National Telecommunication Institute

Digital IC Design using FPGA

Custom APB UART IP

Farah Ihab Zidan

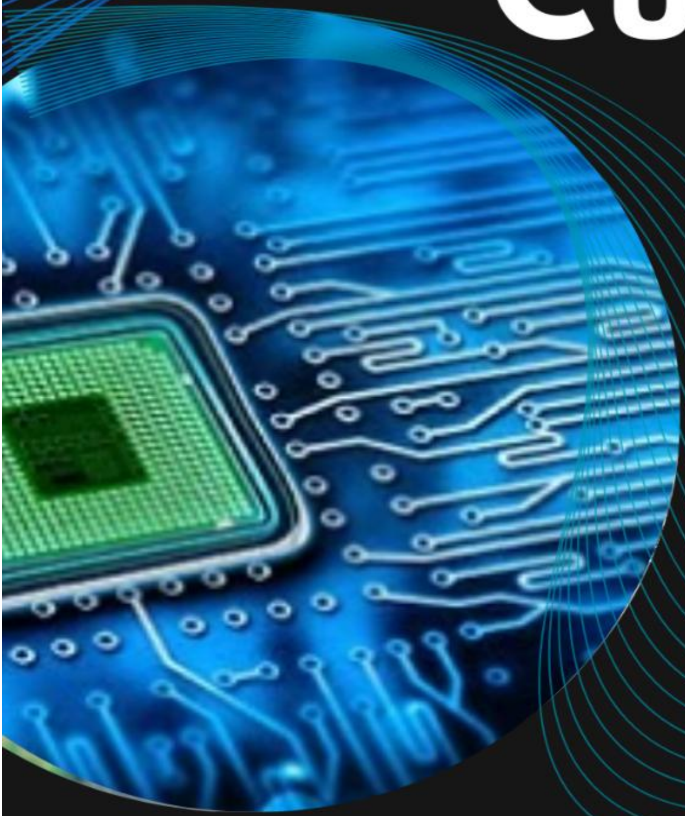


Table of Contents

1.0 INTRODUCTION 3

2.0 DESIGN ANALYSIS 4

2.1 Baud Generator 4

2.2 UART Receiver 4

2.3 UART Transmitter 4

2.4 APB Interface 5

2.5 Register File 5

2.6 Top modules 6

3.0 STATE DIAGRAMS 6

3.1 UART Rx and Tx FSM 6

3.2 APB FSM 6

4.0 DESIGN DECISIONS 7

4.1 Baud Generator 7

4.2 UART Receiver 7

4.3 UART Transmitter 7

4.4 APB Interface 7

4.5 Register file 8

5.0 VERIFICATION STRATEGY 8

5.1 Baud Generator TB 8

5.2 Receiver TB 8

5.3 Transmitter TB 8

5.4 APB and Register file TB 9

5.5 UART APB Top Module TB 9

6.0 SIMULATION RESULTS 10

6.1 Receiver TB 10

6.2 Transmitter TB 10

6.3 APB and Register file TB 10

6.4 UART APB Top Module TB 11

7.0 CONCLUSION 12

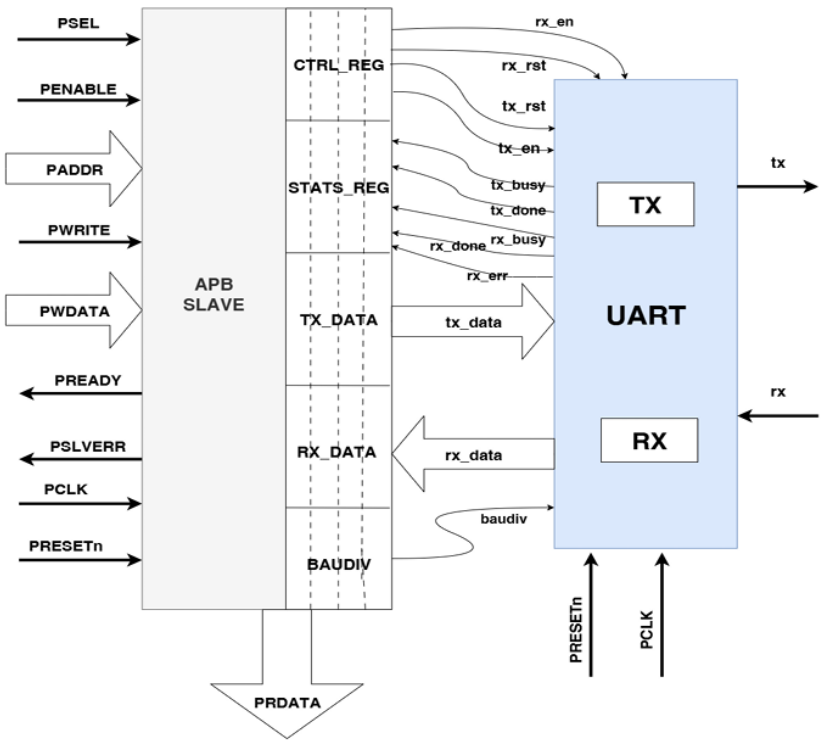
1.0 INTRODUCTION

This project focuses on the design and verification of a custom Universal Asynchronous Receiver/Transmitter (UART) peripheral that is wrapped with an AMBA Advanced Peripheral Bus (APB) interface. The motivation behind this work is to provide a reusable, configurable, and easily integrable IP block that can be deployed as part of a larger System-on-Chip (SoC) design.

