.

A custom APB UART IP

Submitted by : *Mohamed sherif abd el mouniem*

ID: 22P0058

Major: COMM

Level: Junior

Submitted to:

Dr. Ghazal A. Fahmy

Eng. Mohamed Salah

Eng. Mohamed Tareq

INTRODUCTION:

Background:

In modern System-on-Chip (SoC) designs, processors need to communicate efficiently with peripheral devices. To make this easier, the Advanced Microcontroller Bus Architecture (AMBA) has become the standard for connecting different IP blocks in a consistent and scalable way. Within AMBA, the Advanced Peripheral Bus (APB) is a simple, low-power interface commonly used for accessing control registers of peripherals. At the same time, the Universal Asynchronous Receiver and Transmitter (UART) is still one of the most popular serial communication protocols because it is easy to use and reliable. By connecting UART through APB, SoCs can achieve efficient hardware integration while keeping the design accessible for software control.

Project Objective

The goal of this project is to create a custom UART IP core with an AMBA APB slave interface. The UART manages sending and receiving data over a serial link, while the APB wrapper allows access to its control, status, data, and baud rate registers through memory-mapped addressing. The project focuses on:

- Designing a register-mapped peripheral that follows the APB protocol.
- Building UART transmitter and receiver modules that support serial communication with adjustable baud rates.
- Enabling smooth communication between the CPU (or APB master) and the UART core through the wrapper module..
- Verify the design using self-checking Verilog testbenches and simulation results.

Scope and Learning Outcomes

This project goes beyond just building a UART core and its APB wrapper—it also highlights important system design practices. By working on it, students will gain practical experience in:

- **Creating peripheral IPs with APB-compliant interfaces.**.
- **Mapping hardware features to registers for processor-based control.**
- **Managing read and write synchronization between the CPU and the peripheral.**
- **Using testbenches to verify designs and interpret simulation waveforms**.
- **Integrating hardware modules into a complete SoC environment.**

DESIGN ANALYSIS :

UART Transmitter Module

The UART transmitter converts parallel data (tx_data) into a serial stream (tx_serial) following the UART protocol. The design is parameterized for different baud rates, clock frequencies, and word lengths.

Key Parameters:

- **BAUD_RATE, CLK_FREQ, DATA_BITS** – define baud rate (e.g., 9600 bps), system clock (e.g., 100 MHz), and data width (e.g., 8 bits).
- **CLKS_PER_BIT** – number of clock cycles per UART bit (CLK_FREQ / BAUD_RATE). Example: 100 MHz / 9600 ≈ 10416 cycles.
- **Counter widths** – $clog2 is used to size counters for scalability.

FSM (4 States):

1. **IDLE** – Line stays high. If tx_en is asserted, input data is latched and FSM moves to **START_BIT**.
2. **START_BIT** – Sends logic 0 for one baud period. Then goes to **DATA_BITS_STATE**.
3. **DATA_BITS_STATE** – Sends data bits from LSB to MSB, one per baud period. Moves to **STOP_BIT** after all bits are sent.
4. **STOP_BIT** – Sends logic 1 for one baud period, asserts tx_done, then returns to **IDLE**.

Control Signals:

- start_bit_init, data_bit_init, stop_bit_init, stop_bit_end – one-cycle flags to align timing and transitions.

Serial Line Behavior (tx_serial)

- **Idle:** High (1)
- **Start bit:** Low (0)
- **Data bits:** Output from r_tx_data[bit_cnter]
- **Stop bit:** High (1)

Counters:

- clk_cnter – Counts cycles for each baud period.
- bit_cnter – Tracks transmitted data bits.

Status Outputs:

- tx_busy – High during transmission (not IDLE).
- tx_done – Pulses high for one cycle after stop bit, signals completion.

.

UART Receiver Module

The receiver converts the serial input (rx_serial) into parallel data (rx_data). It uses status and handshake signals (rx_en, rx_rst, rx_busy, rx_done, rx_error) and runs on a 100 MHz clock with async active-low reset (arst_n).

