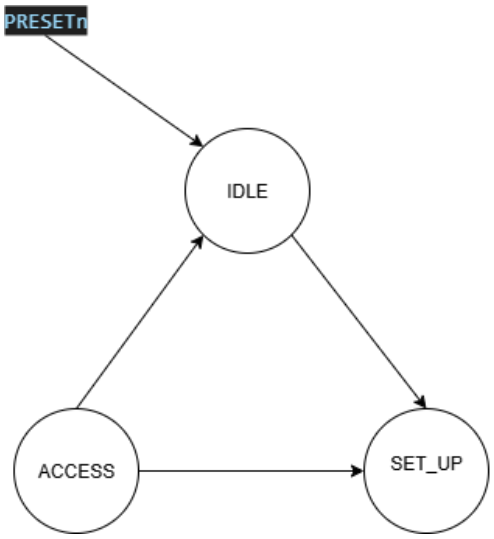


Figure 2: APB_FSM

3. Reset Logic
    - The APB reset PRESETn is active-low.
    - Internally inverted to form sys_rst (active-high).
    - Combines with soft reset bits (tx_rst_soft, rx_rst_soft) for flexible control.
4. Integration with TX and RX
    - On write to TX_DATA, data is latched and transmitted by the TX FSM.
    - On RX completion, received data is latched into RX_DATA.
    - STATS_REG continuously reflects real-time status signals (busy, done).

Operation Flow

- Transmit Path:
    1. CPU writes control word into CTRL_REG with tx_en=1.
    2. CPU writes a byte into TX_DATA.
    3. TX FSM serializes the byte on the tx line.
    4. Completion is signaled via tx_done.
- Receive Path:
    1. CPU enables receiver by setting rx_en=1 in CTRL_REG.
    2. Incoming serial data is captured by RX FSM.
    3. Once a full byte is received, it is stored in RX_DATA and rx_done=1.
    4. CPU reads RX_DATA over APB.

Design Highlights

- Simplicity: FSM keeps APB operations lightweight.
- Modularity: Clear separation between bus logic and UART datapath.
- Scalability: BAUDIV register included for future variable baud support.
- Reliability: Control and status registers ensure software can safely monitor and drive UART without race conditions.

2.2 UART Transmitter (TX)

The UART Transmitter (TX) module is responsible for serializing parallel data and transmitting it over the tx line using the UART protocol (8 data bits, no parity, 1 stop bit, 9600 baud at 100 MHz).

Key Functions

1. Data Framing
    - Automatically generates the start bit (logic 0), followed by 8 data bits (LSB first), and a stop bit (logic 1).
    - Ensures correct synchronization with the receiving UART.
2. Baud Rate Timing
    - A counter divides the system clock (100 MHz) down to the UART baud rate.
    - For 9600 baud, each bit occupies ~10417 cycles.
    - This guarantees precise timing for every transmitted bit.
3. Control Signals
    - Inputs:
        - clk → System clock.
        - rst → Synchronous reset (active-high).
        - data_in → 8-bit data to be transmitted.
        - tx_en → Transmission enable signal.
    - Outputs:

- tx → Serial output line.
- busy → Indicates transmission in progress.
- done → Indicates transmission completion.

FSM Operation

The TX module uses a five-state FSM:

