MERL
Accelerating Engineering Innovation

PY-UVM FRAMEWORK FOR RISC-V SINGLE CYCLE CORE

PROJECT BY:
MERL DSU

Table of Contents

Acknowledgements

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 initial implementation of Py-UVM Framework for RISC-V Single Cycle 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 RISC-V Single Cycle 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) **Reusability:** UVM TLM FIFO can be easily reused across multiple testbenches and verification environments, providing a standardized interface for communication between components.

2) **Interoperability:** UVM TLM FIFO supports multiple interfaces, such as the blocking, non-blocking, and streaming interfaces, allowing components with different communication requirements to communicate seamlessly.

3) **Abstraction:** UVM TLM FIFO provides a level of abstraction that enables components to communicate at a higher level of abstraction.

4) **Asynchronous Communication:** UVM TLM FIFO supports asynchronous communication, allowing components to operate at different speeds and reducing simulation overhead.

5) **Data Integrity:** UVM TLM FIFO provides built-in features for checking data integrity, ensuring that the data transmitted between components is correct.

6) **Configuration Flexibility:** UVM TLM FIFO provides several configuration parameters that allow users to customize its behavior according to their needs.

7) **Debugging:** UVM TLM FIFO 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) **Centralized storage:** The UVM configuration database provides a central location for storing and retrieving configuration information, making it easy to manage and maintain.

2) **Hierarchical configuration:** 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) **Parameterization:** The configuration database can be used to parameterize testbench components, allowing for easy reuse of components across different test scenarios.

5) **Flexibility:** The configuration database allows for different types of data to be stored, including integers, strings, and user-defined types, making it highly flexible.

6) **Easy access:** The UVM configuration database provides a simple API for accessing configuration information, making it easy to use and reducing the likelihood of errors.

7) **Modular design:** The configuration database is designed to be modular, allowing for different implementations to be used based on the needs of the testbench.

8) **Debugging:** The configuration database 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.

Advantages of using UVM interface in verification include:

1. **Reusability:** The UVM interface can be reused across multiple projects, allowing for easier development and maintenance of the verification environment.

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

3. **Easy Integration:** The UVM interface 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. **Debugging:** The UVM interface 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. **Scalability:** The UVM interface 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.

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 of the advantages of using logging in UVM:

1. **Debugging:** Logging 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. **Verification Planning:** 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. **Coverage Analysis:** Logging 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. **Efficiency:** By reducing the time required to debug issues, logging helps to improve the overall efficiency of the verification process.

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

6. **Ease of Use:** 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.

Chapter 3

3.1 Generator

Overview

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

Implementation

The generator design in our framework is responsible for two main tasks which are as follows:

1) Generating Instructions.
2) Generating Random Data points.

Generator class inherits the uvm component from which it was able to get different uvm phases. In our framework the execution of the generator was carried out using UVM three phases, as shown in Figure 3.1.1. To pass various arrays to the various functions of the generator, arrays are constructed during the build phase. Several assertions are included at the end of the elaboration phase to ensure the correct working of the generator. When the start of the simulation phase is executed different functions as shown in Figure 3.1.1 are executed, working of these functions are explained in detail in the section below.

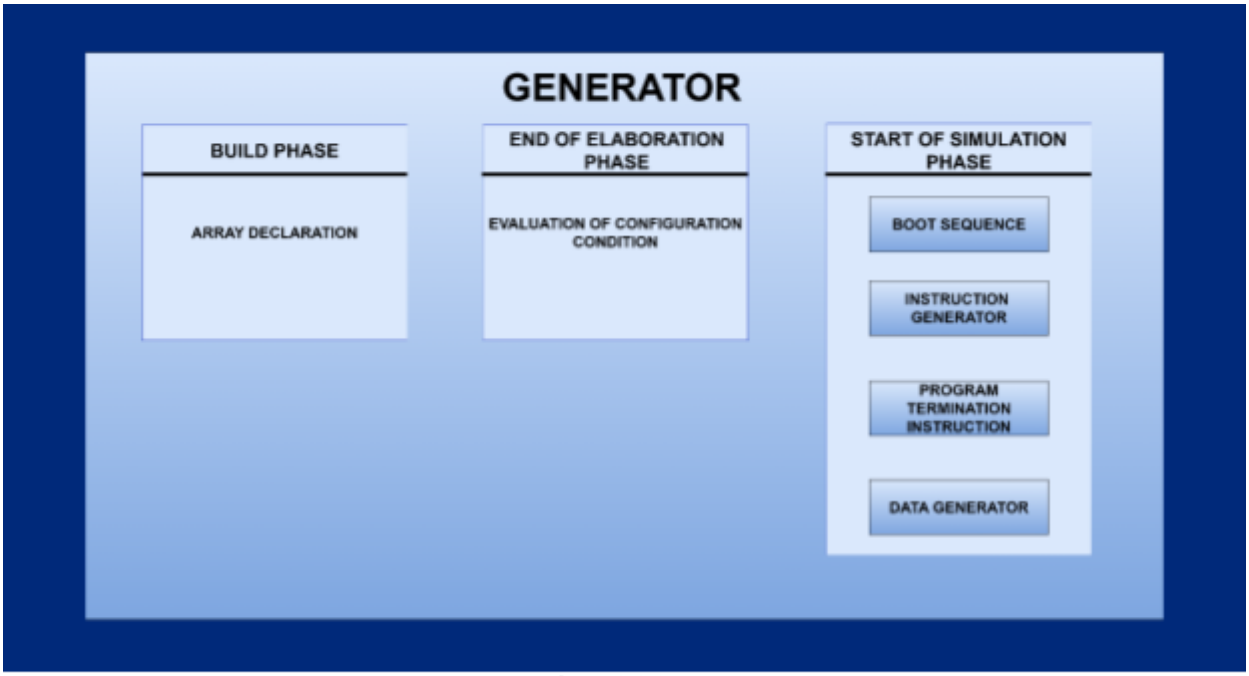

Figure 3.1.1

One of the additional features incorporated in the generator is that the user is provided with a configuration file in which he can specify which type of Extension instruction he wants to generate and which instructions he wants to avoid from being generated before running the test.

As previously mentioned, some of the assertions have been included at the end of the elaboration phase which helps in the smooth execution of the generator. Since our generator does not support many extensions, we initiate the end of elaboration phase by checking the extension entered by the user. if the user requests for an extension that's not supported, an error will be generated notifying the user that "The specified Extension is not supported by Generator" and the test will be terminated.

After the aforementioned assertion, we examine the testname provided by the user. If the user enters a testname which is not supported by the specified Extension, an error message will appear notifying the user "The test name is not supported by the selected Extension" and the test will be terminated.

After passing both assertions, an opcode list is generated depending on user enter extensions and transmitted to other functions. The complete explanation of the flow of the end of elaboration phase was discussed above. A pictorial representation of the flow can be observed in Figure 3.1.2.

