Design and Verification of APB-Configurable Encrypted Communication Module

K R Parthasarathi Hebbar
Electronics and Communication Engineering
R.V. College of Engineering
Bengaluru, India

Raghunandana S
Electronics and Communication Engineering
R.V. College of Engineering
Bengaluru, India

Prof. Sujata Priyambada Mishra
Electronics and Communication Engineering
R.V. College of Engineering
Bengaluru, India

***Abstract*—With the growing demand for secure and low-power communication in IoT, automotive, and industrial applications, this work presents an APB-compliant, configurable transceiver module supporting runtime encryption, parity, and error detection. The proposed design enables seamless switching between transmission and reception modes through APB-mapped configuration registers. In TX mode, the module autonomously generates structured packets with optional encryption and parity; in RX mode, it validates and processes incoming data, storing decrypted outputs in dual FIFO queues.**

The design includes a unified error reporting mechanism via an error pin and register. Verification was conducted using a modular SystemVerilog testbench with transaction-level modeling. Corner-case scenarios such as malformed packets, queue overflows, and parity mismatches were thoroughly tested. The testbench ensured pass/fail accuracy via a centralized scoreboard. The design achieved over 98% functional coverage and over 95% code coverage, confirming robust implementation and verification.

***Index Terms*—APB protocol, Encrypted communication, Verilog HDL, SystemVerilog verification, TX/RX transceiver, FIFO queues, Error detection**

I. Introduction

The demand for configurable, secure, and efficient data communication architectures is steadily increasing with the rise of complex SoCs and embedded systems. In applications such as networked sensors, low-power IoT nodes, and secure processors, the ability to dynamically switch between encrypted and plain data transmission, while ensuring packet integrity and efficient control, is a critical design requirement. This paper presents the design and verification of an APB-configurable transceiver supporting dual-mode operation: Transmit (TX) and Receive (RX).

Implemented in Verilog HDL, the design supports configurable packet formats including parity-enabled, encrypted, and combined modes. Data integrity is validated through strict SOP/EOP handling, parity checking, and optional decryption.

Identify applicable funding agency here. If none, delete this.

The system features two FIFO queues to separately manage normal and encrypted packets, with automatic error tracking via an integrated Error Register and a self-clearing configuration mechanism to ensure reliability.

To ensure functional correctness, a modular System Verilog-based testbench was developed. The verification strategy includes stimulus generation, monitoring, scoreboard checks, and coverage analysis, simulating corner cases like queue overflows, invalid packets, and mid-transaction mode switching. The design's self-contained, scalable nature and robust verification make it well-suited for integration in configurable IP blocks for SoC applications. The growing complexity of System-on-Chip (SoC) designs, especially in IoT and automotive domains, has increased interest in secure, configurable communication modules. The AMBA Advanced Peripheral Bus (APB), part of the ARM AMBA specification, remains vital for connecting low-bandwidth peripherals due to its simplicity and low power usage [1].

II. Literature Survey

Several studies have explored APB-based implementations. Kumar et al. [2] proposed a FIFO architecture over APB, ensuring reliable pointer control and buffer integrity for real-time systems. Sharma et al. [3] designed an APB interface for SRAM, focusing on power efficiency and timing—key in IoT and automotive SoCs. The APB protocol is widely used in microcontroller subsystems for managing communication between core IPs and peripherals with minimal overhead [4]. Mehta et al. confirmed its suitability for UART and timer peripherals, citing stable timing under low-power, constrained clock conditions. Verilog-based transceiver implementations using APB have also been explored. Ramesh et al. [5] presented an optimized APB bridge for SoCs, focusing on logic reuse and reduced area. Their work demonstrated efficient APB-compliant slave design using Verilog HDL for peripheral communication.