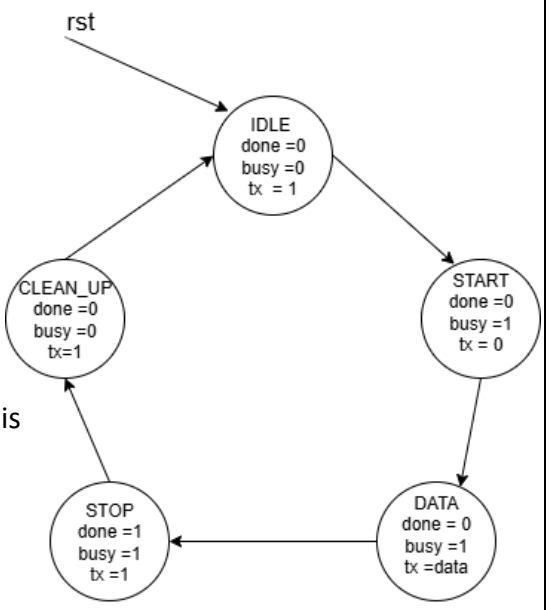


Figure 3: TX_FSM

- IDLE
  - TX line is held high.
  - Waits for tx_en=1 to start transmission.
- START
  - Sends a start bit (tx=0) for one bit time.
- DATA
  - Sequentially transmits 8 bits from data_in, LSB first.
  - Uses r_Bit_Index counter to track which bit is being transmitted.
- STOP
  - Sends a stop bit (tx=1) for one bit time.
  - Sets done=1 and clears busy.
- CLEAN_UP
  - Holds for one cycle to reset flags.
  - Returns to IDLE.

Operation Flow

1. The CPU writes an 8-bit value into the TX_DATA register through the APB wrapper.
2. The wrapper asserts tx_en=1.
3. The FSM transitions from IDLE → START → DATA → STOP → CLEAN_UP.
4. The serialized bits are shifted out through the tx pin.
5. When transmission ends, done=1 is asserted so software can detect completion.

Design Highlights

- Lightweight: Uses a small FSM, an 8-bit register, and counters.
- Synchronous Reset: Guarantees proper restart on reset.
- Handshaking: busy prevents overlapping transmissions, done signals completion.
- Protocol-Compliant: Follows 8-N-1 UART framing with correct baud timing.

2.3 UART Receiver (RX)

The UART Receiver (RX) module performs the reverse function of the transmitter: it deserializes incoming serial data (rx) into an 8-bit parallel word, following the standard UART 8-N-1 protocol (8 data bits, no parity, 1 stop bit, 9600 baud @ 100 MHz).

Key Functions

1. Bit Sampling and Timing
    - Uses a clock counter to align sampling points with the baud rate.
    - Each bit is expected to last ~10417 system cycles (100 MHz ÷ 9600).
    - Samples data at the middle of each bit (HALF_CYCLES) to avoid jitter and ensure stability.
2. Metastability Protection
    - Implements double sampling (r_Rx_Data_R, r_Rx_Data) on the incoming rx signal.
    - This prevents metastability issues from asynchronous serial inputs.
3. Control Signals
    - Inputs:
        - clk → System clock.
        - rst → Synchronous reset.
        - rx_en → Enables the receiver.
        - rx → Serial input data stream.
    - Outputs:
        - rx_data → 8-bit received data word.
        - rx_done → Indicates data reception completed.
        - rx_busy → Indicates ongoing reception.

FSM Operation

The RX FSM consists of five states:

- IDLE
    - Default state, line idle high.
    - Waits for start bit (rx=0) when rx_en=1.
- START

Waits half a bit period to confirm a valid start bit.

    - Ensures that noise or glitches do not trigger reception.
- DATA
    - Samples 8 bits in the middle of each bit period.
    - Stores bits sequentially into r_Rx_Data_Byte.

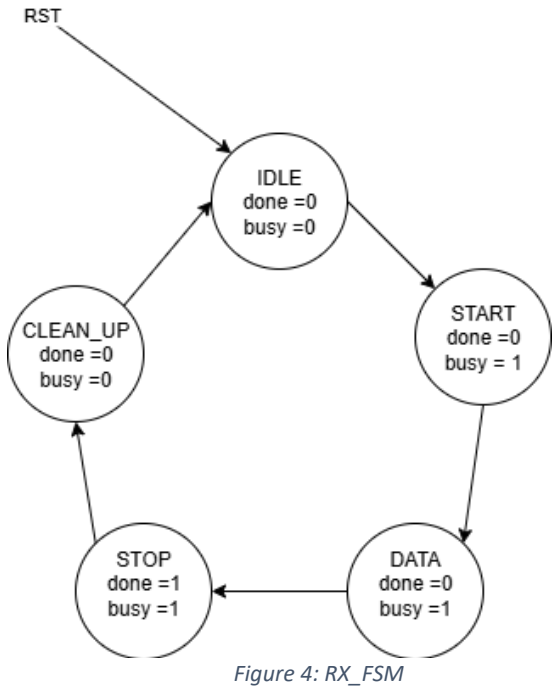


Figure 4: RX_FSM

- STOP
    - Expects a logic high stop bit.
    - Sets rx_done=1 to signal data is ready.
- CLEAN_UP
    - Holds rx_done=1 for one cycle.
    - Clears busy and done signals before returning to IDLE.