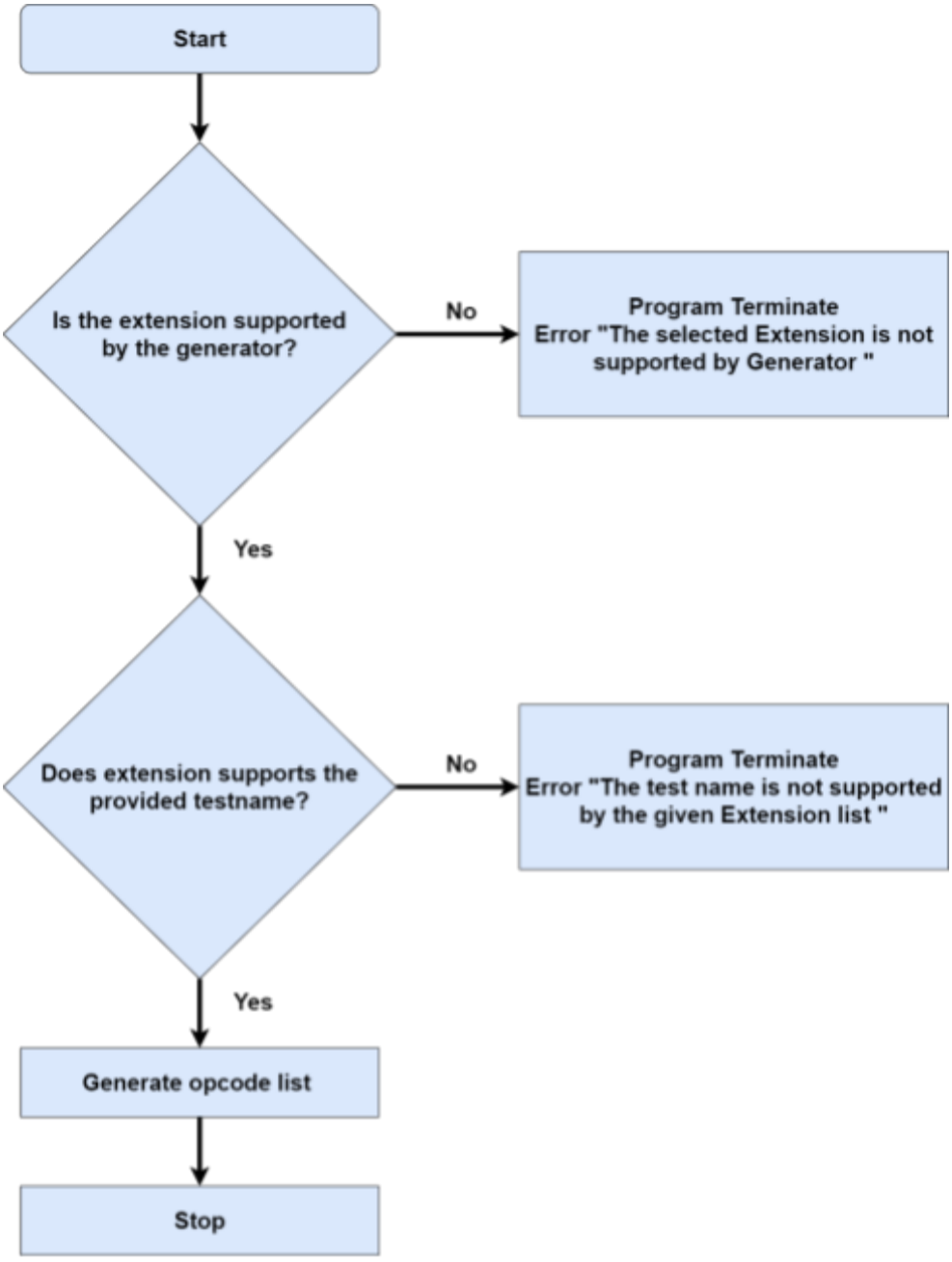

Figure 3.1.2

As explained earlier, different functions inside the start of the simulation phase are executed in the sequence depicted in Figure 3.1.3. The boot sequence is generated first to initialize the stack pointer and registers. To stop a program, a program termination instruction is generated before user-defined test instructions are generated. Following successful generations of instructions, random data points are ultimately generated in order to carry out the efficient execution of the generated program.

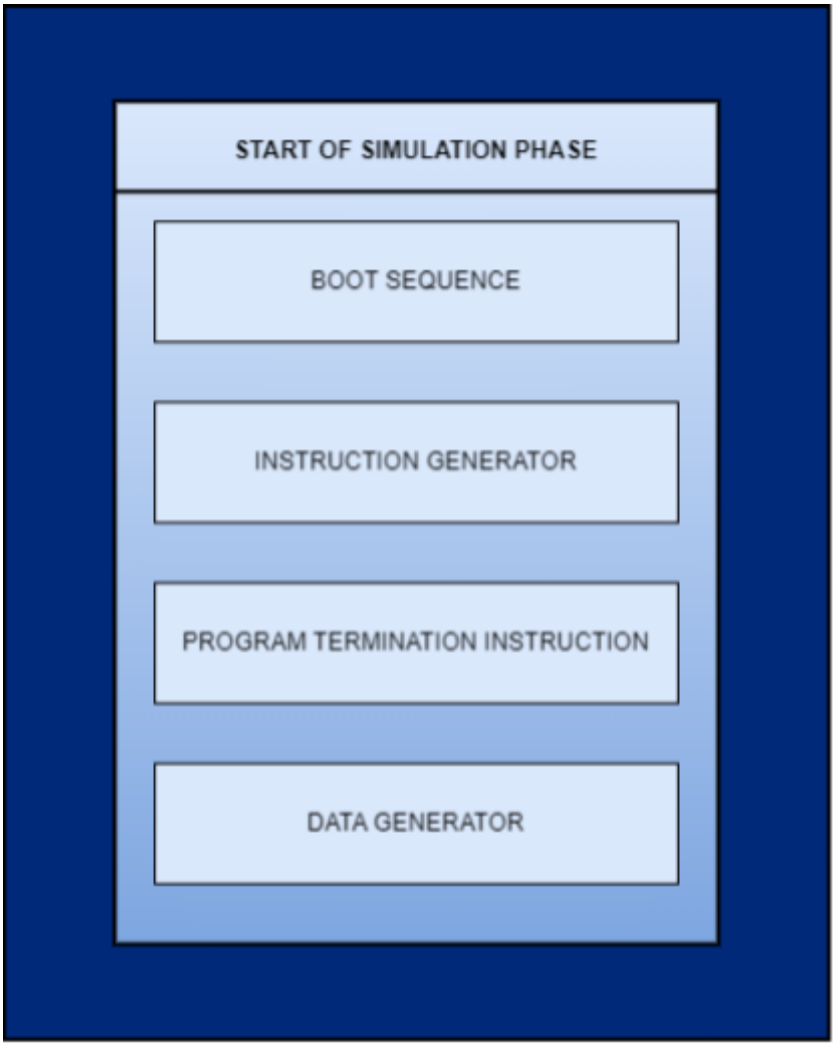

Figure 3.1.3

Instruction Generator is one of the main functions of the generator as it is in charge of generating the instructions required by the user. The function takes three arguments: the test name, the test iteration, and the list of opcodes. The opcode selector function initially selects an opcode randomly from the provided opcode list based on the provided test name. The instruction function generates an instruction based on the opcode that is selected. When appending the generated instruction to a list, a conditional check is performed to ensure that the generated instruction is not on the user's excluded instruction list. If the generated instruction is one of the excluded instructions, it is regenerated. The above process is repeated until the instruction count matches the amount of test iterations.

