Breaking the Wait: Customizable, Real-Time Post-Processing for Data Integrity Verification in SerDes Systems

Chandana K N
chandana.kn@intel.com
Intel, Santa Clara, CA, US 95054

Suresh Gandhi S
suresh.gandhi.subramaniam@intel.com
Intel, Folsom, CA, US 95630

***Abstract* - UVM provides a solid foundation for implementing TLM-based scoreboards using reference models that replicate design behavior. However, building these TLM-based reference models becomes increasingly challenging when supporting multiple design standards—such as PCIe, USB, and DP within a single PHY. This paper introduces a novel, approach to verify data integrity in high-speed SerDes systems through post-processing techniques. Unlike traditional post-processing, which requires waiting for the entire test to finish, our method enables users to access intermediate results during test execution. Furthermore, the level of granularity of these intermediate results can be customized by the user. By leveraging Python's built-in libraries, we eliminate the need for complex TLM based reference models, significantly improving both turnaround time and debugging efficiency. Additionally, the approach offers visual debugging tools that enhance data integrity verification, something traditional scoreboards struggle with due to limited observability.**
Keywords – SerDes, Data Integrity, Post Processing

I. Introduction

Serializer/Deserializer (SerDes) systems are essential for transmitting and receiving data over serial links, converting data between serial and parallel interfaces. Ensuring data integrity is crucial for system reliability, especially as data rates increase, leading to higher chances of errors due to jitter, noise, and signal integrity issues.

Traditionally, data integrity is verified using TLM-based (Transaction Level Modeling) scoreboards, which compare transmitted data with expected data from a reference model. However, as protocols and modulation schemes diversify, developing and debugging these reference models becomes as complex as the design itself. This paper explores the benefits and drawbacks of using post processing techniques versus TLM based scoreboards to verify the data integrity of the design.

II. Data Integrity using Standardized Approach

UVM test benches[1] are composed of several components such as agents, drivers, monitors, sequencers, and scoreboards that work together to validate the DUV. Drivers take transactions from the sequencers that generate protocol-compliant transactions and convert them into pin-level activities that mimic the behavior of the SerDes system. Monitors observe the pin-level activities on both the transmitting and receiving ends and extract transactions from these activities. This dual requirement is what makes traditional scoreboards complex and tightly coupled to the protocol. The scoreboard compares original data sent by the driver with data received and processed by the monitor, reporting mismatches as shown in Figure 1.

The complexity of mimicking the design behavior in the Bus Functional Model (BFM) increases significantly when a single subsystem hosts multiple SerDes protocols as supported by PIPE standard like PCIe, DP, USB and so on. Each protocol has its own set of specifications, signaling methods, and data integrity checks. As data rates increase, BFM monitors must be updated to support higher speeds and modulation schemes like PAM2, PAM3, and NRZ. The scoreboard must bridge these two domains: it needs to understand both the protocol-level transactions and the bit-level SerDes data. This requires significant effort to develop TLM based reference models for each protocol, which are as complex as the design itself and closely tied to the DUV. Debugging data integrity mismatches in SerDes systems can be particularly challenging due to the complex modulation schemes, signal integrity issues (jitter), protocol complexity and limited observability. Pinpointing bit-level mismatches within a continuous data stream resembling a clock signal, lacking clear symbol or byte boundaries, is tedious and time-consuming.

To tackle these challenges in multi-protocol SerDes systems, we propose a Python-based post-processing technique for verifying data integrity, independent of the DUV. This approach aims to simplify the process and reduce the development effort needed for protocol-specific TLM based reference models in a UVM testbench.