Operation Flow

1. Idle line remains high (rx=1).
2. When start bit (rx=0) is detected, FSM moves from IDLE → START.
3. After half a bit time, confirms start bit and transitions to DATA.
4. Samples 8 data bits at midpoints and stores them into the buffer.
5. Moves to STOP and checks stop bit validity.
6. Sets rx_done=1, making the received byte available in rx_data.
7. Clears signals in CLEAN_UP, then returns to IDLE.

Design Highlights

- Reliable Start Detection: Uses half-bit delay to confirm valid start bit.
- Noise Tolerance: Double-registering input helps avoid metastability.
- Synchronization: Samples at bit midpoint to ensure robustness.
- Handshaking: rx_busy shows active reception; rx_done signals completion.
- Parallel Output: Final 8-bit word stored in rx_data register.

3.0 Verification Strategy

The verification methodology was designed to validate functionality at three hierarchical levels:

1. UART Transmitter (TX)
    - Objective: Ensure that the TX FSM correctly generates UART frames (start bit, 8 data bits LSB-first, stop bit).
    - Method:
        - A self-checking testbench (tx_tb.sv) was developed.
        - The test sends a known byte (0xA5) and verifies:
            - Start bit = 0
            - Data bits match input (LSB-first)
            - Stop bit = 1
            - Correct assertion of busy and done signals
        - The testbench includes:
            - Tasks for reset, checking results, waiting baud cycles
            - A scoreboard using error_counter and correct_counter
        - Continuous $monitor displays real-time signal activity.

2. UART Receiver (RX)
    - Objective: Ensure the RX FSM correctly detects start bit, samples incoming serial data, and assembles an 8-bit parallel word.
    - Method:
        - A self-checking testbench (rx_tb.sv) drives a complete UART frame into rx.
        - Frame format: Start bit (0) + 0xA5 (LSB-first) + Stop bit (1).
        - Verification includes:
            - Correct rx_busy assertion during reception
            - Proper rx_done assertion at end of frame
            - Received data matches transmitted word
            - Return to idle state after stop bit
        - Built-in checks confirm reset functionality and metastability protection.

3. APB UART Wrapper
    - Objective: Validate the complete system, including the APB register interface and internal TX/RX operation.
    - Method:
        - A system-level testbench (apb_uart_tb.sv) models CPU-driven APB read/write transactions.
        - Verification flow:
            1. Perform soft reset of TX and RX.
            2. Write TX data (0xA5) and set tx_en.
            3. Observe correct TX waveform (start → data → stop).
            4. Check status register flags (tx_busy, tx_done).
            5. Enable RX and inject a matching UART frame.
            6. Verify RX output (rx_data = 0xA5) and rx_done assertion.
            7. Confirm status flags clear after soft reset.
        - Tasks (apb_write, apb_read, send_data_bits) abstract low-level signal handling, improving readability.
        - $monitor is used for debugging bus activity.