Flow Chart

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

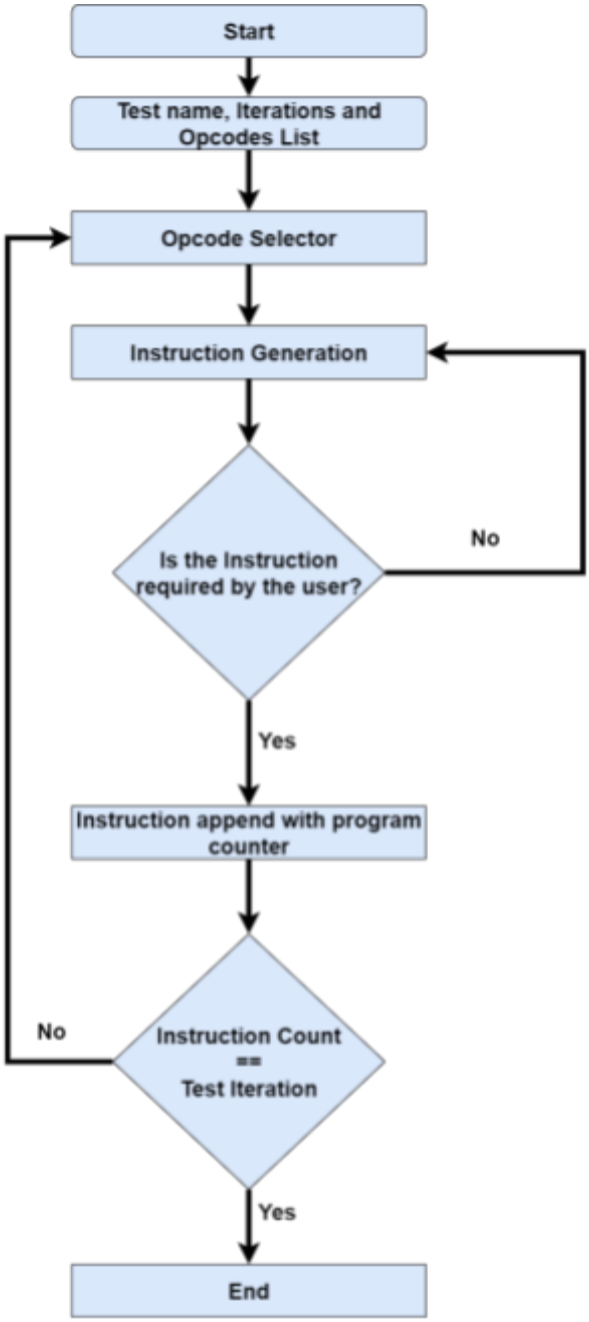
Start
Test name, Iterations and Opcodes List
Opcode Selector
Instruction Generation
Is the Instruction required by the user?
No
Yes
Instruction append with program counter
Instruction Count == Test Iteration
No
Yes
End

Figure 3.1.4

3.2 Bus Function Model (BFM)

Overview

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 AXI, AHB, or PCIe, and is used to generate transactions that can be used to stimulate the design's interface.

Advantages of using a BFM in UVM include:

1. **Faster development:** The use of a BFM allows the verification engineer to quickly develop test scenarios and generate transactions to test the design's interface. This saves time and effort in creating a custom testbench for each design.

2. **Reusability:** Once a BFM is created, it can be reused across multiple projects and designs, making it an efficient and cost-effective way to verify designs

3. **Accuracy:** The BFM models the behavior of the actual bus protocol, providing accurate simulation results and identifying potential bugs or issues.

4. **Simplifies testbench creation:** Using a BFM simplifies testbench creation and reduces the amount of code that needs to be written. This makes it easier for the verification engineer to focus on the actual verification tasks rather than spending time creating a testbench from scratch.

5. **Integration with other components:** The BFM can be integrated with other UVM components, such as monitors, checkers, and scoreboards, to provide a comprehensive verification environment.

6. **Standardization:** The use of a BFM provides a standardized approach to verifying designs that follow a specific bus protocol, ensuring consistency across different projects and designs.

Implementation

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.

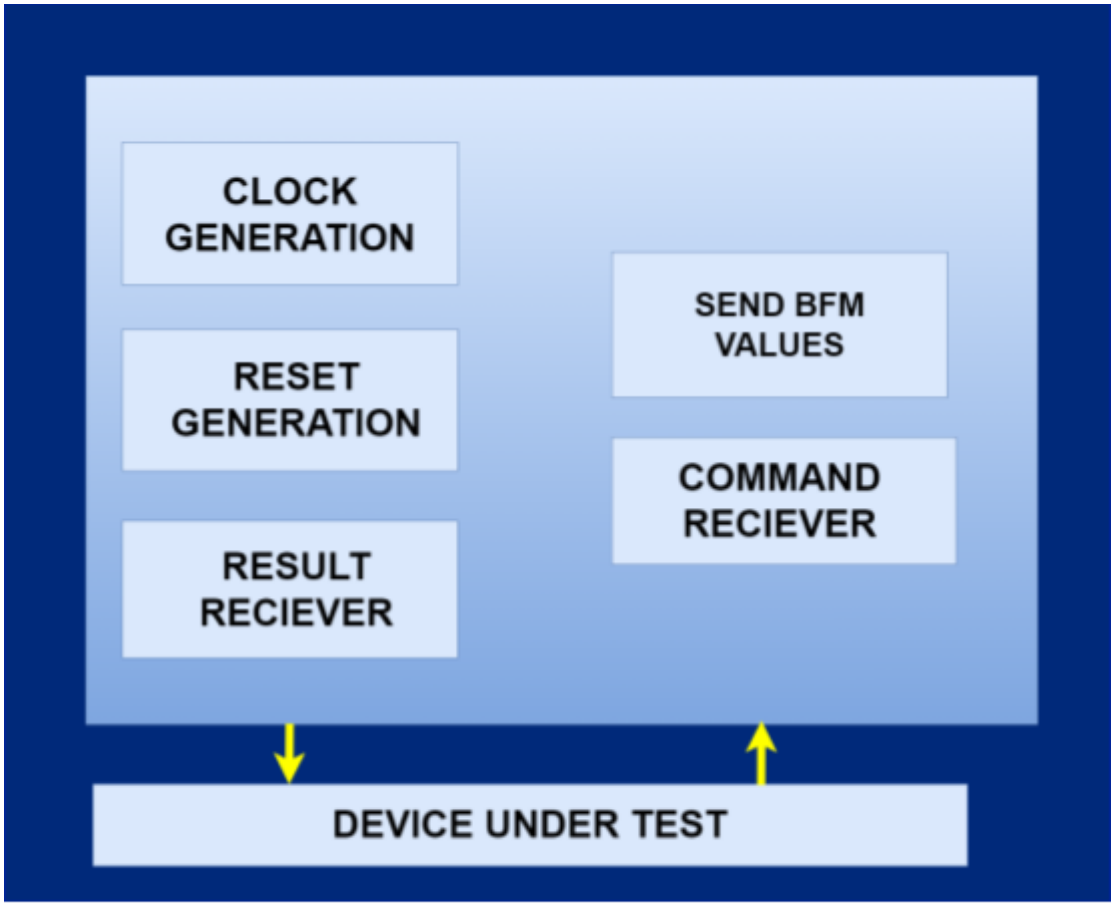

Figure 3.2.1

The block diagram of the BFM is shown in Figure 3.2.1. The BFM contains five functionalities. As two of the functions are responsible for generating Clock and Reset. Some functions are used to transmit data and instructions, while others are used to obtain results and commands from the DUT.

