

Implementing and Verifying RISC-V Nexus Trace Compliant Trace Encoder for High Performance Cores

Sajosh Janarthanam, Rahul Behl, Sharanesh R, Hitesh Pavan Oleti
Tenstorrent

Abstract- Processor trace is a non-intrusive method used for debugging, profiling, and code coverage in RISC-V systems. This paper presents a Nexus-compliant trace infrastructure for high-performance RISC-V cores that support multiple extensions and custom instructions. The design captures multiple branch retirements per cycle while limiting input to two branch records per cycle for efficiency. Key features include early stall determination to prevent packet loss, Branch History Messaging for compression, and virtualization support through VMID and ASID tracking. Trace data is encoded into fixed-length frames for efficient transmission. The trace sink operates in both SRAM and SMEM modes, using SRAM as a buffer to mask SMEM access latency. The trace encoder is verified using a custom C++ model and a two-step verification approach that checks individual trace message fields and validates the reconstructed program counter stream. The approach supports validation across multiple DUT configurations and trigger sources, ensuring functional accuracy and broad integration-level coverage.

I. INTRODUCTION

The escalating complexity and performance demand of modern high-performance processor cores necessitate advanced methodologies for their design, validation, and optimization. Traditional debugging techniques often fall short in providing the holistic, non-intrusive insights required to fully comprehend intricate program behaviors and subtle hardware-software interactions. In this context, **processor instruction trace** emerges as an indispensable technique, offering unparalleled visibility into the dynamic execution flow of a program.

Instruction tracing is an architecturally non-intrusive method that captures, encodes, and transmits a precise record of program execution. Unlike breakpoints or single stepping, which can alter timing and mask elusive bugs, trace solutions operate passively, allowing the core to run at full speed. This inherent non-intrusiveness makes instruction trace a powerful tool for various critical phases of the processor lifecycle, including:

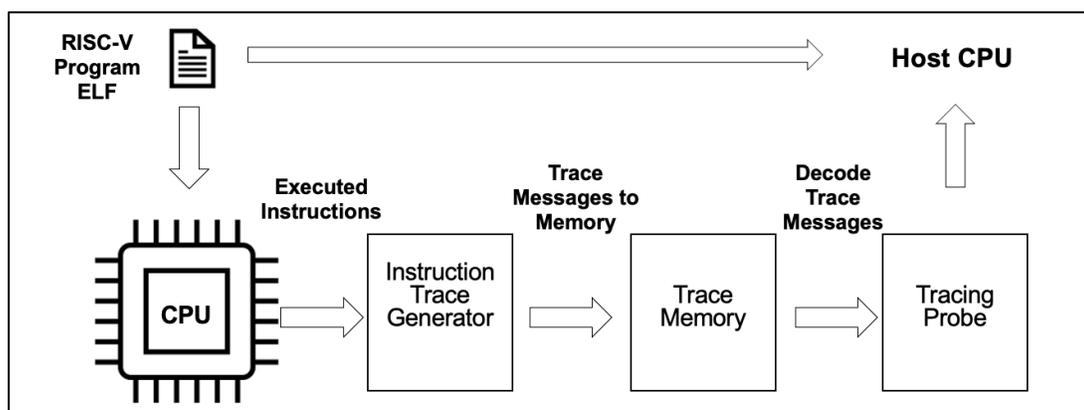


Figure 1. Processor Instruction Trace Overview.



- **Silicon Debug:** Providing a complete historical sequence of executed instructions, enabling forensic analysis of elusive bugs, crashes, and unexpected program behavior that are difficult to reproduce or diagnose with conventional methods.
- **Performance Profiling:** Offering cycle-accurate insights into instruction execution patterns, identifying bottlenecks, stalls, and inefficient code segments, thereby facilitating targeted software and hardware optimizations.
- **Code Coverage Analysis:** Verifying that all intended code paths have been exercised during testing, crucial for validating test suite completeness and ensuring the robustness of critical safety or security applications.

