NTI

Custom APB UART IP Project

JULY 9, 2025

CAROL MICHEAL FAROUK

Table of Contents

1.0 INTRODUCTION

The evolution of System-on-Chip (SoC) design hinges on the principle of reusability and standardized communication. The Advanced Microcontroller Bus Architecture (AMBA), developed by ARM, has become the de facto standard for this purpose, providing a robust framework for integrating diverse intellectual property (IP) blocks into a cohesive system. Within the AMBA family, the Advanced Peripheral Bus (APB) is specifically designed for low-power, low-complexity interfacing with peripheral devices, making it ideal for control registers.

This project focuses on the design and implementation of a fundamental communication peripheral: a Universal Asynchronous Receiver/Transmitter (UART). The core objective is to create a custom UART IP block that is fully compliant with the AMBA APB protocol. This involves designing the UART's core transmitter (TX) and receiver (RX) logic, which handle serial data framing, baud rate generation, and error detection. This core functionality is then wrapped with an APB slave interface, transforming the UART into a memory-mapped peripheral that can be seamlessly integrated into an AMBA-based SoC.

Through this project, we explore the critical process of bridging custom hardware functionality with a standard bus protocol. Key learnings include the implementation of APB state machines for register read/write operations, the mapping of hardware status signals (like busy, done, and error) to software-accessible registers, and the synchronization between the APB clock domain and the internal UART timing. The successful completion of this project demonstrates a practical understanding of how to design, verify, and integrate a reusable peripheral IP, a fundamental skill in modern digital design and SoC engineering.

2.0 DESIGN ANALYSIS

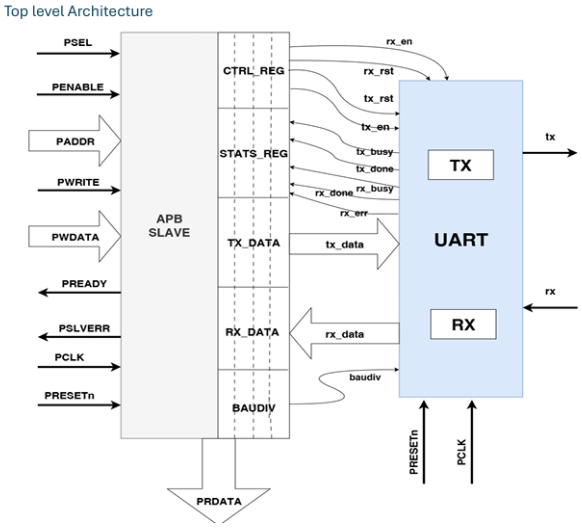
The designed APB UART IP is a modular system comprising three primary components: the **APB Slave Interface**, the **UART Receiver (RX)**, and the **UART Transmitter (TX)**. The system operates by converting parallel data written from the APB bus into a serial output stream (TX) and converting a serial input stream into parallel data readable from the APB bus (RX). All control and status monitoring is performed through a memory-mapped register interface.

2.1 Top-Level Architecture

The top-level module, apb_uart, instantiates and interconnects the three sub-modules. Its primary function is to route signals between the standard APB interface and the internal UART core.

- **APB Interface:** The standard APB signals (PCLK, PRESETn, PADDR, PSEL, PENABLE, PWRITE, PWDATA) are connected directly to the APB Slave Interface module.
- **Internal Bus:** The APB Slave Interface decodes transactions and generates an internal register select signal, write enable, and read data based on the address.

- **Register Access:** Write operations from the APB bus are used to populate the Control Register (CTRL_REG) and the Transmit Data Register (TX_DATA). Read operations allow the processor to read the Status Register (STATS_REG) and the Receive Data Register (RX_DATA).
- **Clock and Reset:** The entire system is synchronized to the APB clock (PCLK) and reset (PRESETn).