The clock function generates a clock for the DUT, while the reset function provides a reset to the DUT. Driver transmits stimulus to the DUT via BFM send_values function without utilizing any for loops. Now the communication between the Driver, BFM and DUT is happening sequentially. For that Driver, transmit each stimulus one at a time to BFM, and BFM sends the same stimulus to the DUT. The command receiver function's main objective is to capture the stimulus transmitted to the DUT via BFM. Similarly, the result receiver function was in charge of capturing the result from the DUT while using some constraints of the while loop and storing it in a tuple, which was later received by the Monitor.

3.3 Sequencer

Overview

The Universal Verification Methodology (UVM) sequencer is a key component in the UVM framework used for verifying complex digital designs. It is responsible for generating and managing transaction sequences that drive stimuli to the design under test (DUT).

In UVM, a sequencer is typically associated with a driver and is responsible for controlling the flow of data from the testbench to the DUT. The sequencer receives transaction requests from the testbench, creates sequence items, and sends them to the driver for transmission to the DUT.

Sequences are typically defined using a hierarchical structure and can be composed of multiple sequences. A sequence can be made up of one or more transactions, and these transactions can be customized using various configuration options.

Advantages of using sequencers in UVM include:

1) **Reusability:** The UVM sequencer is highly reusable, allowing sequences to be easily customized and reused across multiple projects and designs.

2) **Flexibility:** The UVM sequencer is highly configurable, providing users with the ability to create and modify sequences to meet specific testing requirements.

3) **Hierarchical sequencing:** The UVM sequencer supports hierarchical sequencing, allowing sequences to be composed of multiple sub-sequences and enabling the creation of complex, multi-level test scenarios.

4) **Automatic checking:** The UVM sequencer provides automatic checking of sequence items, ensuring that they are correctly formatted and within the defined constraints.

5) **Efficient stimulus generation:** The UVM sequencer generates efficient and targeted stimulus to the DUT, helping to improve the overall efficiency of the verification process.

6) **Error detection and recovery:** The UVM sequencer can detect errors in the transaction flow and recover from errors by automatically retrying transactions or generating an error message.

7) **Integration with other UVM components:** The UVM sequencer seamlessly integrates with other UVM components, such as drivers, monitors, and scoreboards, allowing for a complete verification environment.

Implementation

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.

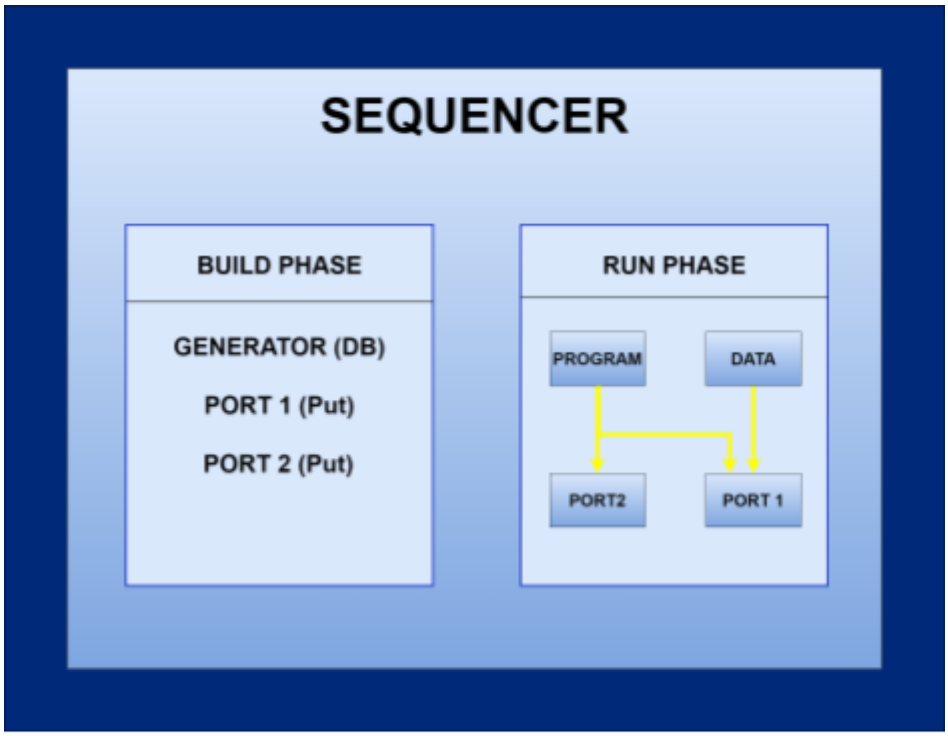

Figure 3.3.1

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 two of the TLM analysis FIFO ports are initialized which are used to connect the sequencer with Driver and Monitor.

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 1. 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

1 in the same pattern as it transmits the instructions. It also sends each instruction to a Monitor for monitoring the instructions which are being transferred to the DUT using TLM analysis FIFO blocking port 2.

Flow Chart

The complete explanation of the flow of the sequencer was discussed in implementation. A pictorial representation of the flow can be observed in Figure 3.3.2 and Figure 3.3.3.