UART is one of the most fundamental and widely adopted serial communication standards. It allows two devices to exchange data asynchronously using just two lines—TX (transmit) and RX (receive)—with minimal hardware complexity. Despite being an “old” protocol, UART remains essential in embedded systems, FPGAs, microcontrollers, and debugging interfaces due to its simplicity and robustness.

In modern SoCs, different communication buses are used to connect CPUs, memories, and peripherals. The APB (Advanced Peripheral Bus), a subset of ARM’s AMBA bus family, is specifically optimized for low-power, low-latency control registers. By wrapping the UART core with an APB slave interface, the peripheral can be memory-mapped, meaning software running on the CPU can configure and control the UART by simply performing read and write operations to predefined addresses.

The design, implementation, and verification of such an IP block allow us to strengthen our skills in digital logic design, bus interfacing, and verification methodologies.



2.0 DESIGN ANALYSIS

2.1 Baud Generator

The baud generator is the timing heart of the UART system. Since UART communication requires precise synchronization between the transmitter and receiver, the baud generator ensures that both blocks operate at the desired baud rate relative to the system clock. In this design, a divisor value is either set by default or configured through the BAUDIV register. The divisor is latched, and a counter decrement from it on every system clock cycle. When the counter reaches zero, a baud tick signal is asserted, which serves as a timing pulse for the TX and RX finite state machines (FSMs). To maintain stability, if the divisor is updated, the counter resets immediately to prevent timing mismatches. This approach makes the design flexible, as it supports dynamic baud rate reconfiguration without requiring a system reset.

2.2 UART Receiver

The UART receiver (RX) is responsible for converting serial data on the input line into parallel data for the processor. Its operation is controlled by an FSM that progresses through distinct states. In the idle state, the RX line is monitored for a falling edge, which indicates the start bit. Once detected, the FSM waits for 1.5-bit periods before sampling the line to confirm the start bit. The receiver then samples incoming bits at the middle of each bit period, shifting them into a Serial-In Parallel-Out (SIPO) register. After collecting all data bits, the FSM verifies the stop bit. If the stop bit is valid, the data is transferred to the RX_DATA register and the rx_done flag is asserted. If the stop bit is incorrect, a framing error is flagged via the rx_error signal. The receiver asserts rx_busy while capturing a frame, and the inclusion of rx_rst ensures that the FSM can be reinitialized at any point.

2.3 UART Transmitter

The UART transmitter (TX) performs the reverse operation by converting parallel data into a properly framed serial bitstream. Its FSM begins in the idle state, waiting for tx_en to be asserted and data to be loaded into the TX_DATA register. Once active, the FSM sends a logic low start bit for one baud period, followed by shifting out eight data bits, least significant bit first. Each bit is held stable on the TX line for one baud interval, synchronized by the baud generator. Finally, a stop bit (logic high) is transmitted for one baud period. Upon completion, the FSM asserts the tx_done signal and deasserts tx_busy, signaling that the transmitter is ready for the next byte. A soft reset, tx_rst, is available to clear the FSM and restart transmission if required.

2.4 APB Interface

The APB interface serves as the bridge between the processor and the UART core. It is designed to comply with the AMBA APB protocol, which provides a simple two-phase transaction consisting of setup and access. During a write operation, the APB interface captures the address and data when PSEL=1 and PWRITE=1, and then asserts PREADY for one cycle to confirm the transfer. During a read operation, when PWRITE=0, the addressed register value is placed onto the PRDATA bus, again with PREADY indicating a valid transaction. Between operations, the FSM returns to the idle state. This design ensures proper synchronization between the processor and the UART registers, making the peripheral behave like any other memory-mapped device in the SoC.