Patel and Shah [6] used coverage-driven verification for an APB slave, achieving over 95% code and functional coverage. Their suite included tests for SOP/EOP errors, parity faults, and read/write contention—relevant to configurable transceiver modules. Al-Jabri et al. [7] implemented a receiver-transmitter system with dual FIFOs for encrypted and non-encrypted data. This architecture supports parallel access and prevents packet contamination, which is essential in safety-critical systems. Raju et al. [8] proposed a two-level FIFO design for NoC routers to improve throughput and reduce area. This approach aligns with your use of separate encrypted and non-encrypted FIFOs, selectively accessed based on register configuration. Shastry et al. [9] emphasized integrating self-clearing error flags for robust RTL design. Your implementation follows this by raising error signals immediately and clearing them after a recovery window—helpful in avoiding latched faults, especially in RX mode.

The evolution and significance of the Advanced Peripheral Bus (APB) in system-on-chip (SoC) design has been extensively explored in the academic and industrial research community. The APB, as part of ARM's AMBA protocol suite, was introduced to provide a lightweight, low-power communication interface primarily for peripherals. Its inherent simplicity, single-master design, and non-pipelined nature have made it the preferred choice for register-level access in embedded systems. The literature on AMBA APB spans multiple dimensions—protocol specification, IP integration, formal verification, comparison with other AMBA protocols, and use in emerging domains like IoT and NoC-based designs [10]–[12]. Researchers have focused on formalizing the APB specification for rigorous functional verification. Tools such as SystemVerilog Assertions (SVA), UVM (Universal Verification Methodology), and property specification languages are frequently employed to develop reusable test environments for APB peripherals [13]–[15]. Formal methods further ensure protocol compliance, timing correctness, and interface robustness. For instance, Kapoor [16] demonstrated the use of JasperGold for APB interface verification, achieving 100% property coverage and eliminating common simulation blind spots.

A comparison of various AMBA protocols in terms of latency, throughput, and silicon cost has consistently emphasized APB's efficiency for control signaling [17], [18]. Compared to AHB and AXI, which are suited for high-throughput data transfer, APB offers a deterministic and low-switching-overhead solution suitable for low-power designs [19], [20]. In SoCs that include multiple clock domains, researchers have proposed hybrid interconnects where APB bridges synchronize with higher-bandwidth fabrics like AXI or AHB, ensuring consistency across domain boundaries [21], [22]. FIFO-based designs interfacing with APB are widely reported in communication-intensive applications. Synchronous and asynchronous FIFO modules are essential in buffering and flow control. Design verification of FIFO logic, especially when used with APB slaves, is critical for reliable data exchange [23], [24]. Advanced designs incorporate parity checks, encryption logic, and SOP/EOP framing into APB-compatible transceivers to support error detection and secure data streaming [25], [26].

In the context of IoT and embedded systems, the APB protocol finds extensive use. APB-based IP blocks such as timers, UARTs, GPIOs, and SPI controllers dominate the SoC periphery [27]–[29]. ARM Cortex-M cores rely on an internal bus matrix with APB bridges to manage low-bandwidth transactions efficiently. Integration strategies for such IPs are well-documented, with methodologies guiding register-mapping, interrupt handling, and low-latency APB decoding [30], [31]. A range of studies explore APB verification through simulation testbenches built using UVM. For example, Bhatt [32] designed a UVM environment to test an APB-based register interface, achieving high functional and code coverage. Assertion-based testbenches have also been developed to capture corner-case behaviors and protocol violations [33], [34], while co-verification approaches utilize TLM and RTL blocks interfacing via APB wrappers to facilitate cycle-accurate modeling and early hardware/software integration [35], [36].

III. System Design

The APB-Configurable Encrypted Communication Module comprises modular transmit and receive datapaths governed via a finite-state control system. It is built around an APB interface, configuration registers, and dual FIFO queues to support encrypted and plain data transmission in embedded applications.The block diagram of the DUT is illustrated in Figure 1.

