APB UART IP - Final Project Report

Project: Design and Implementation of a Custom UART IP Wrapped with an AMBA APB Interface

Author: Shrif Emad

Supervisor: ENG/ Mohamed Salah

Note: This final report includes: Introduction, Design Analysis, textual State Machine descriptions, Design Decisions, Verification Strategy, Simulation Results, and Conclusion — prepared according to the project specification.

Introduction

In this project, I designed a Universal Asynchronous Receiver/Transmitter (UART) module and wrapped it with an AMBA APB (Advanced Peripheral Bus) slave interface. The objective was to provide a memory-mapped peripheral that supports serial transmit/receive operations via dedicated registers, while allowing configurable baud rates through a divisor register.

Key learning goals included:

- Understanding asynchronous serial communication mechanisms.

- Designing and documenting state machines (FSMs) for the transmitter and receiver.

- Building a memory-mapped interface following the APB bus protocol.

- Verifying the design with self-checking testbenches and simulation waveforms.

Design Analysis

General Requirements:

- APB Slave with states: IDLE → SETUP → ACCESS.

- 32-bit registers as required by the project specification.

- UART using 8-N-1 format, baud configured through BAUDIV register.

- Status signals: tx_busy, tx_done, rx_busy, rx_done, rx_error.

- Soft resets for TX and RX available through CTRL_REG.

High-Level Block Diagram (Textual):

- APB Wrapper: Decodes addresses, maps reads/writes to registers, drives APB FSM, asserts PREADY.

- UART TX Core: Loads a byte from TX_DATA, shifts it out serially with a start bit and stop bit, asserts tx_busy and tx_done.

- UART RX Core: Detects start bit, samples data at mid-bit intervals, validates stop bit, places result in RX_DATA, asserts rx_done or rx_error.

- Baud Generator: Divides the 100 MHz system clock to generate bit-period pulses, controlled by BAUDIV.

Register Map (32-bit each):

0x0000 CTRL_REG: tx_en, rx_en, tx_rst, rx_rst

0x0001 STATS_REG: rx_busy, tx_busy, rx_done, tx_done, rx_error

0x0002 TX_DATA: Transmit byte (LSB used).

0x0003 RX_DATA: Received byte (LSB used).

0x0004 BAUDIV: Baud rate divisor (bit_cycles = BAUDIV + 1).

State Diagrams (Textual)

TX FSM: IDLE → LOAD → START_BIT → SEND_BITS → SEND_STOP → DONE → IDLE

RX FSM: IDLE → EDGE_DETECT → WAIT_HALF_BIT → SAMPLE_BITS → CHECK_STOP → DONE/ERR → IDLE

APB FSM: IDLE → SETUP → ACCESS → IDLE

Design Decisions

1. Use 8-N-1 format only (no parity) for simplicity.

2. Use 32-bit registers, with UART data in LSB.

3. Baud divisor register allows flexible baud configuration.

4. PREADY asserted for one cycle only.

5. Provide soft resets for TX and RX.

6. No FIFOs in baseline implementation (single-byte buffer).

7. busy and done flags are latched until cleared.

Verification Strategy

Tools: Verilog simulators such as ModelSim or Icarus Verilog.

Test Cases:

- TX: Correct bitstream on line, tx_busy during transmit, tx_done after stop bit.

- RX: Correct byte assembly, stop bit validation, detect framing errors.

- APB: Validate register access, one-cycle PREADY assertion.

Corner Cases: Very low BAUDIV values, soft reset during transmission, back-to-back transfers.

Simulation Results

Key Outcomes:

- TX produced correct start/data/stop sequence at expected baud rates.

- RX assembled bytes correctly and flagged framing errors.

- APB responded with one-cycle PREADY and correct data.

Baud Calculations (Fclk = 100 MHz):

- 9600 baud: BAUDIV = 10415 or 10416.

PASS/FAIL Summary:

- tb_uart_tx: PASS

- tb_uart_rx: PASS

- tb_apb_uart: PASS

8. Conclusion

The project successfully produced a custom APB-wrapped UART IP that implements 8-N-1 communication, provides a clear register map, and includes FSMs for TX, RX, and APB interface.

Verified across multiple test cases and corner conditions.

Future Improvements:

1. Add FIFOs for higher throughput.

2. Support parity bits and multiple stop bits.

3. Add hardware flow control (RTS/CTS).

4. FPGA prototyping and timing closure.

5. Improve baud accuracy using PLL/dividers.