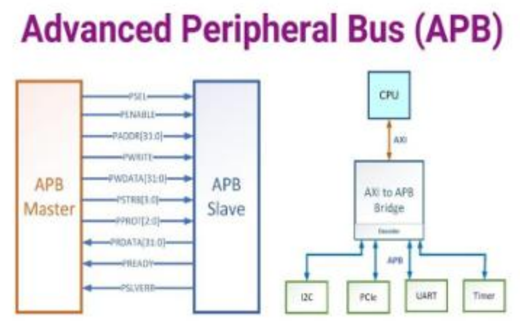


Figure 6: APB Prortocol A

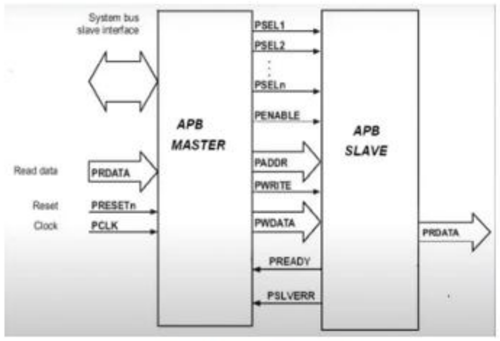


Figure 7: APB Prortocol B

2.5 Register File

Finally, the register file acts as the control and status hub of the design. It includes five memory-mapped registers, each 32 bits wide. The CTRL_REG contains enable and reset bits for both TX and RX (tx_en, rx_en, tx_rst, rx_rst), allowing software to start or reset the UART. The STATS_REG holds read-only flags such as tx_busy, rx_busy, tx_done, rx_done, and rx_error, which reflect the current status of the transmitter and receiver. The TX_DATA register is written by the processor to initiate a transmission, while the RX_DATA register is updated with the most recently received byte when rx_done is asserted. The BAUDIV register holds the divisor for the baud generator, enabling configurable baud rates. Together, these registers form the essential interface through which the processor controls and monitors UART operation.

Address	Name	Description
0x0000	CTRL_REG	Contains bits for tx_en , rx_en , tx_rst & rx_rst
0x0001	STATS_REG	Contains bits for rx_busy , tx_busy , rx_done, tx_done & rx_error
0x0002	TX_DATA	UART Tx data
0x0003	RX_DATA	UART Rx data
0x0004	BAUDIV	Variable baud rate for UART [BONUS]

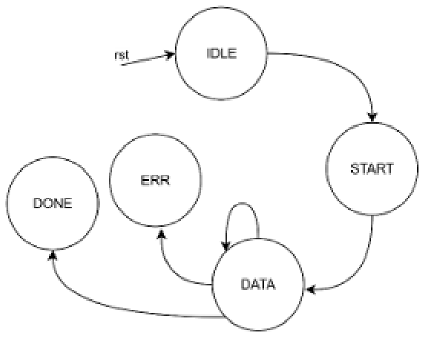
2.6 Top modules

The design was organized into higher-level modules to improve hierarchy and clarity.

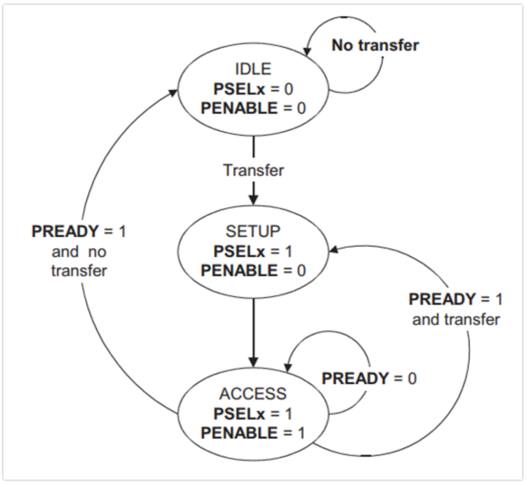
- The UART module encapsulates both the transmitter and receiver, providing a unified interface for serial data handling while keeping TX and RX internally independent.
- The APB_REG module combines the APB interface with the register file, allowing address decoding, read/write operations, and register updates to occur in one place, which simplifies datapath design and reduces signal wiring.
- The TOP module integrates the uart, apb_reg, and the baud generator into a single APB-compliant UART IP. This top module acts as the glue logic, ensuring that register writes trigger UART operations and status updates are correctly reflected back to the processor.

3.0 STATE DIAGRAMS

3.1 UART Rx and Tx FSM



3.2 APB FSM



4.0 DESIGN DECISIONS

4.1 Baud Generator

The baud generator was designed as a standalone, parameterized unit. This modularity allowed easier debugging and reusability across different designs. By latching the divisor and resetting the counter when it changes, the baud generator ensures clean transitions when the baud rate is reconfigured. This prevents glitches and ensures precise timing, which is essential in asynchronous protocols. The use of parameters for system clock frequency, oversampling rate, and baud rate also provides flexibility, enabling the module to be reused in systems with different configurations without modifying the RTL code.