Parameters:

- **BAUD_RATE, CLK_FREQ, DATA_BITS** – configure speed and word size.
- **CLKS_PER_BIT = CLK_FREQ / BAUD_RATE** – cycles per UART bit (≈10416 at 100 MHz/9600).
- Counter widths are set with $clog2 for flexibility.

Timing Strategy:

- Detect start at falling edge.
- Confirm start at the middle of the start bit.
- Sample data bits once per baud period (LSB first).
- Check stop bit at the end of its period.

Synchronizer:

- Two flip-flops (serial_sync0 → serial_sync1) bring rx_serial safely into the clock domain.
- Default reset value = 1 (idle).
- FSM always uses serial_sync1.

FSM States:

- **IDLE** – Wait for low (start bit) when rx_en=1.

- **START_BIT** – Half-bit delay, confirm line still low. If not, return to IDLE.
- **DATA_BITS_STATE** – Capture data bits, LSB first.
- **STOP_BIT** – Wait one bit time, check if line is high (valid frame).

Counters:

- clk_cnter: Measures half-bit in START, full-bit in DATA/STOP.
- bit_cnter: Tracks number of received bits.

Data Path:

- On each data sample, latch serial_sync1 into rx_data[bit_cnter].
- Naturally stores LSB first.
- rx_data clears on reset or rx_rst=1.

Error & Robustness:

- Start bit must be confirmed mid-bit → filters out glitches.
- Stop bit check:
- High → valid frame.
- Low → framing error (rx_error=1).
- rx_done always pulses at stop check (even if error).
- Valid byte = rx_done && !rx_error.

Status Signals

- rx_busy: High during reception (non-IDLE).
- rx_done: Pulses when frame ends.
- rx_error: Pulses if stop bit invalid.

Reset:

- **Arst_n=0** (async): Clears FSM, counters, sync, outputs.
- **rx_rst=1** (sync): Same cleanup, allows abort during frame.

APB UART Wrapper Module

Purpose:

- Connects the APB bus to the UART core (TX + RX).
- Provides CPU access via memory-mapped registers.
- Manages read/write operations, status flags, and error reporting.

Register Map (32-bit wide, lower 8 bits used for data):

- **CTRL_REG (0x0000):** Control bits → tx_en, rx_en, tx_rst, rx_rst.
- **STATS_REG (0x0001):** Status bits → tx_busy, tx_done, rx_busy, rx_done, rx_error.
- **TX_DATA (0x0002):** Data to transmit.
- **RX_DATA (0x0003):** Last received byte.

Control Mapping:

- ctrl_reg[0] → tx_en
- ctrl_reg[1] → rx_en
- ctrl_reg[2] → tx_rst
- ctrl_reg[3] → rx_rst
- tx_data_reg[7:0] → TX parallel data

Status Updates:

- Status register continuously mirrors UART outputs:
  - Bit 0: tx_busy
  - Bit 1: tx_done
  - Bit 2: rx_busy
  - Bit 3: rx_done
  - Bit 4: rx_error
- On valid receive (rx_done=1), received byte is latched into rx_data_reg.

APB FSM (3 Phases):

1. **IDLE:** Wait for PSEL=1, PREADY=0.
2. **SETUP:** Capture address/direction, PENABLE=0.
3. **ACCESS:** Perform read/write, PENABLE=1, PREADY=1, then return to IDLE.

APB Write:

- Done in ACCESS when PWRITE=1.
- Valid addresses: update CTRL_REG, TX_DATA, BAUDIV.
- Invalid → PSLVERR=1.

APB Read:

- Done in ACCESS when PWRITE=0.
- Valid addresses: return CTRL, STATUS, TX, RX, or BAUDIV.
- Invalid → return 0, PSLVERR=1.