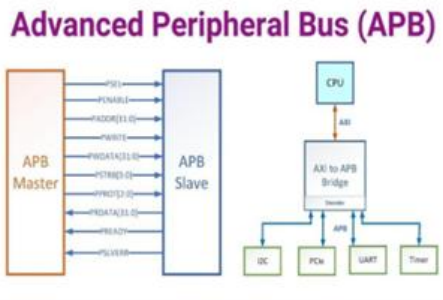


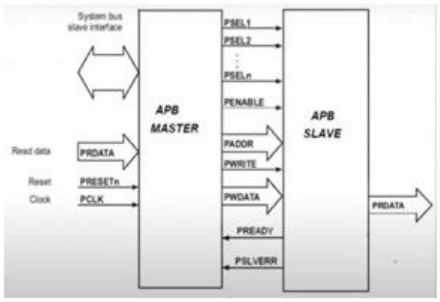
2.2 APB Slave Interface Module

This module is a finite state machine (FSM) that complies with the AMBA APB protocol specification. Its states are IDLE, SETUP, and ACCESS. The FSM transitions to SETUP when PSEL is asserted and to ACCESS when PENABLE is asserted in the SETUP state. It handles read and write cycles by:

- **Write Cycles:** Latching data from PWDATA into the target internal register when a valid write transaction occurs (PSEL, PENABLE, PWRITE are high).
- **Read Cycles:** Multiplexing the data from the addressed internal register onto the PRDATA bus during a valid read transaction.

- **Acknowledgment:** Asserting PREADY during the ACCESS state to signal the completion of the transfer, ensuring the protocol's two-cycle minimum requirement is met.





2.3 UART Receiver (RX) Module

The receiver is designed to detect a start bit, sample the incoming serial data (RX) at the correct baud rate, and assemble a data byte. Its operation, as outlined in the project specification, is based on a suggested architecture consisting of several key components:

- **Edge Detector:** Synchronizes the external RX signal to PCLK and detects a high-to-low transition, indicating a valid start bit and initiating the reception process.
- **Baud Rate Counter:** A down-counter loaded with a value that corresponds to the system clock cycles per bit period. It generates a sample_enable pulse when it reaches zero, signaling the middle of a bit period.
- **SIPO Shift Register:** An 8-bit shift register that is enabled by the RX FSM. On each sample_enable pulse, it shifts in the current value of the RX line, starting with the LSB.
- **RX Finite State Machine (FSM):** This is the controller for the receiver. Its state diagram, inspired by the project guidelines, manages the entire reception sequence:

The FSM controls the baud counter's load value and enable, enables the SIPO register, counts the number of bits received, and checks for a valid stop bit (logic high). A framing error (rx_error) is asserted if the stop bit is low.

2.4 UART Transmitter (TX) Module

The transmitter loads parallel data, appends start and stop bits, and serially shifts the frame out on the TX line at the configured baud rate. Its operation, as outlined in the project specification, is based on a suggested architecture consisting of several key components:

- **Baud Rate Counter:** Functions identically to the RX counter, generating a bit_period_pulse to define the duration of each bit on the TX line.

- **Bit Counter / FSM:** A state machine that controls the transmission process. It loads the parallel data into a shift register, asserts the TX line low for the start bit, shifts out the 8 data bits, and then asserts the line high for the stop bit(s). The tx_busy signal is asserted during this entire process, and tx_done is pulsed upon completion.

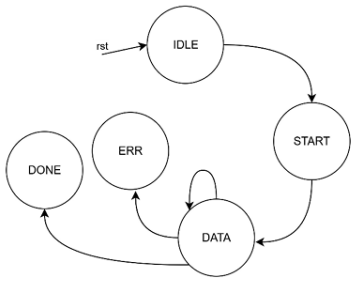
2.5 Register Map

The IP provides a 32-bit register map for processor control and status, as defined in the project specification