4.2 UART Receiver

The receiver design decision is to implement a clear FSM with separate states for idle, start, data, stop, done, and error improves both readability and reliability. By waiting for 1.5 bit-times before sampling the first data bit, the design ensures robust synchronization and minimizes the chance of misinterpreting noise as a valid start bit. Oversampling at 16x improves accuracy, allowing the FSM to sample near the center of each bit period. Additionally, the choice to shift incoming bits into a SIPO register simplifies parallel data reconstruction. The receiver asserts rx_busy during data capture, uses rx_done to signal successful reception, and raises rx_error in case of a framing issue. These decisions were made to make the receiver both error-tolerant and software-friendly, since the CPU can monitor clear status flags.

4.3 UART Transmitter

The transmitter design also uses a clean FSM approach with states for idle, start, data, and stop. A key design choice here is to preload the parallel input data into a shift register at the start of transmission. This guarantees that the byte being transmitted remains stable, even if the CPU writes new data to the TX register mid-transmission. The transmitter outputs bits in a least-significant-bit-first order, consistent with UART convention. The FSM asserts tx_busy during the frame and tx_done at completion, giving clear visibility of the transmitter’s status. A tx_rst input was included to allow soft reset if transmission needs to be aborted or restarted, which is particularly important in noisy or error-prone environments.

4.4 APB Interface

The APB interface was designed to strictly follow the AMBA APB protocol, which uses a two-phase transaction consisting of setup and access states. Register addresses are mapped using PADDR[4:2], ensuring word-aligned addressing where each 32-bit register corresponds to a fixed offset. The interface asserts PREADY for one cycle to acknowledge valid transfers, while generating separate read and write enables for the register file. This decision ensures clean synchronization between the APB master and the UART peripheral while maintaining protocol compliance. Resetting all signals on PRESETn guarantees predictable startup behavior and simplifies SoC integration.

4.5 Register file

The register file was designed to provide a structured memory-mapped interface that exposes both control and status of the UART core to software. Registers include CTRL_REG for enables and resets, STATS_REG for monitoring busy, done, and error signals, TX_DATA for transmitting new bytes, and RX_DATA for storing received bytes. A decision was made to latch the most recent received data to avoid data loss, and to generate a tx_start pulse automatically when TX data is written, reducing software complexity. Read data is driven combinationally, enabling immediate responses during APB reads.

5.0 VERIFICATION STRATEGY

5.1 Baud Generator TB

- Verify correct initialization under reset and ensure that the default divisor produces baud ticks at the expected frequency.
- Confirm that providing an external divisor updates the tick generation correctly, including handling dynamic reconfiguration during operation.
- Validate stable and periodic baud tick pulses over extended operation, with internal signals monitored to ensure alignment with expected timing.

5.2 Receiver TB

- **Functional Reception:** The testbench uses tasks (send_bit, send_byte) to generate valid UART frames at 16× oversampling. Multiple bytes (0x55, 0xF1, 0xA3) are transmitted to verify correct sampling, shifting, and reconstruction into parallel data.
- **Error Handling:** A dedicated task (send_byte_bad_stop) intentionally sends an invalid stop bit to trigger the error state, verifying that rx_error is asserted while rx_done is not, ensuring robustness against framing errors.
- **Control and Status Validation:** Reset tasks (pulse_rx_rst) check proper reinitialization, while monitoring rx_busy, rx_done, and data_out confirms correct FSM progression and status flag behavior across different scenarios.

5.3 Transmitter TB

Transmission Functionality: A task (send_byte) was implemented to load input data, assert tx_start, and monitor the FSM through start, data, and stop states. Multiple patterns (0x55, 0xA5, 0xFF) were transmitted to confirm correct serial framing and LSB-first shifting.

- **Control and Status Monitoring:** Verification checked that tx_busy asserted during transmission and cleared after completion, while tx_done correctly flagged frame completion. These signals ensured synchronization between hardware and software control.
- **Reset and Enable Behavior:** Reset initialization (rst_n) and enable control (tx_en) were validated to guarantee that the transmitter only starts when enabled and properly reinitializes after reset, maintaining deterministic and reliable operation.

