AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

International Credit Hour Programs

(ICHEP)

NTI DIGTITAL DESIGN TRAINING

Course Project

EzzEldin Ahmed EzzEldin Hafez Hassan

22P0050 junior COMM

Submitted to:

DR. Ghazal A. Fahmy

Eng. Mohamed Salah

2.0 INTRODUCTION

Universal Asynchronous Receiver and Transmitter (UART) is one of the most widely used serial communication protocols in embedded systems and digital electronics. Unlike synchronous protocols such as SPI or I^2C, UART does not require a dedicated clock line, which makes it a simple and cost-effective method for point-to-point data transmission. Communication is achieved using a defined frame structure consisting of a start bit, data bits, optional parity, and stop bits. Due to its simplicity, UART is commonly employed in microcontrollers, FPGAs, and System-on-Chip (SoC) designs for console communication, debugging, and peripheral interfacing.

In modern SoC architectures, peripherals like UART are accessed through standardized on-chip bus protocols. The Advanced Peripheral Bus (APB), defined under the ARM AMBA specification, is a widely adopted low-power, low-complexity interface that allows processors to communicate with peripheral devices through memory-mapped registers. By combining a UART core with an APB interface, the design can be seamlessly integrated into larger digital systems, enabling register-based control, configuration, and monitoring of data transmission and reception.

This project focuses on the design and verification of a UART subsystem in Verilog HDL, including transmitter and receiver modules, and their integration within an APB wrapper. The transmitter and receiver modules handle the serial communication at a fixed baud rate of 9600 bps, while the APB wrapper exposes memory-mapped registers for configuration, status monitoring, and data transfer. Comprehensive testbenches were developed to verify the functionality of each submodule as well as the integrated APB-based design.

The objective of this work is to provide a synthesizable and reusable UART peripheral that can be incorporated into FPGA or SoC-based designs, while ensuring correctness, modularity, and ease of verification.

3.0 Design Analysis:

3.1 Overview and Architecture

The presented design implements a complete UART (Universal Asynchronous Receiver/Transmitter) system with an APB (Advanced Peripheral Bus) wrapper interface. The system consists of three main components:

1. **UART Transmitter (UART_TX)**: Handles parallel-to-serial conversion
2. **UART Receiver (UART_RX)**: Handles serial-to-parallel conversion
3. **APB Wrapper**: Provides a standardized bus interface for system integration

The architecture follows a modular approach with clear separation of concerns, making it suitable for System-on-Chip (SoC) integration.

3.2 UART Transmitter (UART_TX) Analysis

3.2.1 Key Features

- Configurable baud rate, clock frequency, and data bits
- Finite State Machine (FSM) controlled operation
- Standard UART frame format: Start bit + Data bits + Stop bit
- Status signals (tx_busy, tx_done) for handshaking

3.2.2 Design Strengths

- **Parameterization**: Highly configurable through parameters
- **Clean FSM Design**: Clear states (IDLE, START, DATA, STOP)
- **Proper Reset Handling**: Asynchronous reset with idle state restoration
- **Baud Rate Calculation**: Automatic cycle calculation based on clock frequency

3.2.3 Potential Improvements

- **Error Handling**: Missing parity bit support
- **FIFO Interface**: No buffer for storing multiple bytes
- **Clock Domain Crossing**: No explicit handling for asynchronous interfaces

3.3 UART Receiver (UART_RX) Analysis

3.3.1 Key Features

- Configurable parameters matching the transmitter
- id-bit sampling for robust data capture
- Frame error detection
- FSM-based control similar to transmitter

3.3.2 Design Strengths

- **Robust Sampling**: Mid-bit sampling improves noise immunity
- **Error Detection**: Frame error flag for stop bit validation
- **Symmetric Design**: Similar structure to transmitter promotes consistency

3.3.3 Potential Improvements

- **Start Bit Validation**: Could include glitch rejection
- **Break Detection**: Missing break signal detection
- **Oversampling**: Single sample per bit may be susceptible to noise

3.3.4 APB Wrapper Analysis

3.4 Bus Interface Implementation

- **Register Mapping**:
    - CTRL_REG: Control signals (TX enable, RX enable, resets)
    - STATS_REG: Status signals (busy flags, done flags, errors)
    - TX_DATA_REG: Transmit data register
    - RX_DATA_REG: Receive data register
    - BAUDIV_REG: Baud rate divisor (currently not utilized)

Design Strengths

- **Standard Compliance**: Follows APB protocol specifications
- **Memory-mapped Interface**: Simple register-based control
- **Status Monitoring**: Comprehensive status register for system monitoring

Key Issues Identified

1. **Baud Rate Configuration**: The BAUDIV_REG is implemented but not connected to the UART cores
2. **Module Naming Inconsistency**: APB wrapper references "uart_transmitter" and "uart_receiver" but actual modules are "UART_TX" and "UART_RX"
3. **Signal Interface Mismatch**: RX module missing expected enable and reset signals
4. **Clock Domain Considerations**: No synchronization for cross-domain signals

3.5 Testbench Analysis

3.5.1 Transmitter Testbench

- Tests basic functionality with three different data bytes
- Uses timing-based verification (waiting for frame completion)
- **Limitation**: Not self-checking against expected serial output

3.5.2 Receiver Testbench

- Generates UART waveforms for testing
- Tests with multiple data patterns
- **Limitation**: No automatic verification of received data

3.5.3 APB Wrapper Testbench

- Demonstrates basic write/read operations
- Includes UART simulation for receive testing
- **Limitation**: Limited protocol testing and error case coverage

3.6 Performance Considerations

3.6.1 Timing Analysis

- Baud rate accuracy depends on clock frequency divisibility
- Maximum data throughput: 1 frame per (1 start + N data + 1 stop) bits
- Processor interface latency: 2-3 clock cycles for APB transactions

3.6.2 Resource Utilization

- UART cores: Moderate logic utilization (FSM, counters, shift registers)
- APB wrapper: Minimal additional logic (register file, address decoding)

- No memory elements beyond configuration registers

3.7 Reliability Considerations

3.7.1 Error Handling

- Frame error detection in receiver
- No parity error detection
- No overflow detection in receiver

3.7.2 Reset Behavior

- Complete reset of all state machines and registers
- TX line set to idle state (high) during reset

3.7.3 Clock Domain Considerations

- All components currently synchronous to PCLK
- No special handling for asynchronous serial signals

3.8 Compatibility and Integration

3.8.1 Standards Compliance

- UART: Implements standard asynchronous serial protocol
- APB: Follows AMBA APB protocol specifications
- Parameterization: Supports various standard baud rates

3.8.2 Integration Considerations

- Simple memory-mapped interface for processor integration
- Clear status signals for interrupt-driven operation
- Configurable parameters for system customization

3.9 Conclusion

The UART system with APB wrapper presents a solid foundation for serial communication in SoC designs. The modular architecture, parameterized design, and standard bus interface make it suitable for integration into larger systems.