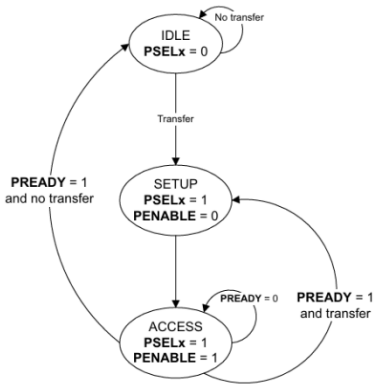
Address	Name	Description
0x0000	CTRL_REG	Contains bits for tx_en , rx_en , tx_rst & rx_rst
0x0001	STATS_REG	Contains bits for rx_busy , tx_busy , rx_done, tx_done & rx_error
0x0002	TX_DATA	UART Tx data
0x0003	RX_DATA	UART Rx data
0x0004	BAUDIV	Variable baud rate for UART [BONUS]

3.0 STATE DIAGRAMS

3.1 Rx/Tx FSM



3.2 APB FSM



4.0 DESIGN DECISIONS

4.1 Baud Generator

Module Role:
The baud_generator module provides a periodic tick signal used to pace the UART transmitter and receiver. This tick defines the sampling and shifting rate for serial communication.

Key Design Decisions:

- **Parameterized Divider (DVSR):**
  The divisor value is parameterized (DVSR = 651) so the baud rate can be easily adapted without changing the RTL code. This value corresponds to dividing the system clock down to achieve the desired baud frequency (e.g., 9600 baud with a 100 MHz clock).
- **Synchronous Counter with Asynchronous Reset:**
  The counter register (r_reg) increments on every clock cycle and resets either via the global async reset (arst_n) or when the terminal count is reached. Using asynchronous reset ensures proper initialization at power-up.
- **Single Output Pulse (tick):**
  The tick signal is asserted for one cycle when the divisor count is reached. This avoids multi-cycle pulses and ensures precise synchronization of TX/RX sampling.
- **16-bit Counter Width:**
  A 16-bit counter was chosen to support a wide range of divisor values without excessive area overhead.
- **Minimal Resource Utilization:**
  The implementation uses only one register for counting and one comparator, keeping the design lightweight for FPGA/ASIC synthesis.

Considered Trade-offs:

- A fractional-N divider could reduce baud error but would add complexity. The simpler integer divider was chosen for clarity and ease of implementation in a course project.
- Using synchronous reset instead of async reset would simplify CDC analysis but async was used to follow APB reset style.

4.2 Receiver

Module Role:
The receiver4 module implements the UART receive logic. It detects the start bit, samples incoming serial data, assembles it into a byte, checks the stop bit, and flags errors.

Key Design Decisions:

- **Finite State Machine (FSM):**
  A 6-state FSM (IDLE, START, DATA, STOP, DONE, ERROR) manages reception. This makes the design explicit, easy to debug, and expandable.
- **Oversampling at 16×:**
  The receiver samples the incoming signal every tick, with SB_TICK = 16 cycles per bit. Mid-bit sampling is performed in the START state to reduce susceptibility to noise and jitter.
- **Input Synchronization:**
  The asynchronous serial input rx is passed through a two-stage synchronizer (rx_sync1, rx_sync2) to prevent metastability. This is critical since rx is not aligned to the system clock.
- **Configurable Data Bits:**
  Parameter DBIT = 8 allows adjusting the number of received data bits, though the default is 8 for standard UART.
- **Framing Error Detection:**
  If the stop bit is not high at the expected time, the FSM transitions to ERROR and asserts rx_error_tick. This provides robustness against corrupted frames.
- **Status Outputs:**

i. rx_done_tick: one-cycle pulse when a valid byte is received.

ii. rx_busy: asserted whenever the FSM is not idle.

iii. rx_error_tick: indicates a framing error.
These signals integrate with the APB status register.

- **Software Reset and Enable:**
The module supports both a global reset (PRESETn) and a software-controlled reset (rx_rst). Additionally, rx_en allows dynamic enabling/disabling of the receiver.

Control & Datapath:

i. A shift register (b_reg) collects received bits in LSB-first order.

ii. Counters track sampling ticks (s_reg) and bit positions (n_reg).

iii. FSM transitions are triggered by s_tick pulses.