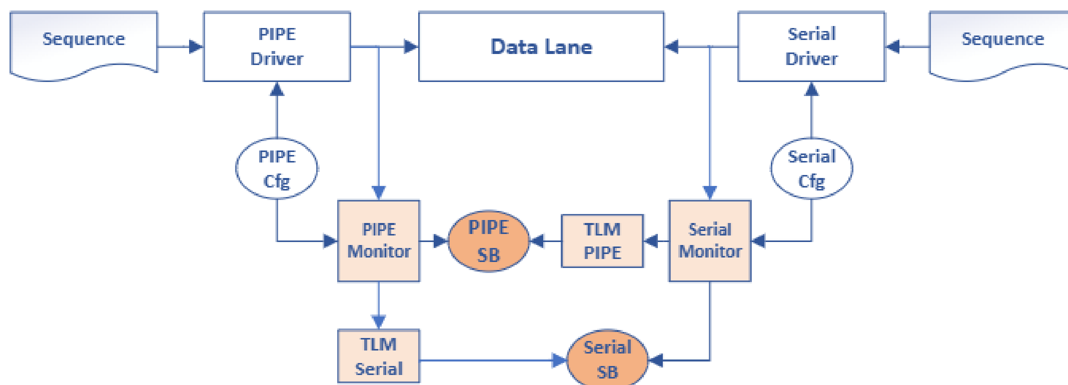

Figure 1 TLM based scoreboard for SerDes Applications

III. Data Integrity using Post Processing Techniques

Postprocessing techniques for data integrity in the context of SerDes and other data transmission systems involve analyzing the received data after it has been captured to ensure that it matches the transmitted data. This is done after the physical transmission and reception processes, hence the term "postprocessing." The goal is to verify that the data has not been corrupted during transmission. We developed a generic Python[2] based script to compare the integrity of the parallel to serial data with coarse and fine granularity based on sliding window mechanism for each burst at the end of each transmission cycle as shown in Figure 2. This approach of enabling per burst comparison eliminates the need to wait until the end of the simulation to verify data integrity failures, a common limitation of post-processing techniques. It also offers a visual debugging tool that generates outputs like tkdiff, providing essential debug information that System Verilog/UVM alone cannot feasibly deliver.

The data to be compared is captured into two text files (*pipe_<laneid>_<burstid>* on the parallel side and *pad_<laneid>_<burstid>* on the serial side) by sampling the interface signals at valid points as per protocol specifications. Raw bit streams are written out by monitors attached to the TX and RX interfaces. These monitors record serialized and parallel data streams into text files, which are then consumed by the Python script. Alternatively, data can be scraped from waveforms using post-processing tools, but direct monitor-based capture is preferred for accuracy and automation. Once the burst of data is completely available, the post processing comparison script is triggered through python system calls with these files and other configurations and tolerance related information as inputs.

A. Post Processing flow

The python based post-processing flow for data integrity verification involves:

1. ***Protocol-Agnostic Data Capture****:* Instead of using protocol-specific monitors, the scripts operate on captured data streams (TX and RX) in their raw form. This information is loaded into text files and is fed to the script for comparison. Each line in the file represents a symbol to be compared. The script converts each hex value to its binary representation, based on the encoding (e.g., 8b, 10b, 16b, 32b, PAM, NRZ). The process does not depend on protocol-specific framing or transaction types—only on the encoding type and data format.

2. ***Data Reconstruction****:* The captured symbols are concatenated to form a long serial bitstream considering protocol-specific framing (number of bits per symbol) or encoding (8b10b, PAM, NRZ etc.). This step reconstructs the serial data that was originally sent and received regardless of the symbol boundaries.

3. ***Data Comparison****:* The serial bitstream is split back into chunks of the encoding width to form a parallel list of data. This allows for symbol-by-symbol comparison, which is useful for score boarding and diff generation. Comparison between the transmitted and reconstructed received data uses a sliding window mechanism, with a provision for both coarse and fine granularity checks (per burst, per symbol, per lane). The sliding window mechanism and configurable tolerances allow the script to align and compare data streams regardless of protocol framing or boundaries.

4. ***Tolerance handling***: The comparison occurs at the end of each burst or for a specified number of symbols, accommodating various tolerances for mismatches. The script may trim extra bits at the start or end (junk data) based on configurable tolerances, ensuring alignment for comparison.

*5. **BER Handling (if enabled)***: For certain modes/rates, the script may reverse the bit order or perform additional processing for BER calculation. It writes the reconstructed bit streams to files for BER analysis.

*6. **Error Identification and Reporting**:* The comparison script uses *HTML* diff inbuilt Python libraries to visually highlight and tag failure points at the bit level granularity with the symbol count information as shown in Figure 3. This facilitates rapid debugging and visual comparison, surpassing traditional scoreboard methods of tracing bit level mismatches from the waves alone.

The comparison is purely data driven. To support a new protocol, we typically only need to adjust the data capture and encoding parameters, not rewrite the entire verification logic. Post-processing is best for data integrity and bit-accurate checks, but not for all protocol or timing checks.

