Efficient Data Width Conversion Using AXI4 Stream Downsizer in Verilog

Pratikhya Raut
Department of Electronics and Communication Engineering
Siddhartha Academy of Higher Education
Deemed to be University
Vijayawada, Andhra Pradesh, 520007, India
sweetpratikshya91@gmail.com

Durga Bhavani Sugrivu
Department of Electronics and Communication Engineering
Siddhartha Academy of Higher Education
Deemed to be University
Vijayawada, Andhra Pradesh, 520007, India
sugrivudurgabhavani@gmail.com

Varshitha Yaramala
Department of Electronics and Communication Engineering
Siddhartha Academy of Higher Education
Deemed to be University
Vijayawada, Andhra Pradesh, 520007, India
varshitha.2505yaramala@gmail.com

Poojitha Yanamareddy
Department of Electronics and Communication Engineering
Siddhartha Academy of Higher Education
Deemed to be University
Vijayawada, Andhra Pradesh, 520007, India
poojithayanamareddy@gmail.com

***Abstract*—In modern digital systems, efficiently managing data between interfaces with different data widths is crucial for high performance. This paper presents a Verilog-based design of an optimized AXI Stream Downsizer that converts data from a wider master interface to a narrower slave interface. The design ensures low latency, high throughput, and efficient use of FPGA resources while fully supporting AXI Stream protocol signals like TVALID, TREADY, TLAST, and TKEEP.This approach uses techniques such as data packing, buffer management, and finite state machines to handle partial data transfers effectively. Experimental results demonstrate that the proposed downsizer achieves better timing performance and uses fewer resources compared to conventional methods. This makes it well-suited for applications like high-speed networking, image processing, and real-time data acquisition, where flexible and reliable data streaming is essential.**

***Keywords*—AXI Stream Protocol, Downsizer, Verilog, Data Width Conversion, Finite State Machine**

I. INTRODUCTION

As digital systems continue to grow in complexity, there is an increasing need for compact, scalable, and standard- ized interfaces to support high-speed data communication between system components. The Advanced Microcontroller Bus Architecture (AMBA), developed by ARM [6] has become a widely adopted on-chip communication standard. Among its protocols, the AXI (Advanced eXtensible Interface) Stream interface is particularly suited for high-throughput, low-latency, unidirectional, burst-based data transfers, as it avoids address-phase overhead. However, in many systems, the data width of the source and destination AXI Stream interfaces often differ due to varying design optimizations or performance requirements. Bridging this mismatch efficiently is critical to maintaining system performance, minimizing latency, and ensuring flexible integration. This paper presents a Verilog-based design of an optimized AXI Stream Downsizer, which converts data from a wider AXI Stream interface to a narrower one. This process, known as data width downsizing, introduces challenges such as maintaining proper data alignment, managing flow control, and adhering to AXI Stream protocol signals (TVALID, TREADY, TKEEP, TLAST). Our proposed solution addresses these challenges using a well-structured architecture that includes finite state machines (FSMs), intelligent buffering, and streamlined control logic. Through extensive simulation we demonstrate that our downsizer achieves improved resource efficiency, lower critical path delay, and sustained high throughput compared to conventional methods.

A. CHOICE OF AXI4 STREAM PROTOCOL :

The AXI4-Stream protocol was selected over alternatives such as AXI4-Lite, full AXI4, ACE/ACE Lite, AMBA AHB, and APB [8] because the target application demands continuous, high-throughput data transfer with minimal latency and low hardware overhead. AXI4 and AXI4-Lite [10] are primarily optimized for memory-mapped transactions that involve addressing and burst management, which introduce unnecessary control overhead when the primary goal is the rapid movement of sequential data between processing modules. Similarly, ACE [7] and ACE-Lite provide cache coherency features tailored for multi-core processors, which are not required in this streaming-oriented design. AHB and APB [9] while simpler, offer limited bandwidth and are more appropriate for low-speed peripheral or register-level communication rather than high-performance streaming.

In contrast, AXI4-Stream eliminates the address phase and focuses solely on payload delivery, enabling lightweight, cycle-by-cycle flow control through its TVALID/TREADY handshake mechanism. Additional sideband signals such as TLAST and TKEEP provide precise packet boundary and partial-word management, both of which are critical for efficient real-time data width conversion. These features make AXI4-Stream particularly well-suited for the proposed downsizer, which aims to deliver low latency, high throughput, and resource-efficient serialization of wide data words into narrower streams. By leveraging AXI4-Stream, the design achieves near-maximum bandwidth utilization with minimal buffering, directly meeting the performance goals of applications such as image processing, high-speed networking, and real-time data acquisition.