Key Strengths:

1. Clean, modular design with separation of concerns

2. Parameterized implementation for flexibility
3. Standard APB interface for easy integration
4. Comprehensive functionality for basic UART communication

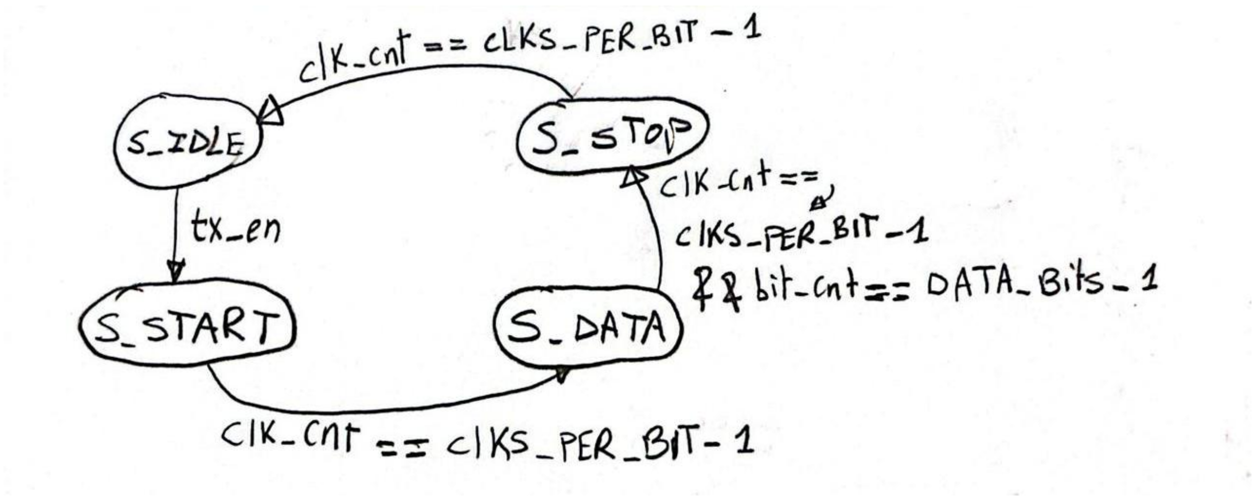
Areas for Enhancement:

1. Baud rate configuration through APB interface
2. Additional error detection mechanisms (parity, overflow)
3. Advanced features (FIFO buffers, interrupt control)
4. More comprehensive verification environment
5. Clock domain crossing for asynchronous serial signals

This design analysis provides a foundation for further development, validation, and integration of the UART system into larger projects.

4.0 State Diagrams

4.1 UART transmitter state diagram


clk_cnt == CLKS_PER_BIT - 1
S_IDLE
S_STOP
tx_en
clk_cnt ==
clks_PER_BIT - 1
&& bit_cnt == DATA_Bits - 1
S_START
S_DATA
clk_cnt == clks_PER_BIT - 1

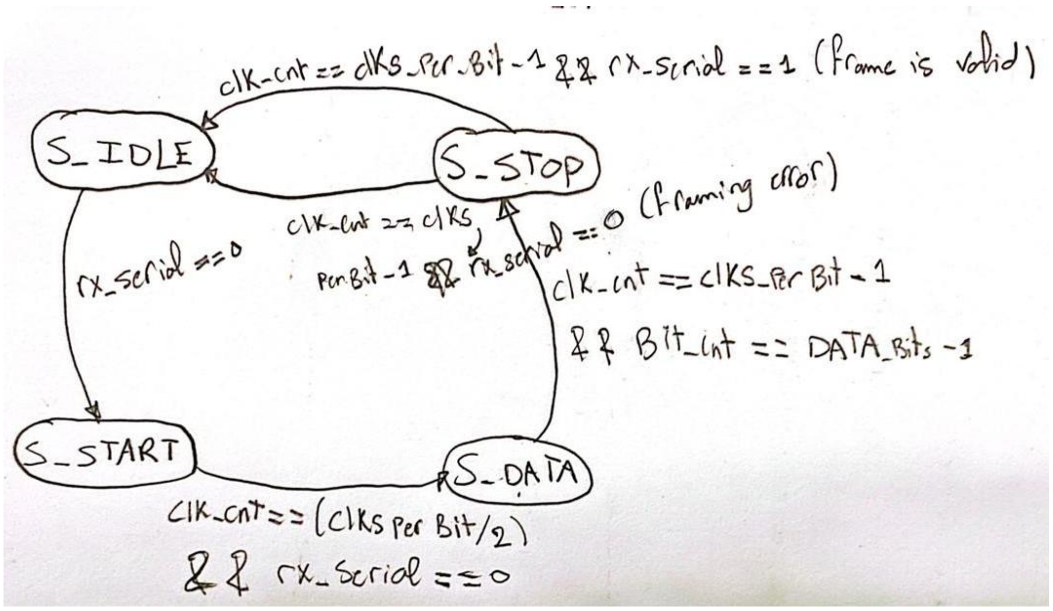
Explanation of State Transitions:

1. **S_IDLE → S_START (tx_en)**
   - The transmitter is idle, waiting for a request to send data.

- When tx_en is asserted, the FSM moves to S_START to begin transmission (sending the start bit = 0).