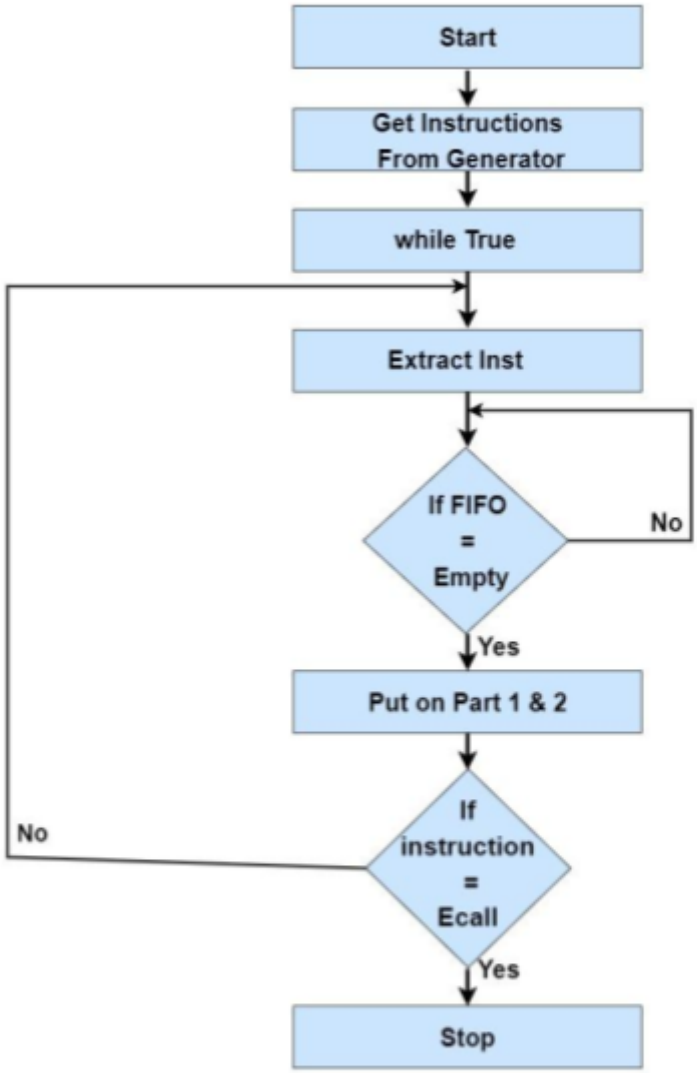

Figure 3.3.2: Instruction Flow Chart

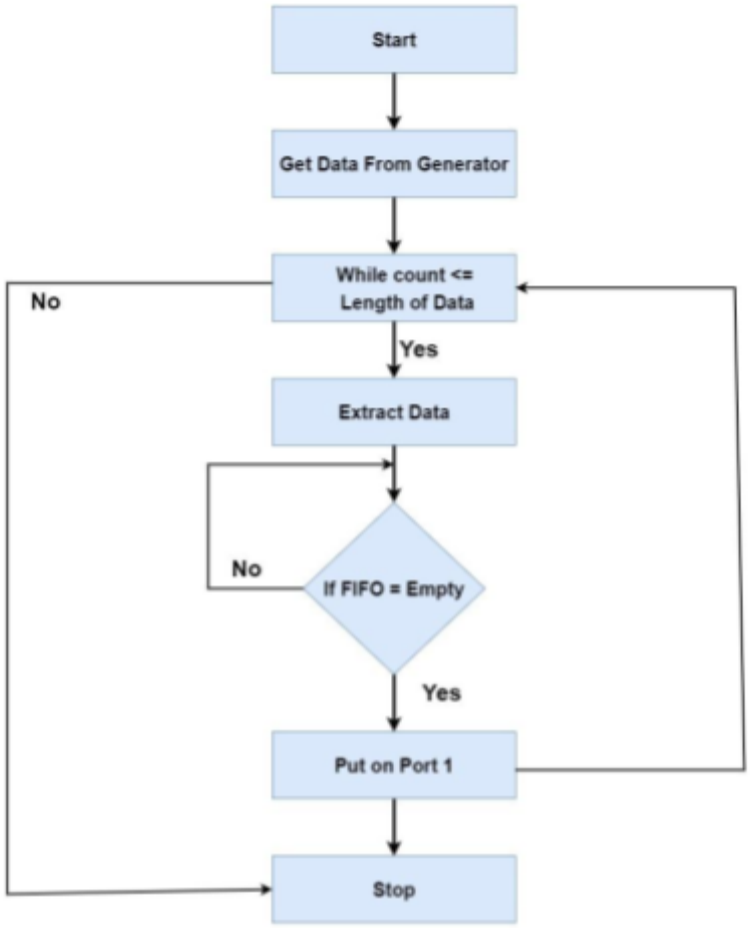

Figure 3.3.3: Data Flow Chart

3.4 Driver

Overview

In UVM (Universal Verification Methodology), a driver is a component that drives stimulus into a Design Under Test (DUT) by translating high-level transaction-level stimuli into low-level signals that the DUT can understand. The driver typically receives transactions from the sequencer and sends them into a BFM that can be applied to the DUT's inputs.

Advantages of using a driver in UVM include:

1. **Abstraction:** The driver provides an abstraction layer between the testbench and the DUT. This allows the testbench to send high-level transactions to the driver, without needing to know the specific details of the DUT's interface.

2. **Reusability:** Since the driver is responsible for translating transactions into signals, it can be reused across different tests and testbenches. This saves time and effort in the verification process.

3. **Modularity:** The driver can be easily swapped out for a different driver that implements the same interface. This allows for easy testing of different implementations of the DUT or for testing different DUTs with the same interface.

4. **Configurability:** The driver can be configured to support different interface protocols and timing requirements. This allows for easy adaptation to different DUTs and test environments.

5. **Debugging:** The driver provides visibility into the signals that are being sent to the DUT. This can help with debugging the testbench and verifying that the DUT is receiving the expected inputs.

Implementation

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.

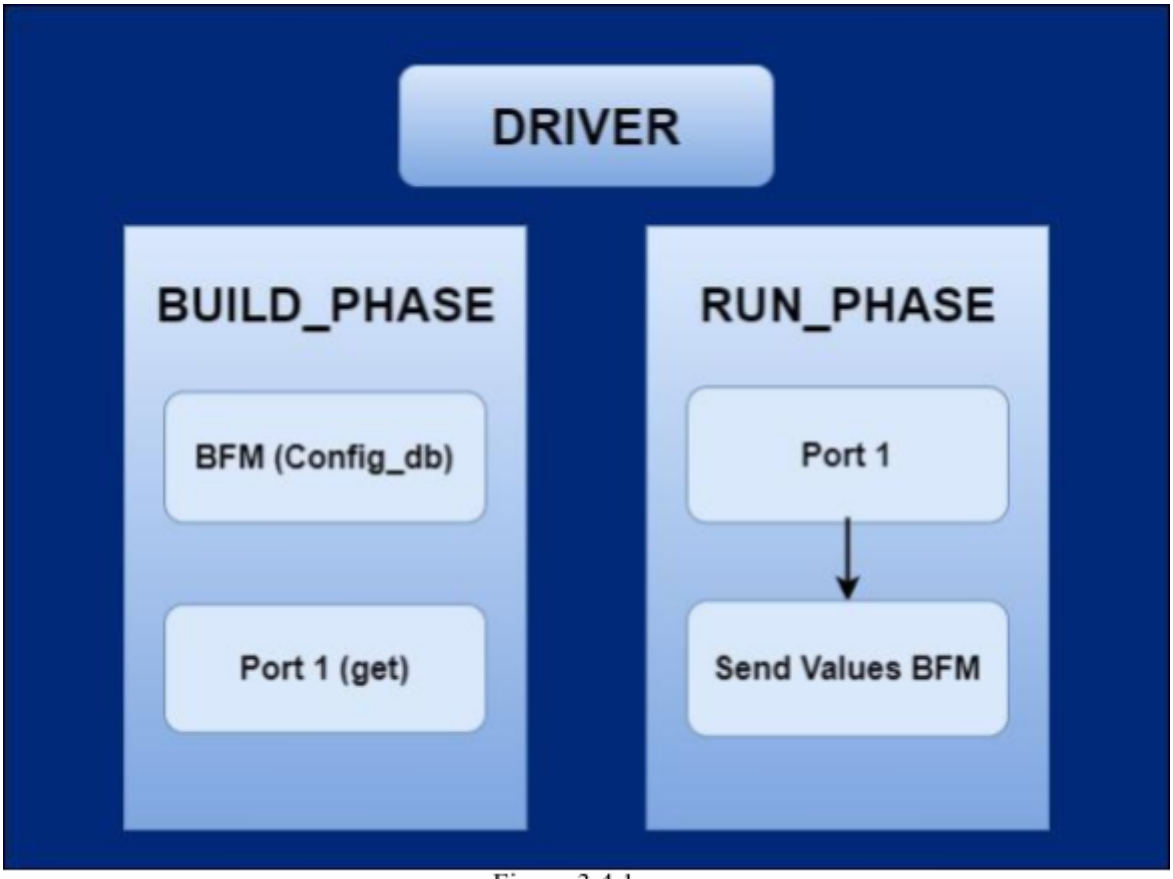

Figure 3.4.1

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 1 and sends it to the DUT via BFM. When the TLM FIFO port 1 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.