5.4 APB and Register file TB

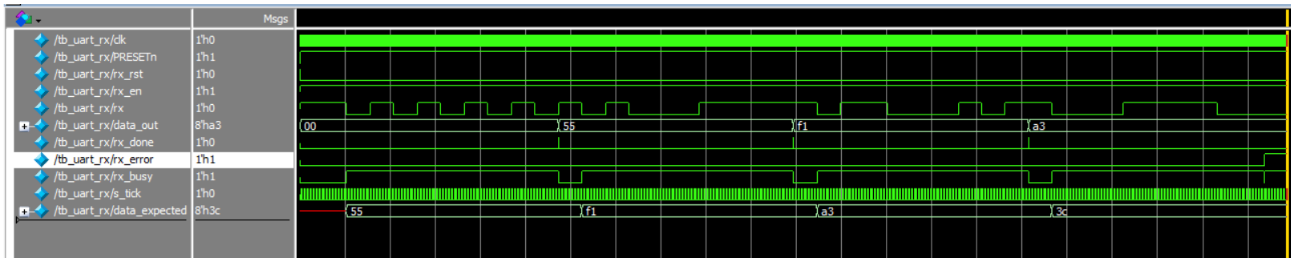
- **Register Access Verification:** Tasks (apb_write, apb_read) were used to validate correct APB protocol operation, ensuring that writes update control, baud, and data registers while reads return the expected values from RX_DATA and STATUS.
- **Control and Status Checks:** Tests confirmed that enabling TX/RX, asserting/deasserting resets, and updating TX_DATA correctly propagated to the module outputs, while status bits reflected UART activity (tx_busy, rx_busy, tx_done, rx_done, rx_error).
- **Functional Integration:** The testbench verified proper datapath connectivity, such as BAUD_DIV programming, TX data transfer, RX data latching, and reset behavior, ensuring the APB interface and register file operate as a cohesive unit within the UART IP.

5.5 UART APB Top Module TB

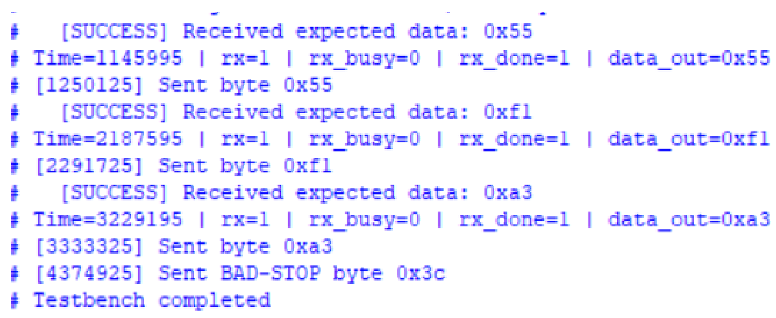
- **APB Register Interface Validation:** The testbench used tasks (do_write, do_read) to exercise memory-mapped operations, ensuring correct read/write functionality for control, status, TX, and RX registers when accessed through the APB bus.
- **End-to-End Data Path Testing:** Bytes were transmitted via APB writes to TXDATA, passed through the UART transmitter, looped back into the receiver, and read back via RXDATA. This confirmed complete TX → RX → APB integration and functional correctness.
- **System-Level Control and Status Checks:** Verification included enabling/disabling TX/RX, monitoring status flags (tx_busy, rx_busy, tx_done, rx_done, rx_error), and ensuring proper reset behavior across the entire integrated UART IP.

6.0 SIMULATION RESULTS

6.1 Receiver TB

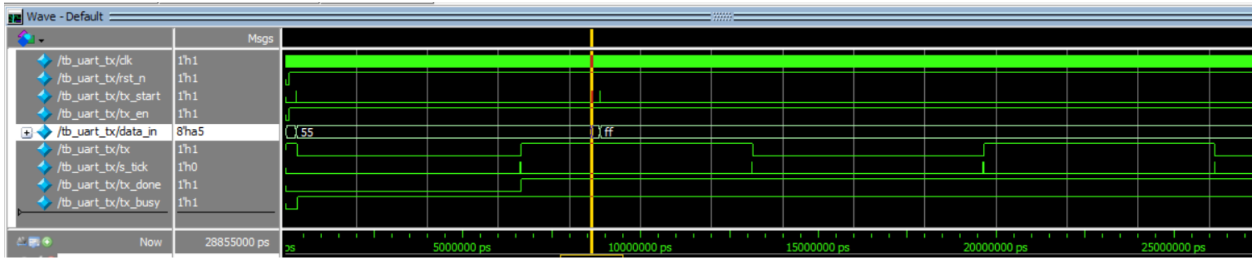


```
/tb_uart_rx/clk            1'h0
/tb_uart_rx/PRESETn        1'h1
/tb_uart_rx/rx_rst         1'h0
/tb_uart_rx/rx_en          1'h1
/tb_uart_rx/rx             1'h0
/tb_uart_rx/data_out       8'ha3
/tb_uart_rx/rx_done        1'h0
/tb_uart_rx/rx_error       1'h1
/tb_uart_rx/rx_busy        1'h1
/tb_uart_rx/s_tick         1'h0
/tb_uart_rx/data_expected  8'h3c
```