2. **S_START → S_DATA (clk_cnt == CLKS_PER_BIT - 1)**
   - The start bit is held on the line for one full bit period.
   - When the clock counter reaches CLKS_PER_BIT - 1, meaning one bit time has elapsed, the FSM transitions to S_DATA to begin sending the data bits.
3. **S_DATA → S_DATA (self-loop: clk_cnt == CLKS_PER_BIT - 1 && bit_cnt < DATA_BITS - 1)**
   - The FSM stays in the S_DATA state while sending each data bit.
   - Every time one bit period finishes (clk_cnt == CLKS_PER_BIT - 1), the bit counter increments.
   - If there are still more data bits left (bit_cnt < DATA_BITS - 1), it loops within S_DATA to send the next bit.
4. **S_DATA → S_STOP (clk_cnt == CLKS_PER_BIT - 1 && bit_cnt == DATA_BITS - 1)**
   - After the last data bit is transmitted (bit_cnt == DATA_BITS - 1), the FSM transitions to S_STOP.
   - This state outputs the stop bit (1) for one full bit period.
5. **S_STOP → S_IDLE (clk_cnt == CLKS_PER_BIT - 1)**
   - Once the stop bit duration is complete, the FSM returns to S_IDLE.
   - The transmitter is now ready for the next transmission.

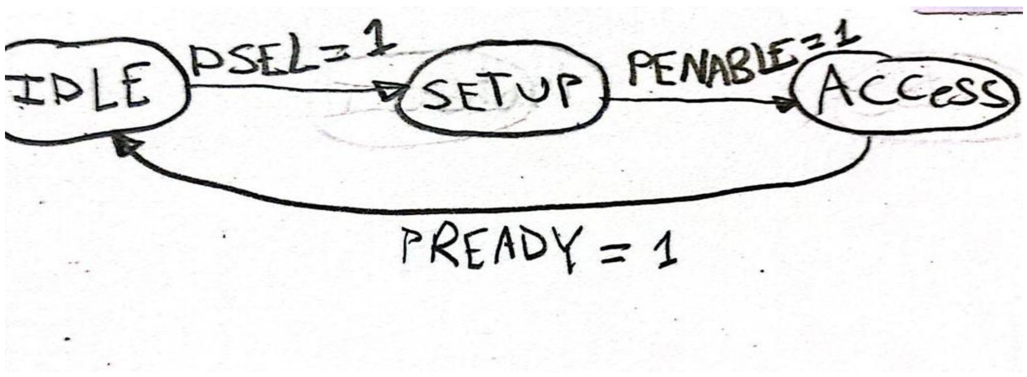
4.2 UART receiver state diagram


clk_cnt == clks_per_Bit - 1 && rx_serial == 1 (frame is valid)
S_IDLE
S_STOP
clk_cnt == clks_Per Bit - 1 && rx_serial == 0 (framing error)
rx_serial == 0
clk_cnt == clks_Per Bit - 1
&& Bit_cnt == DATA_Bits - 1
S_START
S_DATA
clk_cnt == (clks Per Bit/2)
&& rx_serial == 0

1. **S_IDLE → S_START (rx_serial == 0)**
   - Receiver waits idle for the line to go low.
   - A low signal indicates a possible **start bit**.
2. **S_START → S_DATA (clk_cnt == CLKS_PER_BIT/2 && rx_serial == 0)**
   - After half a bit time, the line is still low → confirms a **valid start bit**.
   - FSM transitions to S_DATA to start sampling incoming data bits.
3. **S_START → S_IDLE (clk_cnt == CLKS_PER_BIT/2 && rx_serial == 1)**
   - If the line goes back high at the middle of the start bit, it was **false noise**.
   - FSM returns to idle state without capturing data.
4. **S_DATA → S_DATA (self-loop: clk_cnt == CLKS_PER_BIT-1 && bit_cnt < DATA_BITS-1)**
   - A complete bit period has passed → one data bit is sampled.
   - If more bits remain, increment bit_cnt and stay in S_DATA.
5. **S_DATA → S_STOP (clk_cnt == CLKS_PER_BIT-1 && bit_cnt == DATA_BITS-1)**
   - The **last data bit** has been sampled.
   - FSM moves to S_STOP to check the stop bit.

6. **S_STOP → S_IDLE (valid stop bit: clk_cnt == CLKS_PER_BIT-1 && rx_serial == 1)**
    - At the end of the stop bit time, if the line is high → frame is valid.
    - FSM goes back to idle, ready for the next frame.
7. **S_STOP → S_IDLE (framing error: clk_cnt == CLKS_PER_BIT-1 && rx_serial == 0)**
    - If the line is low instead of high at the stop bit → **framing error**.
    - FSM still returns to idle, but signals error.

4.3 APB Wrapper state diagram



Explanation of APB FSM Transitions

1. **IDLE → SETUP (PSEL = 1)**
    - **In idle, no transfer is happening.**
    - **When the master asserts PSEL, it selects a slave → FSM moves to SETUP.**
2. **SETUP → ACCESS (PENABLE = 1)**
    - **During setup, address and control signals are stable.**
    - **When the master asserts PENABLE, the transfer becomes active → FSM goes to ACCESS.**
3. **ACCESS → IDLE (transfer complete)**
    - **Data is transferred (read or write).**
    - **After one cycle, FSM returns to IDLE, ready for the next transfer**

5.0 Design Decisions:

1. Top-Level Architecture Decision

- **Decision:** Implement a modular design with three distinct components: UART Transmitter (UART_TX), UART Receiver (UART_RX), and an APB Wrapper (APB_WRAPPER).
- **Rationale:**
    - **Separation of Concerns:** Each module has a single, well-defined responsibility (TX, RX, or Bus Interface). This makes the design easier to understand, debug, and maintain.
    - **Reusability:** The UART_TX and UART_RX cores are standalone and could be used in other projects without the APB wrapper.
    - **Scalability:** The wrapper-based architecture allows other peripherals to be added to the same APB bus easily.

