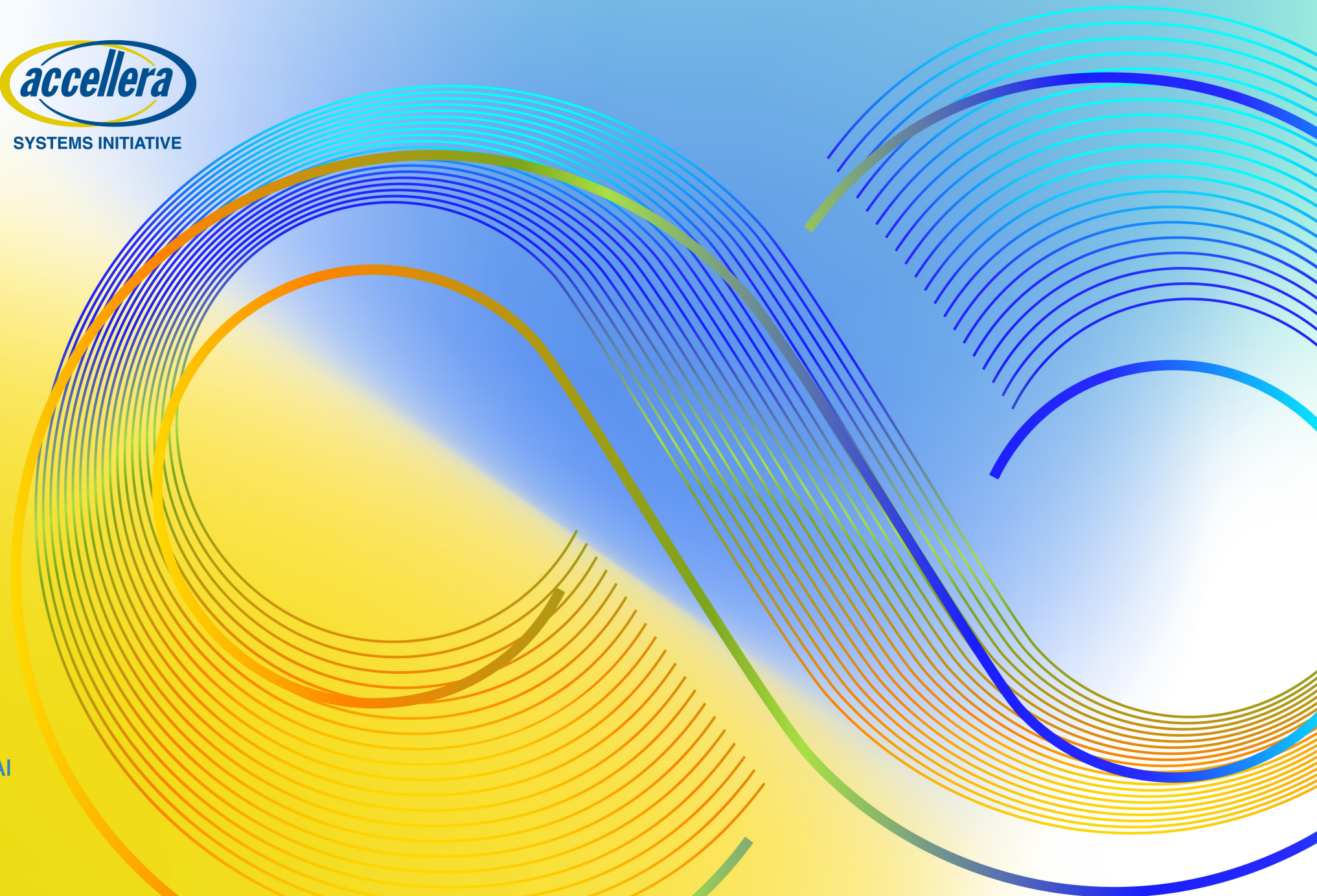
2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

16 Apr 2025 上海 SHANGHAI



2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

A System-Level Random Verification Method for Multi-threaded Processors

Speaker： Li Zhixian

Date: 2025.04.16

16 Apr 2025 上海 SHANGHAI

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

CATALOGUE

16 Apr 2025 上海 SHANGHAI

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

Introduction to Multi-threaded Processor Verification

16 Apr 2025 上海 SHANGHAI

Development and Verification Needs

01

Rapid Growth of AI and Chip Architectures

With the surge of AI, multi- threading in computing has become crucial. It enhances performance but complicates verification, making system- level verification indispensable.

Traditional verification methods fail to meet modern demands due to the dynamic and uncertain nature of real-world applications.

02

Challenges in System-Level Verification

System- level verification generates realistic stimuli, covering all core pipeline modules. However, creating random and valid stimuli for multi-threaded processors is a significant challenge.

Engineers struggle with intricate interactions like memory access synchronization and avoiding infinite loops in random jumps.

03

Importance of Efficient Verification

Efficient verification is vital for timely chip tapeout and reducing failure rates. It ensures that processors meet design specifications and perform reliably in various scenarios.

The proposed method aims to address these challenges by offering a versatile and highly random system- level verification approach.

Contributions of the Proposed Method

Innovative Verification Framework

This paper introduces a system-level random stimulus generation framework using instruction-set modeling. Key innovations include modular API encapsulation for flexible instruction configuration and an automated multi-threaded conflict resolution mechanism to ensure reliability. The framework employs hierarchical constraints to balance randomness with legal instruction flow boundaries, enabling efficient and controlled test stimulus generation for processor verification.

Enhanced Debugging Support

A closed- loop debug support mechanism is established, integrating stimulus generation and debugging.
This significantly reduces debugging costs for complex instruction streams and improves defect localization efficiency.

Practicality and Effectiveness

Extensive experiments demonstrate the method's ability to uncover previously undetected design flaws.
It shows significant improvements in verification efficiency and comprehensive coverage of the instruction set architecture.