II. LITERATURE REVIEW

In recent years, various enhancements to AMBA AXI-based interconnect systems have been proposed, addressing challenges related to performance, reliability, and compatibility. These works provide valuable insights that inform the development of our AXI Stream Downsizer, which focuses on converting wider AXI Stream data widths to narrower ones efficiently, within a latency of around 20 microseconds, while maintaining AXI4-Stream protocol compliance.

Jesús Lázaro et al [1] introduced the AXI Lite Redundant On-Chip Bus Interconnect, which implements a triple-redundant communication path aimed at enhancing fault tolerance and zero-latency data flow in FPGA-based systems. The authors address system reliability in critical environments, whereas our downsizer targets throughput and seamless data streaming by managing data width conversion in AXI Stream interfaces rather than system-level fault mitigation.

Jiang et al [2] proposed AXI-Interconnect RT, a modified AXI interconnect designed to support real-time operations by enhancing timing predictability and ensuring low-latency communication in SoCs. Their work aligns with ours in terms of performance optimization but diverges in scope; while their design focuses on real-time arbitration across AXI infrastructure, our design ensures compatibility between mismatched data widths, maintaining protocol correctness through data packing, slicing, and control signal management (e.g., TVALID, TREADY, TLAST, TKEEP).

Zhang et al [3] developed a high-performance, low-area AXI interconnect, emphasizing hardware efficiency and throughput. Their architecture minimizes area utilization while maintaining robust data transfer. Our project shares similar goals in terms of optimization and scalability, but operates specifically at the AXI Stream layer, enabling flexible data width interoperability between streaming components with varying bus widths.

De Sio et al [4] evaluated Single Event Upset (SEU) effects on AXI interconnects within aerospace-grade SoCs, focusing on the reliability of data communication in radiation-prone environments. Although our work does not deal with fault tolerance under radiation, it contributes to system robustness by ensuring standardized and error-free data streaming across interfaces with different data widths, a key consideration for applications requiring consistent performance.

Emil et al [5] presented a custom AXI interconnect for RISC-V-based systems, targeting communication between customized peripherals and processors. Their contribution lies in improving processor-to-peripheral data handling, while our design is more generic and modular, focusing on processor-independent stream downsizing, making it applicable to various real-time systems such as image processing, networking, and high-speed data acquisition.

Together, these works highlight the evolving landscape of AXI-based design optimizations. Our Verilog-based AXI Stream Downsizer builds on these ideas by offering a latency-efficient, resource-optimized, and protocol-compliant solution to a practical and recurring problem in digital systems — data width mismatch between AXI Stream interfaces.

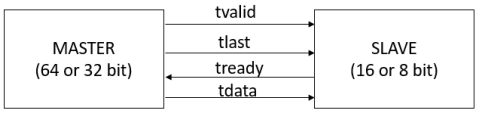

Fig. 1. Block Diagram of Conversion of data width

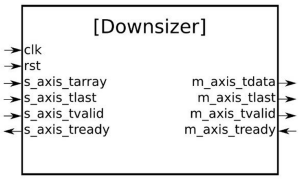

Fig. 2. Block Diagram of AXI Stream Downsizer

III. IMPLEMENTATION OF ALGORITHM

The AXI Stream Downsizer module is created to modify the data width from one AXI Stream interface to another, scaling down data packets according to the input and output widths provided. This module consumes data from an AXI Stream slave interface and outputs reduced-width data on an AXI Stream master interface. Fig 1 illustrates the block diagram of the data width conversion using the AXI4 Stream Downsizer.Implemented in Verilog, the design supports dynamic selection of the input width (either 32-bit or 64-bit) and employs a configurable downsizing ratio, enabling serialization of wide data inputs into smaller-width chunks without violating AXI4-Stream protocol requirements. Key handshaking signals including TVALID, TREADY, and TLAST are fully supported to ensure seamless communication and flow control.Functional verification is carried out using a Verilog-based testbench, validating all supported configurations: 32-bit to 8-bit conversion, 32-bit to 16-bit conversion, 64-bit to 16-bit conversion. The proposed design is modular, resource-efficient, and open-source, making it suitable for a wide range of embedded system applications.