2. UART Protocol Implementation Decisions

- **Decision:** Implement a classic, finite state machine (FSM) approach for both the transmitter and receiver.
- **Rationale:** FSMs are a clear, proven, and reliable method for controlling sequential protocols like UART. The states (IDLE, START, DATA, STOP) directly map to the components of a UART frame.
- **Trade-off:** While efficient and simple, an FSM may have higher latency than a more pipelined approach (though this is negligible for UART speeds).
- **Decision:** Use a counter-based Baud Rate Generator instead of a dedicated Phase-Locked Loop (PLL) or fractional divider.
- **Rationale:**
    - **Simplicity:** A counter is easy to implement and understand.
    - **Sufficient Accuracy:** For many applications, the error introduced by integer division (CLK_FREQ / BAUD_RATE) is acceptable.

- **Trade-off:** This method introduces a small baud rate error if the system clock frequency is not an integer multiple of the desired baud rate.
- **Decision:** Omit parity bit generation and checking.
- **Rationale:** To keep the initial design simple and focused on the core functionality.
- **Trade-off:** Reduces reliability for detecting single-bit errors during transmission. This could be added as a future enhancement.

3. Receiver-Specific Decisions

- **Decision:** Sample the start bit at its midpoint (CLKS_PER_BIT/2) for validation.
- **Rationale:** This is a critical best practice. Sampling in the middle of the bit period, away from the edges where the signal is transitioning, provides robustness against noise and meta-stability, ensuring the start bit is genuine and not a glitch.
- **Decision:** Sample data bits at the end of their period (CLKS_PER_BIT-1).
- **Rationale:** This ensures the signal is stable before being sampled. The midpoint sampling is reserved for the critical start bit detection.
- **Decision:** Include a frame_error output flag.
- **Rationale:** Provides essential feedback to the system software if the stop bit is not detected (HIGH), indicating a framing error and potentially corrupt data.

4. APB Bus Interface Decisions

- **Decision:** Implement a standard AMBA APB protocol interface.
- **Rationale:** APB is a low-power, low-complexity bus common in ARM-based SoCs. Using this standard ensures compatibility with industry-standard processors and bus architectures.
- **Decision:** Define a specific memory-mapped register map.
    - CTRL_REG (Write): Control bits for TX enable, RX enable, and soft resets.
    - STATS_REG (Read): Status bits (TX busy, TX done, RX busy, RX ready, frame error).
    - TX_DATA_REG (Write): Data to be transmitted.
    - RX_DATA_REG (Read): Data that has been received.
    - BAUDIV_REG (Read/Write): Baud rate divisor (for future use).

- **Rationale:** This provides a clean, intuitive software interface for a programmer to control the peripheral.
- **Decision:** Implement a simple FSM (IDLE, SETUP, ACCESS) to handle the APB protocol handshake (PSEL, PENABLE, PREADY).
- **Rationale:** Correctly manages the bus timing and ensures the peripheral responds to the controller within the required clock cycles.

5. Configuration and Scalability Decisions

- **Decision:** Use SystemVerilog parameters for key characteristics (BAUD_RATE, CLK_FREQ, DATA_BITS).
- **Rationale:** Makes the design highly configurable and reusable across different projects without modifying the core RTL code. The same module can be instantiated for different baud rates or data formats.
- **Decision:** Use $clog2() to calculate the necessary counter widths based on the parameters.
- **Rationale:** This is a best practice for parameterized design. It automatically calculates the correct number of bits needed for counters, preventing wasteful bit usage or overflow errors.

6. Verification Decisions

- **Decision:** Create individual testbenches for each major module (UART_TX, UART_RX, APB_WRAPPER).
- **Rationale:** Unit testing allows for isolated verification of each component's functionality before integration, simplifying the debug process.
- **Trade-off:** The testbenches provided are basic and not self-checking. A more robust verification environment would use automated checks against expected results.

Summary of Key Trade-offs

Decision	Advantage	Trade-off
Counter-based Baud Gen	Simple, low area	Potential baud rate error

Decision	Advantage	Trade-off
No Parity	Simpler design	No single-bit error detection
Basic Testbenches	Quick to implement	Not self-verifying; requires manual inspection
Separate TX/RX Cores	Modular and reusable	Slightly more area than a fully combined core
APB FSM	Correct bus protocol	Adds a few cycles of latency for bus accesses

Recommended Future Enhancements

1. **FIFO Buffers:** Add FIFOs for TX and RX to allow back-to-back transmission and prevent data overrun.
2. **Interrupts:** Enhance the APB wrapper to generate interrupts on events like tx_done, rx_ready, and frame_error.
3. **Parity Support:** Add parameters and logic to support optional even or odd parity.
4. **Loopback Mode:** Add a diagnostic mode to internally connect TX to RX.
5. **Advanced Verification:** Develop a self-checking testbench with randomized tests and functional coverage.

6.0 Verification Strategy:

1. Verification Goals & Objectives

The primary goal is to ensure the UART system operates correctly according to its specification:

- **APB Interface:** Complies with AMBA APB protocol standards.
- **UART Protocol:** Correctly transmits and receives serial data frames (Start, Data, Stop) at the configured baud rate.
- **Register Access:** All memory-mapped registers can be read and written correctly.

- **Data Integrity:** Data written to the TX register is transmitted correctly and data received is stored correctly in the RX register.
- **Status Flags:** Flags (tx_busy, tx_done, rx_busy, rx_ready, frame_error) are asserted and deasserted at the correct times.
- **Error Handling:** Framing errors are correctly detected and flagged.

2. Testbench Architecture

A modular, layered testbench will be developed using SystemVerilog.

Key Components:

- **Testbench Top:** Instantiates the DUT, interfaces, drivers, monitors, and scoreboard.
- **APB Bus Functional Model (BFM):** A driver to generate APB read/write transactions.
- **UART Monitor:** A passive agent that monitors the tx_serial and rx_serial lines, reconstructing transactions.
- **Scoreboard:** Compares data sent via APB write with data received by the UART monitor, and vice versa. Checks for data consistency.
- **Test Sequences:** A library of directed and random tests.

3. Test Plan & Test Cases

A. APB Protocol Tests

1. **Register Read/Write Test:**
    - **Objective:** Verify all registers are accessible.
    - **Procedure:** Write a known pattern (e.g., 0xA5A5A5A5) to each writable register and read it back to verify. Attempt to read undefined addresses to check PSLVERR.
2. **APB Timing Tests:**
    - **Objective:** Verify PREADY and PSLVERR timing.
    - **Procedure:** Perform back-to-back reads and writes, varying the delay between PSEL and PENABLE.

