MERL

Accelerating Engineering Innovation

PY-UVM Framework for SweRV EL2 Core

PROJECT BY:

MERL-DSU

Table of Contents

Acknowledgement

First, we would like to express our unending gratitude towards Almighty Allah, the Most Beneficent, and the Most Merciful for blessing us with this opportunity to complete the implementation of PY-UVM Framework for SweRV EL2 Core and for providing us with the intelligence to reach our goal. We are thankful to Dr. Roomi Naqvi (Director MERL) , our advisors, Dr. Salman Zaffar and Engr Umair bin Mansoor and our seniors for guiding us throughout the way and giving us their valuable advice. We appreciate our Verification Team, Engr Ameen Raza Khan, Engr Syed Zeeshan Hussain, and Engr Muneeb-ur-Rehman, for designing the first ever Py-UVM framework for SweRV EL2 Core and crafting an informative and well-written document.

Chapter 1

1.1 Introduction

RTL, or Register Transfer Level, refers to a design abstraction level used in digital circuit design. Verification in RTL is the process of ensuring that the RTL design correctly implements the intended functionality and meets the desired performance and timing requirements.

There are several methods for verifying an RTL design, including simulation-based verification, formal verification, and hardware emulation.Simulation-based verification will be discussed in this document.

Simulation-based verification is a widely used technique for verifying digital designs, particularly at the RTL (Register Transfer Level) stage. In this approach, the design is modeled as a set of digital signals that are simulated over time to check whether they behave correctly according to the design specification. The simulation process generates a large number of test vectors that are used to exercise the design and check its behavior under various conditions.

The main importance of simulation-based verification for RTL designs are:

1. **Ensures Correctness**: Simulation-based verification ensures that the RTL design is functionally correct and meets its specification. It can detect design errors and corner-case scenarios that may not be identified through other verification techniques.

2. **Early bug detection:** Simulation-based verification can be performed early in the design process, enabling design bugs to be detected and corrected before they propagate to later stages of the design flow/

3. **Cost-effective:** Simulation-based verification is a relatively inexpensive verification technique compared to other formal verification methods such as theorem proving or model checking.

4. **Easy to use:** Simulation-based verification is easy to use and is the most widely used verification technique in the industry. It does not require any special skills or expertise to set up and run the simulations.

5. **Enables Validation:** Simulation-based verification is essential for the validation of a design against a set of test cases. It ensures that the design behaves as expected and meets the requirements of the intended application.

Simulation-Based Verification is necessary for RTL designs to ensure the correctness, reliability, and safety of the final product. It is a cost-effective and efficient verification technique that enables designers to detect design errors early in the design process and improve the quality of the design.

1.2 Simulation Based Verification Through PY-UVM

PY-UVM is a modern verification framework that is built on top of UVM (Universal Verification Methodology) and Python. It provides a Pythonic interface to UVM, making it easier for verification engineers to develop and maintain testbenches and test cases.

PY-UVM is an open-source Python package that implements the Universal Verification Methodology (UVM) for hardware verification. UVM is a widely used methodology in the semiconductor industry for verifying complex hardware designs, particularly digital integrated circuits (ICs). It provides a standardized framework for developing testbenches that can be used to verify the functionality and performance of digital ICs.

PY-UVM is designed to provide a simple and flexible implementation of UVM in Python, a popular programming language used in many industries, including semiconductor design and verification. It allows developers to write testbenches using Python syntax, which is easy to learn and read, while still adhering to the industry-standard UVM methodology.

PY-UVM provides many of the features and benefits of UVM, including a hierarchical testbench architecture, a rich set of built-in classes and functions, and support for transaction-level modeling. It also supports debugging features such as message logging and transaction tracing, making it easy to debug complex testbenches.

One of the key advantages of PY-UVM is its interoperability with other verification languages such as SystemVerilog, VHDL, and Verilog. This means that developers can use PY-UVM in conjunction with other verification tools and environments, making it easy to integrate into existing verification workflows.

PY-UVM also promotes code reusability, which is a key advantage in hardware verification where designs can be complex and time-consuming to verify. By using PY-UVM, developers can create testbench components that can be easily reused across multiple projects, saving time and effort.

Overall, PY-UVM is a powerful and flexible tool for developing testbenches for hardware verification. Its ease of use, interoperability, and support for UVM make it a valuable addition to any verification engineer's toolbox.

Verification engineers will find this document useful for creating a PY-UVM framework for RISC-V Single Cycle Core.It is assumed that the reader is familiar with RISC-V ISA and Single Cycle Core design. The design files are available on https://github.com/merldsu/Py_UVM_Framework.git github.

1.3 The Methodology Wars

The methodology conflicts are widely documented in several literature and textbooks, however in this document, we explored the methodology differences between Cocotb, Python OOP-based methodology and PY-UVM.

COCOTB, Python OOP based methodology, and PY-UVM are all used for hardware verification and testing, but they differ in their approach and implementation.

Cocotb is a Python-based verification framework used in the hardware industry. It enables the creation of testbenches for digital designs. With Cocotb, developers can write testbenches in Python, making the process of verification faster and more efficient. Cocotb is an open-source project that is supported by a large community of developers and users. It integrates with popular simulation tools, such as ModelSim and Icarus Verilog, and provides a simple and intuitive interface for writing tests. Overall, Cocotb is a valuable tool for hardware developers looking to improve their verification process and ensure the quality of their designs.

Python OOP (Object-Oriented Programming) based methodology is a way of organizing and structuring Python code to create reusable objects that can be used to model complex systems. This methodology is commonly used in software development, but it can also be used in hardware verification. In this approach, hardware designs are modeled as objects, and testbenches are created using Python classes and methods. This approach provides a way to create modular and reusable code, which can make it easier to maintain and modify testbenches over time.

PY-UVM (Python Universal Verification Methodology) is a Python implementation of the UVM (Universal Verification Methodology), which is a standard methodology used for hardware verification. PY-UVM provides a set of classes and methods for creating testbenches in Python, which are organized into a hierarchy of reusable components. This approach allows designers to create testbenches that are scalable and can be easily modified and extended as needed. PY-UVM is typically used for testing complex designs, such as SoCs (System-on-Chips), and its approach is to use a structured and hierarchical methodology to create a testbench environment that is reusable and scalable.

If we say that UVM surpasses all methodologies until now, we may say that PY-UVM will be the future of simulation-based verification and that many industries will adopt this methodology because of advancements in Python and the benefits of UVM. PY-UVM can be a powerful tool for verification, and some of the benefits are as follows:

1) **Easy to use:** PY-UVM provides a simplified interface for writing UVM code in Python, making it easier to learn and use for those familiar with the language.

2) **Faster development:** The Python language and its ecosystem are known for their ease of use, flexibility, and rapid development capabilities. PY-UVM leverages these advantages to accelerate the development of testbenches.

3) **Code reusability:** PY-UVM allows for code reuse, which is essential in hardware verification. It provides a framework for creating and managing reusable verification components, such as drivers, monitors, and scoreboards.

4) **Improved productivity:** PY-UVM's simplified interface, fast development capabilities, and code reusability lead to improved productivity, allowing engineers to focus on verifying designs instead of writing testbenches.

5) **Integration:** PY-UVM can be easily integrated with other Python-based tools, such as simulation engines and data analysis libraries, to create a complete verification environment.

Overall, PY-UVM offers a streamlined and efficient approach to UVM-based hardware verification in Python, making it a powerful tool for verification engineers.

Chapter 2

2.1 UVM Methodology

The long history of methodology that included eRM,VMM,AVM, and OVM was to come to an end with the introduction of the UVM. The most successful verification methodology is the Universal Verification Methodology. The evolution of the UVM is now in the hands of a group of EDA vendors and end-users.

In this section, we will be discussing the few concepts of Universal Verification Methodology and its features.

2.1.1 UVM Component

A UVM component is a class that encapsulates a certain functionality or capability in the verification environment. Examples of UVM components include drivers, monitors, agents, scoreboards, and sequencers. A UVM component is defined as a subclass of the uvm_component class.

The UVM methodology defines a set of standard phases for these components, which are executed in a specific order during the simulation.There are several main phases in UVM, each with its own set of advantages here we discussed only few of them which were used in PY-UVM framework for Single Cycle Core:

1) **Build Phase:** This phase is used to initialize the component's configuration and build any child components that it may have. During the build_phase, the component's configuration parameters are set, such as its clock and reset signals, and any child components that it depends on are created. The build_phase is called once for each instance of the component.

2) **Connect Phase:** This phase is used to connect the component's interface to other interfaces in the verification environment. During the connect_phase, the component's interface is connected to the interfaces of any other components that it needs to communicate with. The connect_phase is called once for each instance of the component.

3) **End of Elaboration Phase:** This phase is used to perform any final setup before the simulation starts. During the end_of_elaboration_phase, any configuration parameters that have not been set are given default values, and any other necessary setup is performed.

4) **Start of Simulation Phase:** This phase is used to perform any setup that needs to be done before the simulation starts running. During the start_of_simulation_phase, any remaining setup that needs to be done is performed, such as initializing data structures and configuring analysis ports.

5) **Run Phase:** This phase is the main phase of the component and is called repeatedly during the simulation. The run_phase method is responsible for executing the component's functionality. During the run_phase, the component interacts with other components in the verification

environment by sending and receiving transactions, checking for errors, and updating its internal state.

6) **Extract Phase:** This phase is used to extract any information from the component that may be needed for debugging or analysis. During the extract_phase, any relevant data from the component is extracted and made available for analysis.

7) **Check Phase:** This phase is used to verify that the component is functioning correctly. During the check_phase, any relevant checks are performed to ensure that the component is working as expected. This may involve comparing the component's output to expected values, checking for error conditions, or performing other types of verification.

8) **Report Phase:** This phase is used to generate any final reports or summaries. During the report_phase, any relevant data is collected and summarized to provide an overall view of the component's performance.

9) **Final Phase:** This phase is used to perform any final cleanup before the simulation ends. During the final_phase, any remaining cleanup that needs to be done is performed, such as closing files or releasing resources.

By dividing the functionality of a UVM component into these distinct phases, the component can be designed to be modular and reusable, and can interact with other components in a well-defined way. This allows for the creation of complex and scalable verification environments that can be easily maintained and extended.

The advantages of using the UVM component phases are that they provide a structured way to create, connect, and simulate the components of a testbench, and they ensure that the components are properly initialized, connected, and cleaned up. Additionally, the use of standardized phases allows for easy reuse of components and promotes a consistent methodology across the testbench.

2.1.2 UVM TLM FIFO

UVM TLM (Transaction Level Modeling) FIFO is a class in the Universal Verification Methodology (UVM) that provides a transaction-level interface for communication between two or more verification components in a testbench. It acts as a buffer that can store transactions of arbitrary data types and sizes.

The following are some advantages of using UVM TLM FIFO:

1) This can be easily reused across multiple testbenches and verification environments, providing a standardized interface for communication between components.

2) It supports multiple interfaces, such as the blocking, non-blocking, and streaming interfaces, allowing components with different communication requirements to communicate seamlessly.

3) It also provides a level of abstraction that enables components to communicate at a higher level of abstraction.

4) It supports asynchronous communication, allowing components to operate at different speeds and reducing simulation overhead.

5) It provides built-in features for checking data integrity, ensuring that the data transmitted between components is correct.

6) It provides several configuration parameters that allow users to customize its behavior according to their needs.

7) It provides built-in features for debugging and tracing, allowing users to quickly diagnose and fix issues.

UVM TLM FIFO provides a standardized, flexible, and efficient way for components to communicate with each other, improving the overall reliability and performance of a verification environment.

2.1.3 UVM CONFIG DB

The UVM Configuration Database is a powerful tool in the Universal Verification Methodology (UVM) that provides a flexible and efficient way to store and retrieve configuration data used by verification components in a testbench. It is a central repository for configuration information, allowing users to set and access values of various configuration parameters. The configuration database is implemented as a hierarchical tree structure that enables users to organize their configuration data according to their needs. By using the UVM Configuration Database, verification engineers can easily customize and parameterize their testbench components, reducing the amount of code they need to write and making their testbenches more reusable and maintainable.

Here are some advantages of using the UVM configuration database:

1) It provides a central location for storing and retrieving configuration information, making it easy to manage and maintain.

2) The configuration database supports hierarchical configuration, which allows for easy customization of configurations at different levels of the testbench hierarchy.

3) Dynamic configuration: The UVM configuration database supports dynamic configuration, allowing for changes to be made to the configuration during simulation.

4) It supports dynamic configuration, allowing for changes to be made to the configuration during simulation.