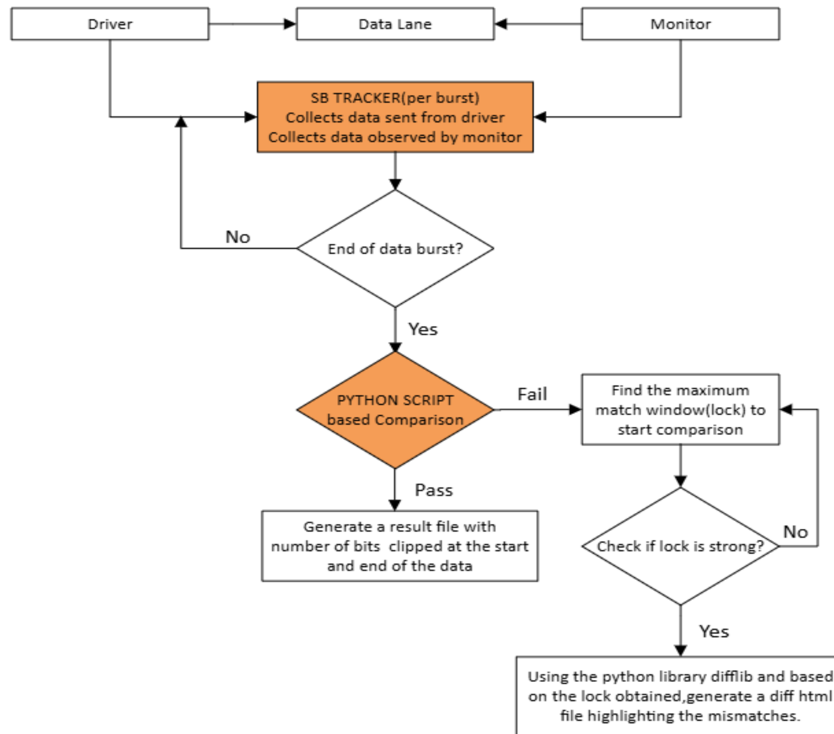

Figure 2 Python script-based scoreboard

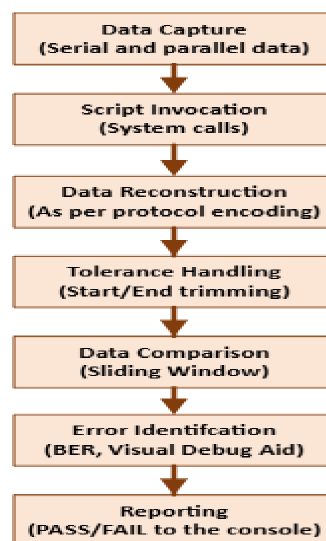

Figure 3 Post Processing script flow

B. Sliding Window Mechanism

The **sliding window** mechanism is a key technique used in the Python scoreboard scripts for aligning and comparing data streams, especially when there may be offsets, extra bits, or mismatches between the transmitted and received data. A fixed-size segment (window) of one data stream is taken and compared bit-by-bit across another data stream to find the best possible alignment or match. This is particularly useful in high-speed IO and SerDes verification, where data may not be perfectly aligned due to protocol overhead, encoding, or physical layer effects. By finding the best alignment, the user can pinpoint exactly where mismatches occur, making it easier to debug data integrity issues.

The steps involved are as follows:

- The script takes the serialized pipe data and the reference (pad) data.
- It tries to align the two by moving a window of a certain length across the pad and pipe data.
- For each position, it checks if the windowed segment matches or calculates a similarity ratio (using SequenceMatcher from difflib Python library).
- If a good match is found (ratio above a certain threshold), it reports a pass; otherwise, it continues sliding the window.
- If a segment matches well enough with few mismatches, it generates a diff report using HtmlDiff from python difflib and flags the result.
- It can handle cases where there are extra bits at the start or end, or where the streams are not perfectly synchronized.
- The window size can be adjusted dynamically based on encoding or protocol requirements.

As an example, consider two streams to be compared as:

TX data: *ABCDEF*, RX data: *XXABCDEFYY*

A direct comparison would fail due to the extra XX at the start and YY at the end. With a sliding window of size 6:

- Window 1: *XXABCD* (no match)
- Window 2: *XABCDE* (no match)
- Window 3: *ABCDEF* (match!)

The script would report a pass at window position 3.

C. Visual Debug Aid

The visual debug aid refers to the use of **HTML diff files** generated by Python scripts during post-processing. These files provide a graphical, interactive way to inspect and debug data integrity mismatches between transmitted and received data streams in SerDes systems.

After capturing and reconstructing the TX and RX data streams, the script compares them using a sliding window mechanism. This comparison is performed at configurable granularities (per burst, per symbol, per lane, etc.). When mismatches are found, the script uses Python's built-in libraries (such as difflib.HtmlDiff) to generate an HTML file that visually highlights the differences between the two data streams.

The HTML diff file displays the TX and RX data side by side. Matching segments are shown normally, while mismatches are highlighted (often in color). The file includes context such as burst number, symbol count, and bit position, making it easy to locate and understand the error. Instead of manually tracing bit-level mismatches in waveforms, engineers can open the HTML file in a browser and instantly see where and what the mismatches are. This is especially useful for long data streams or when the error is subtle (e.g., a single bit flip in thousands of bits) and can dramatically reduce debug time, especially for complex or intermittent issues as highlighted in TABLE I.