4.3 Transmitter

Module Role:
The transmitter module serializes an 8-bit parallel data word into a UART frame consisting of a start bit, data bits, and stop bit(s). It produces the tx output signal while also generating status flags (tx_busy, tx_done_tick).

Key Design Decisions:

- **FSM-based Architecture:**
A 4-state FSM (IDLE, START, DATA, STOP) is used to control the frame transmission. This provides a clean separation of frame phases and simplifies verification.
- **Parameterization:**
  i. DBIT = 8: Default number of transmitted data bits (configurable).
  ii. SB_TICK = 16: Defines the number of ticks per bit for baud timing.
  iii. STOP_BITS = 1: Chosen to match common UART configuration but left parameterized for flexibility.
- **Oversampling Support:**
The transmitter relies on s_tick pulses from the baud generator. Each data bit is held for exactly SB_TICK ticks, ensuring consistent bit timing.
- **Shift Register Design:**
A shift register (b_reg) stores the parallel input word. Bits are shifted out least-significant-bit first, consistent with UART standards.

- **Soft and Hard Reset Handling:**
  Both an asynchronous global reset (arst_n) and a software reset (rst) are supported. This matches the APB control register scheme.

- **Status Flagging:**

  i. tx_busy: Asserted whenever the FSM is not in IDLE.

  ii. tx_done_tick: Pulses for one cycle at the end of transmission (in STOP state).
  These outputs integrate cleanly with the status register for software monitoring.

- **Output Idle Level:**
  The line tx is held at logic '1' (mark level) whenever the transmitter is idle, complying with the UART standard.

4.4 Register File / APB Interface

Module Role:
The register_file module implements the memory-mapped APB register interface for controlling and monitoring the UART. It bridges software (CPU) and hardware (TX/RX/baud).

Key Design Decisions:

- **Memory-Mapped Register Map:**
  Five registers were defined:

  i. CTRL_REG: Control bits (enable/disable, resets).
  ii. STATS_REG: Status flags (busy, done, error).
  iii. TX_DATA: Data to transmit.
  iv. RX_DATA: Received data.
  v. BAUDIV: Baud rate divisor.
  This simple mapping follows APB style (word-aligned, address decode on [4:2]).

- **Separation of Control and Status:**

  i. Control signals (tx_en, rx_en, tx_rst, rx_rst) are directly derived from CTRL_REG.
  ii. Status signals (rx_busy, tx_busy, rx_done, tx_done, rx_error) update sticky bits in STATS_REG so software can poll them reliably.

- **Sticky Status Flags:**
  Done/error bits remain set until software explicitly clears them by writing to the register. This prevents missed events when the CPU is slow to poll.

- **TX Start Pulse Generation:**
  A one-cycle tx_start pulse is auto-generated whenever software writes to TX_DATA while tx_en is asserted. This avoids the need for a separate trigger register.
- **Safe Data Capture:**
  rx_data is stored into a dedicated register when rx_done is asserted, ensuring byte integrity even if software reads later.
- **Baud Rate Flexibility:**
  baudiv_reg holds the divisor and drives the baud generator. A default value (651) is loaded on reset, giving ~9600 baud with 100 MHz clock.
- **Auto-Clear Reset Bits:**
  The TX_RST and RX_RST bits in CTRL_REG automatically clear after being applied, so software doesn't need to toggle them back.

4.5 APB Interface

Module Role:
The APB module implements the standard AMBA APB (Advanced Peripheral Bus) slave interface for the UART. It translates APB protocol transactions into register file read/write requests.

Key Design Decisions:

- **APB Protocol Compliance:**
  The design follows the 3-phase APB protocol (IDLE → SETUP → ACCESS).

  i. **IDLE:** Waits for PSEL assertion.
  ii. **SETUP:** Latches address and asserts read enable if applicable.
  iii. **ACCESS:** Performs the read or write and asserts PREADY.