Error Handling:

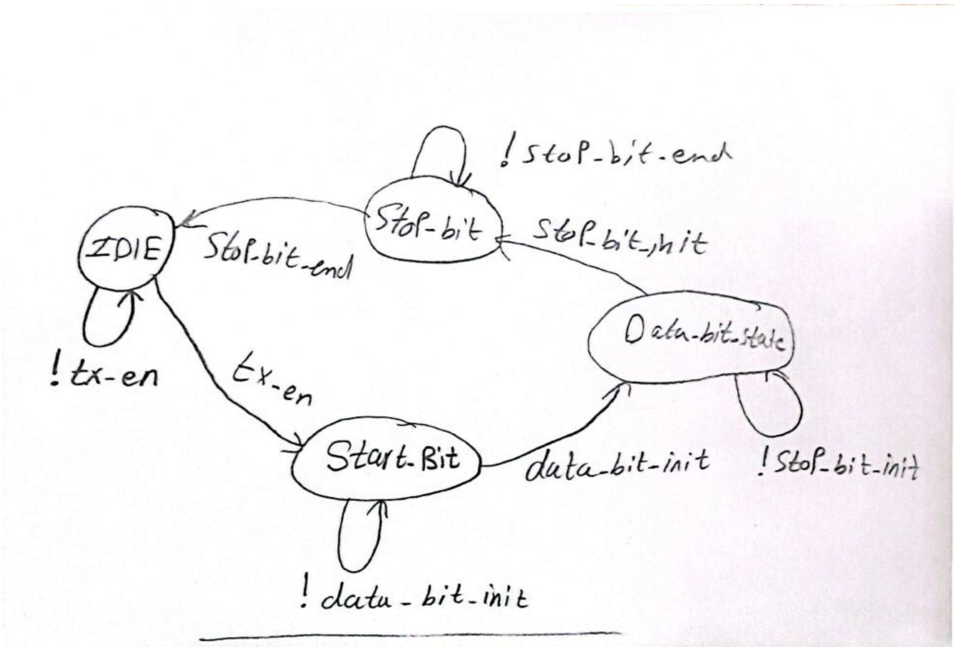
- Invalid address → PSLVERR=1.
- UART framing error → reported in STATS_REG[4].

UART Instantiation:

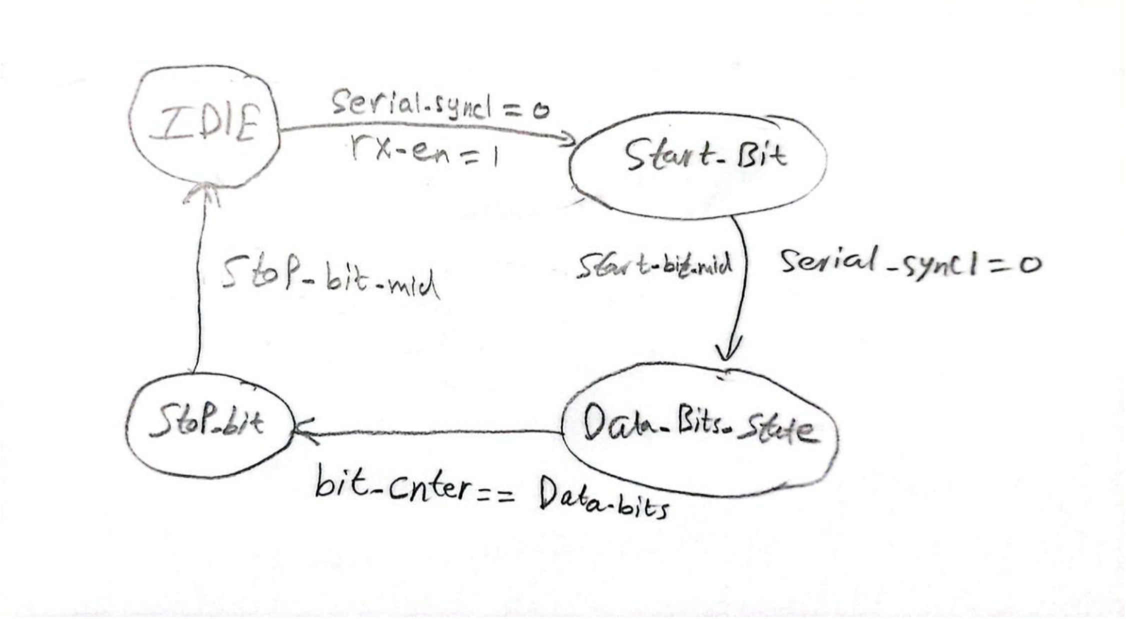
- Wrapper instantiates uart_tx and uart_rx.
- tx_serial and rx_serial connected externally.
- Control/data signals mapped from registers.
- Status fed back into STATS_REG.