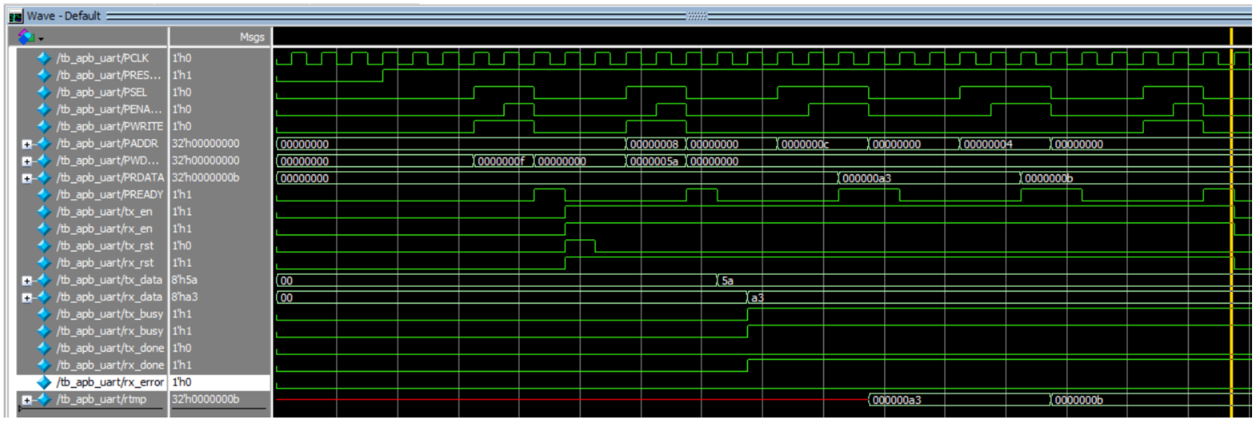
```
#   [SUCCESS] Received expected data: 0x55
# Time=1145995 | rx=1 | rx_busy=0 | rx_done=1 | data_out=0x55
# [1250125] Sent byte 0x55
#   [SUCCESS] Received expected data: 0xf1
# Time=2187595 | rx=1 | rx_busy=0 | rx_done=1 | data_out=0xf1
# [2291725] Sent byte 0xf1
#   [SUCCESS] Received expected data: 0xa3
# Time=3229195 | rx=1 | rx_busy=0 | rx_done=1 | data_out=0xa3
# [3333325] Sent byte 0xa3
# [4374925] Sent BAD-STOP byte 0x3c
# Testbench completed
```

6.2 Transmitter TB

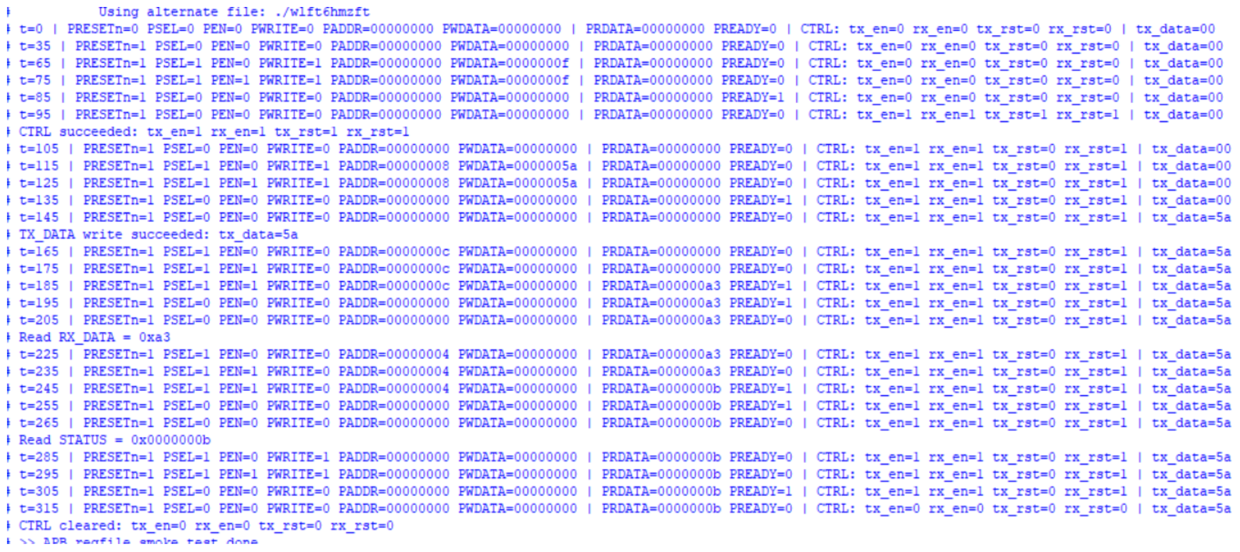


```
/tb_uart_tx/clk       1'h1
/tb_uart_tx/rst_n     1'h1
/tb_uart_tx/tx_start  1'h1
/tb_uart_tx/tx_en     1'h1
/tb_uart_tx/data_in   8'ha5
/tb_uart_tx/tx        1'h1
/tb_uart_tx/s_tick    1'h0
/tb_uart_tx/tx_done   1'h1
/tb_uart_tx/tx_busy   1'h1
Now                   28855000 ps
```