- **Zero-Wait-State Implementation:**
  The interface asserts PREADY in the first ACCESS cycle, ensuring every APB transaction completes in a single cycle. This simplifies timing closure and software driver design.
- **FSM-based Control:**
  A simple 3-state FSM (IDLE, SETUP, ACCESS) manages APB handshaking. This ensures protocol correctness while keeping the logic lightweight.
- **Read Priming in SETUP:**
  For read operations, reg_rd_en is asserted in the SETUP state. This ensures that reg_rd_data is valid by the time the bus reaches ACCESS, meeting the APB requirement for synchronous reads.
- **Address Latching:**
  addr_reg captures PADDR in SETUP and is reused in ACCESS for both reads and writes. This avoids metastability and ensures consistent addressing.
- **Clean Signal Separation:**

  i. reg_wr_en, reg_wr_addr, reg_wr_data: drive register file write.

ii. reg_rd_en, reg_rd_addr: drive register file read.
iii. PRDATA: captures reg_rd_data in the APB domain.
This separation keeps the design modular and reusable.

- **Error Handling:**
  PSLVERR is tied low (0). Error signaling was not required for the project, reducing design complexity.
- **Parameterization:**
  Both ADDR_WIDTH and DATA_WIDTH are parameterized for reuse in other designs, even though 32-bit APB width is used here.

4.6 Top-Level Integration (uart_apb)

Module Role:
The uart_apb module is the **system integration layer** that connects the APB bus interface, register file, UART core blocks (baud generator, transmitter, receiver), and exposes them as a memory-mapped peripheral.

Key Design Decisions:

- **Hierarchical Integration:**
  The UART system is split into modular RTL blocks (APB slave, register file, baud generator, TX, RX). This promotes **reusability** (e.g., the APB interface can be reused in other peripherals) and simplifies verification.

- **Standardized APB Slave Interface:**
  Instead of creating a custom control interface, the top-level exposes a fully AMBA-compliant APB interface. This ensures compatibility with SoC integration tools and allows easy CPU/software access.

- **Single Clock Domain:**
  All submodules share the same PCLK. This eliminates clock-domain crossing issues and ensures deterministic timing between APB accesses, baud generation, and serial logic.

- **Active-Low Global Reset (PRESETn):**
  A synchronous, active-low reset signal is distributed to all submodules. This aligns with APB standard conventions and simplifies SoC reset sequencing.

- **Shared Baud Tick Signal:**
  Both transmitter and receiver use the same s_tick generated by the baud_generator. This guarantees consistent timing and avoids mismatched baud rates, which could occur if separate generators were used.

- **Register File as Central Hub:**
  The register file mediates between APB transactions and UART control signals.

i. Control bits (TX enable, RX enable, resets, baud divisor) are written via APB.

ii. Status bits (TX busy, RX busy, RX error, RX done) are read back by software.

iii. This avoids directly exposing internal signals to APB, ensuring clean abstraction and modularity.

- **Decoupled TX and RX Paths:**
  TX and RX operate independently but are configured via the same register file.
- **Parameterization for Flexibility:**

  i. ADDR_WIDTH and DATA_WIDTH are parameterized for potential reuse in systems with different bus widths.

  ii. DVSR (baud divisor) is parameterized, allowing the UART to be retargeted for different baud rates or system clocks.
- **Error Handling Strategy:**
  Receiver includes rx_error signaling (framing/parity errors). At top-level, these are fed into the register file, allowing software to detect and clear errors. APB PSLVERR is tied low to indicate no bus-level errors (protocol simplification).

5.0 VERIFICATION DECISIONS

5.1 Receiver Verification (receiver4_tb)

- **Stimulus Generation:**
  Serial frames were driven into rx line bit-by-bit, synchronized to s_tick. Both valid frames and framing error cases (send_byte_bad_stop) were tested.
- **Scoreboard Check:**
  Compared dout with expected byte (exp_data) whenever rx_done_tick asserted.
- **Error Detection:**
  Verified rx_error_tick pulses when a bad stop bit is received.
- **Reset Behavior:**
  Included a pulse reset task (pulse_rx_rst) to check DUT recovery mid-operation.