II. LITERATURE SURVEY

Instruction trace mechanisms for RISC-V have gained attention for debugging, performance profiling, and validation. Laghi et al. [1] implemented a trace system in the CVA6 core based on the RISC-V Efficient Trace specification, achieving high compression with low overhead, but it supports only single branch retirement and lacks virtualization or advanced traffic control. Gamino del Río et al. [2] introduced a transparent tracing method with a dedicated pipeline, avoiding instrumentation overhead, but it doesn't handle multi-branch trace capture or compression. Richter et al. [3] proposed DeTRAP for securing return addresses, but it doesn't focus on comprehensive instruction tracing. The RISC-V Trace Task Group [4] defines a trace encoding and compression standard [5], though it doesn't address core-level integration challenges. In contrast, our N-trace infrastructure enables multiple branch retirements per cycle, supports virtualization (VMID/ASID), and integrates early stall detection and fixed-length trace framing. It also incorporates advanced compression and a scalable verification framework addressing the limitations in performance, scalability, and system context awareness seen in prior works.

III. TRACE ARCHITECTURE OVERVIEW

With the advent of the open-source RISC-V Instruction Set Architecture (ISA), the ecosystem has witnessed a surge in diverse core implementations, ranging from simple microcontrollers to complex out-of-order, multi-core System-on-Chips (SoCs). This flexibility, while a core strength, introduces unique challenges in ensuring consistent and effective debugging and validation solutions across disparate designs. To address this, standardization efforts like the RISC-V Nexus Trace (N-Trace) compliant with IEEE-5001 Nexus Standard have become paramount. The RISC-V N-Trace specification outlines a complete, end-to-end trace system for RISC-V cores, harts and System-on-Chip (SoC)/Microcontroller Unit (MCU) designs. This architecture is founded on the established Nexus IEEE 5001 trace standard, and its documentation serves as a guide for N-Trace encoder logic/IP developers, validation teams, and debug and trace tool developers.

The N-Trace architecture is specifically designed to support high-performance RISC-V cores, incorporating mechanisms to handle the complexities of modern processor architectures, including multiple instruction set extensions and custom instructions. The specification details how various program flow changes, such as direct conditional branches, indirect unconditional jumps, exceptions, and interrupts, are precisely traced. Designed for efficiency and enhanced trace compression, the N-Trace architecture defines two primary trace modes: Branch Trace Messaging (BTM) and History Branch Messaging (HTM), with HTM notably offering significantly better compression capabilities. To further minimize trace bandwidth, the architecture integrates advanced compression techniques such as Address Compression and Virtual Addresses Optimization. Additionally, N-Trace supports trace messages that convey essential contextual information, including privilege mode and scontext/hcontext CSR (Control and Status Register) values. This enables trace decoders to accurately associate program execution with specific processes or hypervisor/OS contexts.

This robust and high-performance trace infrastructure is crucial for advanced debugging, profiling, and verification of next-generation RISC-V processors. The specification provides detailed decoding guidelines that explain the algorithm for reconstructing the complete PC flow from the trace messages. This reconstruction process relies on

understanding instruction sizes, types, and branch offsets from the executed code. The architecture also considers aspects like validation and outlines potential future enhancements, such as data trace and system bus trace capabilities

IV. TRACE COMPONENTS OVERVIEW

Instruction execution information from the RISC-V harts is first captured and processed by the Branch Target History Buffer (BTHB), which records branch outcomes and instruction counts. This data is then fed into the N-Trace Encoder, which generates compressed trace packets according to the N-Trace protocol. These packets, along with any relevant vendor-defined or error messages, are then routed through the Trace Network, which includes Trace Hops and Trace Network Interfaces, for aggregation and transmission. The aggregated stream is managed by the Trace Funnel, which can combine multiple trace sources and channels. Finally, the combined trace stream is sent to the Trace Sink, where it is stored in either dedicated on-chip SRAM or buffered for streaming to system memory (SMEM). Additionally, JTAG TAP access provides an out-of-band mechanism for direct configuration and reading of trace-related MMRs and status registers. The system emphasizes robust, lossless transmission through throttling and buffering mechanisms along the entire trace path.