Suppose you have a mismatch in burst 2, symbol 0 as shown in Figure 4. The HTML diff will:

- Show the TX(RHS) and RX(LHS) data for that burst/symbol.
- Highlight the mismatched bits in red.
- Provide navigation links (f for first, n for next etc.) and context so one can quickly jump to the relevant section.
- The symbols number can be used to pinpoint the burst count in the waveforms for confirmation.

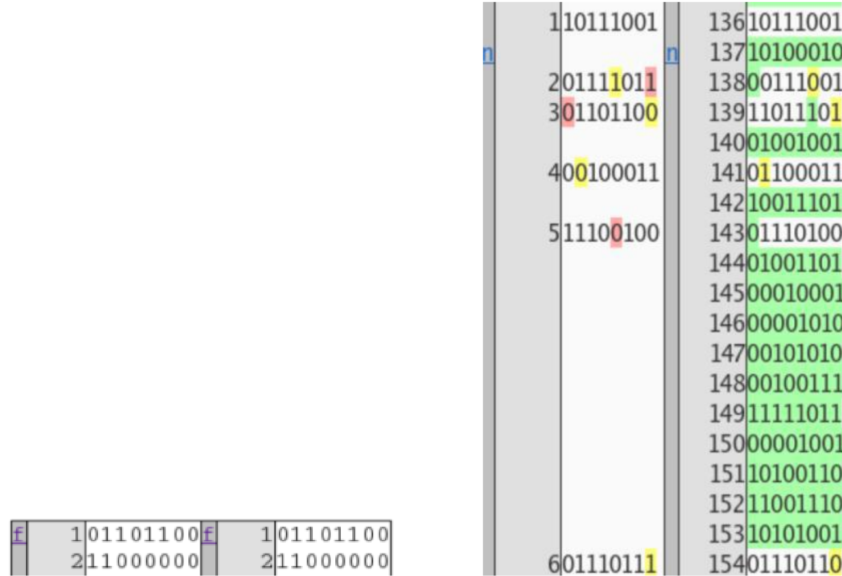

Figure 4 Sample HTML diff (Left: No mismatches, Right: With mismatches)

TABLE I
Comparison of HTML Diff vs Traditional Scoreboard

Feature	TLM based Scoreboard	HTML Diff Scoreboard
Bit-level pinpointing	Manual, slow	Automatic, visual
Protocol dependency	High	Low (agnostic)
Debug time	Long	Short
Usability	Low	High
Realtime feedback	Yes, by default	User configurable
Data capture overhead	Minimum	Raw data stored to disk

IV. Results and Conclusion

The integration of Python script-based scoreboards into SerDes test benches has proven to be a transformative approach, effectively replacing complex TLM-based scoreboards over the past five years. This shift dramatically reduced debug time for complex issues—from weeks to mere days—by eliminating the need to debug the reference model itself. This technique comes with **12% improvement in simulator performance** by eliminating the TLM ports and offloading the comparison to the scripts which can run independently after the simulation.

The HTML diff-based files generated through post-processing techniques further streamlined the debugging process. While post-processing techniques are widely used across various verification domains, our method overcomes the traditional challenge of waiting until the end of simulation, enabling more efficient and timely data integrity verification. The benefits achieved by using this approach are:

1. ***Reduced Complexity****:* Eliminates the need for complex protocol-specific TLM based reference models.

2. ***Protocol Flexibility****:* The process does not depend on protocol-specific framing or transaction types—only on the encoding width and data format. The same verification environment and the script can be easily used for different protocols, making it more flexible and reusable.

3. ***Faster Development****:* Development time is reduced since the focus is on verifying the SerDes behavior rather than the protocol specifics. No need to update or debug multiple reference models when protocols change or new ones are added.

4. ***Scalability****:* The technique can scale changes in data rates and modulation schemes without significant changes to the verification environment.

5. ***Ease of Debug****:* GUI based log files (HTML diff) generated using Python libraries pinpoint exact failure locations at the bit-level mismatch, including burst and symbol count. Focus is on data integrity, not protocol compliance, making it easier to pinpoint and fix issues.

6. ***Granularity***: Users can enable comparisons per burst, per lane, and per link, eliminating the need to wait for the script to run until the end of the simulation.

Though traditional scoreboards provide tight protocol integration, along with functional coverage hooks, the script-based approach can be extended for integrating advanced coverage and functional safety metrics into the post-

processing flow. This paper stresses the advantages and applications of Python-based post-processing data integrity verification for SerDes based protocols; it can potentially be extended to any protocol requiring robust data verification.

ACKNOWLEDGMENT

We thank Batmanaban Pourouchottaman, Pavitra Balasubramanian and Anargha Jatheendran for their expertise and assistance throughout all aspects of our study and for their help in bringing up the concept to implementation.

REFERENCES

[1] IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE Standard 1800.2-2017

[2] https://docs.python.org/3/tutorial/