Flow Chart

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.

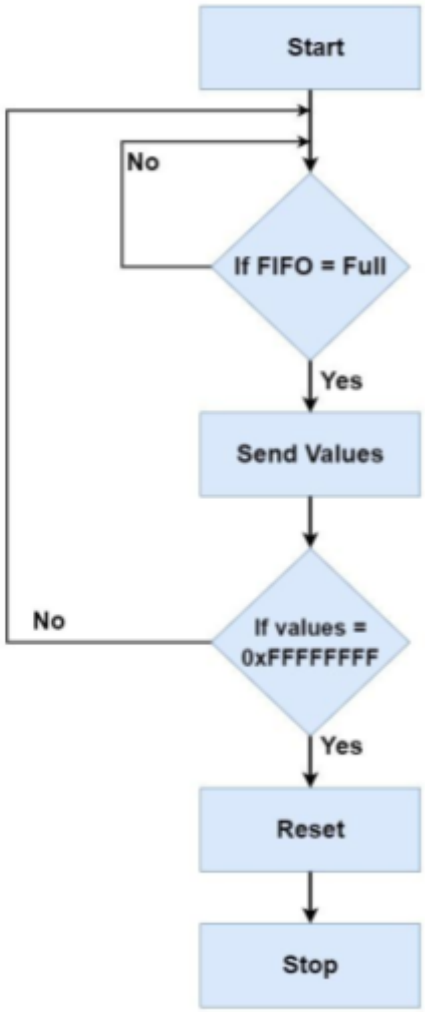
Start
No
If FIFO = Full
Yes
Send Values
No
If values =
0xFFFFFFFF
Yes
Reset
Stop

Figure 3.4.2

3.5 Monitor

Overview

In UVM (Universal Verification Methodology), a monitor is a component that is responsible for monitoring the activity on a design interface. It is an important building block in a verification environment as it allows for the observation of signals or transactions on the design under test (DUT) and receives corresponding data that can be analyzed to verify correct functionality.

There are two main types of monitors in UVM:

1) Command Monitor:

   In UVM (Universal Verification Methodology), a command monitor is a component that is responsible for monitoring commands or transactions sent to a DUT (Design Under Test) during simulation. The primary purpose of a command monitor is to observe the behavior of the DUT and verify that it is responding correctly to the commands according to the design specification.

   The command monitor receives the commands or transactions sent from the testbench and monitors the DUT's response to these commands. It checks whether the DUT responds correctly and generates appropriate notifications or alerts if there is an error or violation of the expected behavior. The command monitor plays an important role in ensuring the functional correctness of a design during verification.

2) Result Monitor:

   In UVM (Universal Verification Methodology), a result monitor is an object that is responsible for monitoring a specific signal, bus, or interface in the design under test (DUT) during the simulation. Its primary role is to capture the values on the interface signals and check them against the expected values.

   The result monitor is typically implemented as a UVM component and is connected to the interface signals through a TLM (Transaction-Level Modeling) port. The monitor receives data from the interface and extracts relevant information to build transactions that can be compared with expected results.

The advantages of using Command and Result Monitors in a verification environment include:

1. **Debugging:** Monitors provide visibility into the behavior of the DUT, allowing for easier debugging of issues that may arise during the verification process.

2. **Protocol Compliance:** Monitors can be used to verify that the DUT is compliant with the protocol being used.

3. **Efficient Test Development:** Monitors can be used to develop and validate tests more efficiently as they provide a mechanism for observing the behavior of the DUT and verifying its correctness.

Implementation

Another component of the PY-UVM framework, which is also inherited from the UVM component, is called Monitor. The primary function of the Monitor component is to capture both RTL-generated results along with stimuli signals that are transmitted to the RTL via the PY-UVM framework.

In our framework the Monitor component has two distinct functions.

1) Command Monitor
2) Result Monitor

The command monitor function's primary purpose is to compare the stimulus generated by the generator and transmitted to the DUT via BFM. The result monitor function, on the other hand, is responsible for acquiring the result generated by the DUT during execution of a program.

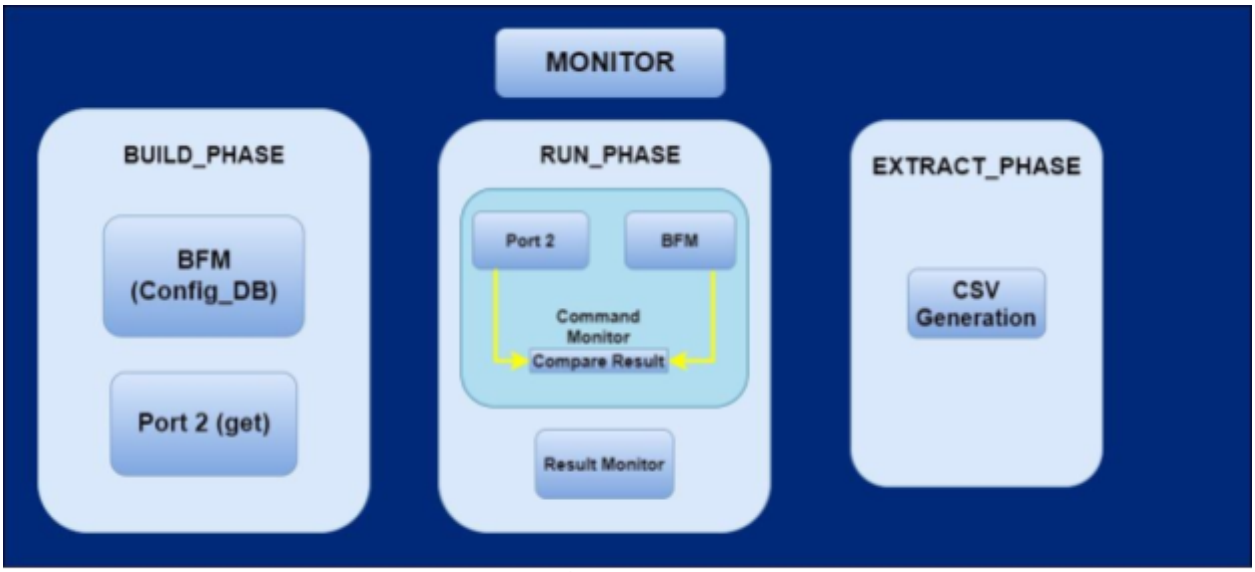

Figure 3.5.1

The block diagram of the monitor component is shown in Figure 3.5.1. In the above figure it is visible that three of the phases have been used to carry out the execution of the monitor component. 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 Monitor.