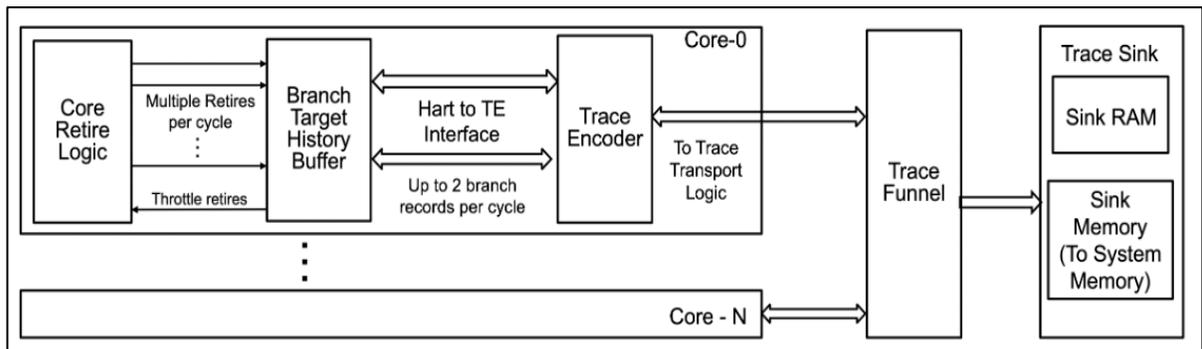


Figure 2. Instruction Trace Hardware Block Diagram.

V. N-TRACE ENCODER MICRO-ARCHITECTURE

The Trace Encoder implements a three-stage pipelined architecture (namely TE0, TE1, TE2) designed to handle the high-throughput requirements of modern processors while maintaining compliance with the N-Trace specification. The MSEO module provides the critical message framing functionality required for proper trace reconstruction and efficient data transmission. Together, these components can enable non-intrusive instruction tracing with zero performance impact on normal program execution while operating without any throttle/stall with appropriate configurations.

A. Three-Stage Pipeline Design

The Trace Encoder employs a sophisticated three-stage pipeline architecture optimized for high-throughput processing of trace data. The pipeline stages are designed to handle the complex requirements of N-Trace packet generation while maintaining low latency and high efficiency.

- **Stage TE0:** The initial stage performs critical computations including instruction count (ICNT), history tracking (HIST), and context field generation. This stage implements variable-length field calculations with XOR compressions, packet framing initialization, and control logic for packet type detection. It also manages Memory-Mapped Register (MMR) write operations and provides the necessary controls for subsequent processing stages.

- Stage TE1: The middle stage focuses on Message Start/End (MSEO) computation, which is essential for N-Trace protocol compliance. This stage handles repeat branch packet controls, packet framing operations, and periodic synchronization counter management. The generated packets coming out of packet framing logic are handled by the FIFO control logic to buffer them in TE2 stage.
- Stage TE2: The last stage manages output control and error handling mechanisms. It implements FIFO pop control logic for managing data flow to the Trace Transport Logic, flush control mechanisms for error recovery, and stall detection logic for backpressure management. This stage ensures robust operation under high traffic conditions and provides the interface to downstream components.

B. Dual Parallel Packet Generators

The encoder architecture incorporates two parallel packet generators to efficiently handle the complex requirements of modern processor architectures. This design enables the processing of up to two branch records per cycle, significantly improving throughput compared to single-path implementations. The parallel architecture supports separate processing paths for different trace events, reducing latency for complex trace scenarios, and enabling efficient resource utilization. The control logic that tracks ICNT, history, and context field computations is enabled to support two independent message generation with accurate information. This parallel approach allows the encoder to maintain high throughput even under peak load conditions, making it suitable for harts with multi-wide retires, and SoC environments where multiple cores may generate trace data simultaneously.