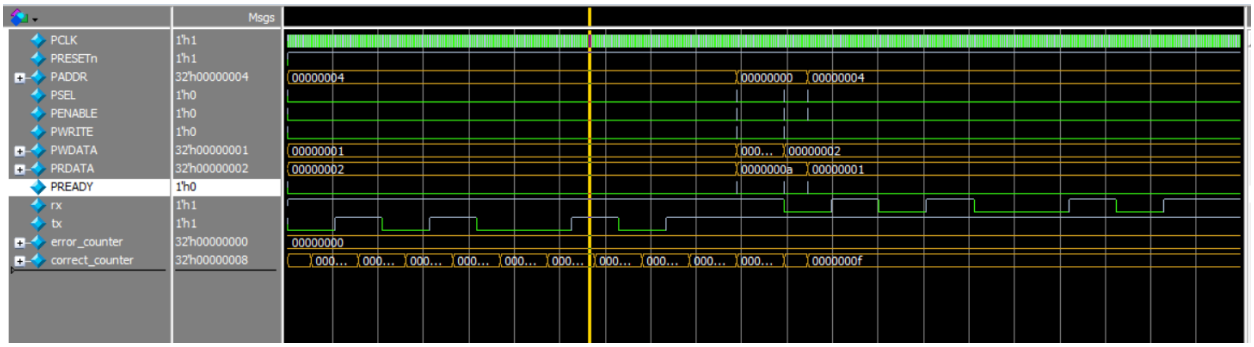
Verification Features

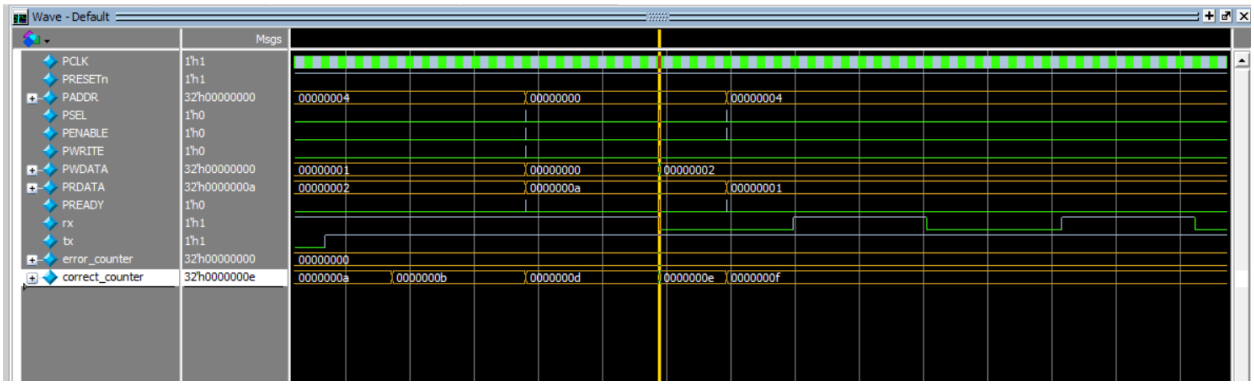
- Self-checking: Testbenches automatically compare DUT outputs with expected results.
- Scoreboard: Pass/fail counters (error_counter, correct_counter) summarize results.
- Reusable tasks: Encapsulated functions for reset, frame sending, and APB transactions.
- Waveform monitoring: $monitor continuously logs DUT signals for debugging.