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

Methodology of the Verification Framework

16 Apr 2025 上海 SHANGHAI

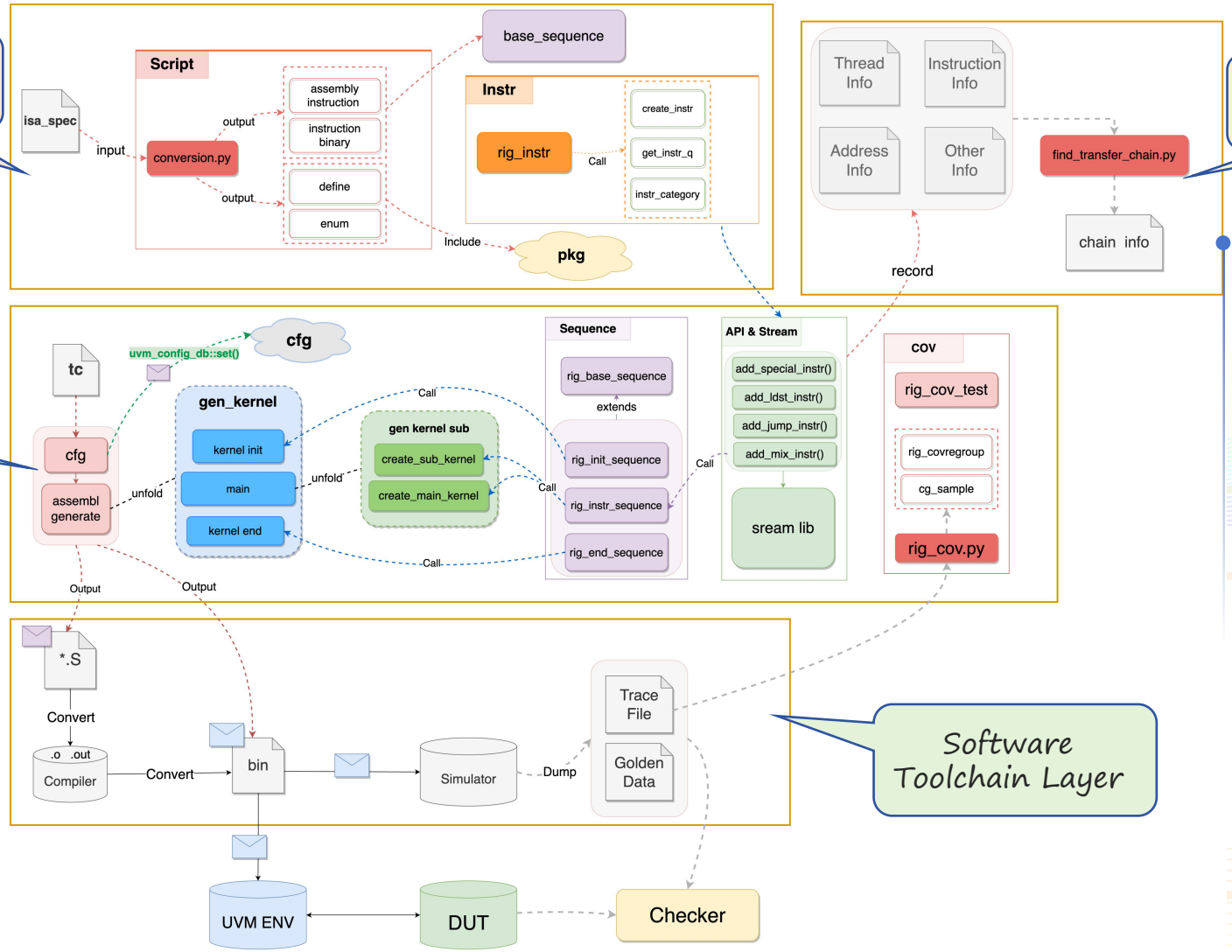
Framework Overview

Instruction Set Abstraction Layer

Instruction Stream Randomization Layer

Debugging Support Layer

Software Toolchain Layer



Hierarchical Structure

The proposed framework consists of four layers: Software Toolchain Layer, Instruction Set Abstraction Layer, Instruction Stream Randomization Layer, and Debugging Support Layer.

This hierarchical structure ensures seamless integration of instruction generation, randomization, and debugging support.

Collaboration of Layers

The Software Toolchain Layer manages the entire process of instruction stream stimuli, from conversion to simulation verification.

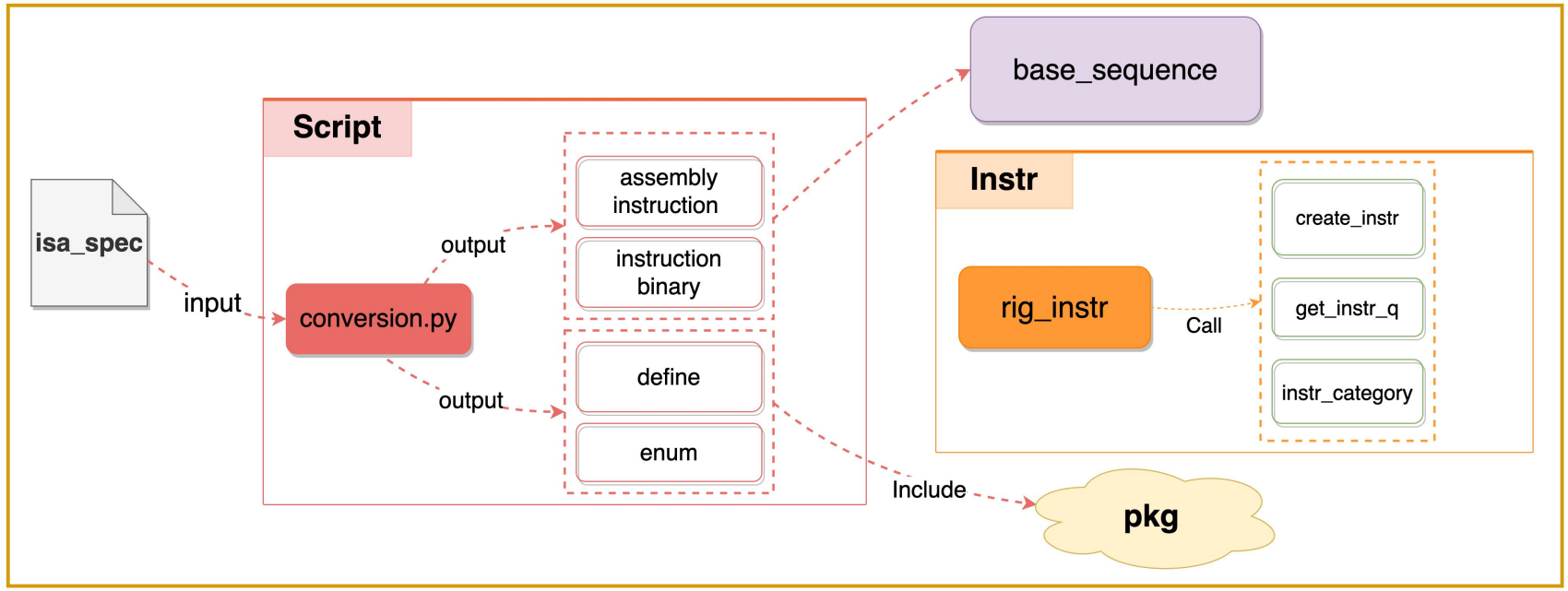
The Instruction Set Abstraction Layer formalizes instruction semantics, enabling structured expression and randomization.