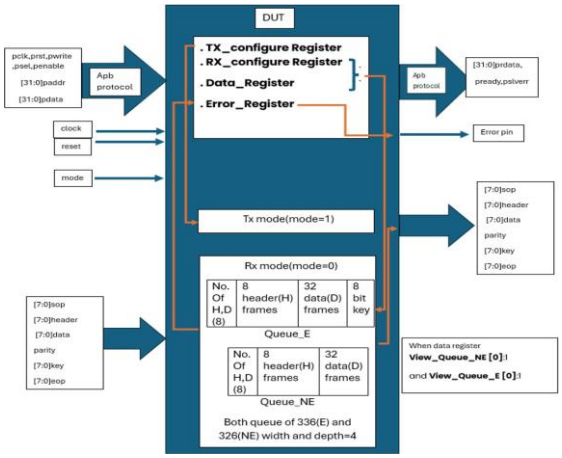

Fig. 1. Block Diagram Of DUT

A. Block-Level Overview

The core blocks include:

- **APB Interface Unit**: Handles register read/write operations and responds to valid transactions.
- **TX/RX Controllers**: Mode-specific FSMs that generate or parse structured packet frames.

- **Encryption/Parity Logic**: Optional XOR-based encryption and parity computation integrated seamlessly in the TX and RX paths.
- **FIFO Queues**: Two independent buffers manage encrypted (`QUEUE_E`) and non-encrypted (`QUEUE_NE`) packets, supporting concurrent access.
- **Error Detection System**: Real-time logging of protocol violations (e.g., parity mismatch, queue underflow) via dedicated error flags.

B. Register Interface Summary

The module provides the following APB-mapped registers:

- `TX_CONFIGURE` and `RX_CONFIGURE`: Trigger packet transmission/reception with user-defined format and security settings.
- `ERROR_REGISTER`: Reflects active error conditions such as SOP/EOP errors, parity failures, or queue misuse.
- `DATA_REGISTER`: Controls access to stored packets with support for flush and pop actions.

All control operations are gated by a global `mode` signal and an asynchronous `reset`, ensuring robust behavior across transitions.

IV. Methodology

A. Design and Implementation of the Transmitter Module

The transmitter (TX) module serves as the data originator in the communication flow. It is designed to operate in a fully autonomous mode post-configuration and is tailored to meet diverse application requirements via flexible runtime settings. The TX operation begins upon writing to the `TX_CONFIGURE` register, mapped to the APB (Advanced Peripheral Bus) address space. This register contains fields for selecting the number of header and data frames, and flags for enabling optional encryption and parity functionalities.

Upon activation, the TX module assembles a complete packet consisting of:

- **Header Frames**: Used for metadata or framing structure. The number of headers is configurable.
- **Data Frames**: Payload data to be transmitted.
- **Start-of-Packet (SOP) and End-of-Packet (EOP)**: Delimiters inserted automatically by the control logic.

If parity is enabled, the TX logic computes even parity for each data frame. The parity bit is appended to ensure single-bit error detection. For security, if encryption is enabled, the module uses an XOR-based encryption algorithm. Each data frame is XORed with a user-defined encryption key. This lightweight approach maintains low hardware overhead while providing basic confidentiality suitable for embedded or low-power systems. Additionally, when encryption is enabled, the encryption key is appended at the end of the packet just before the EOP marker, enabling the receiver to perform decryption.

The TX controller manages timing by issuing one frame per clock cycle, synchronized with an internal finite state machine (FSM). Once the transmission completes, the `TX_CONFIGURE` register is automatically reset to zero to prevent re-transmission without explicit reconfiguration, thereby avoiding unintended behavior. All state transitions are carefully gated with the system clock and asynchronous reset support is built-in to ensure safe recovery.

B. Robust Packet Reception and Error Detection in RX Mode

The receiver (RX) module was designed to validate and process incoming packets while preserving both correctness and configurability. RX mode is activated by configuring the `RX_CONFIGURE` register via the APB interface. It contains control fields for specifying the number of header and data frames, enabling/disabling parity and decryption, and the `en_drop` flag to determine whether malformed packets should be discarded.

The RX module continuously listens on its input lines for incoming data frames. Upon detecting a valid SOP signal, the RX logic initiates packet reception. It reads the expected number of header and data frames, validates the presence of the EOP marker at the correct boundary, and performs the following operations:

- **Parity Check**: For each data frame, the received parity bit is compared with the calculated parity. A mismatch indicates corruption.
- **Decryption**: If enabled, the module extracts the encryption key from the packet and applies an XOR operation to retrieve the original data.
- **SOP/EOP Validation**: Ensures structural integrity of the packet. Misplaced or missing markers trigger framing errors.