A. KEY FUNCTIONALITIES

As shown in Fig. 2 is the block diagram of the AXI Stream downsizer.The proposed AXI Stream Downsizer module includes the following key functionalities:

- **Input Width Selection:** The module supports an `input_width_select` signal to choose between 32-bit and 64-bit input widths, represented by `2'b00` and `2'b01`, respectively. Data is sampled from `s_axis_tdata` when `s_axis_tvalid` is asserted and `s_axis_tready` indicates readiness.
- **Output Width Selection:** The output width is selected using the `output_width_select` signal, where `2'b00` represents 8-bit and `2'b01` represents 16-bit output widths.
- **Data Buffering:** The input data from `s_axis_tdata` is buffered into `data_buffer`. Based on the selected

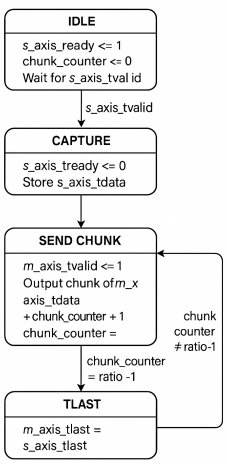

Fig. 3. Finite State Machine (FSM) of Downsizer

output width, the data is divided into smaller segments. The required number of output chunks (ratio) is calculated dynamically.

- **Chunking Process:** A `chunk_counter` keeps track of the number of chunks processed. It is initialized to zero when new input data is received. The downsizer then slices the input data into segments equal to the selected output width (8 or 16 bits).
- **Data Transmission:** The `m_axis_tdata` output is assigned the corresponding chunk from `data_buffer` based on the `chunk_counter` value. For example, 64-bit data is emitted in 8-bit or 16-bit chunks as configured.
- **TLAST Propagation:** The `m_axis_tlast` signal is asserted on the final output chunk, indicating the end of a data transfer. It is derived from the `s_axis_tlast` signal to maintain synchronization with the downstream master interface.
- **Control Signals:**
  - `s_axis_tready`: Indicates readiness to accept new input data.
  - `m_axis_tvalid`: Indicates that valid data is present on the output.
  - `m_axis_tready`: Indicates that the master is ready to receive the next chunk.
- **Reset and Initialization:** Upon reset, internal states such as `chunk_counter`, `s_axis_tready`, `m_axis_tvalid`, `m_axis_tlast`, and the data buffer are reset. The downsizer recalculates the output-to-input width ratio based on the current settings.

B. FINITE STATE MACHINE (FSM) OPERATION

As shown in Fig. 3, the AXI Stream downsizer is controlled by a finite state machine (FSM) to manage data width conversion.The downsizer is controlled by a four-state FSM: **IDLE**, **CAPTURE**, **SEND_CHUNK**, and **TLAST**. This FSM sequences capture, chunking, and transmission while preserving AXI4-Stream semantics.

- **IDLE.** The module advertises readiness to accept a new input beat by asserting `s_axis_tready`. The `chunk_counter` is cleared. A transition occurs when the upstream asserts `s_axis_tvalid`; at that point a handshake (`s_axis_tvalid && s_axis_tready`) moves the FSM to **CAPTURE**.
- **CAPTURE.** `s_axis_tready` de-asserts to block further inputs while the current input word (32 or 64 bits) is latched into `data_buffer`. The downsizer computes the ratio:

$$\texttt{ratio} = \frac{\texttt{input\_width}}{\texttt{output\_width}} \tag{1}$$

  For example, From the Equation 1, the ratio is calculated based on input and output widths. 64→16 gives 4, and 32→8 also gives 4. This ratio determines how many output chunks must be emitted before termination. The FSM then advances to **SEND_CHUNK**.
- **SEND_CHUNK.** The module presents the next chunk (8 or 16 bits) from `data_buffer[chunk_counter]` and asserts `m_axis_tvalid`. The FSM only increments `chunk_counter` after a successful downstream handshake (`m_axis_tvalid && m_axis_tready`), ensuring data stability under backpressure. This state repeats until `chunk_counter == ratio−1`.
- **TLAST.** On the final chunk, `m_axis_tlast` is asserted and propagated from `s_axis_tlast` to mark the end of the frame. After the final handshake, the FSM returns to **IDLE** and re-asserts `s_axis_tready` to accept the next input beat.