6.3 APB and Register file TB

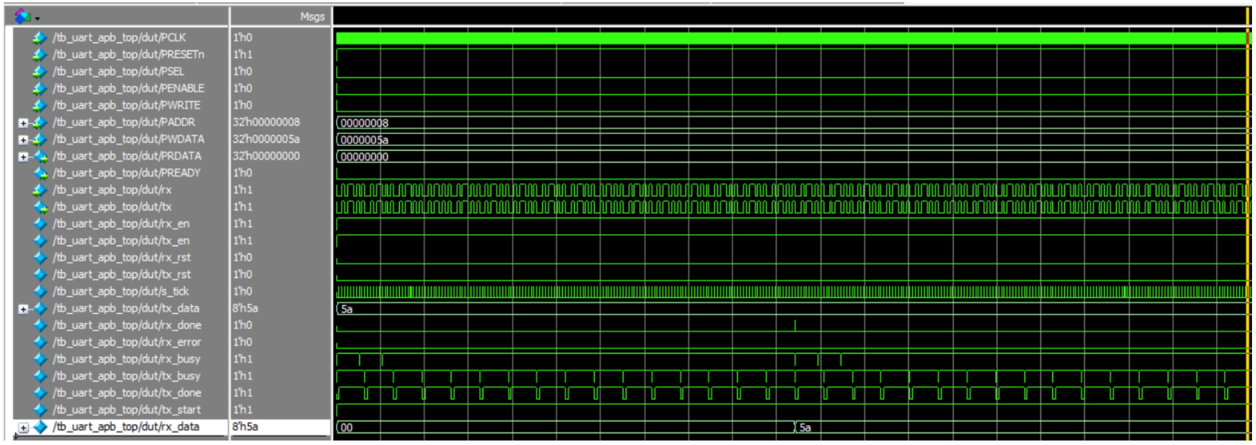


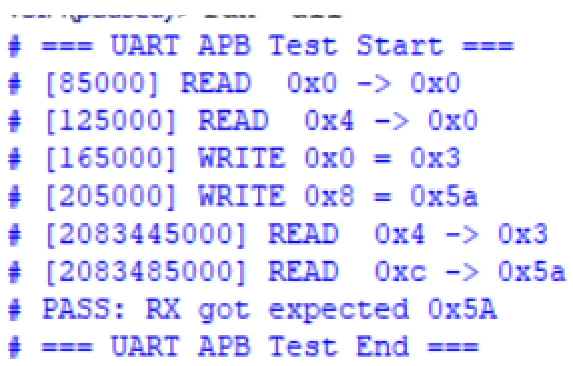
```
/tb_apb_uart/PCLK      1'h0
/tb_apb_uart/PRES...   1'h1
/tb_apb_uart/PSEL      1'h0
/tb_apb_uart/PENA...   1'h0
/tb_apb_uart/PWRITE    1'h0
/tb_apb_uart/PADDR     32'h00000000
/tb_apb_uart/PWD...    32'h00000000
/tb_apb_uart/PRDATA    32'h0000000b
/tb_apb_uart/PREADY    1'h1
/tb_apb_uart/tx_en     1'h1
/tb_apb_uart/rx_en     1'h1
/tb_apb_uart/tx_rst    1'h0
/tb_apb_uart/rx_rst    1'h1
/tb_apb_uart/tx_data   8'h5a
/tb_apb_uart/rx_data   8'ha3
/tb_apb_uart/tx_busy   1'h1
/tb_apb_uart/rx_busy   1'h1
/tb_apb_uart/tx_done   1'h0
/tb_apb_uart/rx_done   1'h1
/tb_apb_uart/rx_error  1'h0
/tb_apb_uart/rtmp      32'h0000000b
```



```
#           Using alternate file: ./wlft6hmzft
# t=0 | PRESETn=0 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=00
# t=35 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=00
# t=65 | PRESETn=1 PSEL=1 PEN=0 PWRITE=1 PADDR=00000000 PWDATA=0000000f | PRDATA=00000000 PREADY=0 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=00
# t=75 | PRESETn=1 PSEL=1 PEN=1 PWRITE=1 PADDR=00000000 PWDATA=0000000f | PRDATA=00000000 PREADY=0 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=00
# t=85 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=1 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=00
# t=95 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=1 rx_rst=1 | tx_data=00
# CTRL succeeded: tx_en=1 rx_en=1 tx_rst=1 rx_rst=1
# t=105 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=00
# t=115 | PRESETn=1 PSEL=1 PEN=0 PWRITE=1 PADDR=00000008 PWDATA=0000005a | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=00
# t=125 | PRESETn=1 PSEL=1 PEN=1 PWRITE=1 PADDR=00000008 PWDATA=0000005a | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=00
# t=135 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=00
# t=145 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# TX_DATA write succeeded: tx_data=5a
# t=165 | PRESETn=1 PSEL=1 PEN=0 PWRITE=0 PADDR=0000000c PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=175 | PRESETn=1 PSEL=1 PEN=1 PWRITE=0 PADDR=0000000c PWDATA=00000000 | PRDATA=00000000 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=185 | PRESETn=1 PSEL=1 PEN=1 PWRITE=0 PADDR=0000000c PWDATA=00000000 | PRDATA=000000a3 PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=195 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=000000a3 PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=205 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=000000a3 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# Read RX_DATA = 0xa3
# t=225 | PRESETn=1 PSEL=1 PEN=0 PWRITE=0 PADDR=00000004 PWDATA=00000000 | PRDATA=000000a3 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=235 | PRESETn=1 PSEL=1 PEN=1 PWRITE=0 PADDR=00000004 PWDATA=00000000 | PRDATA=000000a3 PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=245 | PRESETn=1 PSEL=1 PEN=1 PWRITE=0 PADDR=00000004 PWDATA=00000000 | PRDATA=0000000b PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=255 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=265 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# Read STATUS = 0x0000000b
# t=285 | PRESETn=1 PSEL=1 PEN=0 PWRITE=1 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=295 | PRESETn=1 PSEL=1 PEN=1 PWRITE=1 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=0 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=305 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=1 | CTRL: tx_en=1 rx_en=1 tx_rst=0 rx_rst=1 | tx_data=5a
# t=315 | PRESETn=1 PSEL=0 PEN=0 PWRITE=0 PADDR=00000000 PWDATA=00000000 | PRDATA=0000000b PREADY=0 | CTRL: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0 | tx_data=5a
# CTRL cleared: tx_en=0 rx_en=0 tx_rst=0 rx_rst=0
# >> APB regfile smoke test done
```