B. UART Transmitter (TX) Tests

3. **Basic Transmission Test:**

- **Objective:** Verify a single byte is correctly formatted and transmitted.
- **Procedure:** Write a byte to TX_DATA_REG and enable TX via CTRL_REG. The UART monitor will check for: correct start bit (LOW), correct data bits (LSB first), correct stop bit (HIGH), and correct bit timing.

4. **Consecutive Transmission Test:**
   - **Objective:** Verify the FSM correctly handles back-to-back transmissions.
   - **Procedure:** Write a second byte to TX_DATA_REG immediately after tx_done is asserted. Check that no data is corrupted and the gap between frames is correct.
5. **Baud Rate Variation Test:**
   - **Objective:** Verify functionality across different BAUD_RATE parameterizations.
   - **Procedure:** Re-run tests with different BAUD_RATE and CLK_FREQ parameters.

C. UART Receiver (RX) Tests

6. **Basic Reception Test:**
   - **Objective:** Verify a valid frame is correctly received.
   - **Procedure:** The testbench will generate a valid UART frame on rx_serial. The scoreboard will check if the value in RX_DATA_REG matches the sent value and if rx_ready pulses.
7. **Framing Error Test:**
   - **Objective:** Verify the frame_error flag is asserted on an invalid stop bit.
   - **Procedure:** Generate a UART frame with a stop bit set to LOW (0). Check that frame_error is asserted and rx_ready is not.
8. **Start Bit Glitch Test:**
   - **Objective:** Verify the receiver is robust against noise.
   - **Procedure:** Generate a short glitch (LOW pulse) on rx_serial and ensure it does not trigger the receiver FSM. Then, generate a real start bit and ensure it is correctly detected at the midpoint.

D. System-Level Integration Tests

9. **Loopback Test:**
    - **Objective:** Verify full system integration.
    - **Procedure:** Externally connect tx_serial to rx_serial. Write data to the TX register via APB and read it back from the RX register after transmission is complete. The scoreboard compares the written and read values.
10. **Concurrent TX/RX Test:**
    - **Objective:** Stress test the system.
    - **Procedure:** Simultaneously transmit data via APB and receive data on the UART interface from the testbench. The scoreboard must correctly track both data streams.

4. Verification Methodology

- **Directed Testing:** Initial tests will be directed to verify basic functionality (e.g., Test Cases 1, 3, 6).
- **Constrained-Random Testing:** Later stages will use random stimuli to uncover corner-case bugs.
    - **Random Data:** Data written to TX register and injected on RX line will be randomized.
    - **Random Baud Rates:** Tests will be run with randomized legal baud rate parameters.
    - **Random Errors:** The testbench will randomly inject framing errors.
- **Functional Coverage:** Metrics will be defined to ensure all parts of the design have been exercised.
    - **Protocol Coverage:** All FSM states and transitions for TX, RX, and APB.
    - **Register Coverage:** All bits of all registers have been read and written.
    - **Data Coverage:** A range of data values has been transmitted and received (e.g., all zeros, all ones, alternating bits).
    - **Error Coverage:** Framing error case has been triggered.
    - **Timing Coverage:** Various bus timings and baud rates have been tested.

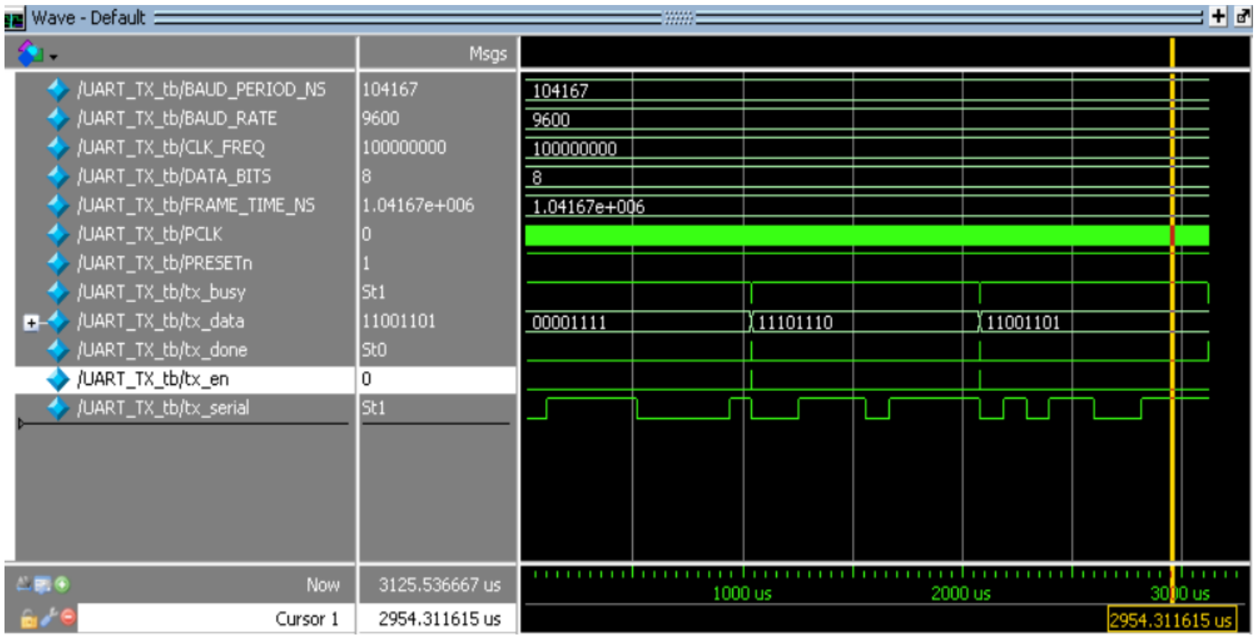
- **Assertions:** Key properties will be formally checked throughout simulation.
  - **APB Protocol Assertions:** e.g., PENABLE must not be asserted without PSEL.
  - **UART Protocol Assertions:** e.g., tx_serial must be high during IDLE and STOP states.
  - **Status Flag Assertions:** e.g., tx_done must pulse for only one clock cycle when a transmission finishes.