As previously stated, all run phases of UVM components execute in parallel; therefore, when the sequencer provides instructions on the port 2 of the TLM FIFO, the monitor receives instructions from two components. One from the TLM FIFO's get port and the other from the DUT through BFM. Both of the received instructions are fed into the assertion logic, which validates both data points.

As soon as the stimuli transaction is completed, the result monitor function is executed, which captures the DUT results. The capture results are converted into a log file during the extract phase.

Flow Chart

The complete explanation of the flow of the monitor was discussed in implementation. A pictorial representation of the flow can be observed in Figure 3.5.2.

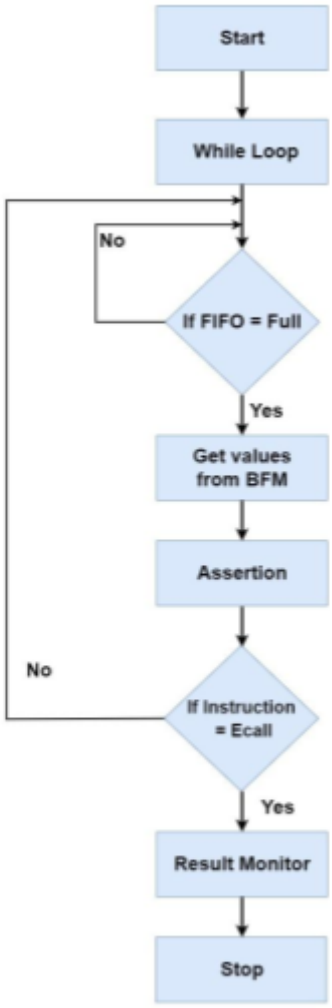

Figure 3.5.2

3.6 Instruction Set Simulator (ISS)

Overview

An Instruction Set Simulator (ISS) is a software tool that simulates the behavior of a microprocessor or microcontroller based on a given instruction set architecture (ISA). The RISC-V ISA is an open, free and scalable architecture that has gained popularity in recent years.

One of the major advantages of using an ISS for a RISC-V based architecture is that it can be used as a golden model to verify the correctness of hardware implementations of the architecture. Some advantages of using a RISC-V based ISS as a golden model include:

1. **Accurate Simulation:** An ISS for RISC-V architecture provides a highly accurate simulation of the processor's behavior. It can simulate the execution of individual instructions and provide detailed information about the processor's state during execution. This level of accuracy is essential for debugging and testing software that runs on the processor.

2. **Flexibility:** An ISS can be used to simulate different configurations of the RISC-V architecture, such as varying the size of the instruction and data caches, the number of processor cores, and the interconnect topology. This can help in optimizing the design for specific applications or use cases.

3. **Cost-effective:** An ISS for RISC-V architecture is a cost-effective solution for testing software. It eliminates the need for expensive hardware-based development boards and allows developers to test their code on a virtual platform

Overall, an ISS for RISC-V architecture is a valuable tool for software development, testing, and research. It provides an accurate, fast, flexible, and cost-effective way to simulate the behavior of the processor and test software in a virtual environment.

Implementation

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.

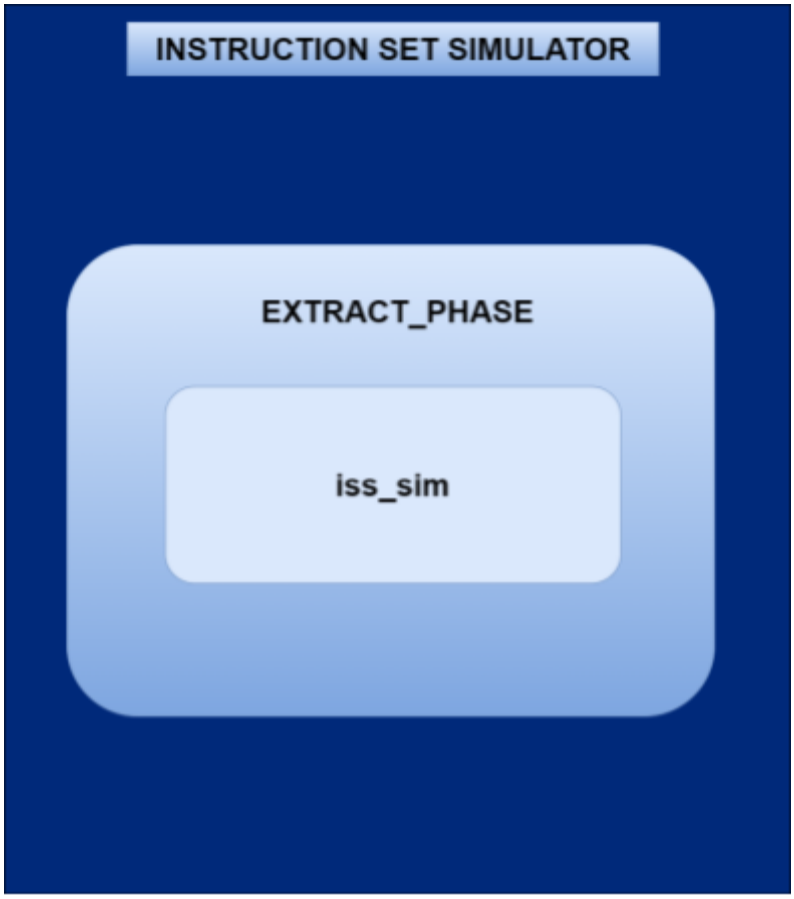

Figure 3.6.1

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.

Flow Chart

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

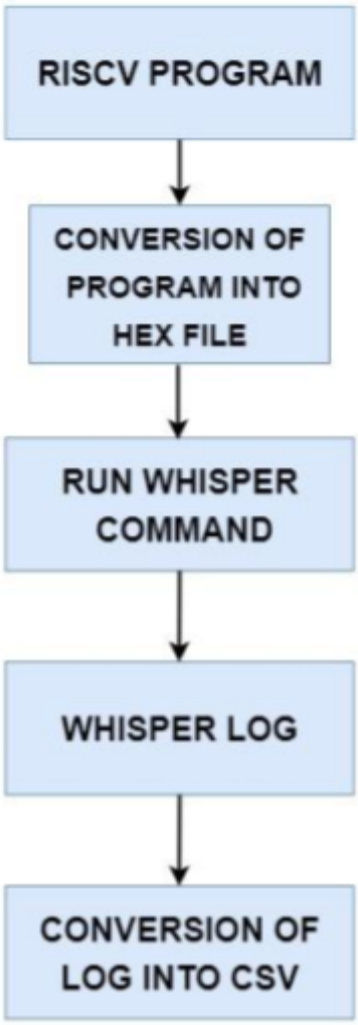
RISCV PROGRAM
CONVERSION OF PROGRAM INTO HEX FILE
RUN WHISPER COMMAND
WHISPER LOG
CONVERSION OF LOG INTO CSV

Figure 3.6.2

3.7 Scoreboard

Overview

In UVM (Universal Verification Methodology), a scoreboard is a verification component that checks the correctness of the output produced by the DUT (Design Under Test) with the expected output. The scoreboard receives outputs from the DUT and compares them with the expected output that is fed into the scoreboard either directly or through another reference model.