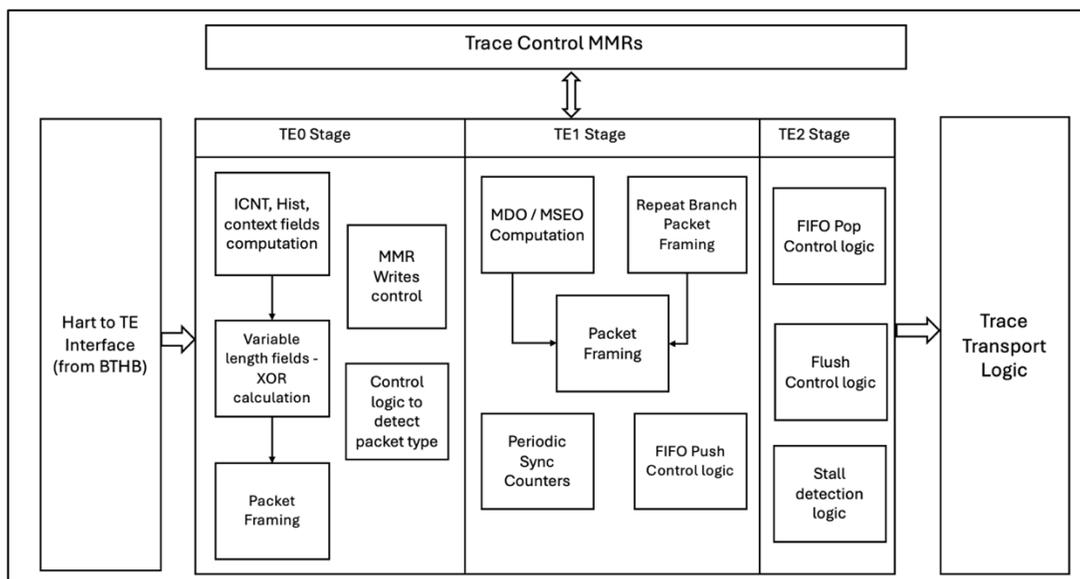


Figure 3. N-Trace Encoder Block Diagram.

C. Backpressure Management

The encoder implements sophisticated backpressure management mechanisms to ensure lossless data transmission. The architecture includes an internal FIFO buffer that stores trace messages, enabling the system to handle peak trace traffic without data loss. Stall detection logic monitors pipeline conditions and triggers appropriate responses when backpressure is detected. The system employs a multi-level throttling mechanism that propagates backpressure from the Trace Transport Logic through the entire pipeline to the retire logic. This ensures that when downstream components cannot process data fast enough, the core can throttle its trace generation, prevent data overflow and maintain system integrity.



D. MSEO Block

The MSEO module provides critical message framing capabilities required for proper N-Trace protocol implementation. This module computes Message Start/End data that enables trace reconstruction tools to properly interpret and decode trace messages. The MSEO functionality is essential for maintaining trace data integrity and ensuring accurate program counter reconstruction. It handles three primary functions: Message Start (MS), Message End (ME), and Message Data Out (MDO). Message Start indicators signal the beginning of a new trace message sequence, while Message End indicators denote the completion of a message, along with trace message sent as part of Message Data.

E. Protocol Compliance

The Encoder module ensures full compliance with the RISC-V N-Trace specification by implementing all required message framing requirements as per History Tracking (HTM) mode. The module supports all nine transport codes (TCODE) defined in the specification, including PROGTRACESYNC, OWNERSHIP, INDIRECTBRANCHHIST, RESOURCEFULL, ERROR, INDIRECTBRANCHHISTSYNC, REPEATBRANCH, PROGTRACECORRELATION and VENDORDEFINED

The module also implements all nine synchronization codes (SYNC) for proper trace synchronization, including external triggers, reset exit, periodic sync, debug mode, instruction overflow, trace events, FIFO overflow, and power-down scenarios. This comprehensive implementation ensures that the trace system can handle all possible execution scenarios and provide complete trace coverage.

F. Scalability and Multi-core Support

The encoder architecture is designed for scalability across multiple cores in modern SoC designs. Each core can have its own dedicated BTHB and Trace Encoder, enabling independent tracing without interference between cores. The architecture supports configurable core throttling mechanisms that allow individual cores to adjust their trace generation rate based on system conditions.

VI. VERIFICATION STRATEGY

Accurate trace generation and verification are essential for debugging, profiling, and ensuring correctness in modern processor architectures, particularly in the context of high-performance and heterogeneous systems. In recent years, instruction trace encoders compliant with industry specifications such as Arm's CoreSight and now the emerging RISC-V Nexus Trace standard have become central to observability infrastructures. Verification of such encoders demands rigorous modelling, multi-layer trace validation, and robust support for cross-DUT applicability. To address these needs, we present a scalable and comprehensive verification methodology for our RISC-V Nexus Trace-compliant encoder, designed for high-throughput RISC-V implementations. Our strategy mirrors best practices in formal and dynamic trace validation, leveraging both functional simulation and post-silicon debug requirements to ensure trace integrity.

The implemented N-Trace encoder is validated using a custom-built, in-house C++ reference model that emulates both encoder and decoder logic. This model serves as a golden reference for verifying field-level accuracy of trace messages and the correctness of the reconstructed program counter (PC) stream. The verification flow employs a two-step mechanism: first, it checks the correctness of trace messages at the field level (Message-Stream check), and second, it performs full PC stream reconstruction and comparison against a reference PC sequence (PC-Stream check). This dual-layer approach guarantees fine-grained validation while maintaining architectural correctness across a wide range of instruction sequences and core retire behaviour. All the supported Ntrace message types are thoroughly verified. Periodic synchronization messages and trace filtering mechanisms are also supported and verified through integrated checkers.