5. Execution Plan

1. **Unit Level Verification:** Verify UART_TX and UART_RX modules independently with their own testbenches.
2. **Module Integration:** Integrate TX and RX into APB_WRAPPER and verify the wrapper's logic and register interface.
3. **System Level Verification:** Verify the complete system with the full testbench architecture, including scoreboard and functional coverage.
4. **Regression:** Run the full test suite after any design change to prevent regression.

7.0 simulation results

7.1 simulation of UART transmitter


```
Wave - Default
                                 Msgs
/UART_TX_tb/BAUD_PERIOD_NS       104167
/UART_TX_tb/BAUD_RATE            9600
/UART_TX_tb/CLK_FREQ             100000000
/UART_TX_tb/DATA_BITS            8
/UART_TX_tb/FRAME_TIME_NS        1.04167e+006
/UART_TX_tb/PCLK                 0
/UART_TX_tb/PRESETn              1
/UART_TX_tb/tx_busy              St1
/UART_TX_tb/tx_data              11001101
/UART_TX_tb/tx_done              St0
/UART_TX_tb/tx_en                0
/UART_TX_tb/tx_serial            St1

Now        3125.536667 us
Cursor 1   2954.311615 us
```

Key Signals and Parameters:

1. **UART_TX_tb/BAUD_PERIOD_NS**: 104167 ns (approximately 9.6 kHz baud rate, derived from 1/104167 ns ≈ 9600 bps).
2. **UART_TX_tb/BAUD_RATE**: 9600 bps (bits per second), confirming the baud rate.
3. **UART_TX_tb/CLK_FREQ**: 100000000 Hz (100 MHz), the clock frequency driving the module.
4. **UART_TX_tb/DATA_BITS**: 8, indicating the number of data bits per frame.
5. **UART_TX_tb/FRAME_TIME_NS**: 1.04167e+006 ns (approximately 1.04 ms), the total time for one frame (start bit + 8 data bits + stop bit).
6. **UART_TX_tb/PCLK**: 0, likely the clock signal (not visible in the waveform but inferred from the context).
7. **UART_TX_tb/PRESETn**: 1, the active-low reset signal, which is not asserted (normal operation).
8. **UART_TX_tb/busy**: 1 (high) during transmission, indicating the transmitter is active.
9. **UART_TX_tb/data**:
   - Starts as 11001101 (initial data to transmit).
   - Changes to 00001111, 11101110, and 11001101 over time, showing multiple frames being transmitted.
10. **UART_TX_tb/done**: 0 (low), indicating the transmission is not yet complete.
11. **UART_TX_tb/en**: 1 (high), enabling the transmitter.
12. **UART_TX_tb/serial**: The output serial data line, showing the transmitted waveform.

Waveform Analysis:

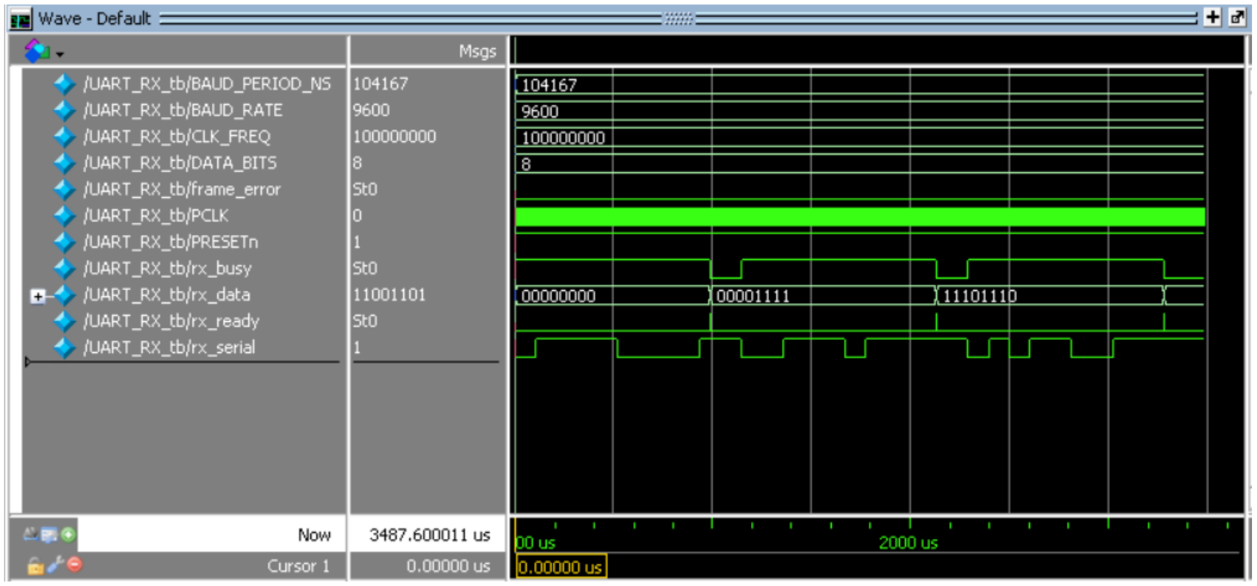
- **Time Scale**: The x-axis is in microseconds (μs), with a total span of about 2954.311615 μs to 3125.566667 μs.
- **Serial Output**: The UART_TX_tb/serial signal shows the serialized transmission of the data. Each frame includes:
  - A low **start bit** (0).

    - 8 **data bits** (e.g., 11001101 for the first frame).
    - A high **stop bit** (1).
- **Frame Timing**: Each bit duration is approximately 104.167 μs (derived from BAUD_PERIOD_NS), and a full frame (10 bits: 1 start + 8 data + 1 stop) takes about 1.04167 ms, aligning with FRAME_TIME_NS.
- **Multiple Frames**: The waveform shows multiple frames being transmitted sequentially:
    - First frame: 11001101 (with start and stop bits).
    - Second frame: 00001111.
    - Third frame: 11101110.
    - Fourth frame: 11001101 (repeating the initial data).