C. AXI4-STREAM COMPILANCE GUARANTEES

Handshake correctness & stability: The FSM advances the output pointer only on `m_axis_tvalid && m_axis_tready`, so `m_axis_tdata`/`m_axis_tlast` remain stable when `m_axis_tready` is de-asserted (downstream backpressure). Likewise, `s_axis_tready` is de-asserted outside IDLE to prevent overwriting `data_buffer`, ensuring lossless capture. Simulation results confirm continuous one-chunk-per-cycle emission after capture and correct backpressure behavior.

Framing: `TLAST` is asserted only on the final chunk of a frame and is driven from `s_axis_tlast`, preserving packet boundaries end-to-end.

IV. RESULTS

The AXI4 Stream Downsizer was also successfully used to carry out dynamic data width conversion among AXI Stream interfaces. The module correctly partitioned input data into smaller-sized output pieces while ensuring the AXI protocol through proper assertion of m_axis_tvalid, s_axis_tready and the forwarding of s_axis_tlast to m_axis_tlast.

A. TEST CASE 1: 32-bit to 8-bit Conversion

From the Fig. 4, The simulation waveform demonstrates the behavior of the AXI Stream Downsizer module configured for

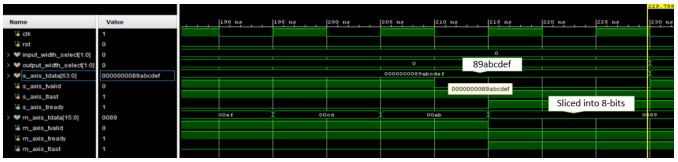

Fig. 4. Simulation waveform showing downsizing of a 32-bit input (0x89ABCDEF) into four 8-bit outputs.

a 32-bit input width and 8-bit output width. The control signals are set as follows:

- `input_width_select = 2'b00` ⇒ 32-bit input
- `output_width_select = 2'b00` ⇒ 8-bit output

Once both `s_axis_tvalid` and `s_axis_tready` are asserted, the 32-bit input data `0x89ABCDEF` is captured into the internal buffer.As shown in Table I, The downsizer then slices this data into four 8-bit chunks, which are transmitted sequentially over consecutive clock cycles on `m_axis_tdata`, as shown below:

TABLE I
32-BIT TO 8-BIT DOWNSIZER OUTPUT PER CLOCK CYCLE(0X89ABCDEF)

Clock Cycle	m_axis_tdata Output
1	0xEF
2	0xCD
3	0xAB
4	0x89

For each output chunk:

- `m_axis_tvalid` is asserted to indicate valid data m_axis_tready controls the handshake mechanism to proceed with data transfer.

B. TEST CASE 2: 32-bit to 16-bit Conversion

From the Fig. 5, The simulation output confirms the correct operation of the AXI Stream Downsizer module configured to convert a 32-bit input into 16-bit output data chunks. The control signals are set as follows:

- `input_width_select = 2'b00` ⇒ 32-bit input
- `output_width_select = 2'b01` ⇒ 16-bit output

When `s_axis_tvalid` is asserted, the 32-bit input data `0x12345678` is captured into the internal buffer.As shown in Table II, The downsizer then divides this data into two 16-bit chunks, which are transmitted on the `m_axis_tdata` output over two clock cycles as follows:

TABLE II
32-BIT TO 16-BIT DOWNSIZER OUTPUT PER CLOCK CYCLE (0X12345678)

Clock Cycle	m_axis_tdata Output
1	0x5678
2	0x1234

During each transfer:

- `m_axis_tvalid` is asserted to indicate valid data m_axis_tready controls the handshake mechanism to proceed with data transfer.

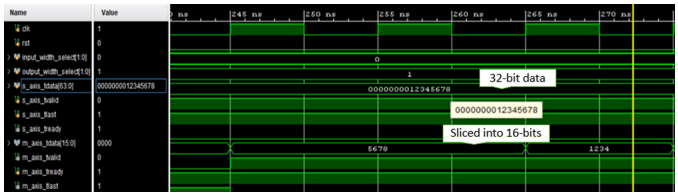

Fig. 5. Simulation waveform showing downsizing of a 32-bit input (0x12345678) into four 16-bit outputs.