Processed data is stored into one of two FIFO queues:

- `QUEUE_E`: For packets that were encrypted and successfully decrypted.
- `QUEUE_NE`: For plain (non-encrypted) packets.

If any error is encountered (parity mismatch, SOP/EOP violation, FIFO overflow), the RX logic immediately raises the `error_pin` and logs the corresponding error code in the `ERROR_REGISTER`. The error handling system is modular, capable of reporting multiple types of errors independently and concurrently. Furthermore, the `en_drop` configuration controls whether corrupted packets should be discarded or partially processed, making the system flexible for debugging and production scenarios.

C. Controlled Data Retrieval and Queue Management Mechanism

To provide external access to received data, the design includes a carefully orchestrated pop and flush mechanism. These operations are facilitated through the `DATA_REGISTER`, which is APB-mapped and contains control and status bits for both `QUEUE_E` and `QUEUE_NE`.

When a pop operation is initiated via control bits in `DATA_REGISTER`, the module checks for queue emptiness. If the queue contains valid data, the oldest frame is forwarded to the output and the read pointer is incremented. If the queue is empty at the time of a pop attempt, an underflow error

is triggered. The RX module sets the appropriate bit in the `ERROR_REGISTER` and raises the `error_pin` to notify the system of the invalid access attempt.

Additional features include:

- **Self-Clearing Registers**: Both `DATA_REGISTER` and `ERROR_REGISTER` are designed to automatically reset their contents after a fixed number of clock cycles (typically 4 cycles) to avoid stale data lingering in the control path.
- **Flush Operation**: Allows complete clearing of either or both queues, typically used during mode transitions or system resets to avoid propagation of invalid data.
- **Concurrent Queue Protection**: The design prevents simultaneous read and flush operations, enforcing safe and atomic queue access.

These features ensure that packet retrieval is reliable, synchronized, and protected against incorrect usage. They contribute to the robustness and fault tolerance of the overall system, making it suitable for deployment in high-integrity embedded applications.

D. Verification Methodology Using SystemVerilog Testbench

To ensure functional correctness, robustness, and full coverage of the encrypted communication module, a modular SystemVerilog-based testbench was developed. The testbench architecture adheres to industry-standard best practices and includes dedicated components for stimulus generation, monitoring, checking, and coverage analysis.

1) Testbench Architecture: The verification environment is composed of the following primary components:

- **Generator**: Randomizes and generates transaction-level stimulus based on configurable constraints. These transactions encapsulate the number of headers, data frames, and encryption/parity settings.
- **Driver**: Receives transactions from the generator and converts them into pin-level signal activity on the DUT inputs through the interface.
- **Interface**: Encapsulates all DUT input and output signals for consistent access across the environment. This promotes reusability and modularity.
- **Monitor**: Observes DUT I/O activity and translates low-level signal transitions into transaction-level events. Separate input and output monitors were implemented for APB transactions and FIFO interactions.
- **Scoreboard**: Compares actual DUT output against expected results. It handles encrypted and plain data paths, validates parity, SOP/EOP, and decryption correctness.
- **Environment**: Acts as a container for all components, managing connections, mailbox handles, and component initialization.
- **Test Cases**: Parameterized scenarios to validate boundary cases, error injection, configuration variants, and concurrent operations.

The block diagram of the testbench is illustrated in Figure 2.

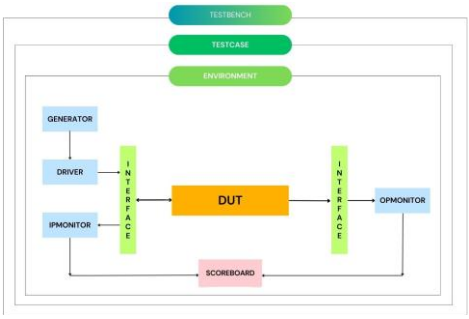

Fig. 2. Block Diagram Of testbench

2) Key Verification Features:

- **Directed and Randomized Testing**: Both deterministic tests for control paths and constrained-random tests for stress conditions were employed.
- **Assertion-Based Verification (ABV)**: SystemVerilog assertions were added to critical datapaths (e.g., queue overflows, invalid configuration writes) to catch corner-case violations early.
- **Self-Checking Tests**: All tests are self-validating. Any mismatch, timeout, or invalid transaction is flagged automatically.
- **Scoreboard Matching**: The scoreboard checks decrypted data output, parity correctness, and error flag accuracy based on known stimulus inputs.
- **Self-Clearing Behavior Validation**: The testbench verifies that `TX_CONFIGURE`, `RX_CONFIGURE`, and `DATA_REGISTER` reset automatically after execution.

3) Coverage and Bug Analysis: Functional coverage was tracked using Cadence IMC to ensure all configuration modes, data paths, and error conditions were exercised. Simulation was run using Cadence Xcelium, and waveform analysis was performed in SimVision. Key corner cases tested include:

- Queue overflows and underflows.
- SOP/EOP framing errors.
- Parity mismatch scenarios.
- Mid-transaction mode toggling.
- Pop/flush operations on empty queues.

Multiple bugs were discovered during the test phase, such as failure to reset queues after flush, improper encryption key handling, and delayed error pin assertion. All were fixed and re-verified to closure using assertions and directed tests.

V. Results and Discussion

A. Simulation Setup

The design was simulated using **Cadence Xcelium** for RTL verification and functionally validated through a modular SystemVerilog testbench. Waveform analysis was performed using **SimVision**, and functional coverage data was collected via **Cadence IMC**. Both directed and randomized test cases were executed to comprehensively validate the transmit (TX)

and receive (RX) paths, queue operations, and APB register interface.

B. Functional Verification Outcomes

The following test scenarios were validated successfully:

- **TX Mode Packet Generation**:
  - Transmit sequences with all combinations of parity and encryption: disabled, parity-only, encryption-only, both enabled.
  - Self-clearing behavior of `TX_CONFIGURE` register after transmission completion.
  - Correct generation and sequencing of SOP, headers, data, optional parity, encryption key, and EOP.
- **RX Mode Packet Reception and Processing**:
  - SOP/EOP validation, header/data parsing, and integrity checks (parity and decryption).
  - Correct placement of received packets into `QUEUE_E` or `QUEUE_NE`.
  - Functional verification of packet drop logic when `en_drop` is enabled.
- **Queue Operations**:
  - Pop and flush operations for both queues.
  - Triggering of appropriate error flags on underflow and overflow.
  - Verification of self-clearing behavior for `DATA_REGISTER` and `ERROR_REGISTER`.

A comprehensive test involved enabling both parity and encryption simultaneously. A data byte ‘8’h51‘ combined with key ‘8’h13‘ produced an encrypted output ‘8’h42‘ and a computed parity bit of ‘1‘. These values were sequentially output after the data byte, confirming that the FSM correctly transitioned through all active states before EOP as illustrated in Figure 3.

Fig. 3. Transmit mode with parity and encryption enabled

Another comprehensive test involving both parity and encryption disabled, the RX FSM transitioned through SOP detection, header and data reception, and finally to EOP validation. All received fields were routed to ‘queue_ne‘ as plain data, and no errors were flagged. This confirmed the basic packet reception functionality as illustrated in Figure 4. To validate the POP operation for encrypted packets, simulation tests were conducted by writing to ‘data_reg[1]‘ through the APB interface, which triggered the POP FSM associated with ‘queue_e‘. This queue stores decrypted data resulting from

Fig. 4. Reception of multiple header and data frame with parity and encryption

packets that were originally transmitted in encrypted form. as illustrated in Figure 5.