Key Components

The Instruction Stream Randomization Layer encapsulates APIs for generating diverse instruction types and formats.

The Debugging Support Layer provides critical tools for tracing and diagnosing errors in complex instruction streams.

Instruction Set Abstraction Layer



Formalizing Instruction Semantics

The core objective of this layer is to formalize the semantics of the instruction set, providing manipulable variables for randomization.
It categorizes the ISA into clusters such as computational, control, memory, and system instructions, defining their opcodes, operand constraints, and special field parameters.

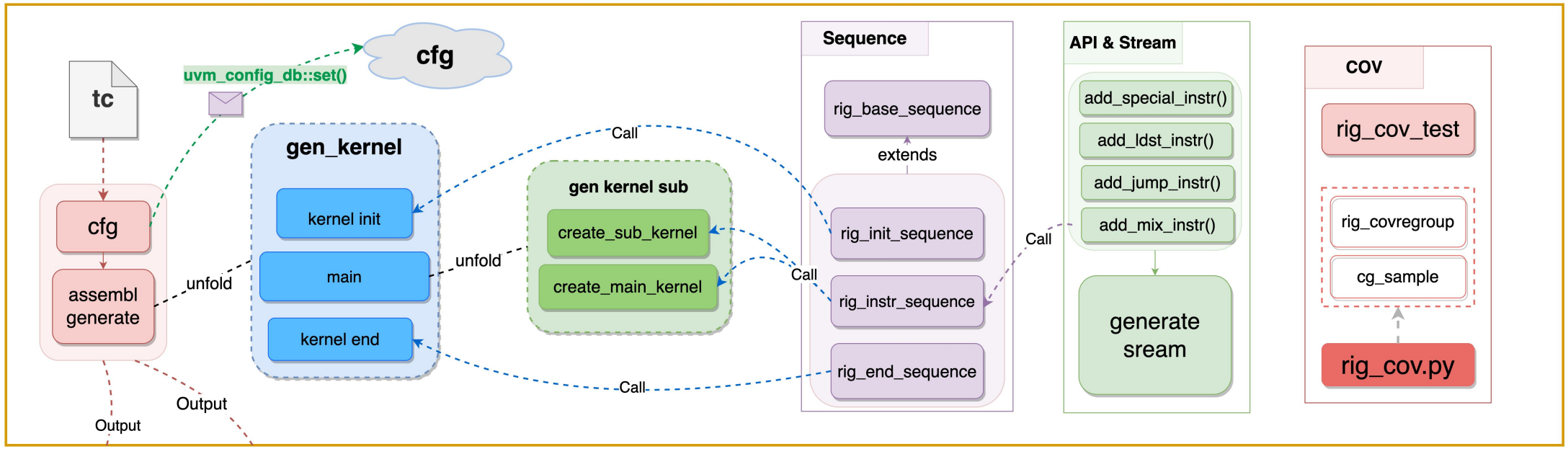
Dynamic Instruction Object Generation

The UVM factory pattern dynamically generates instruction objects based on the defined ISA specifications.
The VCS constraint solver ensures operand legality, avoiding issues like writes to reserved registers.

Assembly Instruction Generation

Instantiated instruction objects are converted into target assembly syntax using a stringification engine.
This process enables the generation of diverse and valid instruction streams for system- level verification.

Instruction Stream Randomization Layer



Randomizing Individual Instructions

This layer aims to randomize individual instructions and construct instruction stream stimuli.
It encapsulates the instruction stream logic into multiple API functions, allowing customization of desired instruction types and formats.

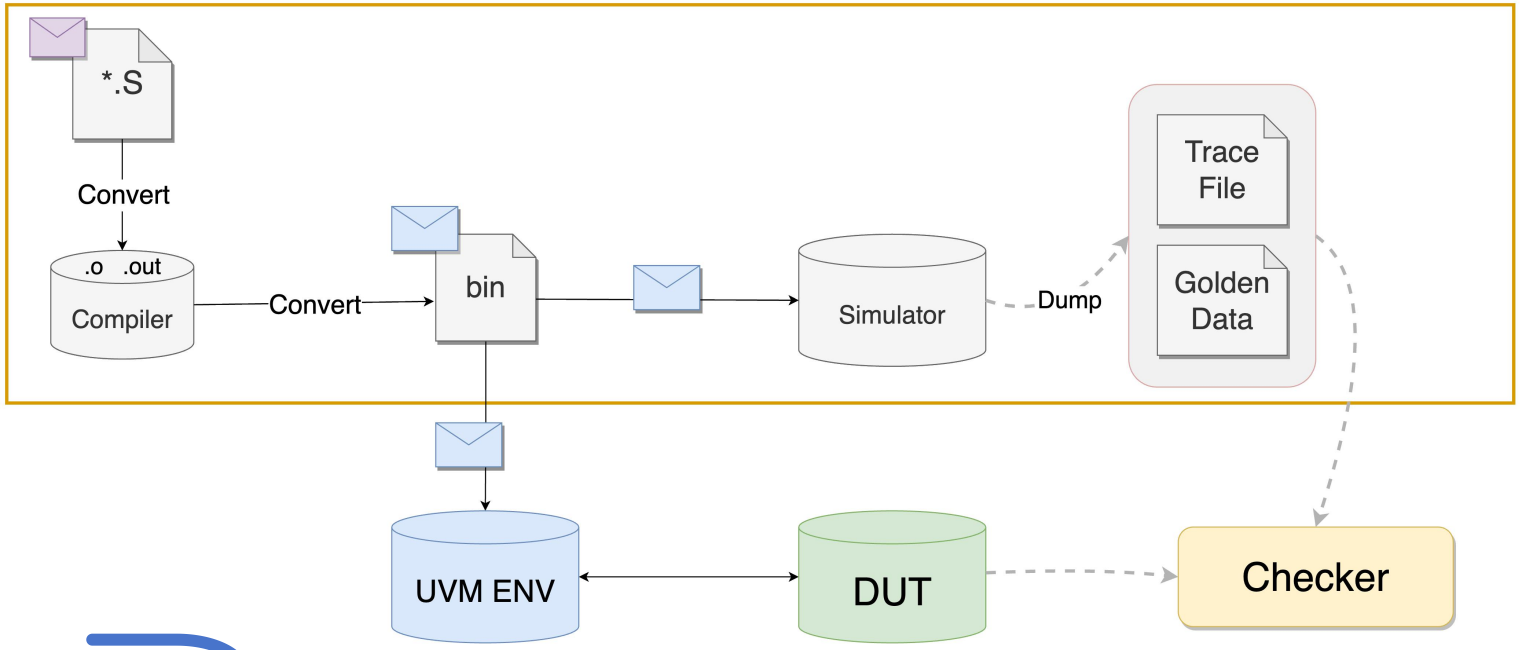
API-Driven Instruction Generation

The APIs enable automated synchronization of memory access instructions and dynamic insertion of branch instructions.
They also simplify instruction mixing and type weighting, facilitating the generation of mixed instruction types for multi-threaded processors.