Our verification infrastructure is built for scalability across design abstraction levels, from unit-level testing without a core to integration-level testing with full system configurations. The verification framework supports ELF-based

stimuli generation and post-processing-based trace validation, making it easily portable across different designs. Furthermore, it handles trace start/stop triggers from diverse sources (Hardware and Software debug Triggers) and external asynchronous signals. Special emphasis is placed on validating transitions in and out of debug mode, as well as the generation and propagation of trigger-based trace packets. This verification strategy not only improves trace fidelity and visibility but also lays the foundation for robust silicon bring-up, runtime observability, and high-performance profiling in next-generation RISC-V systems. With the recent ratification of the RISC-V trace specification, our encoder implementation is aligned with the standard.

A. Ntrace Verification Flow

Ntrace design verification environment provides a scalable framework for validating diverse DUT configurations. It supports both ELF-based and retiree log-based stimulus, with a reusable post-processing checker infrastructure across DUTs. Figure-4 illustrates the overall Verification flow.

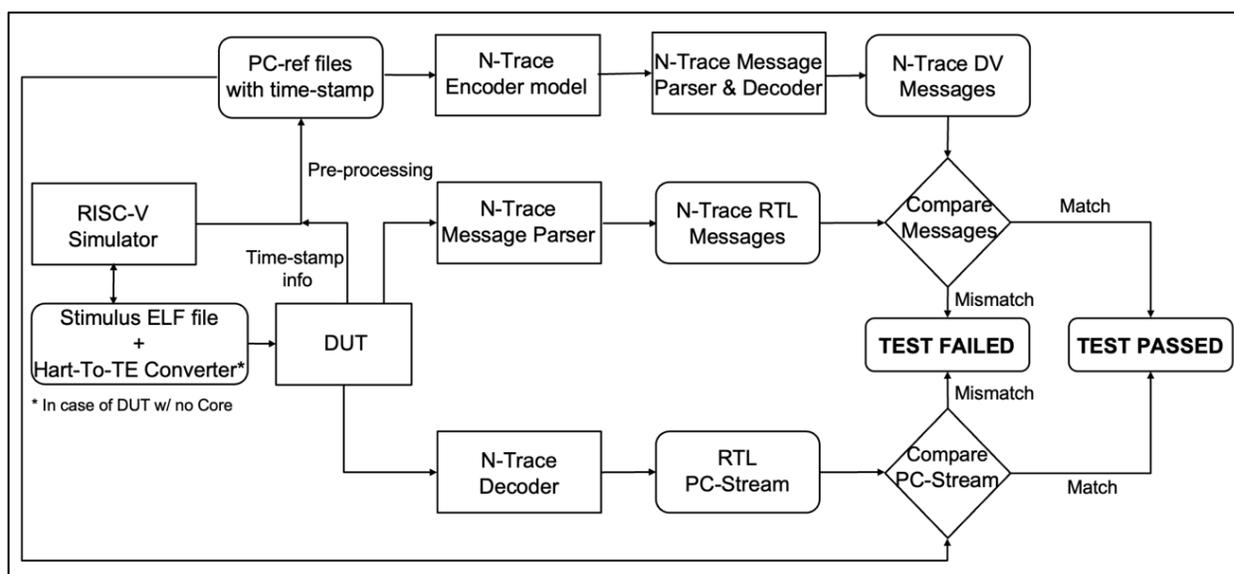


Figure 4. N-Trace Verification Flow Diagram.

Stimulus (either ELF or Hart-to-TE log) is driven to the DUT. During test execution, trace data is captured in real time based on the active trace mode and logged into Ntrace message dump files using dedicated monitors. At the end of the trace programming sequence, the Ntrace checker is invoked via a DPI call. The checker consumes the RTL generated binary trace stream and initiates the Ntrace software encoder model upon encountering a trace-start message. This model generates the expected message stream, which is then compared against the actual RTL message stream. After validation of the trace messages, the encoded RTL stream is passed to the Ntrace decoder to reconstruct the actual program counter (PC) stream. This reconstructed stream is then compared with a golden reference PC stream produced by a RISC-V simulator. The test is considered a pass only if both the message stream and PC stream validations succeed, alongside all functional checks.