STATE DIAGRAMS:

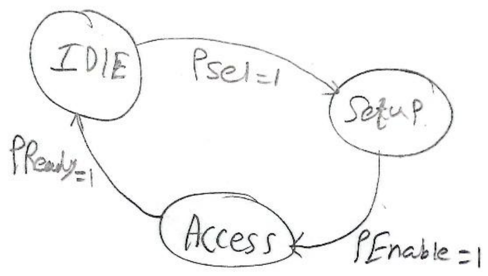
UART Transmitter state diagram



UART Receiver state diagram



APB Wrapper state diagram



DESIGN DECISIONS:

Key Design Decisions:

Parameterization for Flexibility

- Transmitter and receiver are parameterized by **baud rate, system clock, and data width**.
- Allows reuse across systems (e.g., 50 MHz or 100 MHz clock, 9600 or 115200 baud, 7/8-bit frames).
- Trade-off: must carefully calculate CLKS_PER_BIT to avoid TX/RX mismatch.

FSM-Based Control:

- Both TX and RX use a **4-state FSM**: IDLE → START_BIT → DATA_BITS → STOP_BIT.
- FSMs improve readability, debugging, and verification compared to counter-only designs.
- Slightly more code, but more reliable and maintainable.

Cycle Counters for Bit Timing:

- A **clk_cnter** counts system clock cycles per bit instead of using a separate baud clock.
- Keeps everything in one clock domain → avoids metastability and jitter.

- Example: 100 MHz / 9600 baud ≈ 10416 cycles/bit.
- Trade-off: counter width must support large clock-to-baud ratios.

Receiver Sampling Strategy:

- Start bit sampled at **mid-bit** (CLKS_PER_BIT/2) → filters glitches.
- Data bits sampled at **full-bit intervals** → ensures alignment with TX.
- More robust than edge-aligned sampling.

Error Detection:

- Receiver checks **stop bit** at expected time.
- If low → set rx_error (framing error).
- Simpler than adding parity but still catches corrupted frames.

Status & Handshake Signals:

- TX: tx_busy, tx_done. RX: rx_busy, rx_done, rx_error.
- Prevents overwriting or missing data.
- Provides safe interaction with CPU or other logic.

APB Wrapper Integration:

- UART mapped into APB registers:
  - CTRL_REG (enable/reset)
  - STATS_REG (busy/done/error)
  - TX_DATA / RX_DATA (data I/O)
- APB chosen for simplicity and compatibility with SoC designs.
- Trade-off: wrapper adds complexity, but enables clean CPU access.

Structured Testbenches:

- Separate testbenches for TX, RX, and APB wrapper.
- Each uses tasks (send_data, apb_write, apb_read) for clarity.
- Benefits: easier debugging, modular verification, supports regression testing.

VERIFICATION STRATEGY :

UART Transmitter Verification

Objective
The purpose of this testbench is to confirm that the transmitter correctly serializes parallel data into start, data, and stop bits with precise baud-rate timing.