Maintaining Instruction Stream Records

The framework maintains records of generated instruction streams, which is crucial for debugging and tracing errors.
This ensures that the verification process is traceable and any issues can be quickly identified and resolved.

Software Toolchain Layer



01

Instruction Stream Management

This layer focuses on managing the conversion of instruction stream stimuli and the simulation process of simulators, ensuring smooth integration with the UVM environment.
It supports syntax parsing of Instruction Set Architecture (ISA) and converts the generated instruction stream into binary files for simulation.

02

Integration with UVM Environment

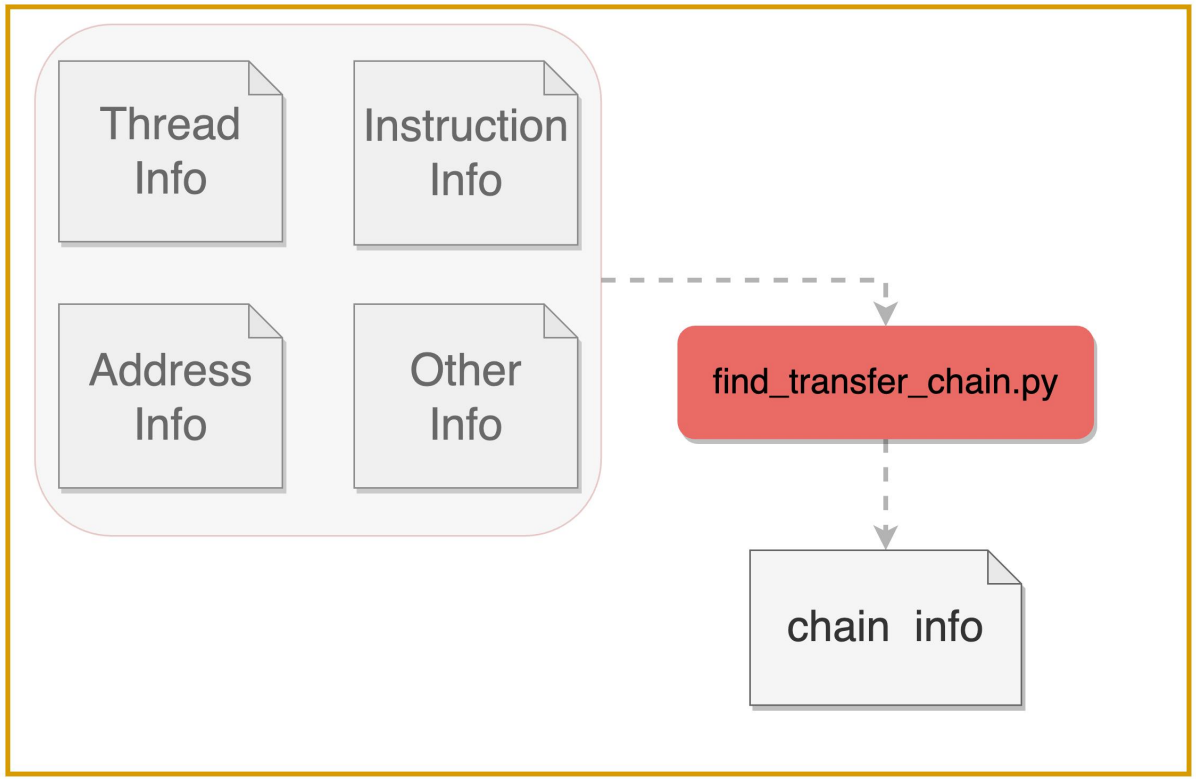
The integrated software toolchain includes assembler and simulator interfaces, facilitating the conversion of instruction streams into binary format.
This seamless integration enables efficient verification of the Design Under Test (DUT) against specification expectations.

03

Golden Data Dump

The binary files generated from instruction streams serve as golden data for dynamic simulation.
This ensures that the DUT's behavior aligns with the expected specifications, enhancing the reliability of the verification process.

Debugging Support Layer



Critical Debugging Tools

This layer provides essential tools for debugging complex instruction stream stimuli.
It includes mechanisms for identifying faulty threads and tracing back through the instruction stream to locate erroneous instructions.

Improving Debugging Efficiency

By leveraging recorded debug information, the debugging subsystem can quickly pinpoint the root cause of errors.
This significantly reduces the time and effort required for debugging, enhancing overall verification efficiency.

Visualizing Dependency Chains

The debugging support layer also visualizes dependency chains across threads, providing a clear understanding of the relationships between instructions.
This helps in diagnosing complex issues such as cache coherence problems and memory access conflicts.

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

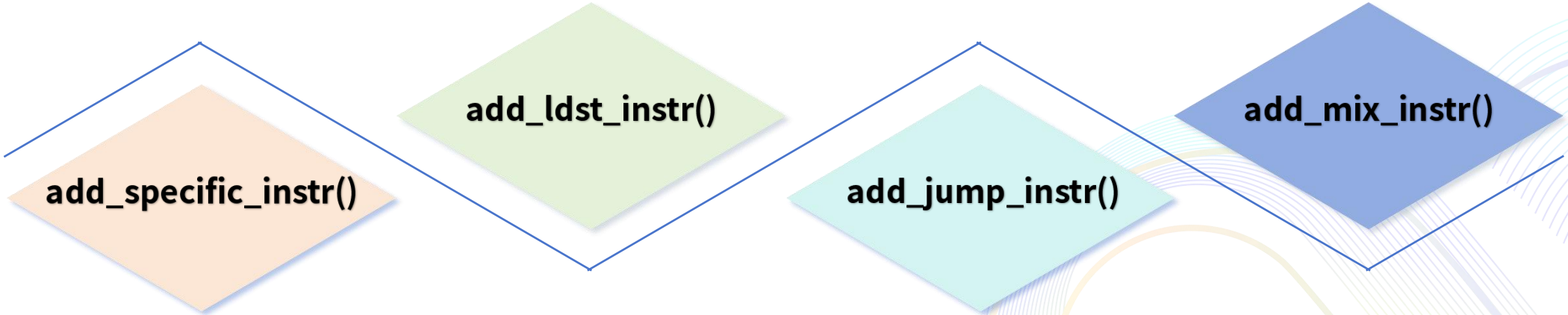
API encapsulation and instruction flow constraints

16 Apr 2025 上海 SHANGHAI

API Functions

This API specializes in generating load/store (memory access) instructions.
It internally models an address management and synchronization mechanism to avoid cache coherence issues in multi- threaded processors.