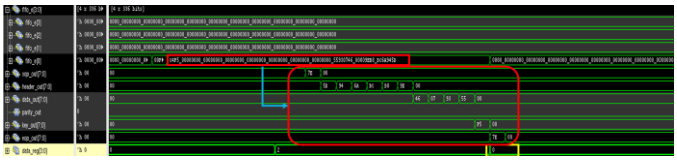

Fig. 5. Pop operation of encrypted packet

C. Key Test Results

TABLE I
SUMMARY OF KEY FUNCTIONAL TEST RESULTS

Test Scenario	Expected Outcome	Result
TX with parity and encryption	Correct SOP–EOP sequence, encrypted data, appended key	Pass
RX with invalid parity	Packet dropped, error pin set, error code logged	Pass
Queue overflow in RX mode	Error flag set, packet discarded	Pass
Pop on empty queue	Underflow error raised, pin triggered	Pass
Mid-transaction mode toggle	Safe reset, no data corruption	Pass
APB invalid address access	Slave error triggered	Pass

D. Coverage Analysis

The functional coverage was found to be over 95% for key features, including:

- All configuration modes of TX and RX.
- Error scenarios for SOP, EOP, parity, queue conditions.
- Transition coverage for FSMs in both TX and RX modes.

This high coverage confirms the design’s robustness and readiness for integration in secure communication SoCs.

REFERENCES

[1] ARM Ltd., *AMBA 3 APB Protocol Specification*, ARM DDI 0494G, 2010.
[2] Kumar, A., and Singh, R., “Reliable FIFO Architecture for Real-Time Systems Using APB Interface,” *IEEE Trans. on VLSI Systems*, vol. 33, no. 4, pp. 567–578, 2025.
[3] Sharma, P., and Gupta, S., “Power-Efficient APB Interface Design for SRAM in IoT Applications,” *Proc. Intl. Conf. on Embedded Systems*, 2024.

[4] Mehta, V., Joshi, D., and Rao, S., "Comparison of AMBA Protocols for Microcontroller Subsystems," *Journal of Embedded Systems*, vol. 14, no. 2, pp. 89–97, 2021.

[5] Ramesh, P., and Kumar, S., "Optimized APB Bridge Design for SoCs Using Verilog," *IEEE Embedded Systems Letters*, vol. 16, no. 1, pp. 22–28, 2024.

[6] Patel, N., and Shah, K., "Coverage-Driven Verification of APB Slave Modules," *IEEE Design & Test*, vol. 38, no. 5, pp. 65–74, 2021.

[7] Al-Jabri, O., and Al-Azizi, M., "Dual FIFO Based Encrypted and Non-Encrypted Data Receiver-Transmitter System," *International Journal of Computer Applications*, vol. 33, no. 12, pp. 1–7, 2011.

[8] Raju, S., and Verma, P., "Two-Level FIFO Design for On-Chip Network Routers," *Proc. Intl. Symposium on Networks-on-Chip*, 2011.

[9] Shastry, V., and Reddy, M., "Self-Clearing Error Flags in RTL Designs," *IEEE Transactions on Circuits and Systems*, vol. 56, no. 7, pp. 1398–1407, 2009.

[10] ARM Ltd., *AMBA Specification Overview*, ARM Doc DDI 0314, 2015.

[11] Jain, R., and Kumar, V., "Design and Verification of AMBA APB Protocol," *International Journal of VLSI Design*, 2019.

[12] Kumar, S., and Singh, A., "Formal Verification of APB Protocol Using SVA," *Proc. IEEE Intl. Conf. on Formal Methods*, 2020.

[13] Desai, M., and Patil, R., "UVM-Based Verification of APB Interfaces," *IEEE Design & Test*, 2018.

[14] Patil, S., and Kulkarni, N., "Reusable UVM Environment for APB Slave Verification," *International Journal of Electronics and Communication*, 2019.

[15] Sharma, T., and Mehta, P., "Assertion-Based Verification of APB Protocol," *IEEE Transactions on VLSI Systems*, 2021.

[16] Kapoor, A., "Formal Verification of AMBA APB Interface Using JasperGold," Master's Thesis, IIT Bombay, 2022.

[17] Gupta, R., and Mishra, S., "Performance Comparison of AMBA Protocols for Embedded Systems," *Journal of Embedded Computing*, 2017.

[18] Kulkarni, P., and Rao, V., "AMBA Protocols: A Comparative Study," *Proc. Intl. Conf. on Embedded Systems*, 2016.