Methodology
The testbench instantiates the uart_tx module with parameters for baud rate, clock frequency, and data width. A clock generator toggles every 5 ns to emulate a 100 MHz system clock. Reset (arst_n) is asserted low at the beginning, then released to initialize the design.

A task named send_data drives parallel data into the transmitter. For each transaction:

- tx_data is loaded with the desired byte (e.g., 8'h55, 8'hAF).
- The enable (tx_en) is pulsed high to start transmission.
- The task then waits for the full frame time (WAIT_TIME), ensuring the serial sequence completes before another transmission begins.

The testbench monitors the following behaviors:

- **tx_serial waveform**: verifies the expected sequence of bits (start = 0, data bits LSB-first, stop = 1).
- **tx_busy signal**: remains high throughout the transmission period.
- **tx_done pulse**: occurs exactly one clock cycle after the stop bit.

Results

Simulation confirmed that:

- The serialized output matched the expected patterns (e.g., 8’h55 = 01010101, 8’hAF = 11110101, transmitted LSB-first).
- Each bit lasted precisely CLKS_PER_BIT cycles, consistent with the baud rate divider.
- The transmitter returned to the IDLE state immediately after the tx_done pulse.

UART Receiver Verification

Objective

Confirm that the receiver correctly detects the start bit, samples incoming data bits at mid-bit intervals, and reconstructs the transmitted byte. Additionally, verify that error detection (framing error) works when the stop bit is invalid.

Methodology

- The testbench generated a valid UART frame on the rx_serial line using the send_byte task.
- A start bit (0) was driven, followed by 8 data bits (LSB first), and finally a stop bit (1).
- Sampling was aligned to the middle of each bit period (CLKS_PER_BIT/2) to ensure robustness against noise.
- The reconstructed data (rx_data) was compared against the expected value.
- Error detection was tested by intentionally holding the stop bit low, simulating a framing error.

Results

- Simulation waveforms confirmed correct byte reconstruction for input values such as 8'h55.
- The rx_done flag asserted exactly at the completion of the stop bit.
- During the error test, the rx_error signal was asserted, demonstrating reliable framing error detection.

APB Wrapper Verification

Objective

Confirm that the UART core can be properly controlled and accessed via the APB bus through memory-mapped registers, ensuring seamless integration with a CPU.

Methodology

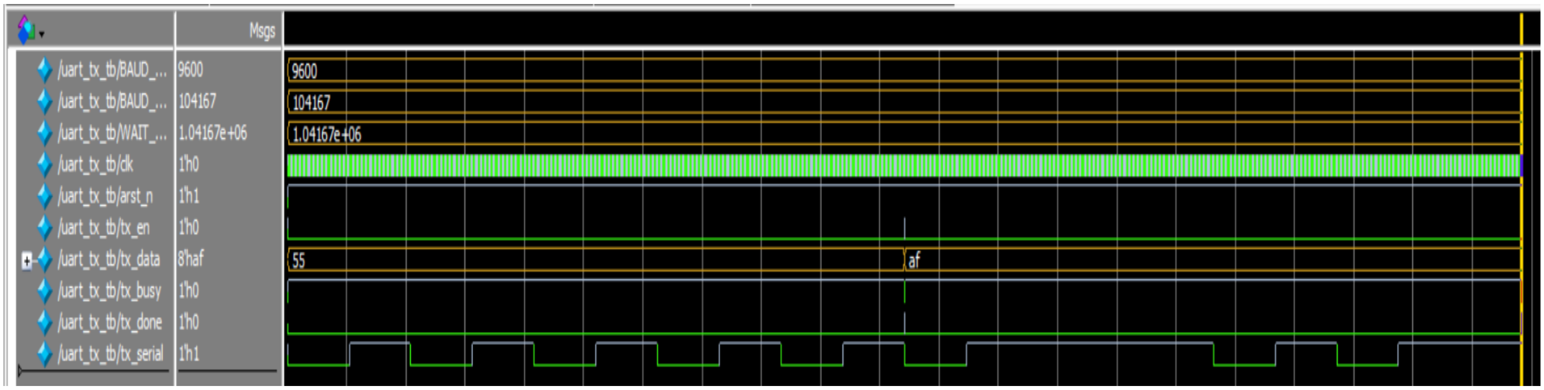
- The testbench implemented APB transactions using dedicated tasks:
- apb_write for register writes.
- apb_read for register reads.
- The following scenarios were exercised:
- Writing a byte to the **TX_DATA** register and confirming that transmission started.
- Writing to the **CTRL_REG** to enable and reset TX/RX functionality.
- Reading the **STATS_REG** to verify status flags (tx_busy, tx_done, rx_busy, rx_done, rx_error).
- Simulating a received byte (8'hA5) on the rx_serial line and checking that it was latched into **RX_DATA**.
- Modifying the **BAUDIV** register to confirm correct update of the baud divider and timing.