This API randomly inserts instructions into existing instruction streams.
It allows users to specify the number of instructions to insert, the starting position, and the instruction queue.



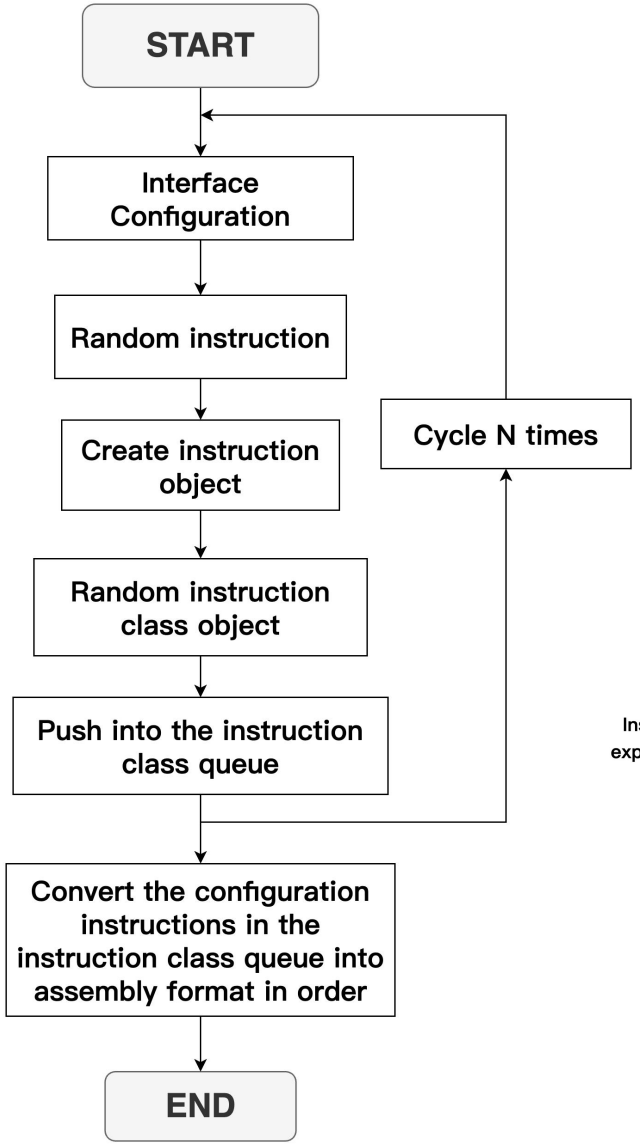
This embedded API serves as the foundation for instruction generation in the instruction stream.
It provides users with interfaces for configuring instruction types, registers, and special instruction fields while balancing high reusability and randomness.

This API handles the insertion of branch/jump instructions.
It ensures that random jumps adhere to constraints such as legal jump ranges and avoiding jumps into synchronization regions.

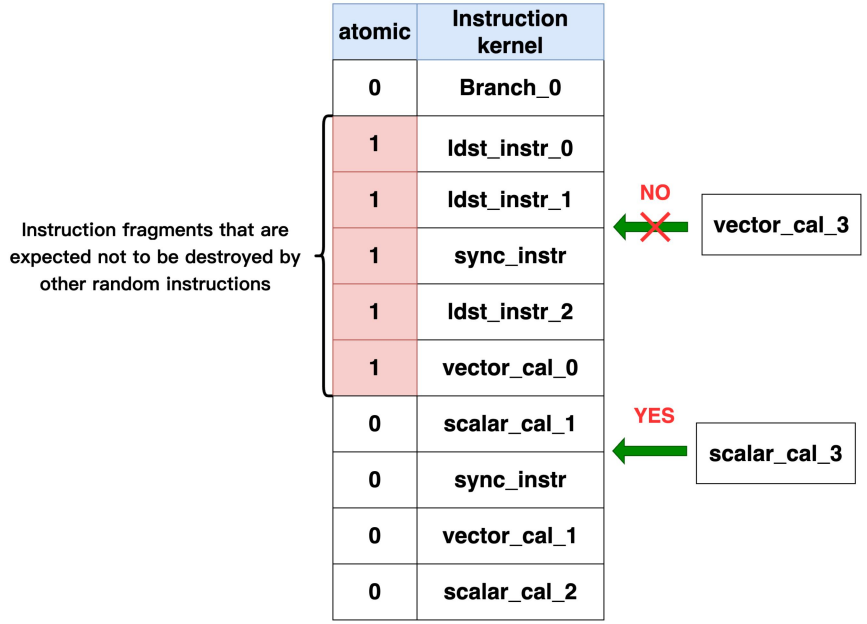
Basic API - add_specific_instr()

API Interface

- Instruction queue
- Register interface(vector register/scalar register/special register/immediate)
- Functional configuration args for special instructions
- Instruction comment
- Jump instruction target instruction label
- Atomic switch between instructions
- Interface fully configured check switch