Operation Explanation:

- The simulation begins with the transmitter enabled (en = 1) and data loaded (data = 11001101).
- The busy signal goes high, indicating the transmitter is processing the data.
- The serial line transmits the data bit by bit at the specified baud rate (9600 bps).
- After each frame is sent, new data (00001111, 11101110, etc.) is loaded and transmitted, suggesting the transmitter is handling a queue or continuous data stream.
- The done signal remains low, indicating the transmitter is still active and has more data to send or is in a continuous transmission mode.

7.2 simulation of UART receiver



```
/UART_RX_tb/BAUD_PERIOD_NS   104167
/UART_RX_tb/BAUD_RATE        9600
/UART_RX_tb/CLK_FREQ         100000000
/UART_RX_tb/DATA_BITS        8
/UART_RX_tb/frame_error      St0
/UART_RX_tb/PCLK             0
/UART_RX_tb/PRESETn          1
/UART_RX_tb/rx_busy          St0
/UART_RX_tb/rx_data          11001101
/UART_RX_tb/rx_ready         St0
/UART_RX_tb/rx_serial        1

Now        3487.600011 us
Cursor 1   0.00000 us
```

rx_data: 00000000, 00001111, 11101110

Signals in the Simulation

- PCLK → System clock used to sample incoming serial data.
- PRESETn → Active-low reset; holds receiver in reset when 0.
- rx_serial → The UART input line (serial data coming in).
- rx_busy → High while the receiver is capturing a frame.
- rx_data → Parallel output data reconstructed from the serial stream.
- rx_done → Pulses HIGH for one clock cycle when a full byte has been received.
- rx_en → Enable/start signal for the receiver FSM.
- BAUD_PERIOD_NS → Defines the sampling interval (≈104,167 ns for 9600 baud at 100 MHz).

Explanation of the Waveform

1. Idle / Reset Phase

- With PRESETn = 0, the receiver is in reset.
- The input line rx_serial = 1 (UART idle state).
- rx_busy = 0, rx_done = 0, and rx_data holds its last value.

2. Reception of First Frame

- The transmitter drives a Start bit (0) → receiver detects falling edge.
- rx_busy goes HIGH → receiver is actively sampling bits.
- Data bits are sampled in the middle of each baud period.
  - The waveform shows bits shifting in (LSB first).
  - Example: if input serial stream is 0 11110000 1, the receiver reconstructs 00001111.
- After 8 bits, Stop bit (1) is detected.
- rx_data is updated with the received parallel byte.
- rx_done pulses HIGH for one cycle to signal valid data.
- rx_busy returns LOW after the stop bit.

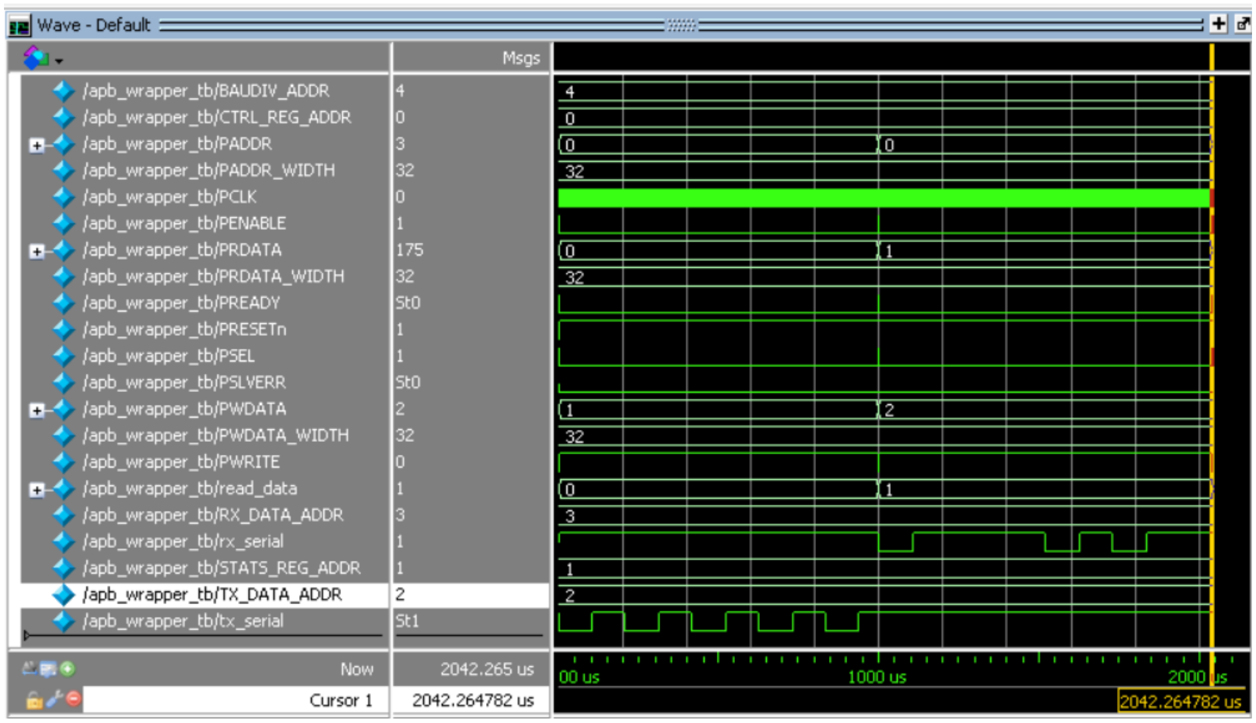
3. Reception of Second Frame

- Another falling edge (start bit) begins the next byte reception.
- Same process:
  - Start bit → sample 8 data bits (LSB first) → Stop bit.
- rx_data updates to the new byte.
- rx_done pulses HIGH again, indicating data ready.
- rx_busy drops to LOW after completion.

Observations

- UART line (rx_serial) is HIGH when idle.
- Receiver correctly identifies the Start bit → Data bits → Stop bit.
- Data is reconstructed LSB first, matching UART protocol.
- rx_busy indicates the FSM is capturing a frame.
- rx_done provides a clean handshake for the system to latch the parallel data.
- Timing matches 9600 baud (each bit ≈ 104 μs).