5.2 Transmitter Verification (Transmitter_tb)

- **Stimulus Generation:**
  Data was loaded via din and tx_start, simulating APB-driven writes.
- **Frame Monitoring:**
  check_tx_frame task sampled TX line at baud intervals to validate:

Start bit (0), Data bits (LSB-first), Stop bit (1).

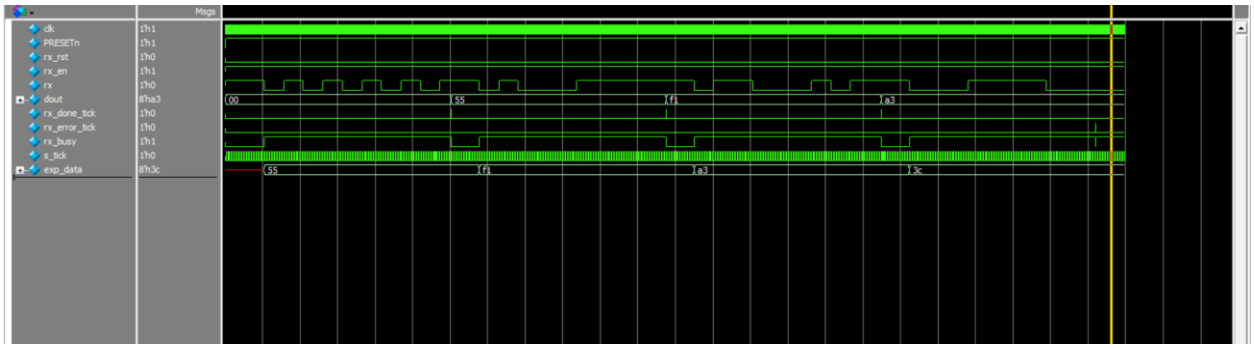
- **Busy/Done Flags:**
  Verified tx_busy asserts during transmission and tx_done_tick pulses at the end.

5.3 APB_UART Verification

- **Register Read/Write Verification:**
  Control, status, TX, RX, and baud registers were accessed through APB tasks. Each write was followed by a read-back to ensure correct register-mapped behavior.
- **Transmit/Receive Functional Test:**
  A known pattern (0xA5) was written to TX, transmitted on the serial line, and received back through RX.
- **Status Register Checks:**
  After transmission, the STATS_REG was read to confirm DUT flag updates.
- **Frame Timing Consideration:**
  Frame duration was calculated (FRAME_CLKS = DVSR × SB_TICK × (start+data+stop bits)) to ensure enough simulation cycles were run before checking RX.