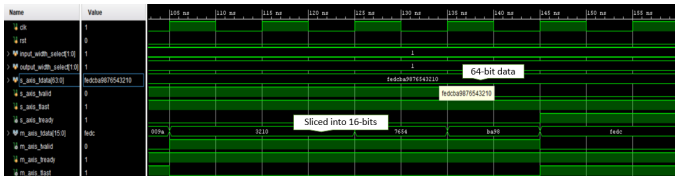

Fig. 6. Simulation waveform showing downsizing of a 64-bit input (0xFEDCBA9876543210) into four 16-bit outputs.

This simulation result demonstrates that the module correctly preserves the data integrity and adheres to AXI4-Stream protocol standards, including proper frame boundary signaling through the `m_axis_tlast` signal.

C. TEST CASE 3: 64-bit to 16-bit Conversion

From the Fig. 6, The simulation waveform illustrates the functionality of the AXI Stream Downsizer configured to convert a 64-bit input stream into 16-bit output chunks. The module is set with the following control signals:

- `input_width_select = 2'b01` ⇒ 64-bit input
- `output_width_select = 2'b01` ⇒ 16-bit output

When both `s_axis_tvalid` and `s_axis_tready` are asserted, the 64-bit input data `0xFEDCBA9876543210` is latched into the internal buffer.As shown in Table III, The downsizer then sequentially slices the buffered data into four 16-bit segments and transmits them on the `m_axis_tdata` line in the following order:

TABLE III
64-BIT TO 16-BIT DOWNSIZER OUTPUT PER CLOCK CYCLE
(0XFEDCBA9876543210)

Clock Cycle	m_axis_tdata Output
1	0x3210
2	0x7654
3	0xBA98
4	0xFEDC

This simulation result demonstrates that the module correctly preserves the data integrity and adheres to AXI4-Stream protocol standards, including proper frame boundary signaling through the `m_axis_tlast` signal.

D. CREATING AN IP MODULE

The AXI Stream Downsizer IP module converts wider AXI4 Stream inputs into narrower outputs while maintaining protocol compliance. The AXI4 Stream Downsizer IP block As shown in the Fig. 7 is intended to transform wider AXI4

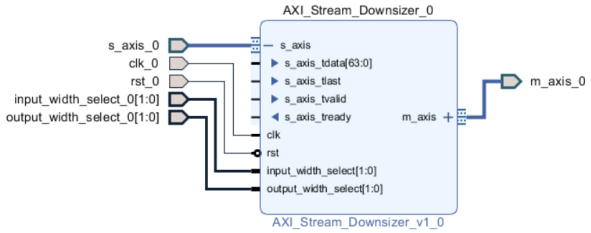

Fig. 7. AXI Stream Downsizer IP module

Stream data inputs into narrower AXI Stream outputs by dividing each input data word into several shorter output pieces.Such a functionality is necessary for bridging data bus mismatched components in AXI4 Stream-based designs and facilitating easy data width conversion without compromising the integrity of data or protocol compliance.

E. PERFORMANCE ANALYSIS

The proposed AXI Stream Downsizer is designed to deliver both low latency and high throughput

Low Latency:

Latency is minimized through a streamlined, pipelined FSM and internal data buffering mechanism. Data is captured as soon as `s_axis_tvalid` and `s_axis_tready` are asserted, and the first output chunk is transmitted in the immediate next clock cycle. In contrast to traditional approaches, which often include deep buffering and complex control logic, our architecture reduces unnecessary delays and reacts quickly to input availability.

High Throughput:

The design achieves high throughput by eliminating idle cycles between chunk transmissions. Once input data is captured, the module continuously emits output chunks over consecutive cycles. For example, during a 64-to-16 bit conversion, the module delivers 4 consecutive 16-bit outputs in 4 clock cycles, demonstrating near-maximum bandwidth utilization. The dynamic ratio computation enables adaptable serialization without stalling the data pipeline.

V. CONCLUSION

AXI4 Stream Downsizer module has been successfully designed, implemented, and verified to allow for smooth adaptation of data widths between AXI Stream interfaces of varying widths. Simulation outputs, as illustrated by the waveform, unequivocally show the proper operation of the module: a 64-bit or 32-bit input data width is properly broken down into successive 16-bit or 8 bitoutput chunks, with correct treatment of the AXI Stream protocol's handshaking and packet boundary signals. The module dynamically controls data buffering, chunking, and flow control to provide safe and lossless data transfer over changing interface widths. This architecture is extremely useful for system-on-chip (SoC) designs where interoperability between modules of varying data bus widths needs to be ensured. As shown in the table IV, the proposed downsizer outperforms traditional width converters in terms of efficiency and output timing.