Results

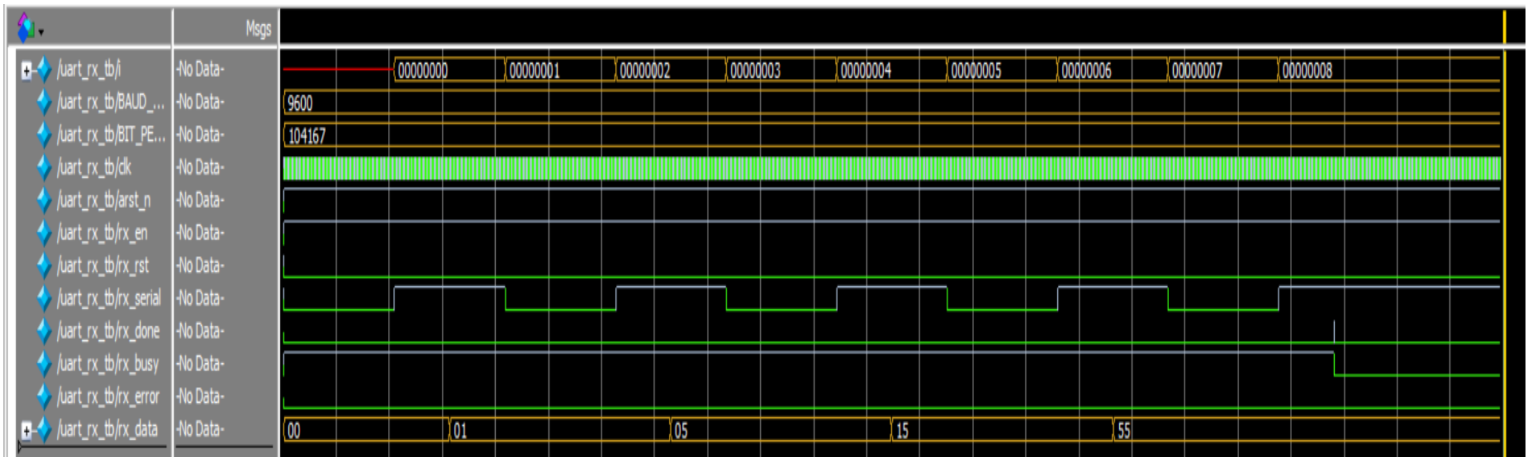
- All APB registers responded correctly to read/write operations.
- Status flags updated in synchronization with TX and RX activities.
- Received data was successfully captured into the RX register.
- Changes to the baud rate divider affected the bit timing as expected.
- No spurious PSLVERR signals were observed during valid transactions.