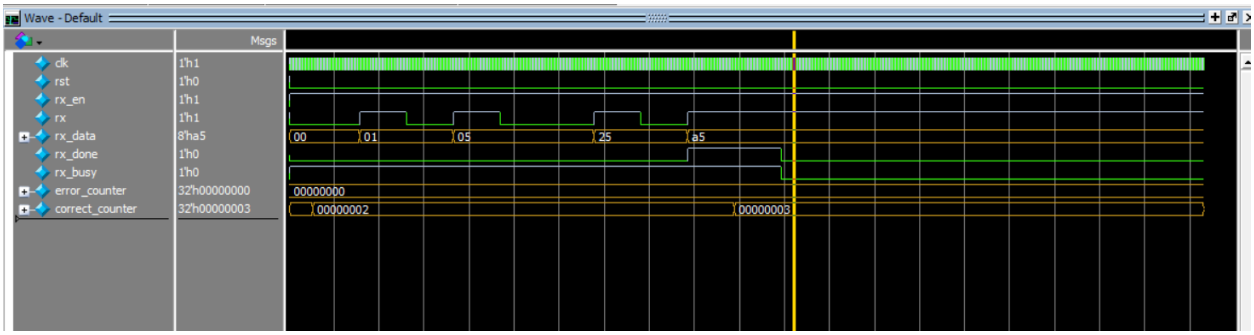
4.0 Simulation Results

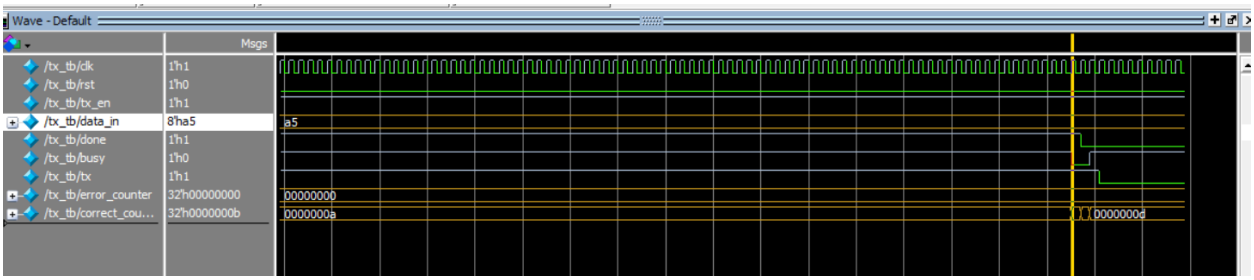
```
# t=0 | PRESETn=0 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=0 tx=x rx=1
# t=5 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=25 | PRESETn=1 PSEL=1 PENABLE=0 PWRITE=1 PADDR=00000000 PWDATA=00000004 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=35 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000004 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=55 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000004 | PRDATA=xxxxxxxx PREADY=1 tx=1 rx=1
# t=65 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000000 PWDATA=00000004 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=75 | PRESETn=1 PSEL=1 PENABLE=0 PWRITE=1 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=85 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=105 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=1 tx=1 rx=1
# t=115 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=125 | PRESETn=1 PSEL=1 PENABLE=0 PWRITE=1 PADDR=00000008 PWDATA=000000a5 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=135 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000008 PWDATA=000000a5 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=155 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000008 PWDATA=000000a5 | PRDATA=xxxxxxxx PREADY=1 tx=1 rx=1
# t=165 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000008 PWDATA=000000a5 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=175 | PRESETn=1 PSEL=1 PENABLE=0 PWRITE=1 PADDR=00000000 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=185 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=205 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=1 PADDR=00000000 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=1 tx=1 rx=1
# t=215 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000000 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=0 tx=1 rx=1
# t=225 | PRESETn=1 PSEL=1 PENABLE=0 PWRITE=0 PADDR=00000004 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=0 tx=0 rx=1
# t=235 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=0 PADDR=00000004 PWDATA=00000001 | PRDATA=xxxxxxxx PREADY=0 tx=0 rx=1
# t=255 | PRESETn=1 PSEL=1 PENABLE=1 PWRITE=0 PADDR=00000004 PWDATA=00000001 | PRDATA=00000002 PREADY=1 tx=0 rx=1
# t=265 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000004 PWDATA=00000001 | PRDATA=00000002 PREADY=0 tx=0 rx=1
```

```
# error_counter = 0, correct_counter = 18
# t=2136045 | PRESETn=1 PSEL=0 PENABLE=0 PWRITE=0 PADDR=00000004 PWDATA=00000002 | PRDATA=00000000 PREADY=0 tx=1 rx=1
# ** Note: $stop    : apb_uart_tb.sv(169)
#    Time: 2136145 ns  Iteration: 0  Instance: /apb_uart_tb
# Break in Module apb_uart_tb at apb_uart_tb.sv line 169
```









5.0 conclusion

In this project, a complete UART system with APB interface was designed, implemented, and verified. The design included three major components: the APB wrapper, the UART Transmitter (TX), and the UART Receiver (RX). Each component was carefully analyzed through its finite state machine (FSM) and integrated into a cohesive top-level architecture.

A structured verification strategy was applied at both the module and system level. The TX and RX modules were validated with self-checking testbenches to ensure correct bit-level UART communication, including proper handling of start, data, and stop bits. The APB wrapper was verified by simulating CPU-driven transactions, confirming correct register operations, data transfers, and status flag updates.

The results demonstrated that the design meets its specifications:

- The transmitter correctly generates UART frames at the configured baud rate.
- The receiver accurately samples and reconstructs incoming serial data.
- The APB interface provides seamless communication between the UART core and an external bus master.

Overall, this project successfully achieved its objectives of implementing a reliable APB-based UART module, gaining experience in digital design, FSM modeling, bus interfacing, and verification methodology. The work provides a strong foundation for extending the design with additional features such as parity support, configurable stop bits, or interrupt-driven operation in future iterations.