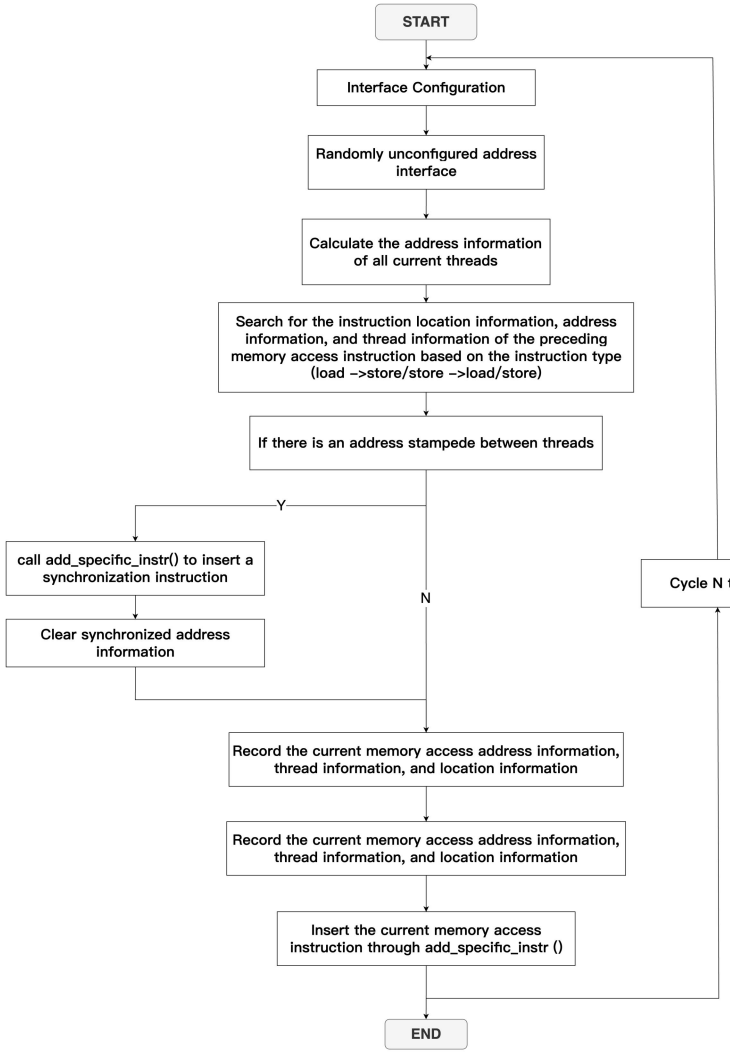
Atomic schematic diagram

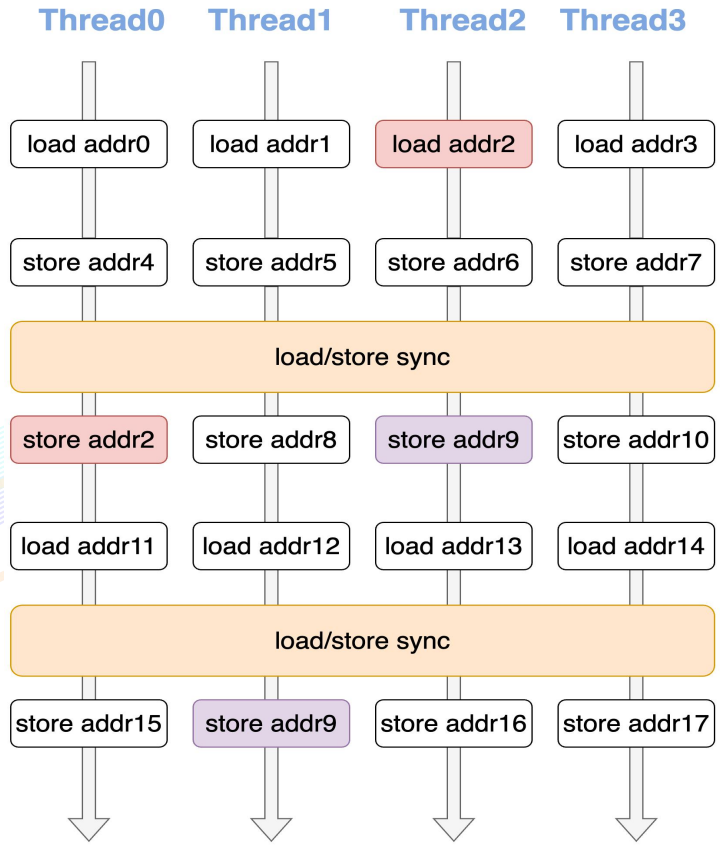


Load/Store instruction API - add_ldst_instr()

API Interface

- Load/Store instruction queue
- Saved address register
- offset address
- exception insert switch

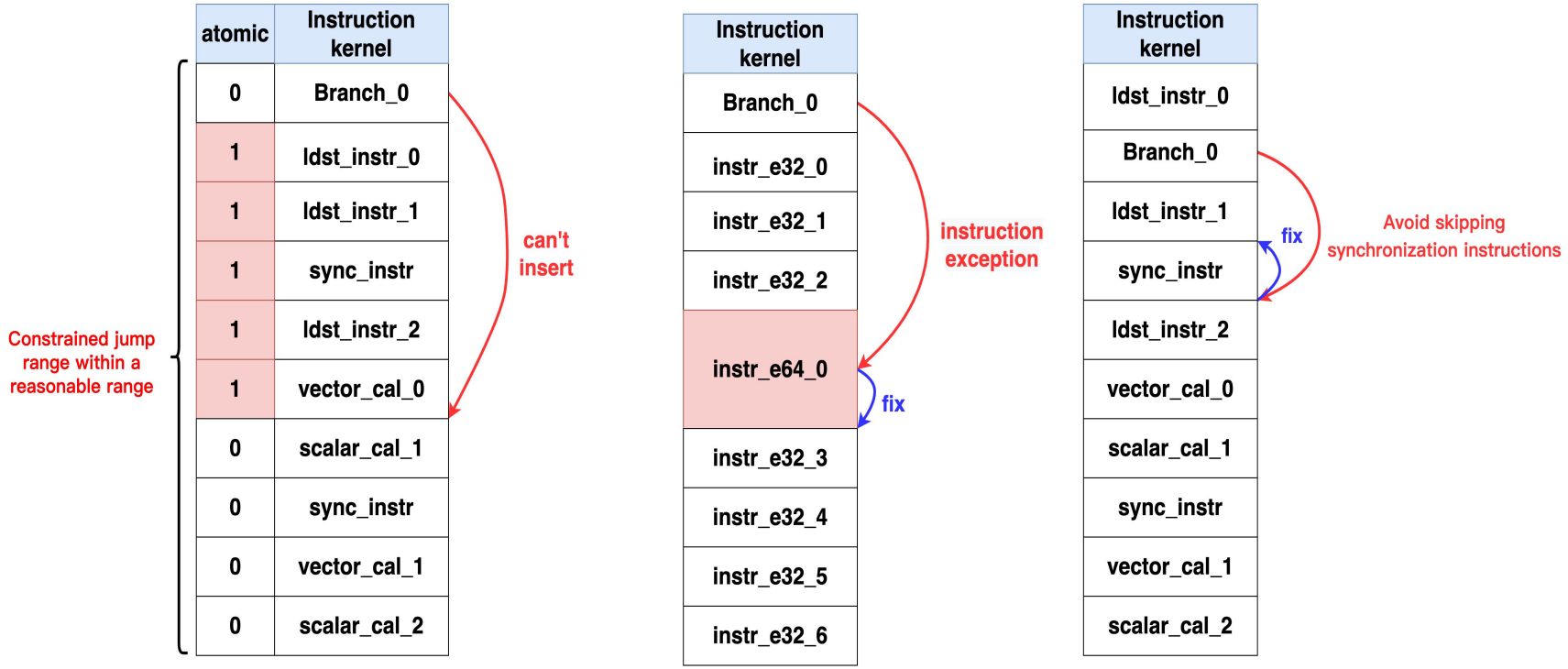




Jump instruction API - add_jump_instr()