5) It can be used to parameterize testbench components, allowing for easy reuse of components across different test scenarios.

6) It allows for different types of data to be stored, including integers, strings, and user defined types, making it highly flexible.

7) It is designed to be modular, allowing for different implementations to be used based on the needs of the testbench.

8) It can be used to store and display information about the configuration of the testbench, making it easier to debug issues.

Overall, the UVM configuration database provides a powerful and flexible mechanism for managing configuration information in a verification environment, making it an essential tool for developing complex testbenches.

2.1.4 UVM Interface

One of the key components of UVM is the use of interfaces to connect the verification environment to the Design Under Test (DUT).

An interface is a set of signals or methods that define a standard communication protocol between two components. In the context of UVM, an interface is used to define the communication protocol between the verification environment and the DUT. The UVM interface is defined in SystemVerilog and provides a way for the testbench to interact with the DUT.

Here are some advantages of using the UVM Interface :

1. It can be reused across multiple projects, allowing for easier development and maintenance of the verification environment.

2. The use of a standard interface promotes consistency and reduces errors by enforcing a common communication protocol between the testbench and the DUT.

3. It simplifies the process of integrating the DUT with the testbench. By using a standard interface, designers can easily connect their DUT to the verification environment, reducing the time and effort required for integration.

4. It provides a clear and concise way to debug issues related to the communication between the testbench and the DUT. By using a standardized interface, debugging becomes more efficient and effective.

5. It is scalable, allowing for easy expansion of the verification environment as the complexity of the DUT increases. As more functionality is added to the DUT, the UVM interface can be modified and extended to accommodate the new requirements.

Overall,using Interface in UVM offers advantages such as modularity, abstraction, ease of use, standardization, interoperability, debugging support, and improved simulation performance. These benefits contribute to efficient and effective verification of digital designs, reducing development time and improving overall product quality.

2.1.5 UVM Logging

In the context of the Universal Verification Methodology (UVM), logging is a technique used to generate messages or records about the various activities or events happening during the verification process. These records are typically stored in a log file, which can be analyzed later to identify errors, debug the design, or improve the verification process.

Here are some advantages of using the UVM Logging :

1. It provides a detailed view of the various events happening during the simulation, making it easier to track down the source of errors and debug the design.

2. By reviewing the log files, verification engineers can get an idea of the overall progress of the verification process, and make informed decisions about the next steps.

3. It can help in the analysis of code coverage, functional coverage, and assertion coverage by providing a detailed view of the events that occurred during the verification process.

4. By reducing the time required to debug issues, logging helps to improve the overall efficiency of the verification process.

5. Log files serve as a documentation of the verification process, providing an audit trail of the activities performed during the simulation.

6. The UVM framework provides a built-in logging mechanism that makes it easy to implement logging in the testbench. This saves time and reduces the chance of errors.

Overall, logging in UVM offers a systematic approach to capturing and analyzing information during hardware verification. It enhances debugging capabilities, facilitates progress tracking, improves design understanding, aids performance analysis, ensures coverage completeness, supports regression analysis, and fosters effective communication among team members.

2.1.6 UVM Agent

UVM provides a standardized framework for creating reusable, scalable, and modular verification environments. An important component of the UVM methodology is the UVM Agent, which plays a crucial role in coordinating the interaction between the Design Under Test (DUT) and the testbench.

A UVM Agent is responsible for generating stimulus, capturing responses, and monitoring the behavior of a specific block or interface in the DUT. It acts as an intelligent interface between the testbench and the DUT, facilitating communication and verification activities. The agent encapsulates the necessary components, such as driver, monitor, sequencer, and scoreboard, to perform its functions effectively.

Here are some advantages of using the UVM Agents :

1. It is designed to be reusable, enabling efficient verification of multiple designs or instances of the same design. By encapsulating the verification components within an agent, you can easily reuse the agent across different projects, reducing development time and effort.

2. It can be scaled to handle complex designs and growing verification requirements. As the design complexity increases, agents can be instantiated or interconnected to create a hierarchical and scalable verification environment, allowing for efficient verification of large designs.

3. It promotes a modular approach to verification, making it easier to develop, maintain, and debug the verification environment. Each agent encapsulates a specific functionality, such as stimulus generation or response checking, making the verification environment more manageable and enabling easy integration of new agents as the project evolves.

4. It leverages the power of concurrent programming, allowing multiple agents to operate simultaneously. This concurrency enables efficient and parallel execution of various verification tasks, such as generating stimulus and checking responses, improving overall verification performance.

5. It can be configured to adapt to different test scenarios and requirements. Through configuration mechanisms provided by UVM, you can control the behavior and features of an agent dynamically, enabling the reuse of agents with different configurations for different test cases.

6. It also provides portability across different simulation platforms and tools. The UVM methodology is based on a standard class library and methodology, making it easier to transfer verification environments across different projects or organizations, as long as they adhere to the UVM guidelines.

Overall, agents in UVM offer a modular, reusable, and configurable approach to verification. They ensure protocol compliance, facilitate concurrency, synchronization, and parallelism, and provide visibility for efficient debugging. Agents play a critical role in achieving comprehensive verification of complex designs by accurately modeling and verifying the functional blocks of the DUT.

2.1.7 UVM Environment

In the Universal Verification Methodology (UVM), the environment is a fundamental component used to model and verify the Design Under Test (DUT). It is responsible for creating the testbench infrastructure and coordinating the various verification components. The environment encapsulates the test stimulus generation, checking mechanisms, and other components necessary for verification.

Here are some advantages of using the UVM Environment :

1. It promotes a modular approach to verification. It allows for the separation of concerns, enabling different teams or individuals to work on different aspects of the verification process concurrently. This modularity enhances reusability and maintainability.

2. It follows a hierarchical structure, which allows for the creation of a multi-level verification environment. This structure enables the organization of verification components into layers, such as block-level, subsystem-level, and full-chip level. The hierarchical nature of the environment facilitates scalability and improves simulation performance.

3. It provides configurability options, allowing users to adapt the verification environment to different test scenarios. Parameters can be set at different levels of hierarchy, providing flexibility to change the behavior of the testbench without modifying the underlying code.

4. It serves as a central entity responsible for coordinating the activities of various components within the testbench. It manages the scheduling of tests, the synchronization between different verification components, and the flow of data and control.

5. It encourages the development of reusable verification components. Once a component is developed and validated, it can be reused across different projects or reused within the same project with different configurations. This reusability reduces verification effort and improves productivity.

6. UVM provides a layered abstraction model, where different components of the environment can be organized into layers based on their functionality. This layered approach facilitates the separation of concerns and simplifies the debugging and maintenance of the verification environment.

7. It integrates various verification components, such as sequences, scoreboards, monitors, and drivers, to form a complete and comprehensive testbench. This integration ensures efficient communication and data transfer between these components, enabling effective verification of the DUT.

8. With the UVM environment, tests can be written independently of the DUT's implementation. This portability allows tests to be reused across different DUT implementations or even different projects, promoting efficiency and accelerating the verification process.

Overall, the environment in UVM provides a standardized and scalable methodology for developing reusable and configurable verification environments. It improves productivity, promotes collaboration, and enables effective verification of complex designs.

2.1.8 UVM Sequencer

In the Universal Verification Methodology (UVM), a sequencer is a fundamental component of UVM that is responsible for generating stimulus transactions to be applied to the Design Under Test (DUT). The sequencer interacts with the driver to control the flow of stimulus generation and manages the sequencing and ordering of transactions.

Here are some advantages of using UVM Sequencer :

1. The primary role of a sequencer is to generate stimulus transactions that exercise the DUT's functionality. It provides a higher level of abstraction, allowing test scenarios to be defined using transaction-level models rather than low-level signal toggling. This abstraction simplifies test development and improves readability.

2. It provides fine-grained control over the generation of stimulus. It enables the test engineer to define the sequence of transactions, including their order and timing. This controllability allows for the creation of complex test scenarios and the ability to target specific corner cases.

3. It is designed to be reusable across different test scenarios and projects. Once a sequencer is implemented and validated, it can be easily reused by connecting it to different drivers and monitors, facilitating the development of a modular and scalable testbench infrastructure.

4. It offers configurability options to adapt to different test requirements. Parameters can be adjusted to modify the behavior of the sequencer, such as the transaction ordering policy, the timing constraints, or the burst size of transactions. This configurability enhances the flexibility and adaptability of the testbench.

5. It allows the definition of sequencing constraints, such as transaction dependencies or transaction orderings, which are critical for accurate stimulus generation. By specifying these constraints, the sequencer ensures that transactions are generated in the correct sequence, mimicking real-world scenarios and verifying the DUT's behavior accurately.

6. It is designed to work concurrently and in parallel with other sequencers and components within the testbench. This concurrency enables multiple sequencers to generate stimulus simultaneously, reflecting real-world scenarios and increasing the overall efficiency of the verification process.

7. It provides visibility into the transactions generated and the order in which they are sent to the DUT. This visibility aids in debugging by allowing engineers to observe and analyze the stimulus at various stages of the verification process. It helps identify any issues or unexpected behavior in the stimulus generation flow, improving the efficiency of bug detection and resolution.

Overall, sequencers in UVM provide controllability, reusability, and configurability for stimulus generation. They enable the definition of sequencing constraints, support concurrency and parallelism, and integrate with coverage and monitoring for effective verification. Sequencers play a crucial role in generating accurate and comprehensive stimuli to thoroughly verify the functionality of the DUT.

2.1.9 UVM Driver

In the Universal Verification Methodology (UVM), a driver is a key component of UVM that is responsible for driving the generated stimulus transactions from the sequencer to the Design Under Test (DUT). The driver converts the abstract stimulus transactions into appropriate 33 signal-level or protocol-level signals to stimulate the DUT and manages the timing and synchronization of the transaction execution.

Here are some advantages of using UVM Driver :

1. It translates the abstract stimulus transactions generated by the sequencer into appropriate signal-level or protocol-level signals that the DUT can understand. This conversion enables the testbench to drive the DUT with the correct signals and ensure that the DUT is properly stimulated.

2. It manages the timing and synchronization aspects of transaction execution. It ensures that the signals are applied to the DUT at the correct time, respecting the DUT's clock domains and any specified timing constraints. This timing accuracy is crucial for accurate and reliable verification.

3. It ensures that the generated stimulus signals conform to the communication protocol or interface used by the DUT. It verifies that the signals adhere to the protocol specifications and correctly represent the intended stimulus, validating the DUT's compliance with the specified protocol.

4. It takes care of maintaining the integrity of the generated stimulus signals. It handles any necessary signal conditioning or adjustments to ensure that the signals meet the required voltage levels, drive strengths, or other electrical characteristics. This signal integrity ensures proper signal propagation and reduces the risk of signal-related issues during verification.

5. It provides controllability over the execution of the generated stimulus transactions. It can implement features such as transaction-level retries, delays, or error injections to test specific scenarios. Additionally, the driver may include monitoring capabilities to capture and analyze the DUT's responses to the stimuli, enabling efficient checking and analysis.

6. It is designed to be reusable across different test scenarios and projects. Once a driver is implemented and validated, it can be easily reused by connecting it to different sequencers and monitors, facilitating the development of a modular and scalable testbench infrastructure.

7. It can operate concurrently and in parallel with other drivers and components within the testbench. This concurrency allows for multiple drivers to drive stimulus to different parts of the DUT simultaneously, reflecting real-world scenarios and increasing the overall efficiency of the verification process.

8. It provides visibility into the signals being driven to the DUT. This visibility aids in debugging by allowing engineers to observe and analyze the stimulus at various stages of the verification process. It helps identify any issues or unexpected behavior in the signal generation flow,

improving the efficiency of bug detection and resolution.

Overall, drivers in UVM provide signal-level conversion, timing and synchronization, protocol compliance, and signal integrity. They offer controllability and monitoring capabilities, support concurrency and parallelism, and aid in debugging and visibility. Drivers play a vital role in driving the generated stimulus to the DUT accurately and efficiently, facilitating the verification of the DUT's functionality.

2.1.10 UVM Monitor

In the Universal Verification Methodology (UVM), a monitor is a key component used for capturing and analyzing signals during the verification process of a design. It plays a crucial role in monitoring and collecting data from the Design Under Test (DUT) and converting it into transaction-level information for analysis.

Here are some advantages of using UVM Monitor :

1. It provides visibility into the internal workings of the DUT by capturing signals of interest. They enable the verification engineer to observe and analyze the behavior of the design at a transactional level, which is critical for debugging and identifying potential issues.