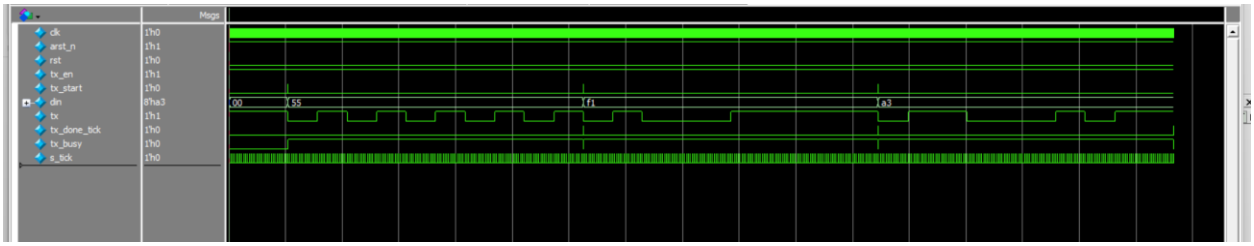
6.0 SIMULATION RESULTS

6.1 Receiver Simulation



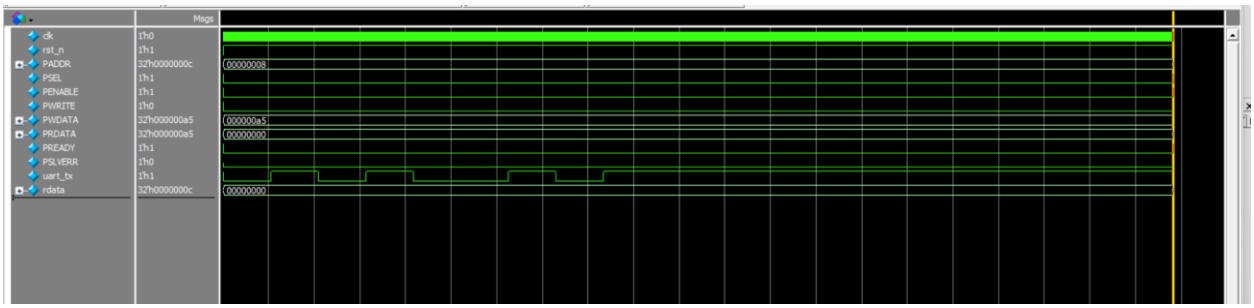

```
# PASS: Expected 0x55, got 0x55
# [1354275] Sent byte 0x55
# PASS: Expected 0xf1, got 0xf1
# [2500035] Sent byte 0xf1
# PASS: Expected 0xa3, got 0xa3
# [3645795] Sent byte 0xa3
# ERROR detected at time 4641845
# [4791555] Sent BAD-STOP byte 0x3c
# Testbench completed
```

6.2 Transmitter Simulation




```
PASS: Sent 0x55, observed 0x55 at time 1152465
PASS: Sent 0xf1, observed 0xf1 at time 2194065
PASS: Sent 0xa3, observed 0xa3 at time 3235665
TX Testbench completed
```

6.3 APB_UART Simulation




```
# [195000] READ  0x4 -> 0x0
# [245000] WRITE 0x0 = 0x3
# [295000] WRITE 0x8 = 0xa5
# [2083545000] READ  0x4 -> 0xc
# [2083595000] READ  0xc -> 0xa5
# PASS: RX got expected 0xA5
```

7.0 CONCLUSION

This project successfully achieved its primary objective: the design and implementation of a fully functional Universal Asynchronous Receiver/Transmitter (UART) intellectual property (IP) core, wrapped with a compliant AMBA Advanced Peripheral Bus (APB) interface. The resulting IP is a reusable, memory-mapped peripheral ready for integration into a System-on-Chip (SoC) environment.

The design process involved the meticulous development of three core components: the APB Slave Interface, the UART Receiver (RX), and the UART Transmitter (TX). The APB interface was implemented as a finite state machine (FSM) that rigorously adheres to the AMBA protocol, correctly handling all read and write transactions, generating the PREADY signal, and managing the two-cycle access requirement. The UART core itself was built to reliably perform serial communication, employing edge detection, precise baud rate counting, and shift registers to frame data with start and stop bits. Key status signals such as busy, done, and error were implemented and made accessible to a system processor through the defined register map.

Throughout the project, several valuable insights were gained. First, the importance of standardized bus protocols like APB was underscored; it provides a clear, consistent method for integrating diverse IP blocks, drastically reducing design complexity. Second, the challenge of synchronizing asynchronous signals (like the external RX line) to the system clock was addressed, highlighting a critical consideration for real-world digital design. Finally, the project served as an excellent practical exercise in modular design, state machine design, and the creation of a testbench strategy to verify both individual modules and the system as a whole.

In conclusion, this custom APB UART IP project served as a comprehensive exercise in digital systems design. It effectively bridged the gap between theoretical concepts of bus protocols and serial communication and their practical hardware implementation. The skills developed in creating a spec-compliant, modular, and verified IP core are fundamental to the field of SoC and FPGA design.