API Interface

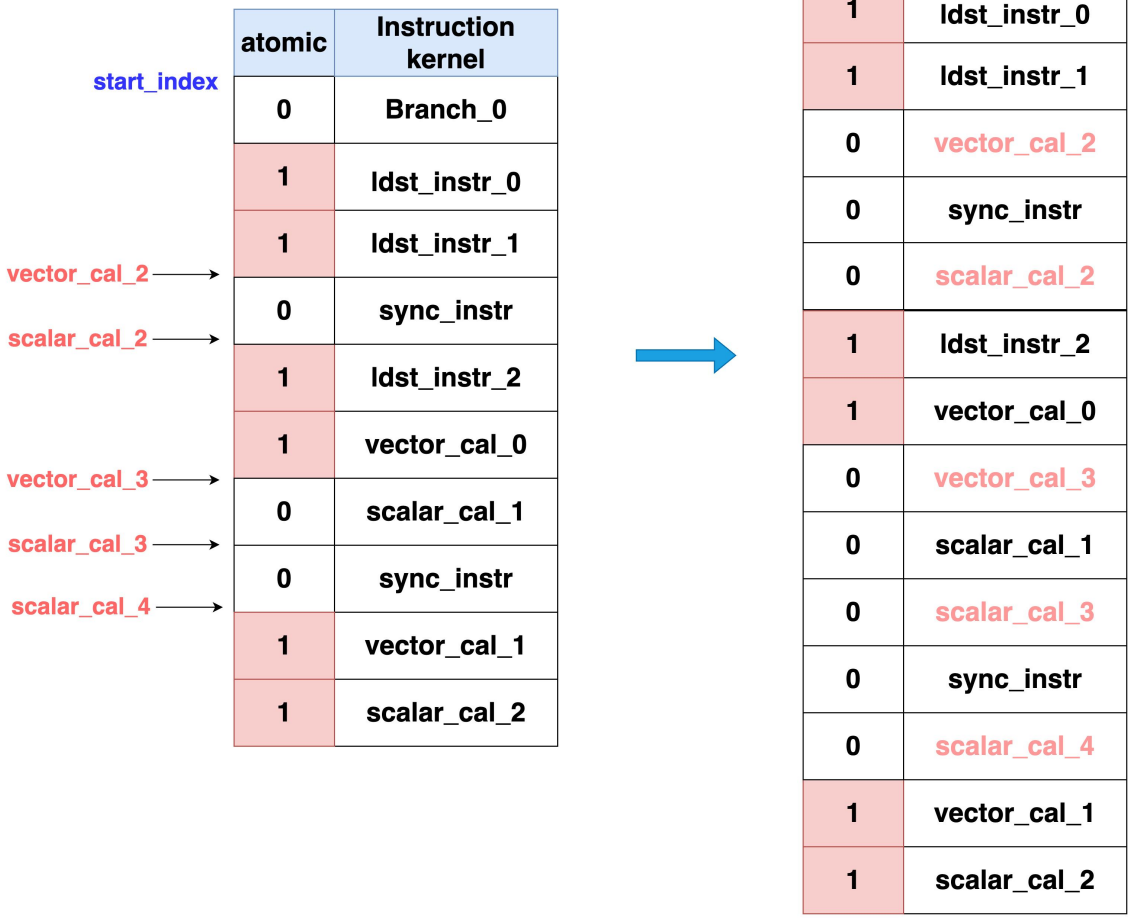
- Instruction queue
- Instruction count
- Starting instruction stream index
- The maximum jump range for jump instructions



Mix instruction API - add_mix_instr()

API Interface

- Instruction queue
- Instruction count
- Starting instruction stream index



Instruction Stream Constraints

Hierarchical Constraint Mechanism

The API interfaces ensure instruction legality through a hierarchical constraint mechanism.
This includes mandatory constraints that serve as the baseline for legal instruction streams and user- configurable constraints for specific verification scenarios.

Mandatory Constraints

Examples of mandatory constraints include register number range constraints, instruction format alignment, and prevention of illegal instruction encoding.
These constraints ensure that the generated instruction streams strictly comply with the Instruction Set Architecture (ISA).

User-Configurable Constraints

Examples of user- configurable constraints include address legality control, customizable jump range settings, and address exception trigger. These constraints allow users to tailor the instruction streams to meet specific verification requirements and test various edge cases.

Differentiated Constraint Strategies

The framework employs differentiated constraint strategies based on instruction types:
Calculational instructions: Constrained within the valid register range according to the ISA spec and avoiding contaminating reserved registers in the environment
Memory access instructions: In addition to the constraints of registers, it is also necessary to constrain the range of addresses and data synchronization between memory access instructions
Jump instructions: Strict control over jump ranges and protection of synchronization instructions and avoid jumping to illegal instruction segments.

Example Usage of APIs

01. Rapid stimulation Kernel Construction

By combining these APIs, users can quickly construct an ISA- based random kernel.
For example, generating a kernel with 8 load/store instructions, 80 computational instructions, and 8 branch instructions is fully randomized and automated.

02. High randomness

When generating a kernel through APIs, users only need to configure the interfaces they care about. For unconfigured interfaces, parameters will be randomly assigned within the legal range, and the instruction stream sequence will also reflect the high randomness inherent in the APIs.

03. High flexibility

The use of APIs significantly simplifies the process of generating complex instruction streams. Users can define the scale and type of instructions they wish to randomize, leaving intricate details to the API's automated mechanisms

```
Code:
    repeat(8)
    add_ldst_instr(
        .ldst_instr_q({load_instr, store_instr}),
        .addr_reg(addr0),
        .offset_addr(offset),
        .exception_insert(0));
```

Instruction kernel
store_instr_0
load_instr_0
store_instr_1
sync_instr
store_instr_2
load_instr_1
load_instr_2
sync_instr
store_instr_3
store_instr_4

start_index

```
Code:
    add_mix_instr(
        .instr_q(vector_cal, scalar_cal),
        .instr_cnt(10),
        .index(start_index));
```

Instruction kernel	
store_instr_0	vector_cal_2
load_instr_0	vector_cal_3
store_instr_1	sync_instr
vector_cal_0	scalar_cal_3
scalar_cal_0	vector_cal_4
sync_instr	store_instr_3
scalar_cal_1	scalar_cal_4
store_instr_2	store_instr_4
vector_cal_1	
load_instr_1	
scalar_cal_2	
load_instr_2	

start_index

```
Code:
    add_jump_instr(
        .instr_q(branch),
        .instr_cnt(4),
        .index(start_index),
        .max_imm(5));
```

Instruction kernel	
store_instr_0	branch_2 imm=0
branch_0 imm=2	scalar_cal_2
load_instr_0	load_instr_2
store_instr_1	vector_cal_2
branch_1 imm=1	vector_cal_3
vector_cal_0	sync_instr
scalar_cal_0	scalar_cal_3
sync_instr	branch_3 imm=3
scalar_cal_1	vector_cal_4
store_instr_2	store_instr_3
vector_cal_1	scalar_cal_4
load_instr_1	store_instr_4

The closed-loop mechanism of debugging

Debug data recording

This framework records the thread ID, global sequence number, operation type, and memory access address of memory access instructions during the instruction stream generation process.