2. It helps ensure that the DUT conforms to the specified protocol or standard. By monitoring the signals associated with the protocol, they can check for protocol violations, detect unexpected behavior, and generate appropriate error messages or alerts.

3. It contributes to the achievement of functional coverage goals. They can be used to track the occurrence of specific events, sequences, or transactions in the DUT. By collecting coverage data, monitors help assess the completeness of the verification environment and provide valuable information for coverage-driven verification closure.

4. It extracts relevant information from the DUT's signals and transactions. This information can be further processed and analyzed to extract meaningful metrics, statistics, or performance data. It enables verification engineers to gain insights into the design's operation and performance characteristics.

5. It is typically designed to be reusable across different testbenches and projects. Once a monitor is implemented, it can be easily integrated into various verification environments and connected to different DUTs. This reusability factor enhances productivity, reduces development effort, and promotes standardization within an organization.

6. It seamlessly integrates with other UVM components, such as drivers, sequencers, and scoreboards. They facilitate data transfer and synchronization between these components, ensuring efficient communication and coordination within the verification environment.

7. This aids in debugging the DUT by providing a detailed view of its internal signals and transactions. When an issue arises, monitors can help narrow down the problem area, identify the root cause, and assist in the resolution process.

Overall, monitors in UVM offer valuable insights into the behavior of the Design Under Test, assist in protocol adherence, contribute to functional coverage goals, enable data extraction and analysis, promote reusability, facilitate integration with other UVM components, and aid in debugging and error diagnosis. These advantages collectively enhance the efficiency and effectiveness of the verification process.

2.1.11 UVM Coverage

In the context of Universal Verification Methodology (UVM), coverage refers to the measurement and analysis of the completeness and quality of the verification process for a digital design. It involves tracking various aspects of the design, such as signals, functional behavior, and code coverage, to ensure that all the required functionality has been exercised and tested adequately.

"UVM supports functional coverage, which allows you to define coverage goals based on specific design functionality. Functional coverage metrics enable you to track the progress of verifying specific features or scenarios. By setting functional coverage targets, you can ensure that all the required functionalities are exercised and tested adequately.

Functional coverage in UVM allows you to set coverage goals based on specific functionalities or scenarios of the design under verification. It helps you ensure that all the intended behaviors, features, and corner cases of the design are thoroughly tested. By defining coverage points and bins, you can monitor the progress of verification by tracking how often different aspects of the design are exercised. This is especially useful for verifying complex designs where it’s crucial to ensure that all required functionalities are adequately tested."

Here are some advantages of using UVM Coverage :

1. It allows you to gauge the extent to which your design has been verified. By setting coverage goals and tracking the progress, you can ensure that all parts of the design are exercised during the verification process. This helps in identifying areas that need additional testing and ensures that your verification effort is comprehensive.

2. Its metrics can help in identifying untested or under-tested parts of the design. By analyzing the coverage results, you can identify areas that have low coverage and focus your verification efforts on those specific areas. This increases the likelihood of detecting bugs or corner-case scenarios that may have been missed otherwise.

3. The Coverage analysis provides a quantitative measure of the quality of the verification process. By monitoring coverage metrics, you can assess the effectiveness of your test suite and identify areas where additional tests may be required. It helps in evaluating the robustness of the design

and ensuring that it meets the desired functional requirements.

4. The Coverage analysis provides valuable feedback for improving the test suite. By analyzing coverage results, you can identify redundant or ineffective tests that do not contribute significantly to the coverage metrics. This helps in optimizing the test suite, removing unnecessary tests, and adding new tests to target untested areas, thereby improving the efficiency of the verification process.

Overall, coverage in UVM plays a crucial role in ensuring comprehensive and high-quality verification of digital designs. It provides quantitative measures of verification completeness, helps in bug detection, assesses the effectiveness of the test suite, and guides improvements in the verification process.

2.1.12 UVM Scoreboard

In Universal Verification Methodology (UVM), a scoreboard is a key component used to compare the results produced by a Design Under Test (DUT) and with those generated by a Golden Model. The purpose of a scoreboard is to verify the correctness of the DUT by comparing its outputs against the expected outputs from the Golden Model. This comparison helps in ensuring that the DUT is functioning as intended and that it meets the desired specifications.

Here are some advantages of using UVM Scoreboard :

1. It enables comprehensive functional verification by comparing the expected results with the DUT's actual output. It helps identify any discrepancies or errors in the design early in the verification process.

2. When a discrepancy occurs between the expected and observed behavior, the scoreboard assists in the debugging process. By closely examining the inputs, outputs, and internal state of the DUT, engineers can trace the source of the error and identify problematic areas in the design.

3. It can provide valuable coverage metrics to assess the quality and completeness of the verification process. By tracking the number of transactions processed and the coverage of different scenarios, engineers can determine the effectiveness of the testbench and identify any gaps in verification.

4. It can be reusable across different testbenches and projects. Once implemented and verified for a specific interface or protocol, they can serve as a reference model for future projects, saving time and effort in developing new verification components.

5. It can handle complex verification scenarios involving multiple inputs, outputs, and protocols. They can be designed to support various levels of abstraction and can scale with the complexity of the design, making them suitable for verifying large and intricate systems.

6. The presence of a well-validated and functioning scoreboard provides confidence in the verification process. It demonstrates that the DUT meets the expected requirements and helps in

achieving verification sign-off, indicating that the design is ready for deployment or further stages of the development cycle.

Overall, scoreboards in UVM are essential components that enhance functional verification, aid in debugging, provide coverage analysis, enable reusability, and ensure scalability. They contribute to the overall reliability and quality of the verification process, ultimately leading to more robust and dependable designs.

2.2 Instruction Set Simulator (ISS)

An Instruction Set Simulator (ISS) is a software tool that simulates the behavior of a computer's instruction set architecture (ISA). It allows developers to analyze and evaluate the execution of programs without the need for physical hardware. One popular ISA that has gained significant attention in recent years is RISC-V.

The RISC-V instruction set simulators that are mostly utilized in the industry are as follows:

1. **Spike:** Spike is the reference RISC-V ISS provided by the RISC-V Foundation. It is a highly configurable and accurate simulator written in C++. Spike is primarily used for architectural exploration, development of operating systems, and running early-stage software.

2. **Whisper:** Whisper ISS is a valuable tool for anyone who is involved in the development or use of RISC-V processors or software. It is a free and open-source simulator that is easy to use and provides a wide range of features.

Here are some advantages of using RISC-V Instruction Set Simulator :

1. RISC-V is an open-source ISA, and the corresponding ISS tools are also open-source. This openness fosters collaboration, innovation, and customization, allowing developers to modify and extend the simulator to suit their specific requirements.

2. It offers a high degree of configurability, allowing developers to simulate different RISC-V architectures, extensions, and system configurations. This flexibility enables architectural exploration, performance analysis, and experimentation with novel designs.

3. Since RISC-V is designed to be platform-independent, RISC-V ISS tools can be run on various operating systems and hardware platforms. This portability ensures that developers can utilize the simulator in their preferred environment without being tied to specific hardware
.

4. It provides rich debugging and analysis capabilities, enabling developers to trace the execution of instructions, inspect register values, set breakpoints, and analyze program behavior. These

features aid in understanding and troubleshooting software at the instruction level.

5. It facilitates early-stage development and testing of software before the availability of physical hardware. This capability allows developers to start software development and validation at an early stage, reducing time-to-market and enabling rapid prototyping.

6. RISC-V ISS simulators like RARS offer a user-friendly interface and serve as valuable educational tools for learning RISC-V assembly programming. They provide a visual representation of the execution flow, aiding students in understanding the concepts of computer architecture and programming.

Overall, RISC-V ISS tools provide an open-source, flexible, and portable environment for simulating RISC-V architectures. They offer advantages such as configurability, debugging capabilities, early development support, and educational value, making them valuable tools for software development, research, and education.

Chapter 3

3.1 Generator

A generator plays a crucial role of generating stimuli which serves as the basis of design verification and allows engineers to produce accurate and productive stimulus for complex digital designs. The design under test (DUT) requirement for data and instructions are fulfilled by the generator for the purpose of effective verification. In the context of the Python implementation a Generator is a python class that generates stimuli to test a digital design or system.

Generators can be designed to produce different types of stimulus, such as random or deterministic sequences, and can be configured to control various aspects of the generated stimulus, such as timing and data rates. They can also be synchronized with other generators and components in the testbench to ensure proper sequencing of events

In our framework, the Generator class serves as a versatile component responsible for two fundamental tasks: generating random instructions and random data points. It seamlessly integrates into the UVM architecture and efficiently operates within UVM phases as shown in Figure 3.1.1.

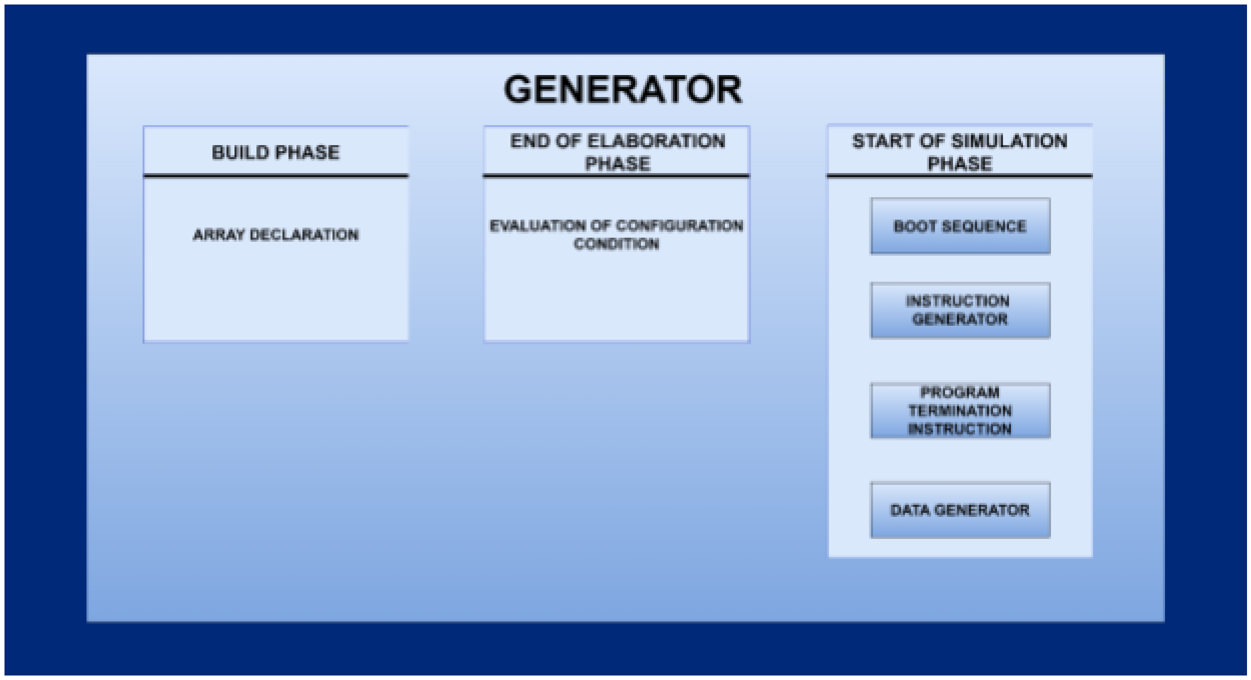

Figure 3.1.1: Block Diagram of Generator Component

During the build phase, arrays are meticulously crafted to pass a variety of data to the generator functions. The elaboration phase plays a pivotal role, incorporating essential assertions that ensure the precise functioning of the generator. If a user requests an unsupported extension, the generator promptly issues an error message, terminating the test. Similarly, if an incompatible test name is provided, a corresponding error message appears, leading to the test's termination.

Upon successfully navigating these initial checks, the generator proceeds to construct an opcode list based on the user-specified extensions, which can include RV32I, RV32M, and CSR instructions. The entire flow of the end of the elaboration phase is meticulously depicted in Figure 3.1.2, providing a visual representation of the process.