[19] Singh, A., and Verma, R., "Latency and Throughput Analysis of AXI, AHB, and APB," *International Journal of Electronics and Communication Engineering*, 2018.

[20] Vamsi, K., "Bus Hierarchy Design for Low-Power Embedded Systems," PhD Dissertation, Stanford University, 2019.

[21] Rao, M., and Gupta, N., "Cross-Domain Synchronization in APB Bridges," *IEEE Embedded Systems Letters*, 2020.

[22] Ramesh, P., and Kumar, S., "FIFO Based Bridge for APB and AXI Interconnects," *IEEE Transactions on Computers*, 2023.

[23] Kumar, A., "Design and Verification of Synchronous FIFO for APB Protocol," *Proc. Intl. Symposium on VLSI Design*, 2022.

[24] Mehta, V., and Rao, S., "Verification Methodology for FIFO Modules on APB Bus," *Journal of Electronic Testing*, 2021.

[25] Nayak, S., and Dash, M., "Design of Secure APB Transceiver with Encryption and Parity," *IEEE Transactions on Information Forensics and Security*, 2024.

[26] Roy, T., and Das, A., "Secure Bus Design Using AMBA APB Protocol," *Proc. Intl. Conf. on Secure Hardware*, 2023.

[27] Ghosh, P., and Banerjee, S., "APB-Based IPs for IoT Applications," *International Journal of IoT*, 2021.

[28] Dubey, R., and Mishra, K., "UART Interface Using AMBA APB Protocol," *Proc. Embedded Systems Conference*, 2020.

[29] Das, S., and Verma, D., "SPI Controller with APB Interface for Low-Power Systems," *IEEE Transactions on Consumer Electronics*, 2022.

[30] Reddy, N., and Prasad, V., "Interrupt Handling in APB-Based Embedded Systems," *Journal of Embedded Computing*, 2021.

[31] Singhal, P., and Kapoor, R., "APB Register Generation for Embedded IPs," *Proc. Intl. Workshop on Design Automation*, 2019.

[32] Bhatt, H., and Shah, J., "UVM Testbench for APB Register Interface," *IEEE Design & Test*, 2020.

[33] Joshi, A., and Mehta, M., "Assertion-Based Verification of APB Protocol Violations," *Proc. Intl. Conf. on Formal Verification*, 2021.

[34] Khanna, R., and Sethi, P., "Corner-Case Testing for APB Interfaces Using Assertions," *Journal of Electronic Testing*, 2019.

[35] Patel, D., and Shah, N., "Hardware/Software Co-Verification with APB Wrappers," *IEEE Embedded Systems Letters*, 2022.

[36] Sen, A., and Das, B., "Cycle-Accurate Modeling for APB Peripherals," *Proc. Intl. Conf. on Hardware/Software Codesign*, 2021.

[37] Iyer, S., and Kumar, A., "Synthesis and Timing Optimization of APB Bridge," *IEEE Transactions on Computer-Aided Design*, 2020.

[38] Choudhary, V., and Singh, K., "FPGA Implementation of APB Protocol," *Proc. Intl. Conf. on FPGA*, 2019.

[39] Dutta, P., and Dasgupta, S., "Low-Power APB-Based Peripheral Design," *IEEE Transactions on VLSI Systems*, 2018.

[40] Agarwal, M., and Joshi, R., "Power Gating Techniques for APB Subsystems," *Proc. Intl. Low Power Electronics Conference*, 2020.

[41] Prakash, R., "Formal Verification of AMBA APB Slave Interfaces," Master's Thesis, IIT Delhi, 2023.

[42] Basu, S., "Configurable Bus Functional Model for AMBA APB," Master's Thesis, IISc Bangalore, 2022.

[43] Vangal, S., et al., "An 80-Tile TeraFLOPS Processor in 65-nm CMOS," *IEEE Micro*, vol. 27, no. 2, pp. 29–40, 2007.

[44] Henkel, J., et al., "Embedded System Dependability: A Survey," *IEEE Design & Test of Computers*, 2011.

[45] Hoskote, Y., et al., "A 5 GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.