SIMULATION RESULTS :

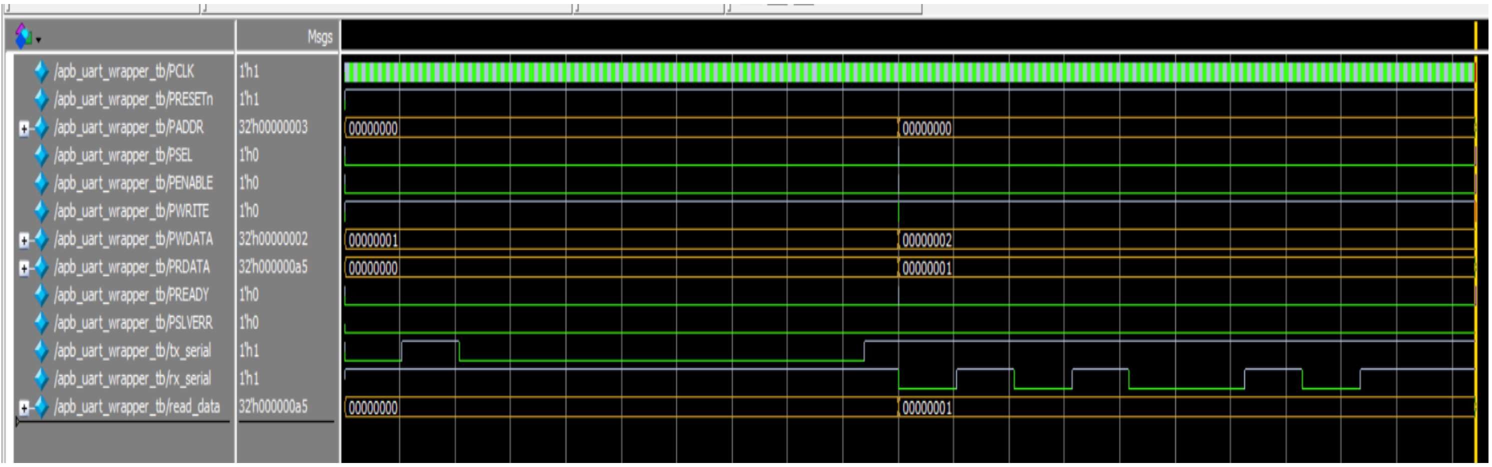
UART transmitter simulation

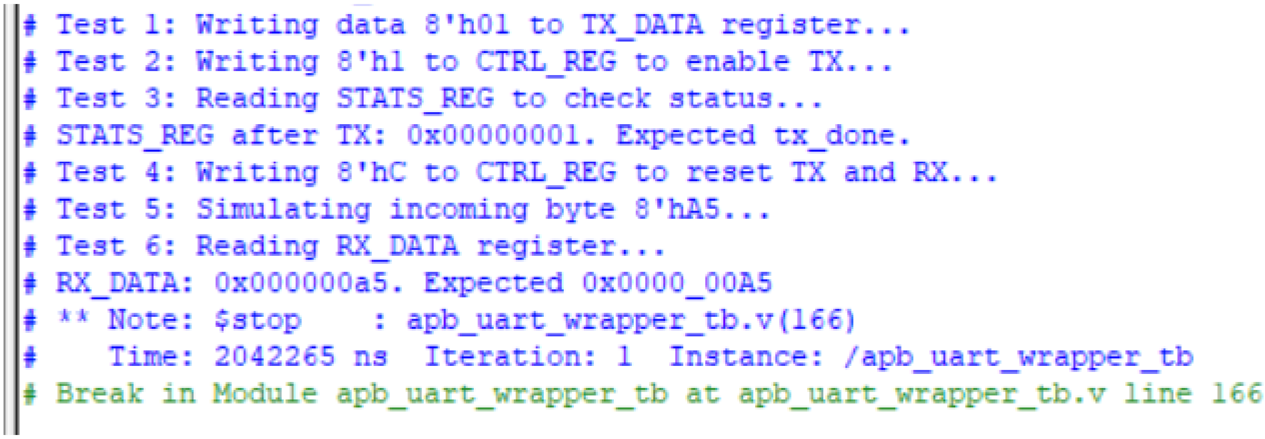


UART receiver simulation


Msgs
/uart_rx_tb/i -No Data-
/uart_rx_tb/BAUD_... -No Data-
/uart_rx_tb/BIT_PE... -No Data-
/uart_rx_tb/clk -No Data-
/uart_rx_tb/arst_n -No Data-
/uart_rx_tb/rx_en -No Data-
/uart_rx_tb/rx_rst -No Data-
/uart_rx_tb/rx_serial -No Data-
/uart_rx_tb/rx_done -No Data-
/uart_rx_tb/rx_busy -No Data-
/uart_rx_tb/rx_error -No Data-
/uart_rx_tb/rx_data -No Data-
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
00000008
9600
104167
00
01
05
15
55

APB Wrapper simulation


Msgs
/apb_uart_wrapper_tb/PCLK 1'h1
/apb_uart_wrapper_tb/PRESETn 1'h1
/apb_uart_wrapper_tb/PADDR 32'h00000003
/apb_uart_wrapper_tb/PSEL 1'h0
/apb_uart_wrapper_tb/PENABLE 1'h0
/apb_uart_wrapper_tb/PWRITE 1'h0
/apb_uart_wrapper_tb/PWDATA 32'h00000002
/apb_uart_wrapper_tb/PRDATA 32'h000000a5
/apb_uart_wrapper_tb/PREADY 1'h0
/apb_uart_wrapper_tb/PSLVERR 1'h0
/apb_uart_wrapper_tb/tx_serial 1'h1
/apb_uart_wrapper_tb/rx_serial 1'h1
/apb_uart_wrapper_tb/read_data 32'h000000a5
00000000
00000000
00000001
00000002
00000000
00000001
00000000
00000001



```
# Test 1: Writing data 8'h01 to TX_DATA register...
# Test 2: Writing 8'h1 to CTRL_REG to enable TX...
# Test 3: Reading STATS_REG to check status...
# STATS_REG after TX: 0x00000001. Expected tx_done.
# Test 4: Writing 8'hC to CTRL_REG to reset TX and RX...
# Test 5: Simulating incoming byte 8'hA5...
# Test 6: Reading RX_DATA register...
# RX_DATA: 0x000000a5. Expected 0x0000_00A5
# ** Note: $stop    : apb_uart_wrapper_tb.v(166)
#    Time: 2042265 ns  Iteration: 1  Instance: /apb_uart_wrapper_tb
# Break in Module apb_uart_wrapper_tb at apb_uart_wrapper_tb.v line 166
```