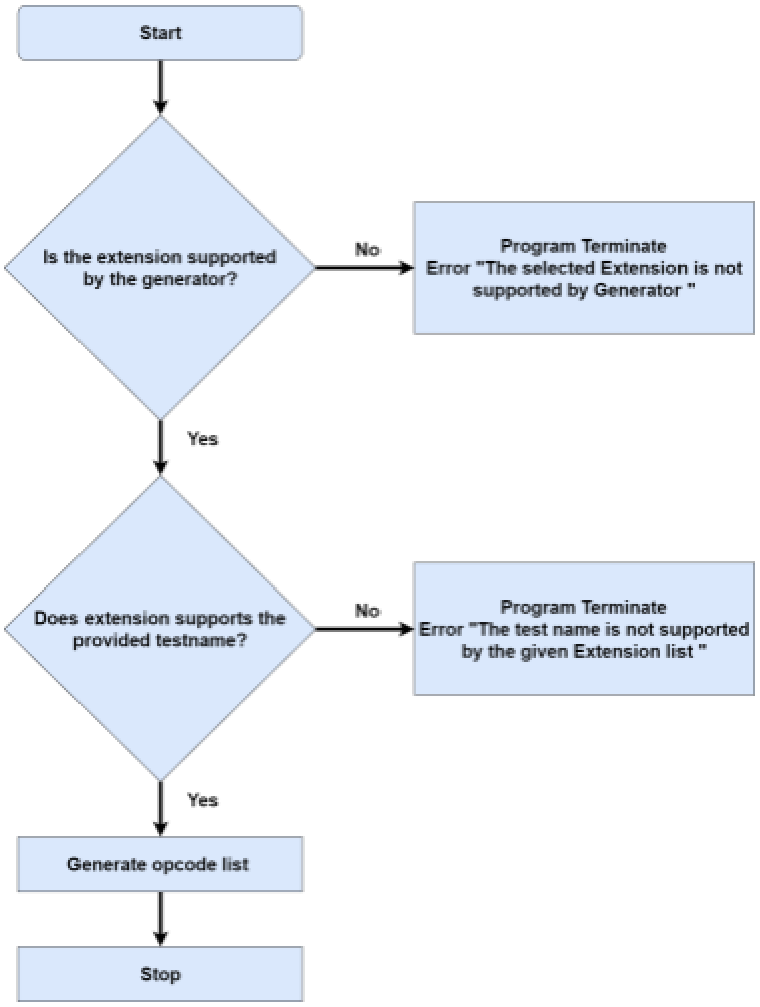
Start
Is the extension supported by the generator?
No
Program Terminate Error "The selected Extension is not supported by Generator "
Yes
Does extension supports the provided testname?
No
Program Terminate Error "The test name is not supported by the given Extension list "
Yes
Generate opcode list
Stop

Figure 3.1.2: Execution Flow Chart of Generator End of Elaboration Phase

Transitioning to the simulation phase, the generator executes a series of functions, meticulously choreographed as illustrated in Figure 3.1.3. It commences by generating a boot sequence, a critical step in initializing stack pointers and registers. To ensure program termination, a specific instruction is crafted. Subsequently, user-defined test instructions are generated, encompassing the instructions from

RV32I, RV32M, and CSR sets. To facilitate efficient program execution, random data points are thoughtfully generated.

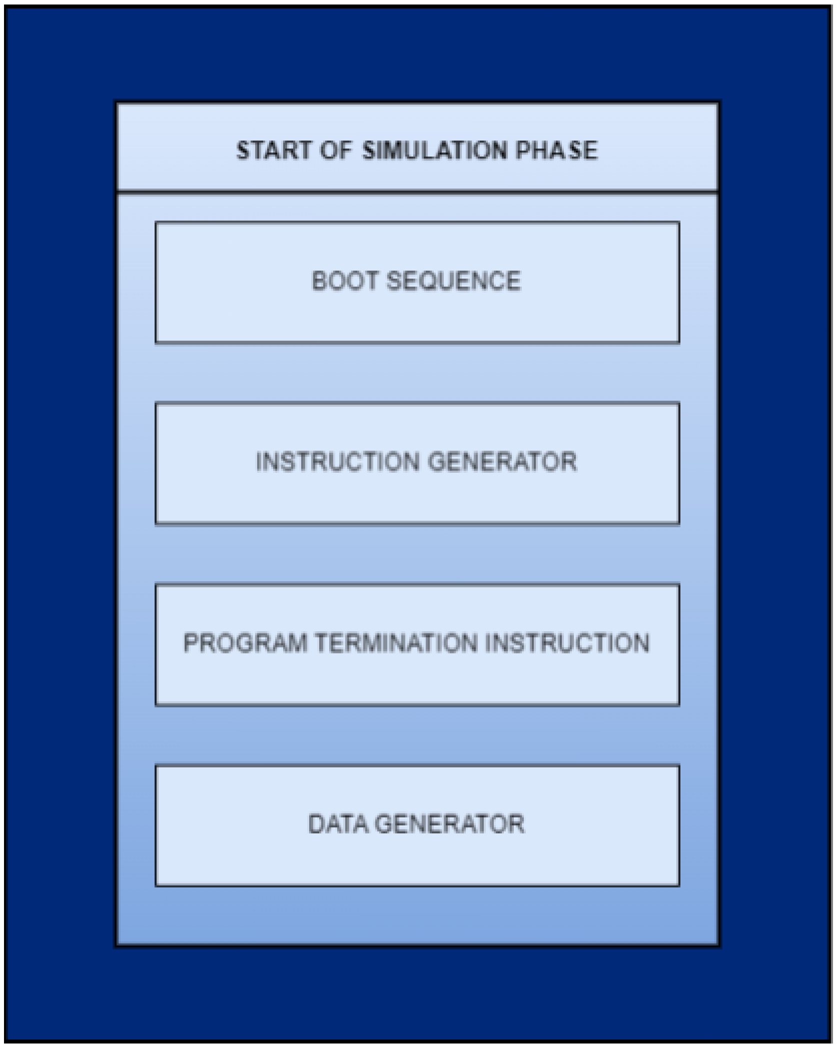

Figure 3.1.3: Execution Sequence of Functions within the Start of Simulation Phase

At the heart of this framework lies the Instruction Generator function, an indispensable component responsible for crafting user-desired instructions from RV32I, RV32M, and CSR instruction sets. This function takes as input the test name, test iteration, and the opcode list, which may contain instructions from these sets. It employs a sophisticated opcode selector to randomly choose an opcode based on the provided test name. The instruction function skillfully generates an instruction corresponding to the selected opcode. Prior to appending the generated instruction to the list, a meticulous check ensures it is not among the excluded instructions designated by the user. This iterative process continues until the instruction count aligns with the specified test iterations.

The generator's enhanced capabilities now encompass the ability to generate instructions from RV32I, RV32M, and CSR instruction sets, significantly broadening its functional scope. Users are granted enhanced configurability through the configuration file, allowing them to selectively include or exclude specific instructions and CSR numbers. Furthermore, users can configure distinct privilege modes, which dictate access to system resources, memory management, and operational oversight. Each privilege mode offers varying levels of access to hardware resources and instruction sets.

While generating CSR test instructions, it checks if the extra included CSR numbers in the configuration file are supported by the generator. If any of the CSR numbers are not supported, it raises an assertion and terminates the test.

In summary, the SweRV EL2 framework's generator has undergone substantial enhancements, expanding its capabilities to support a broader range of instructions, including RV32I, RV32M, and CSR instructions. Its enhanced configurability and seamless integration of extensions have made it an even more powerful tool for testing and validation within the framework. The generator's ability to validate CSR numbers in the configuration file ensures the accuracy and reliability of CSR test generation.

The complete explanation of the flow of the generator was discussed above. A pictorial representation of the flow can be observed in Figure 3.1.4.

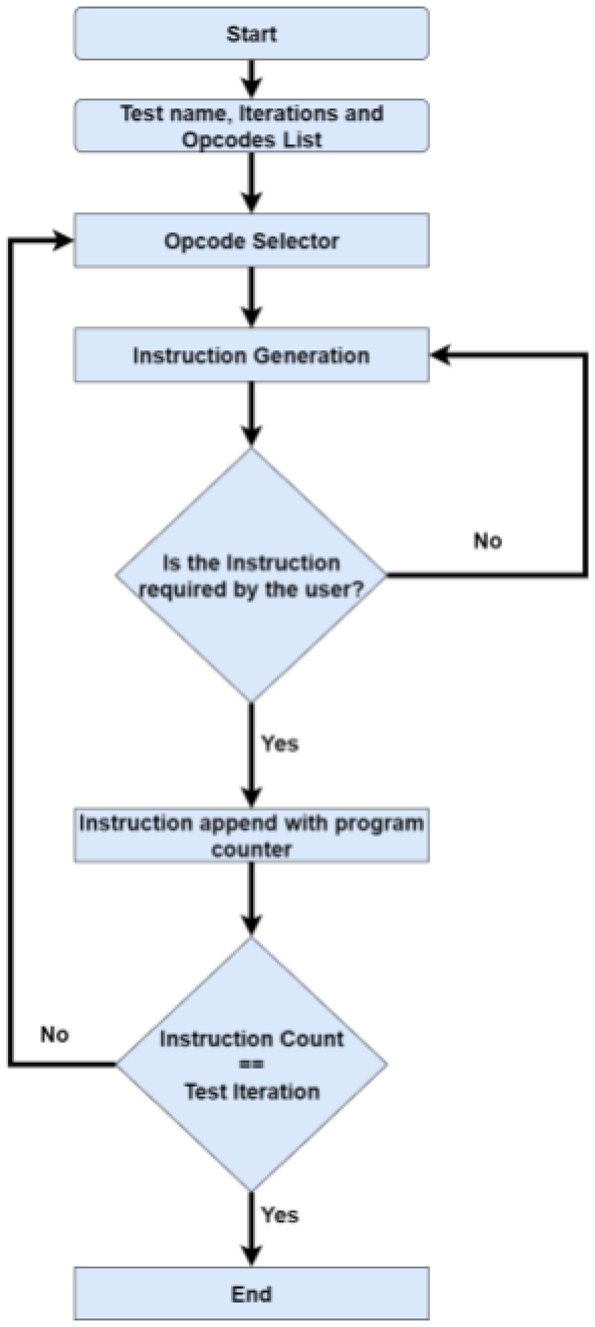
Start
Test name, Iterations and Opcodes List
Opcode Selector
Instruction Generation
Is the Instruction required by the user?
No
Yes
Instruction append with program counter
Instruction Count == Test Iteration
No
Yes
End

Figure 3.1.4: Execution Flow Chart of Generator Component

.

3.2 AXI4 Bus Functional Model (BFM)

The Bus Functional Model (BFM) is a fundamental component of the Universal Verification Methodology (UVM) that allows verification engineers to model the behavior of a design's interface protocol. The BFM represents a specific bus protocol, such as AXI4, AHB, or PCIe, and is used to generate transactions that can be used to stimulate the design's interface.

AXI4 (Advanced eXtensible Interface 4) stands as a prominent bus protocol frequently employed in computer architecture and digital system design, particularly in System-on-Chip (SoC) frameworks. Developed and maintained by ARM, a distinguished technology company renowned for its contributions to semiconductor intellectual property (IP) and microprocessor architectures, AXI4 is tailored for high-performance, on-chip communication.

AXI4 offers a range of features that enhance system efficiency and performance. Burst transactions, separate read and write channels, adaptable addressing modes, data width scalability, and Quality of Service (QoS) prioritization collectively contribute to swift movement of multiple data items, concurrent read and write activities, flexible addressing, configurable data widths, and prioritized critical data transfers. With these attributes, AXI4 serves as a versatile and high-performance interface, particularly for intricate System-on-Chip (SoC) designs

The Interface in standard UVM is referred to as the Bus Function Model (BFM) throughout this document. In the PY-UVM framework BFM is kept separate from the framework and designed as a separate module to interact with the DUT. The main objective to keep the BFM separately is that it can be changed as per the design requirement.

BFM is not inherited from the UVM component and is designed as a singleton class. The reason for designing it as a singleton class is because in our framework we have only one BFM whose main objective is to communicate with the DUT using the BFM object.

Effective communication between different components is a critical aspect of system design, particularly in the context of complex architectures like the SweRV EL2 core. To facilitate seamless interaction between the Design Under Test (DUT) and the framework, the Bus Functional Model (BFM) was updated within the PY-UVM framework. This adaptation is specifically tailored to accommodate the utilization of the AXI4 bus protocol by the SweRV EL2 core.

The BFM comprises four primary functions, as depicted in Figure 3.2.1. Among these, the "clock" function is responsible for generating the clock signal, while the "reset" function provides the necessary reset signals to the design.