Ntrace DV Components: The Ntrace DV flow is built around three key components namely the Hart-to-TE Converter, the Ntrace Monitor, and the Ntrace Checker.

- 1) **Hart-to-TE Converter:** Since the Ntrace Encoder requires stimulus in the Hart-to-TE format, ELF files are first executed through a RISC-V simulator to generate retire logs. These logs are then converted into the required format by the Hart-to-TE converter, ensuring compliance with the Ntrace specification for encoder input.

- 2) **Ntrace Monitor:** This component includes the Retire Instruction Monitor, which captures retire data such as privilege level, context, and trap status, and the Trace Funnel Monitor, which samples trace control and RAM read operations. It dumps trace output into Ntrace data files and triggers the checker via DPI calls at the end of trace collection.
- 3) **Ntrace Checker:** The checker validates the correctness of trace encoding and supports multi-core verification. It includes three main subcomponents:
 - a) **Ntrace Encoder:** A software model that encodes the PC stream into binary messages following the Ntrace specification. By default, it operates in HTM mode, but it can be configured to operate in BTM mode and to generate RepeatBranch, History-Repeat, and Periodic Sync messages.
 - b) **Ntrace Message Parser:** Parses the binary message stream into structured message representations. It performs format validation and generates human-readable dumps to aid debugging.
 - c) **Ntrace Decoder:** Reconstructs the PC stream from the encoded messages and verifies it against the reference PC sequence. It supports all message types defined in the Ntrace specification and provides statistics such as decoded instruction count, message count, compression ratio, and encoding efficiency.

The unit level multicore testbench implementation of the above mentioned Ntrace verification environment can be seen in Figure-5.

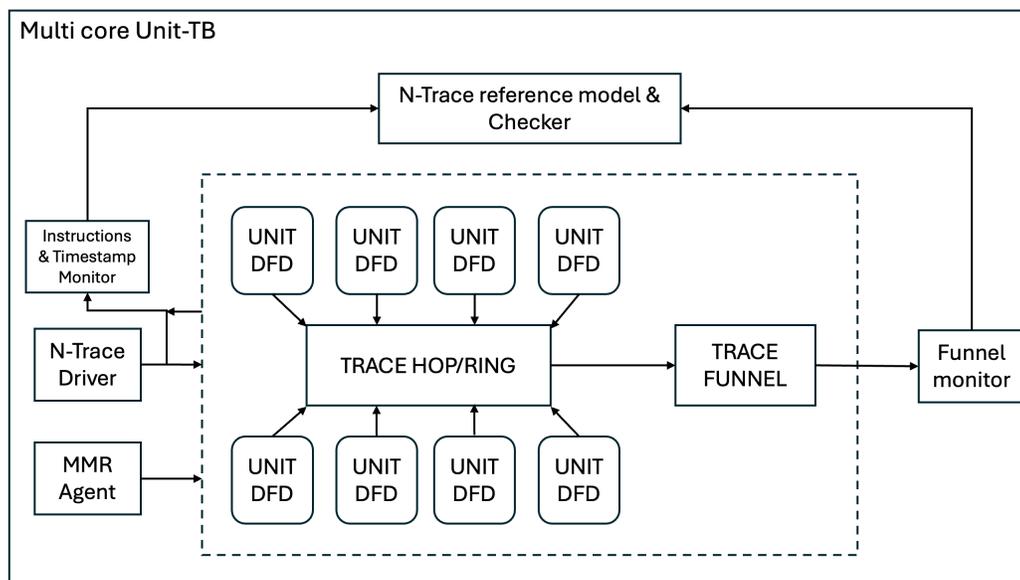


Figure 5. N-Trace Unit level Test bench Diagram.

Two-Step Verification: To improve debug efficiency, we implemented a two-step verification approach. The software encoder generates the expected message stream, which is then compared with the message stream produced by the design. This enables field-level granularity in checking, helping accurately locate design bugs through field mismatches and corresponding timestamps. In contrast, PC-only verification offers limited visibility and may miss lost messages, especially during trace stop events. The two-step method ensures comprehensive coverage and simplifies debugging.