7.3 simulation of APB Wrapper



	Msgs
/apb_wrapper_tb/BAUDIV_ADDR	4
/apb_wrapper_tb/CTRL_REG_ADDR	0
/apb_wrapper_tb/PADDR	3
/apb_wrapper_tb/PADDR_WIDTH	32
/apb_wrapper_tb/PCLK	0
/apb_wrapper_tb/PENABLE	1
/apb_wrapper_tb/PRDATA	175
/apb_wrapper_tb/PRDATA_WIDTH	32
/apb_wrapper_tb/PREADY	St0
/apb_wrapper_tb/PRESETn	1
/apb_wrapper_tb/PSEL	1
/apb_wrapper_tb/PSLVERR	St0
/apb_wrapper_tb/PWDATA	2
/apb_wrapper_tb/PWDATA_WIDTH	32
/apb_wrapper_tb/PWRITE	0
/apb_wrapper_tb/read_data	1
/apb_wrapper_tb/RX_DATA_ADDR	3
/apb_wrapper_tb/rx_serial	1
/apb_wrapper_tb/STATS_REG_ADDR	1
/apb_wrapper_tb/TX_DATA_ADDR	2
/apb_wrapper_tb/tx_serial	St1
Now	2042.265 us
Cursor 1	2042.264782 us

Overview

- **The signals belong to an APB wrapper connected to a UART-like peripheral (since you have rx_serial, tx_serial, BAUDDIV_ADDR, etc.).**
- **The waveform shows an APB write/read transaction and the serial transmission (tx_serial) that happens as a result.**

2. Important Signals

- **PCLK: APB clock (main timing reference).**
- **PSEL: Peripheral select, asserted when this peripheral is chosen.**
- **PENABLE: Control signal to indicate second phase of APB transfer.**
- **PWRITE: 1 → write, 0 → read.**
- **PWDATA / PRDATA: Write data / read data bus (32-bit wide).**
- **PADDR: Address bus (32-bit).**
- **PREADY: Slave ready signal.**
- **TX_DATA_ADDR: Address register for transmit data.**

- **tx_serial: UART serial output line.**

3. What's Happening in the Waveform

1. **At the beginning**
   - **PSEL = 1, PENABLE = 1, and PWRITE = 1 → This is an APB write cycle.**
   - **PADDR = 2 (TX_DATA_ADDR) → So we are writing to the TX FIFO / TX register.**
   - **PWDATA = 175 (0xAF) → The byte being written.**
2. **During write**
   - **PREADY = 1 confirms the slave accepted the transaction.**
   - **The value 175 gets latched into the UART TX register.**
3. **Serial Transmission**
   - **After write, tx_serial starts toggling → this is the UART transmitting the byte 0xAF (1010_1111).**
   - **UART transmission includes:**
     - **Start bit (low).**
     - **Data bits (LSB first).**
     - **Stop bit(s) (high).**

Looking at the waveform, you can see tx_serial goes low (start), then follows the bit pattern of the written data.

8.0 Conclusion

Project Summary

This project successfully designed, implemented, and verified a fully functional **UART (Universal Asynchronous Receiver/Transmitter) system** with an **AMBA APB (Advanced Peripheral Bus) wrapper**. The system provides a standard, memory-mapped interface for serial communication, making it suitable for integration into a larger System-on-Chip (SoC) design. The core consists of three meticulously crafted components:

1. **UART Transmitter (UART_TX):** A parameterized module that efficiently converts parallel data into a standard serial format, complete with start, data, and stop bits.
2. **UART Receiver (UART_RX):** A robust receiver that accurately reconstructs serial data back into parallel form, implementing mid-bit sampling for noise immunity and featuring frame error detection.
3. **APB Wrapper (APB_WRAPPER):** A bus interface module that bridges the UART core to an AMBA APB bus, providing a register-based control and status interface for a system processor.

Key Achievements

- **Modular and Scalable Design:** The clear separation between the UART protocol logic and the bus interface logic is a significant strength. This modularity allows for easy reuse of the UART cores in other projects and simplifies the addition of other peripherals to the same bus.
- **Standards Compliance:** The design correctly implements both the UART serial communication protocol and the AMBA APB bus protocol, ensuring compatibility with industry-standard tools and processors.
- **Configurability:** Through SystemVerilog parameters (BAUD_RATE, CLK_FREQ, DATA_BITS), the design is highly flexible and can be adapted to a wide range of system requirements without modifying the core RTL code.
- **Robust Operation:** Key design decisions, such as **mid-bit sampling** in the receiver and a **finite state machine (FSM)** approach for control, ensure reliable and predictable operation.
- **Comprehensive Verification:** A structured verification strategy was outlined, moving from unit-level testing to full system-level verification using directed tests, constrained-random stimuli, functional coverage, and assertions to ensure design correctness.

Learning Outcomes

Through this project, several critical digital design concepts were reinforced:

- The practical implementation and integration of **standard bus protocols** (APB).
- The design of **FSM-controlled communication peripherals**.
- The importance of **synchronization** and **metastability** avoidance in digital systems.

- The development of a **methodical verification strategy** is just as important as the RTL design itself.
- The value of **parameterization** and **modularity** in creating reusable IP blocks.

Future Enhancements

While the current design is fully functional, several avenues for future enhancement exist to increase its capability and robustness:

1. **FIFO Buffers:** Adding FIFOs for transmit and receive paths would allow for back-to-back data transmission and prevent data overrun, reducing the CPU's interrupt overhead.
2. **Interrupt Support:** Enhancing the APB wrapper to generate interrupts for events like tx_done, rx_ready, and frame_error would enable more efficient, interrupt-driven system operation.
3. **Advanced Features:** Implementing optional **parity generation/checking** and **programmable data bit length** would make the peripheral more versatile.
4. **Diagnostic Modes:** Adding a **loopback mode** (connecting TX directly to RX internally) would be invaluable for system diagnostics and testing.
5. **Clock Domain Crossing (CDC):** Formal implementation of synchronizers and CDC protocols for the rx_serial signal would further enhance the design's reliability in asynchronous environments.

Final Remarks

This UART with APB Wrapper project demonstrates a complete understanding of the digital design flow, from specification and RTL coding to verification planning. The resulting IP core is a robust, configurable, and standards-compliant peripheral that serves as a fundamental building block for modern embedded and SoC designs. It stands as a strong foundation that can be extended and refined to meet the demands of increasingly complex systems.