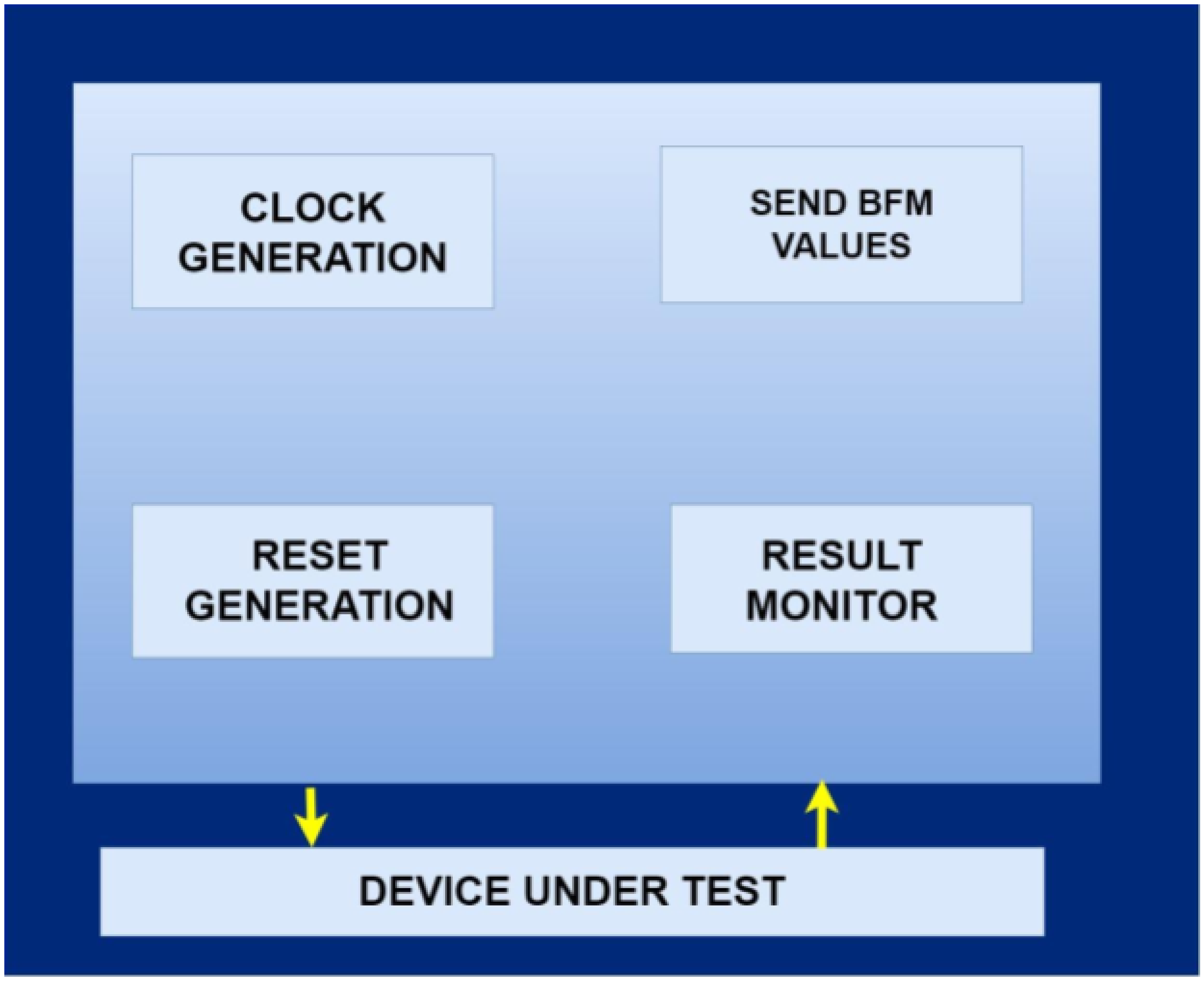

Figure 3.2.1: Block Diagram of AXI4 Bus Functional Model

In the "send BFM value" function, the BFM now incorporates the AXI4 protocol for handling data and instructions directed towards the DUT. This process involves receiving values from the driver, storing them sequentially in a designated queue, and ensuring compliance with the AXI4 protocol's requirement for a 64-bit value. The queue's length is continuously monitored, and when it reaches a count of 2, indicating the assembly of a complete 64-bit value, the value is dispatched to the DUT for further processing. If the count remains below 2, the procedure continues to gather additional values from the driver. This iterative process ensures the accurate assembly and formatting of driver-received values. Figure 3.2.2 provides a visual representation of this operational flow.

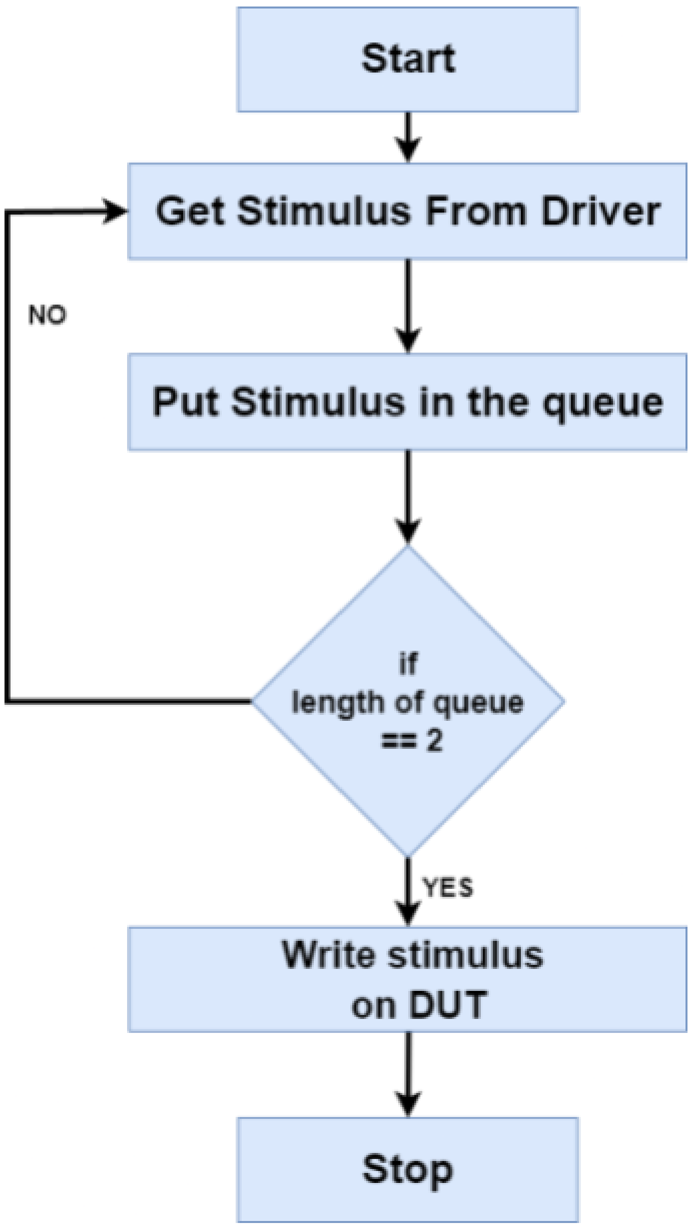

Figure 3.2.2: Execution Flow Chart of AXI4 BFM

Unlike traditional BFM result monitoring, the PY-UVM framework for SweRV EL2 introduces an innovative approach. Instead of depending on dedicated ports, we now employ the monitoring of the "finish ecall" signal generated by the DUT. When this signal transitions to a high state, it signifies the completion of the DUT's program execution. This transition provides access to the log file containing the results obtained from the DUT, facilitating efficient analysis and evaluation.

3.3 Sequencer

Imagine you have a DUT that needs to follow a set of instructions in order to perform a task. The DUT can't perform by itself, so it needs a "sequencer" to tell it what to do. The sequencer that is implemented in the PY-UVM framework is a little bit different from the sequencer that is implemented in the UVM. The purpose of the sequencer in the framework is that it takes stimuli from the generator and transfers it to the DUT via Driver.

Sequencer is a class that is inherited from the uvm_component class and as stated above, when a class inherits from the uvm component class, it inherits all of the uvm phases as well.

Initially in the build phase function, a handler of Generator Class is initialized by using the ConfigDB. After that one of the TLM analysis FIFO ports are initialized which are used to connect the sequencer with Driver.

Next, the execution of the run_phase function starts. This is where the Sequencer really starts doing its job. The Sequencer starts sending the instructions generated by the generator to the DUT via Driver using a TLM analysis FIFO blocking port. The main advantage of using a blocking port is that it will not put new data in the FIFO until the FIFO is empty. Sequencer transmits the stimulus by looping through each instruction in the generated instructions list, and sending it one at a time until it encounters the ECALL instruction. ECALL instruction indicates that all of the generated instructions are transmitted and now the sequencer can transmit the random data generated by the generator for the smooth execution of the program. The sequencer transmits the generated data to the DUT via Driver using the same blocking port in the same pattern as it transmits the instructions.

3.4 Driver

When the stimulus is generated by the generator for the DUT and the sequencer receives that stimulus in order to transfer that stimulus to the DUT, we use Driver to transfer that stimulus.

Driver is one of the PY-UVM framework components inherited from the UVM component as well as it inherits all the uvm phases too. The Driver component's main function is to transfer the stimulus generated by the generator / tester to the DUT via a specific protocol.

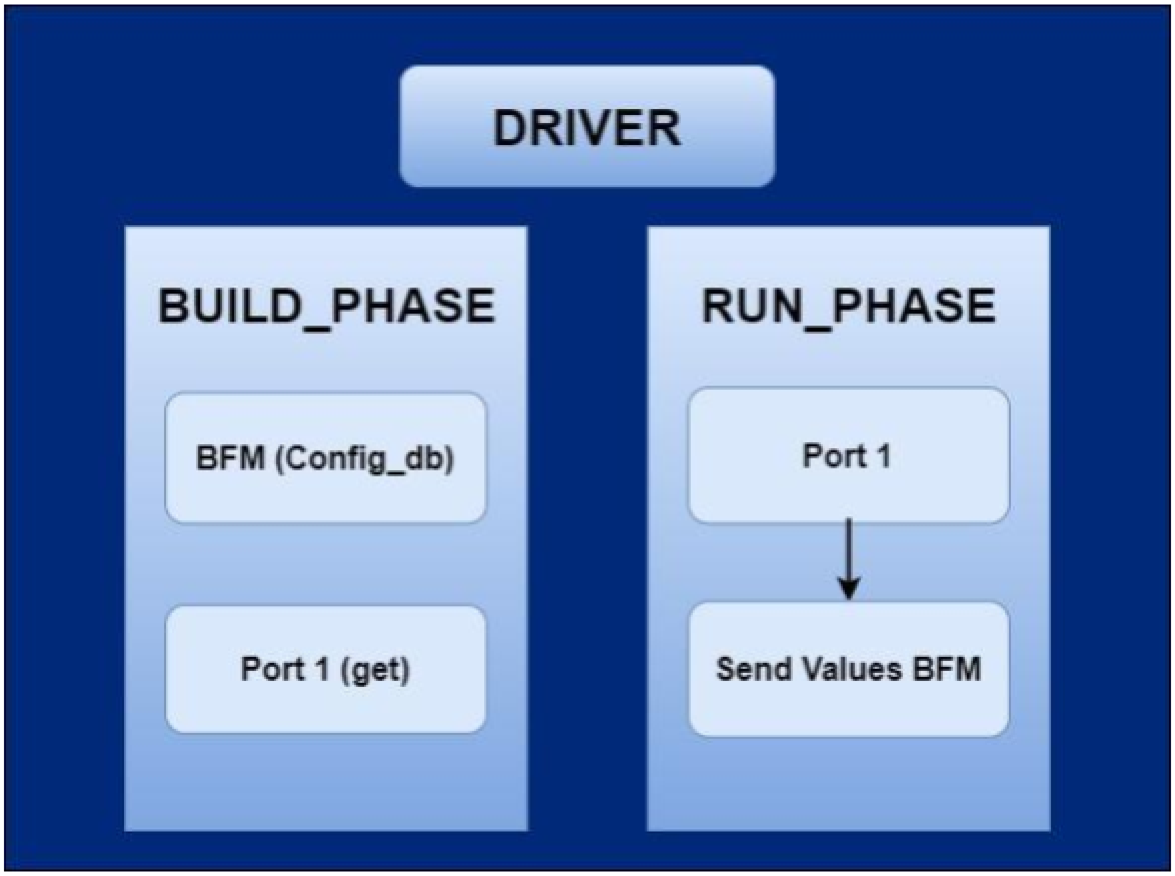

Figure 3.4.1: Block Diagram of Driver Component

Initially in the build phase function, a handler of BFM Class is initialized by using the ConfigDB. After that one of the TLM analysis FIFO ports are initialized which is used to connect the sequencer with Driver.