Checker Resynchronization: While the Ntrace Encoder supports trace-notify and periodic sync messages, their generation can be unpredictable due to asynchronous triggers and design-specific behaviour. Modelling these messages would tightly couple the DV with the design. To avoid this, the checker resynchronizes at each periodic sync or notify message, regenerating the expected stream from that point. This supports reliable two-step verification for both periodic-sync and trigger modes while maintaining Design-DV isolation.

B. N-Trace Programming Sequence

TABLE I
N-TRACE PROGRAMMING SEQUENCE

	Trace Programming	Description
1	Trace Funnel & RAM configurations	Ntrace RAM configurations <ul style="list-style-type: none"> • RAM mode (SRAM/SMEM) • RAM wrap mode (Stop-on-wrap/Overflow) • RAM pointers (start, limit, read and write pointers) • RAM frame length Trace Funnel configurations <ul style="list-style-type: none"> • Disable inactive trace inputs of funnel
2	Ntrace Encoder mode configurations	Configure operating modes supported by Ntrace Encoder <ul style="list-style-type: none"> • Periodic-sync mode • Filter mode • Triggers (Hardware / Software) • Frame mode
3	Enable Ntrace	Enable Ntrace RAM, Funnel and Encoder
4	Disable Ntrace	<ul style="list-style-type: none"> • Disable Ntrace RAM and check for RAM empty condition • Disable trace Funnel • Disable Ntrace Encoder and check for Encoder empty condition
5	Read Trace SRAM Content	SRAM mode <ul style="list-style-type: none"> • Read encoded data from Ntrace SRAM and trigger checker SMEM mode <ul style="list-style-type: none"> • Monitor encoded data sent to system memory and trigger checker

C. Functionality checks on RTL trace data

TABLE II
FUNCTIONALITY CHECKS ON RTL TRACE DATA

	Trace Functionality checks	Description
1	Ntrace software model and decoder functionality checking	Validates software encoder functionality by comparing its decoded PC-stream with reference PC-Stream (triggered during each resync)
2	Two step verification - Message and PC stream checking	Compare individual fields of all the generated Ntrace messages and compare PC stream (pc-pc checking)
3	Transport code check	TCODE should fall in one of the supported Ntrace messages
4	Field count check	Field count should match with that of the corresponding Ntrace message
5	Timestamp continuity check	Checks for monotonic increment in timestamp field of the Ntrace messages
6	Ntrace message ordering checks	<ul style="list-style-type: none"> • Error → Error or ProgTraceSync or ProgTraceCorrelation • ProgTraceCorrelation → ProgTraceSync or ProgTraceCorrelation • Sync (ProgTraceSync, IndirectBranchHistSync) → Ownership • RepeatBranch → Error or IndirectBranchHist or ProgTraceCorrelation
7	Periodic sync message count check	Validates Periodic-sync message count consistency between Design and DV
8	Sync field check for IBHS variants	Sync field check for IBHS message variants (Sequential-Instruction-Counter or Periodic-Sync or Trace-Notify message)



VII. CONCLUSION

In conclusion, the Ntrace system offers a balanced solution for efficient trace generation and reliable verification. The three-stage pipelined encoder, with dual packet generators, achieves high throughput while maintaining low latency, allowing up to two branch records to be processed per cycle. On the verification side, a streamlined approach using both ELF-based and retire-log based stimuli, along with a post-processing checker, ensures key functional aspects are validated. Basic message and PC stream checks, combined with support for essential trace message types and trigger mechanisms, provide sufficient coverage for current use cases. Overall, the design and verification efforts together establish a solid foundation for trace functionality across diverse DUT configurations.

REFERENCES

- [1] U. Laghi, S. Manoni, E. Parisi, and A. Bartolini, *Efficient Trace for RISC-V: Design, Evaluation, and Integration in CVA6*, arXiv preprint arXiv:2504.01972, 2025.
- [2] Gamino del Río, I.; Martínez Hellín, A.; R. Polo, Ó.; Jiménez Arribas, M.; Parra, P.; da Silva, A.; Sánchez, J.; Sánchez, S. *A RISC-V Processor Design for Transparent Tracing*. *Electronics* 2020, *9*, 1873.
- [3] I. Richter, J. Zhou, and J. Criswell, *DeTRAP: RISC-V Return Address Protection With Debug Triggers*, arXiv preprint arXiv:2408.17248, 2024.
- [4] RISC-V International, *The Importance of a Standardized Processor Trace*, March 2020.
- [5] RISC-V International, *RISC-V N-Trace (Nexus-based Trace) Specification*.