This comprehensive memory access data record can accurately track and debug complex instruction streams.

Reverse-Tracing Mechanism

The debugging mechanism implements a reverse tracing mechanism for multi-threaded memory operations.

After detecting data errors at a specific address, the script quickly extracts the spatiotemporal context of all relevant instructions, allowing for precise backtracking to the first erroneous memory operation.

Visual memory access address delivery chain

This mechanism also visualizes the address data transfer chain across threads, providing a clear relationship between memory access instructions.

This helps diagnose complex issues such as cache consistency and memory access conflicts.

Example of debugging memory access address:

```
python3 find_transfer_chain.py

Enter M0/M1 address (hex): 1b7e69a4d2
Address type (m0/m1): m0
Access type (read/write): write
Complete Transfer Chain:
thread[5], dma_instr_5, dma_type=m0_to_m1, dma_len=15, len_id=18, read_m0_addr=b5e69a23d
thread[5], dma_instr_5, dma_type=m0_to_m1, dma_len=15, len_id=18, write_m1_addr=1269
thread[0], dma_instr_9, dma_type=m1_to_m0, dma_len=9, len_id=0, read_m1_addr=1269
thread[0], dma_instr_9, dma_type=m1_to_m0, dma_len=9, len_id=0, write_m0_addr=1b7e69a4d2
```

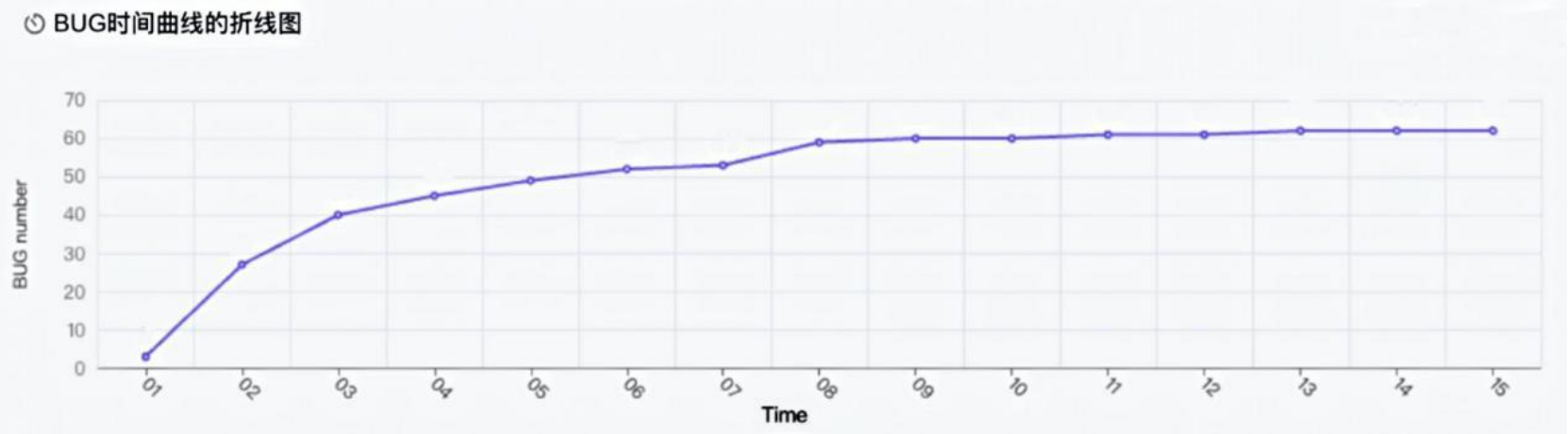
2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

Experimental Validation and Conclusion

16 Apr 2025 上海 SHANGHAI

Experimental Results



Rapid Bug Detection

The random instruction streams constructed using APIs in the RIG environment uncovered 60+ bugs in the project.
60% of bugs were detected within the first two months of the verification cycle, with the Device Under Test (DUT) stabilizing around the six- month mark.

Identification of Undetected Bugs

The API- generated random instruction streams identified 3 bugs that were undetectable by module- level verification or semi- directed single- instruction- type stimulus.
These bugs required a mix of instruction types and sufficient randomness to manifest, proving the effectiveness of the stochastic approach.

Comprehensive Functional Coverage

Functional coverage data collected over two weeks of regression testing shows that:
All instructions defined in the ISA SPEC were triggered.
Every instruction exercised the full register set.
Memory addresses covered boundary conditions and hardware- specific special addresses.
Jump instructions spanned all legal boundary ranges.
This confirms that the randomness of API- generated stimulus comprehensively covers the ISA specification and critical architectural features.

Conclusion

Efficient Verification Framework

This paper presents an efficient system-level random verification framework for multi-threaded processors.
It integrates four layers: Software Toolchain Layer, Instruction Set Abstraction Layer, Instruction Stream Randomization Layer, and Debugging Support Layer.

Key Innovations

The framework features modular instruction-type combinations through embedded APIs and an automated multi-thread conflict resolution mechanism.
It also establishes a closed-loop debug support mechanism, significantly improving debugging efficiency and defect localization.

Significant Code Reduction

The framework demonstrates substantial improvements in multi-threaded instruction stream construction efficiency.
Users need only write 10+ lines of code to generate mixed stimulus streams containing 5,000–10,000 instructions, reducing code volume by over 90% compared to manual methods.

Future Work and Enhancements

Expanding Framework Capabilities

Future work includes expanding the framework's capabilities to support more complex instruction sets and processor architectures.
This will involve enhancing the API functions and constraint mechanisms to cover a broader range of verification scenarios.

Improving Debugging Tools

Further improvements to the debugging tools will focus on providing more detailed and intuitive error analysis.
This will help validation engineers quickly identify and resolve issues, further reducing verification cycles.

Enhancing Randomization Techniques

Continuous enhancement of randomization techniques will ensure comprehensive coverage of the instruction set architecture.
This will involve exploring new algorithms and methods to improve the randomness and effectiveness of instruction stream generation.

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

Thank You

16 Apr 2025 上海 SHANGHAI