Next, the execution of the run_phase function starts. This is where the Driver really starts doing its job. The Driver starts receiving the stimulus from the Sequencer using a TLM analysis FIFO blocking port and sends it to the DUT via BFM. When the TLM FIFO port is full, we get the stimuli and send them to the DUT via the BFM function ‘send values bfm'. During the entire transaction, the driver does not perform another transaction until it is completed. This process is carried out in the while loop and continues until the monitor's run phase drops the objection.

The complete explanation of the flow of the driver was discussed in implementation. A pictorial representation of the flow can be observed in Figure 3.4.2.

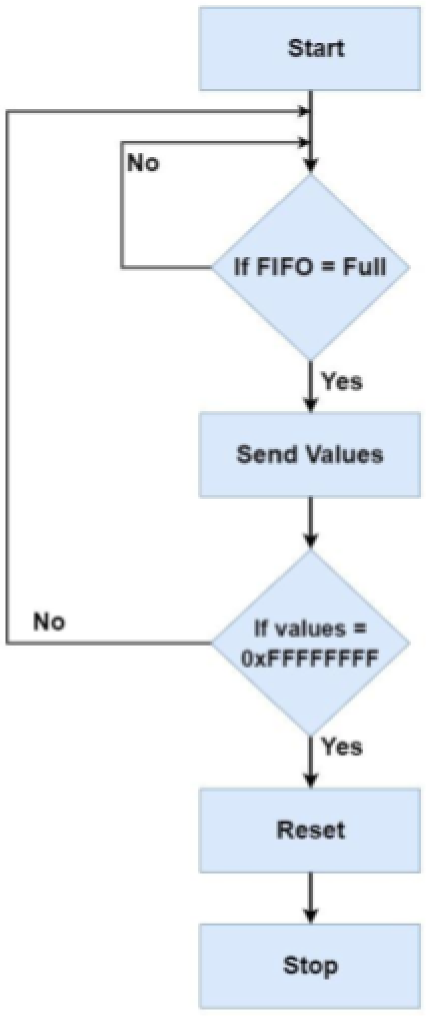

Figure 3.4.2: Execution Flow Chart of Driver Component

3.5 Monitor

The monitor component integrated into the PY-UVM framework of SweRV EL2 assumes a pivotal role in the verification process. This component is designed to meticulously observe and record outcomes emerging from the Design Under Test (DUT) during the testing phase which is shown in Figure 3.5.1 below.

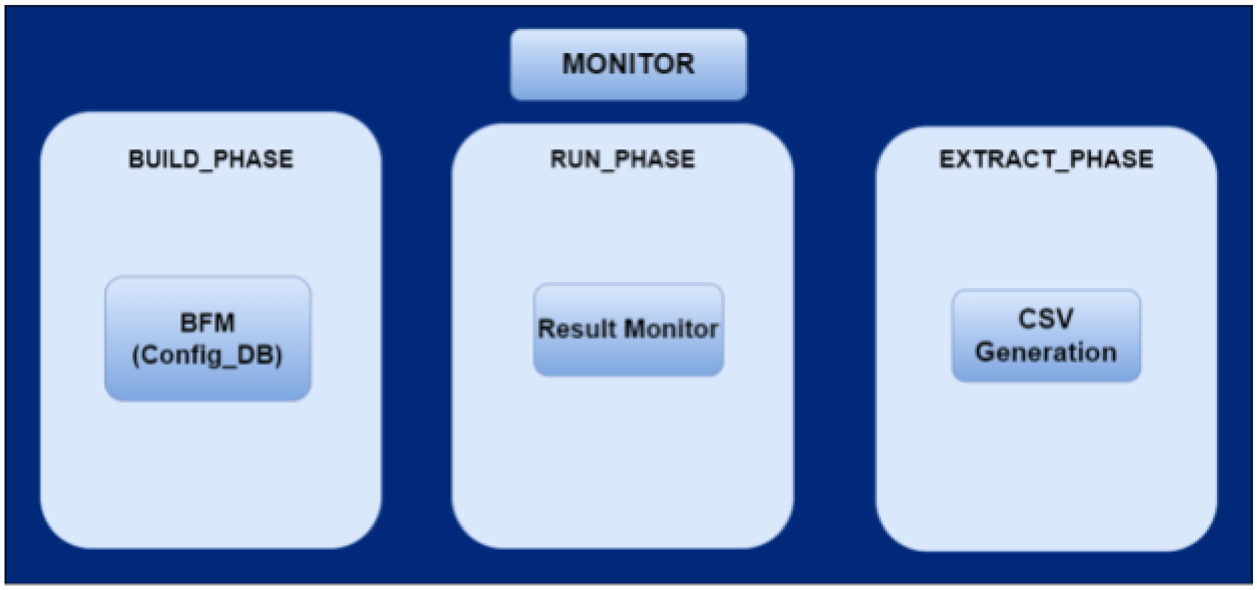

Figure 3.5.1: Block Diagram of Monitor Component

The monitoring procedure commences with the establishment of vital lists and variables. It encompasses the conversion of a log file sourced from the SweRV EL2 core into a structured dataframe, achieved through the application of scripting techniques. This dataframe is subsequently subjected to manipulation to extract pertinent data, essential for subsequent analytical stages.

To enhance data accuracy, the monitor systematically focuses on the "Result" column, systematically eliminating extraneous details such as register names. This curation process streamlines the data, retaining only pertinent information for subsequent analysis. The curated data is then methodically stored as a CSV file, poised for future reference and in-depth scrutiny.

In conclusion, the monitor component's systematic approach guarantees accurate data extraction, coupled with an organized methodology for refinement. This process substantially enhances the quality of analysis and comprehension, thereby bolstering the overall effectiveness of the verification process.

Furthermore, the illustrative flowchart in Figure 3.5.2 provides a visual representation of the Monitor Component's pivotal role. It depicts the sequential process of extracting, refining, and organizing results from the Design Under Test (DUT), reinforcing the comprehensive approach taken in this verification framework.

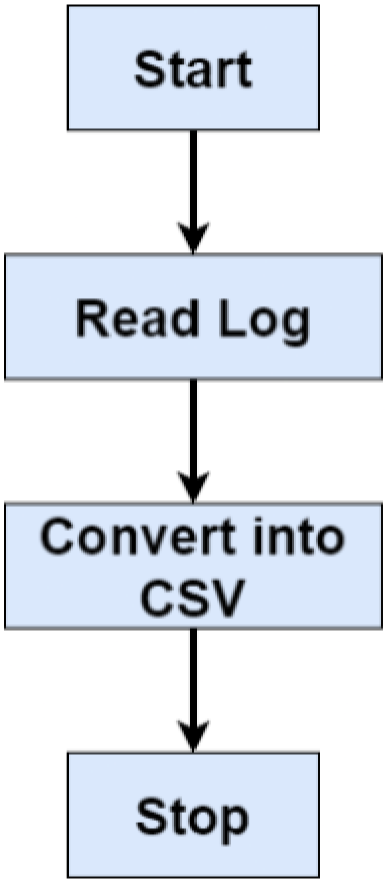

Figure 3.5.2: Execution Flow Chart of Monitor Component

3.6 Instruction Set Simulator (ISS)

ISS is one of the PY-UVM framework components that was inherited from the UVM component. The primary goal of the ISS component is to execute the generated stimulus on whisper, which refers to the Golden Model of RISC-V based architecture.

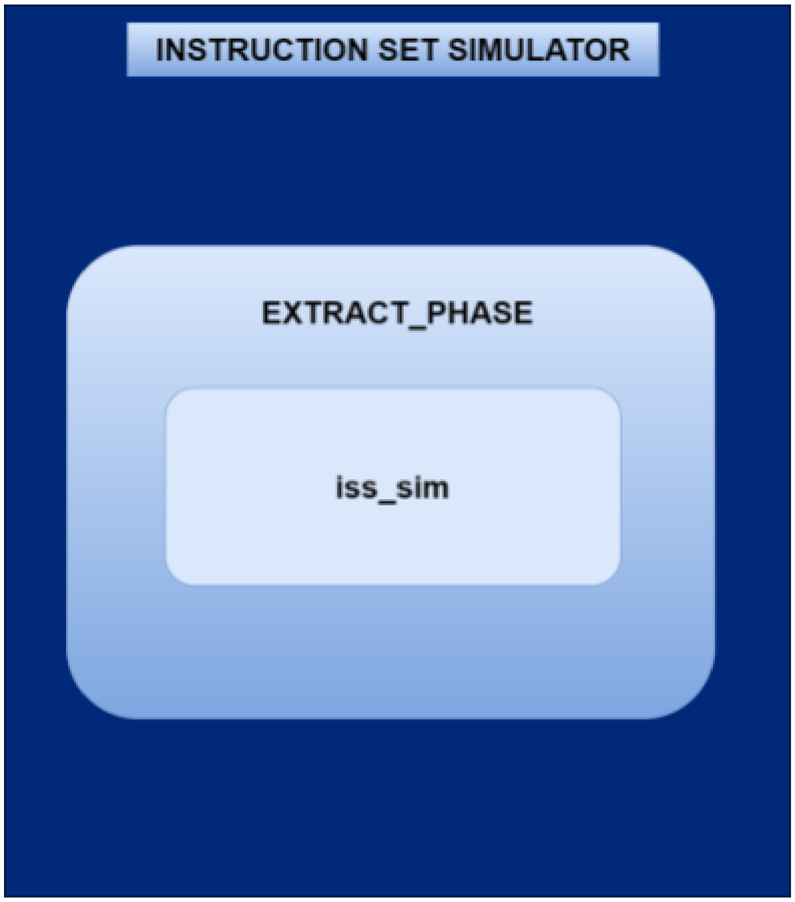

Figure 3.6.1: Block Diagram of ISS Component

As illustrated in Figure 3.6.1, we use the UVM extract phase, which executes after the UVM run phase, during which ISS SIM performs its functionality. ISS SIM initially reads the csv file generated by the generator during the extraction phase. Following that, we extract data and instructions from the csv file one by one and convert them to hex format in accordance with whisper's specifications. Following conversion, the produced hex file is simulated on whisper using 'os command'. Whisper ISS generates a log file after a successful simulation. Lastly, the log file is converted into CSV, which is then fed into the scoreboard for comparison.

The complete explanation of the flow of the ISS was discussed above. A pictorial representation of the flow can be observed in Figure 3.6.2.

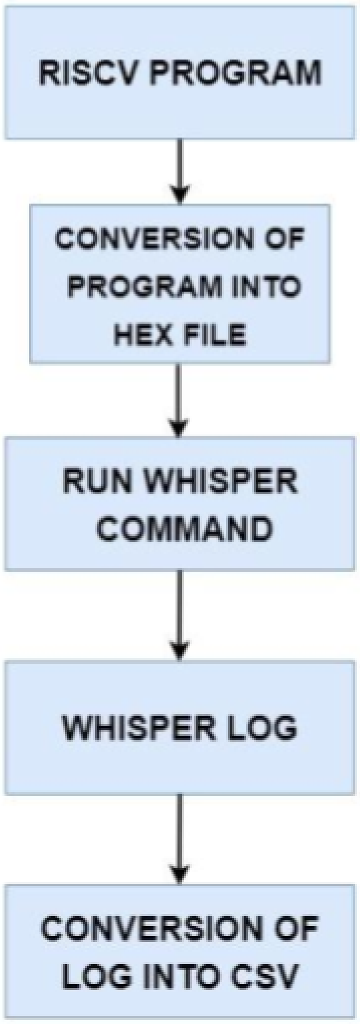
RISCV PROGRAM
CONVERSION OF PROGRAM INTO HEX FILE
RUN WHISPER COMMAND
WHISPER LOG
CONVERSION OF LOG INTO CSV

Figure 3.6.2: Execution Flow Chart of ISS Component

3.7 Scoreboard

Scoreboard is one of the PY-UVM framework components inherited from the UVM component. The main objective of the scoreboard component is to compare the results obtained from the design under test (DUT) and Golden Model (Whisper ISS).