6.4 UART APB Top Module TB


Msgs
/tb_uart_apb_top/dut/PCLK 1'h0
/tb_uart_apb_top/dut/PRESETn 1'h1
/tb_uart_apb_top/dut/PSEL 1'h0
/tb_uart_apb_top/dut/PENABLE 1'h0
/tb_uart_apb_top/dut/PWRITE 1'h0
/tb_uart_apb_top/dut/PADDR 32'h00000008
/tb_uart_apb_top/dut/PWDATA 32'h0000005a
/tb_uart_apb_top/dut/PRDATA 32'h00000000
/tb_uart_apb_top/dut/PREADY 1'h0
/tb_uart_apb_top/dut/rx 1'h1
/tb_uart_apb_top/dut/tx 1'h1
/tb_uart_apb_top/dut/rx_en 1'h1
/tb_uart_apb_top/dut/tx_en 1'h1
/tb_uart_apb_top/dut/rx_rst 1'h0
/tb_uart_apb_top/dut/tx_rst 1'h0
/tb_uart_apb_top/dut/s_tick 1'h0
/tb_uart_apb_top/dut/tx_data 8'h5a
/tb_uart_apb_top/dut/rx_done 1'h0
/tb_uart_apb_top/dut/rx_error 1'h0
/tb_uart_apb_top/dut/rx_busy 1'h1
/tb_uart_apb_top/dut/tx_busy 1'h1
/tb_uart_apb_top/dut/tx_done 1'h1
/tb_uart_apb_top/dut/tx_start 1'h1
/tb_uart_apb_top/dut/rx_data 8'h5a
00000008
0000005a
00000000
5a
00
5a



```
# === UART APB Test Start ===
# [85000] READ  0x0 -> 0x0
# [125000] READ  0x4 -> 0x0
# [165000] WRITE 0x0 = 0x3
# [205000] WRITE 0x8 = 0x5a
# [2083445000] READ  0x4 -> 0x3
# [2083485000] READ  0xc -> 0x5a
# PASS: RX got expected 0x5A
# === UART APB Test End ===
```

7.0 CONCLUSION

This project successfully demonstrated the design and verification of a custom UART peripheral wrapped with an AMBA APB interface. Through a hierarchical and modular approach, the system was divided into key functional blocks: the baud generator, transmitter, receiver, APB interface, and register file. Each block was implemented with parameterization and clear FSM control, ensuring flexibility, reusability, and synthesizability for FPGA-based systems.

The verification process validated each module independently before integrating them into the top-level APB UART IP. Tasks and structured testbenches were used to confirm protocol compliance, correct data transmission and reception, error detection, and proper control/status reporting. End-to-end loopback tests confirmed that the design operates as a fully functional memory-mapped peripheral, capable of reliable serial communication while seamlessly interfacing with an SoC bus.

Overall, the project strengthened understanding of digital design, hardware/software interfacing, and verification methodology. Future improvements may include the addition of programmable parity, multi-stop bit support, FIFO buffering, and interrupt-driven operation, which would extend the design towards industrial-grade UART IP cores.