CONCLUSION

In this project, a Universal Asynchronous Receiver-Transmitter (UART) was designed, implemented, and verified using Verilog HDL. The design included both a transmitter and a receiver, each managed by finite state machines to ensure reliable serialization and deserialization of data. Parameterization for baud rate and data width, mid-bit sampling, and framing error detection were key design choices that improved flexibility and robustness.

To enable integration into processor-based systems, an APB wrapper was developed. This made the UART accessible as a memory-mapped peripheral, simplifying control and enhancing reusability in SoC environments.

Verification was carried out at both the module and system levels. Module-level tests confirmed the correct operation of the transmitter, receiver, and APB wrapper, while integration-level loopback tests validated end-to-end communication. Additional corner cases, such as back-to-back transfers, framing errors, and baud rate mismatches, were also tested to ensure reliability.

The results confirmed that the UART met all functional and timing requirements. It successfully transmitted and received data, detected errors, and interfaced seamlessly with the APB bus. The design is reliable, scalable, and well-suited for extension.

Future improvements could include adding FIFO buffers for higher throughput, parity or CRC error checking for greater reliability, and interrupt support for easier processor interaction.

Overall, this project demonstrates the complete digital design flow — from architecture to RTL implementation and verification — and highlights the importance of building reusable hardware IP cores for modern SoC systems.