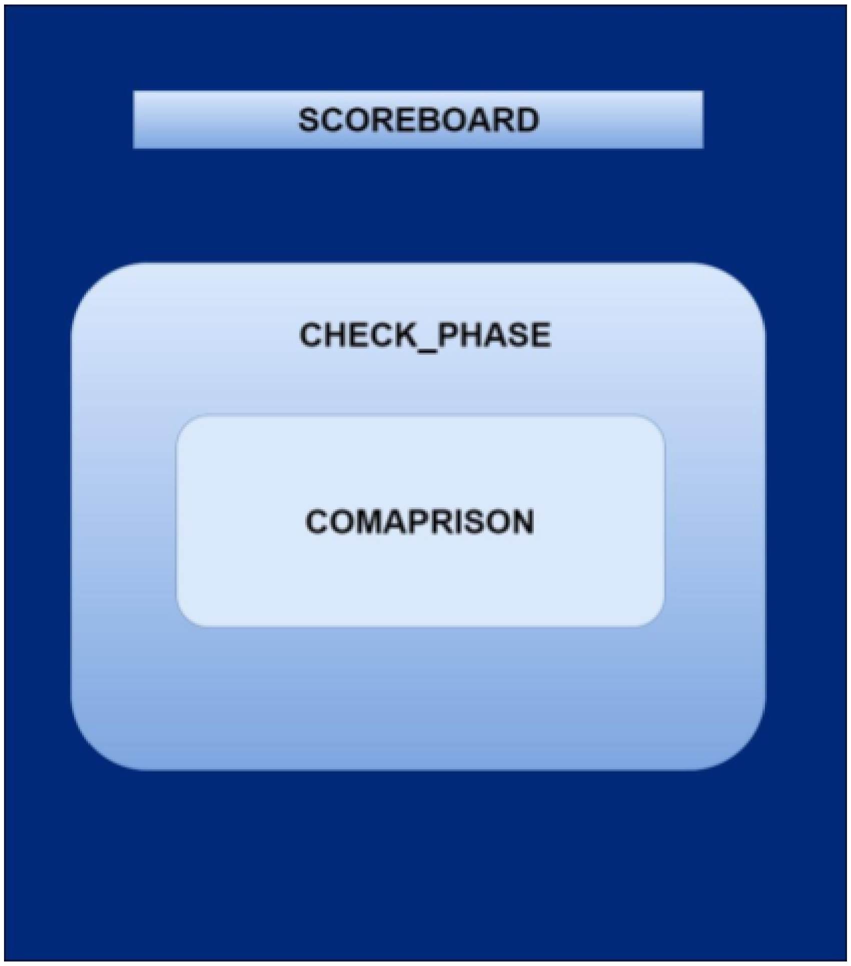

Figure 3.7.1: Block Diagram of Scoreboard Component

As previously stated, inheriting a UVM component gives us the ability to utilize its inherent phases. The UVM Check phase has been utilized to execute the scoreboard as shown in Figure 3.7.1. The scoreboard analyzes both the CSV files generated by the DUT and the ISS and compares them sequentially and informs the user about the test status.

The complete explanation of the flow of the scoreboard was discussed above. A pictorial representation of the flow can be observed in Figure 3.7.2.

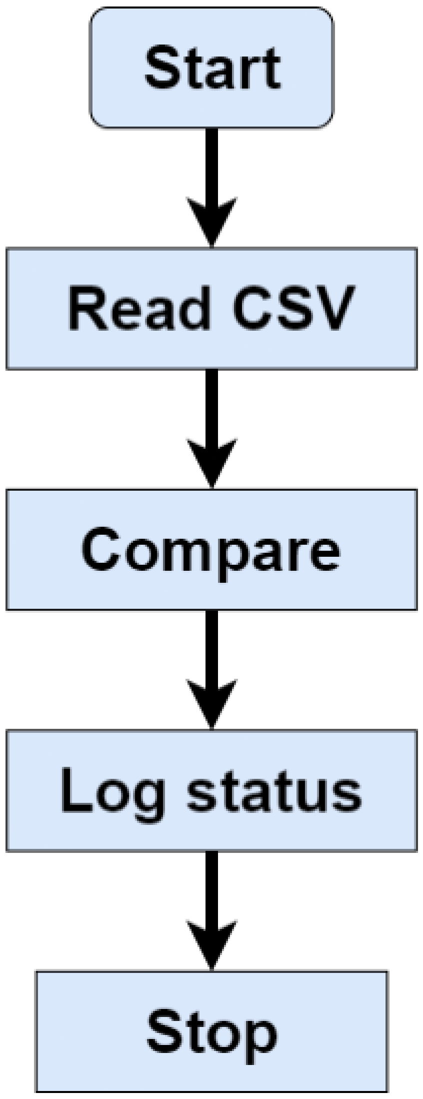

Figure 3.7.2: Execution Flow Chart of Scoreboard Component

3.8 Coverage

In the SweRV EL2 framework, one of the inherited components from the UVM architecture is the Coverage component, which plays a crucial role in reporting both functional and code coverages achieved by the framework. The primary focus of the Coverage component is to provide insights into the verification engineer's progress in achieving the required cover points defined by the Design Under Test (DUT). The Coverage component, depicted in Figure 3.8.1, operates within two inherent UVM component phases.

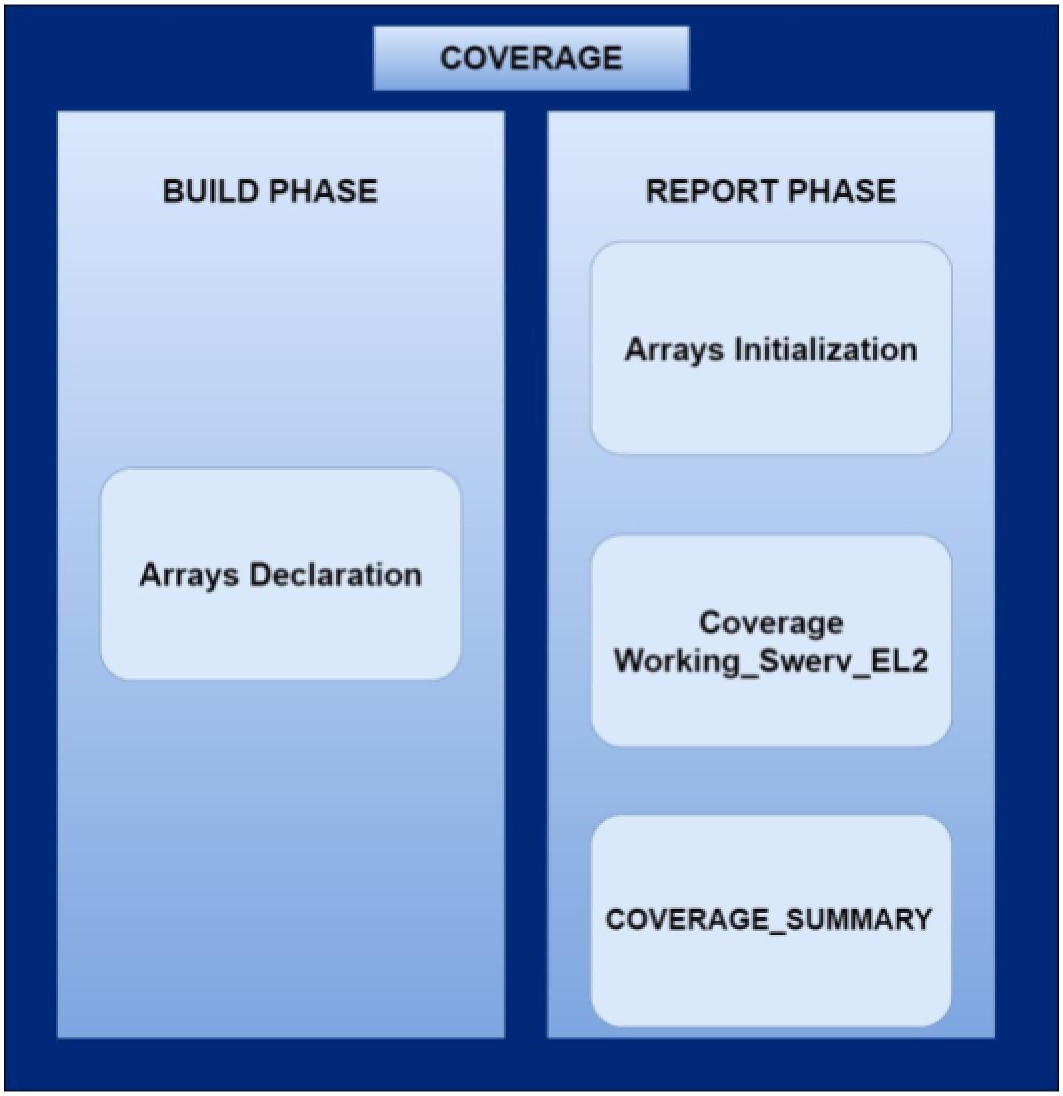

Figure 3.8.1: Block Diagram of Coverage Component

During the build phase, all necessary array declarations are made. Since coverage assessment occurs at the end of each test, the report phase is used instead of the run phase. When the report phase begins execution, the previous coverage report is invoked, and previous information is initialized to the arrays. In the absence of a previous report, the arrays are initialized with default values. Following initialization, another function is called to evaluate the following functional coverages:

1) Instructions: This coverage metric informs the verification engineer about the number of instructions that have been functionally verified.

2) Registers: It tracks how many times a particular register has been used as both a source and a destination during testing.

3) Sign of Immediates: This coverage metric provides insights into the types of immediate numbers generated by the generator.

The fundamental benefit of coverage analysis is its ability to indicate the verification engineer's progress in achieving the required cover points established by the DUT.

In the context of the SweRV EL2 framework, a key implementation is the generation of a comprehensive summary report. This report provides a consolidated presentation of crucial metrics, including the count of executed instructions for each extension. It offers a clear understanding of the types of instructions utilized during testing. Additionally, the report furnishes insights into the quantity of unique registers accessed during execution, providing essential data for evaluating test coverage effectiveness.

The complete explanation of the flow of the coverage was discussed above. A pictorial representation of the flow can be observed in Figure 3.8.2.

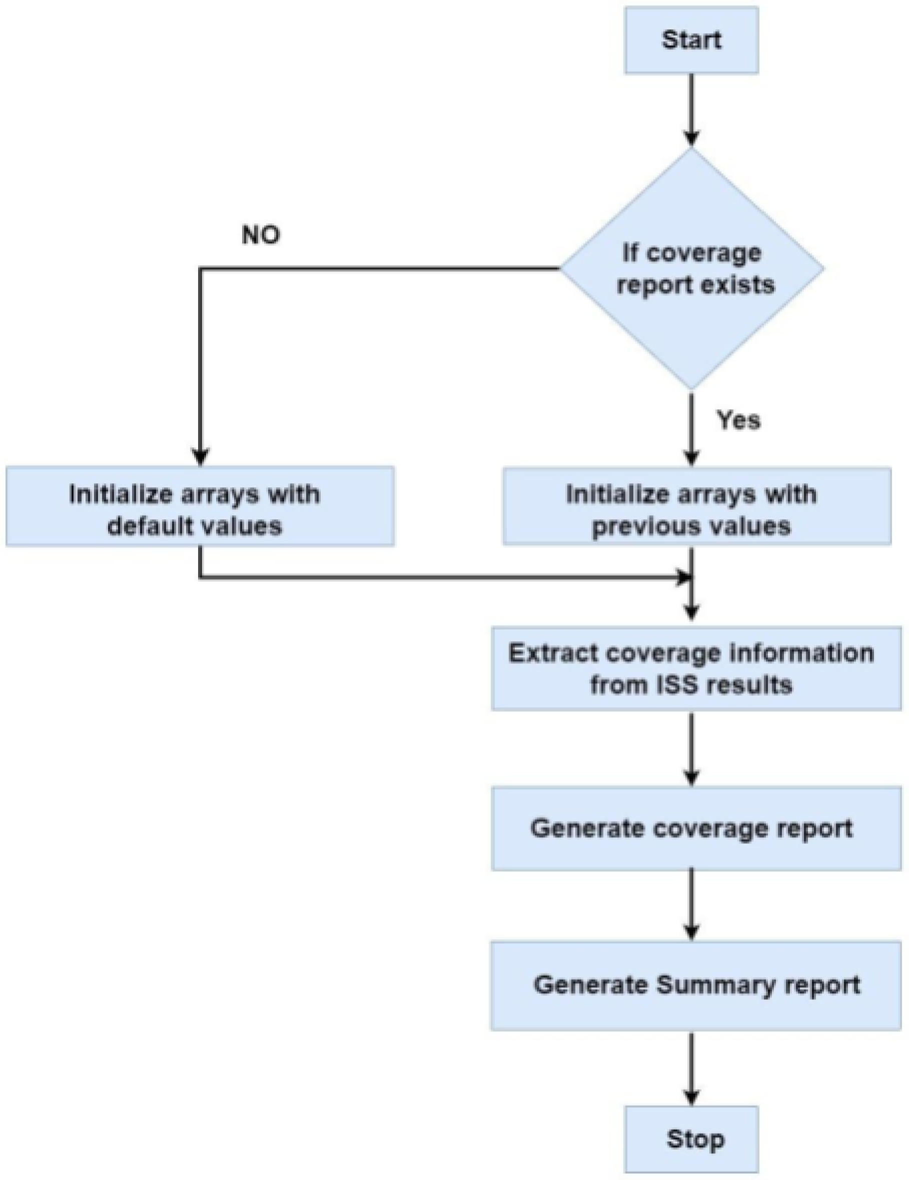
Start
If coverage report exists
NO
Yes
Initialize arrays with default values
Initialize arrays with previous values
Extract coverage information from ISS results
Generate coverage report
Generate Summary report
Stop

Figure 3.8.2: Execution Flow Chart of Coverage Component

3.9 PY-UVM Framework TOP for SweRV EL2 Core

The PY-UVM framework customized for SweRV EL2 embodies a high-level implementation meticulously crafted to enhance functionality and execution flow, ensuring comprehensive verification and validation of the Design Under Test (DUT). This comprehensive framework comprises various key components, depicted in Figure 4.41. These components, including the Generator, Sequencer, Driver, Monitor, Scoreboard, Instruction Set Simulator (ISS), Bus Function Model (BFM), Coverage (CVG), TLM FIFOs, and Agent, collaboratively streamline the verification process, yielding precise and reliable results.