TABLE IV
COMPARISON BETWEEN TRADITIONAL WIDTH CONVERTERS AND PROPOSED DOWNSIZER

Metric	**Traditional Converters**	**Proposed AXI Stream Downsizer**
Latency	Higher due to multi-stage FIFO buffering	Lower due to pipelined FSM and immediate chunking
Throughput	Moderate; may have idle cycles	High; continuous chunk transfer without gaps
AXI4-Stream Compliance	Partial; limited `TLAST`, `TKEEP` handling	Full support for `TVALID`, `TREADY`, `TLAST`
Scalability	Limited to fixed configurations; needs redesign for changes	Scalable across systems with varying width demands
Design Flexibility	Rigid; often tailored to specific applications	Configurable input/output widths, reusable across platforms
Power Efficiency	May consume more due to idle stages	Power-efficient with potential for clock gating

REFERENCES

[1] J. Lázaro, A. Astarloa, A. Zuloaga, J. Á. Araujo, and J. Jiménez, "AXI Lite Redundant On-Chip Bus Interconnect for High Reliability Systems," *IEEE Transactions*, vol. 73, no. 1, Mar. 2024.

[2] Z. Jiang, N. Audsley, D. Shill, K. Yang, N. Fisher, and Z. Dong, "Brief Industry Paper: AXI-Interconnect RT: Towards a Real Time AXI Interconnect for System-on-Chips," in *Proc. IEEE 27th Real Time and Embedded Technology and Applications Symposium (RTAS)*, May 2021, pp. 437–440.

[3] Y. Zhang, G. Cai, and Z. Huang, "High Performance and Low Area Interconnect Structure Based on AXI," in *Proc. Workshop on Electronics Communication Engineering (WECE 2023)*, vol. 12973, Jan. 2024, pp. 39–45.

[4] C. De Sio, S. Azimi, and L. Sterpone, "On the Evaluation of SEU Effects on AXI Interconnect Within AP-SoCs," in *Architecture of Computing Systems–ARCS 2020: 33rd Int. Conf., Aachen, Germany*, May 2020, pp. 215–227, Springer.

[5] D. Emil, M. Hamdy, and G. Nagib, "Development of an Efficient AXI-Interconnect Unit Between Set of Customized Peripheral Devices and an Implemented Dual-Core RISC-V Processor," *The Journal of Supercomputing*, vol. 79, no. 15, pp. 17000–17019, 2023.

[6] Aashima and L. M. Saini, "Design, Analysis and Verification of AMBA AXI4.0," in *Proc. 2025 Int. Conf. on Visual Analytics and Data Visualization (ICVADV)*, pp. 131–136, 2025.

[7] P. Dukare, A. Gokhale, and V. Ingale, "Development of AMBA ACE protocol," in *Proc. 7th Int. Conf. on Computing in Engineering & Technology (ICCET)*, Online Conference, pp. 222–225, 2022.

[8] Adiya and L. M. Saini, "Overview of AMBA AHB2APB Bridge for SOC Interconnects," in *Proc. 2024 Int. Conf. on System, Computation, Automation and Networking (ICSCAN)*, Puducherry, India, pp. 1–6, 2024.

[9] N. Deshpande and R. Sadakale, "AMBA AHB to APB Bridge Protocol Verification Using System Verilog," in *Proc. 2023 1st Int. Conf. on Advances in Electrical, Electronics and Computational Intelligence (ICAEECI)*, Tiruchengode, India, pp. 1–3, 2023.

[10] N. Gaikwad and V. N. Patil, "Verification of AMBA AXI On-Chip Communication Protocol," in *Proc. 2018 4th Int. Conf. on Computing Communication Control and Automation (ICCUBEA)*, Pune, India, pp. 1–5, 2018.

[11] P. Sivaranjani, S. Sasikala, A. Lavanya, et al., "Design and Verification of Low Latency AMBA AXI4 and ACE Protocol for On-Chip Peripheral Communication," *Wireless Personal Communications*, vol. 136, pp. 1811–1824, 2024.