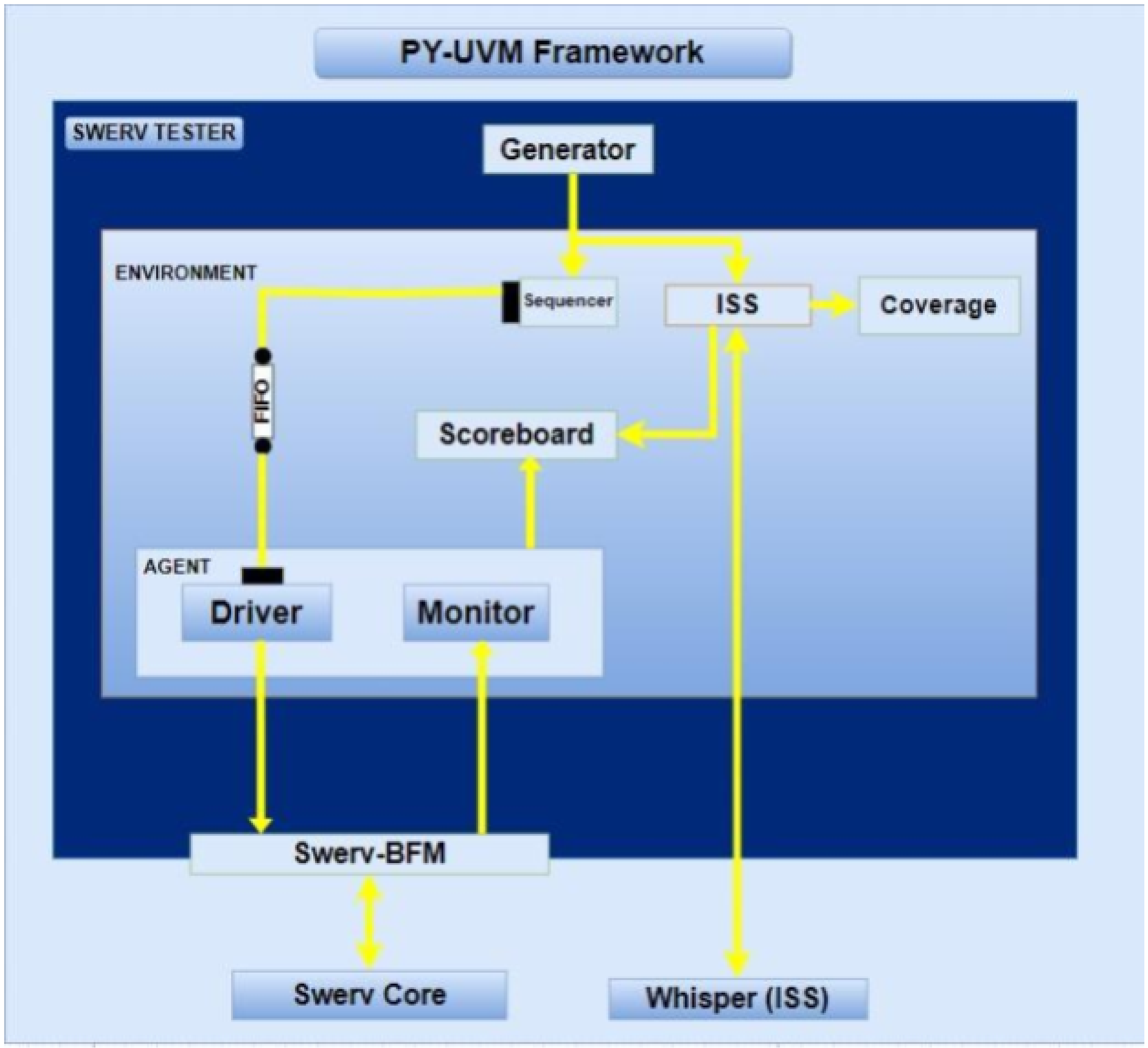

Figure 3.9.1: Block Diagram of PY-UVM Framework for SweRV EL2 Core

Figure 3.9.1 provides a block diagram of the framework implementation, indicating the initialization of the following components:

1) Generator
2) Sequencer
3) Agent
4) Driver
5) Monitor
6) Scoreboard
7) Instruction Set Simulator (ISS)
8) Bus Function Model (BFM)
9) Coverage (CVG)
10) TLM FIFOs

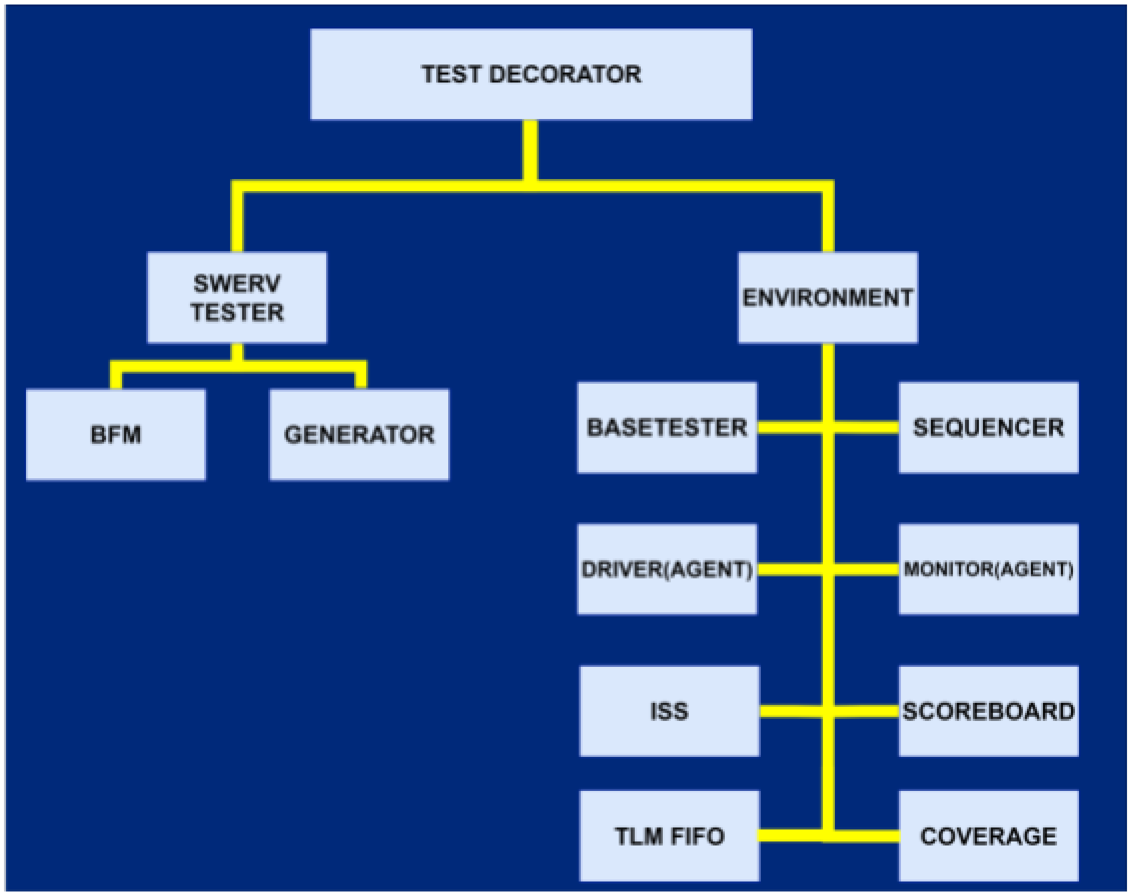

Figure 3.9.2: Component Hierarchy in PY-UVM Framework for SweRV EL2 Core

The framework follows a well-defined execution sequence. After the build phase for all components is completed, the connect phase of the Base Tester component establishes crucial connections between the Sequencer and agent components using TLM FIFOs. Within the Agent class, instances of the Driver and Monitor components are instantiated, and their respective handlers are defined during the Agent Class's build phase.

During the Start of Simulation phase, the Generator generates random stimuli. Once the Generator's task concludes, the run phase of the Sequencer, Driver, and Monitor components is initiated. In this phase, the Sequencer transmits stimuli to the FIFOs, while the Driver receives these stimuli from the FIFOs and conveys them to the DUT through the Bus Functional Model (BFM). The successful completion of these transactions triggers the generation of a log file, which is then used by the Monitor component to collect the DUT's results.

In the subsequent Extract phase, the same stimuli generated by the Generator are directed to the Golden Model (Whisper ISS) to validate the DUT's outcomes. After the Golden Model execution, the Scoreboard component comes into play during the check phase, meticulously comparing the results between the DUT and the Golden Model. Throughout this process, the coverage component remains consistent in its function, with the additional feature of generating a summary report.

In conclusion, the synergistic functioning of the components within the PY-UVM framework, bolstered by UVM phases, TLM FIFOs, UVM agents, and functional coverage analysis, plays a pivotal role in enabling comprehensive verification. This framework emerges as a potent instrument for a thorough assessment of SweRV EL2, guaranteeing meticulous attention to all critical facets and ultimately leading to the creation of a reliable and top-tier design. A carefully designed flowchart has been created to illustrate the workflow of the PY-UVM framework for the SweRV EL2 core, shown in Figure 3.9.3. This flowchart provides a visual representation of the high-level operations involved in the framework.

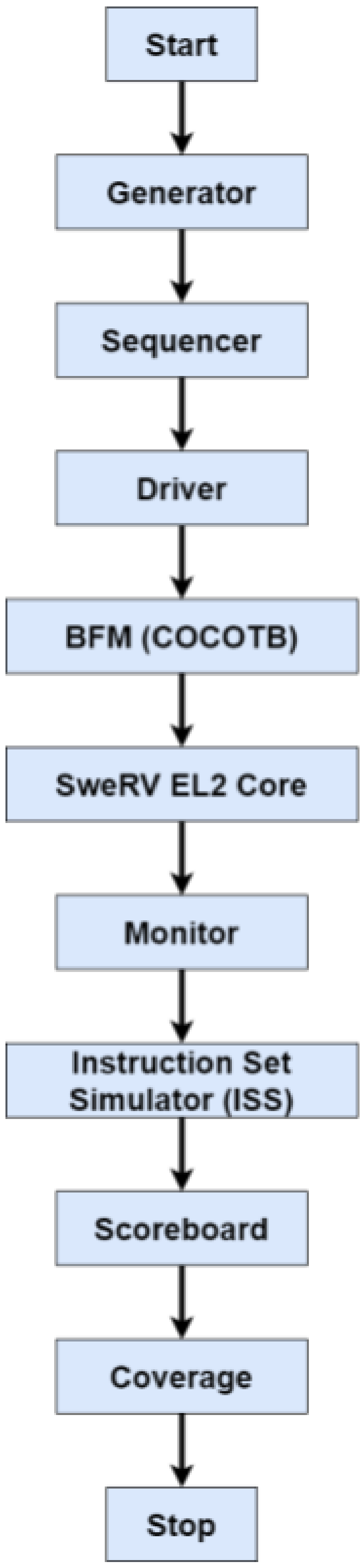

Figure 3.9.3: Execution Flow Chart of PY-UVM Framework TOP for SweRV EL2 Core