The scoreboard has several advantages in the verification process:

1. **Detecting errors:** The scoreboard is responsible for checking the correctness of the output produced by the DUT. It compares the actual output of the DUT with the expected output, and if there is a mismatch, it reports an error. Thus, it helps in detecting errors in the design.

2. **Debugging:** The scoreboard helps in debugging the design. If there is a mismatch between the actual and expected outputs, the scoreboard can provide information about the inputs that led to the error. This information can be used to debug the design.

3. **Coverage:** The scoreboard can also help in improving the verification coverage. By checking the correctness of the output produced by the DUT, the scoreboard can provide information about the functional coverage of the design.

4. **Reusability:** The scoreboard can be reused across different testbenches and designs. This helps in reducing the verification effort and improving the verification quality.

5. **Independence:** The scoreboard is independent of the DUT implementation. This means that the same scoreboard can be used to verify different implementations of the same design.

Implementation

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).

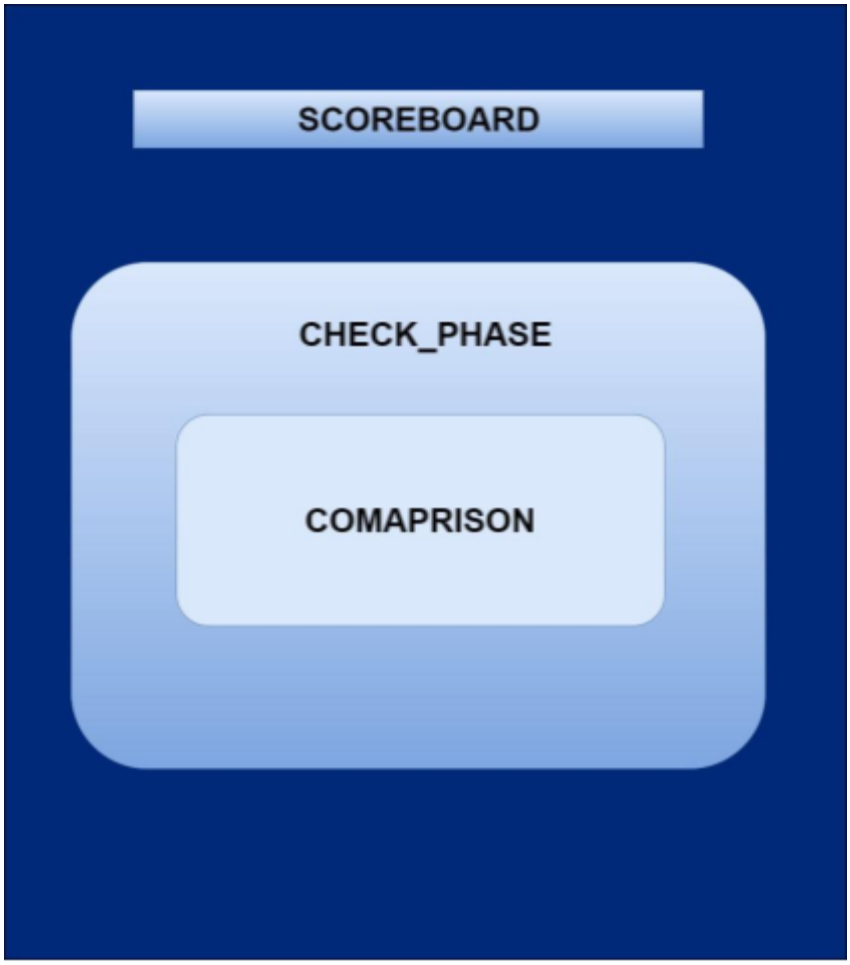

Figure 3.7.1

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.

Flow Chart

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

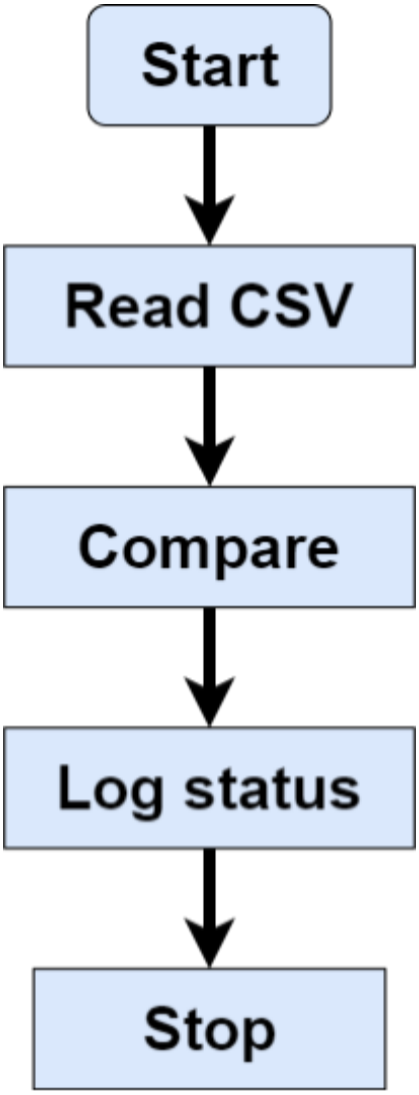

Figure 3.7.2

3.8 Coverage

Overview

In UVM (Universal Verification Methodology), coverage refers to the measurement of how thoroughly a design has been tested. Coverage is an essential aspect of functional verification, as it helps to ensure that the design meets its requirements and specifications.

In UVM, coverage is typically implemented using a coverage model, which defines the coverage items to be measured and the methods used to collect coverage data. A coverage model can include both functional and code coverage items, which can be collected at various levels of abstraction, such as transaction-level or cycle-level.

Advantages of implementing coverage in UVM include:

1. **Improved Verification Quality:** By measuring the extent to which a design has been tested, coverage allows for a more thorough verification process, leading to higher quality designs.

2. **Early Bugs Detection:** By identifying untested portions of the design, coverage enables early detection of bugs, which can be addressed before they cause more significant problems.

3. **Efficient Test Development:** By providing feedback on the completeness of the verification process, coverage helps test developers focus their efforts on the areas of the design that need additional testing.

4. **Faster Time-to-Market:** By enabling more efficient and effective verification, coverage can help reduce the time required to bring a product to market.

5. **Compliance With Industry Standards:** Many industry standards, such as ISO 26262 for automotive safety, require verification coverage as a part of the certification process. Implementing coverage in UVM can help ensure compliance with these standards.

Implementation

Coverage is one of the Py-UVM framework components inherited from the UVM component. Reporting the functional and code coverages covered by the framework is the main objective of a coverage component. Currently, only the functional coverage can be assessed by the coverage component. The main benefit of the coverage is that it informs the verification engineer that how many cover points he was able to achieve that were required by design under test.

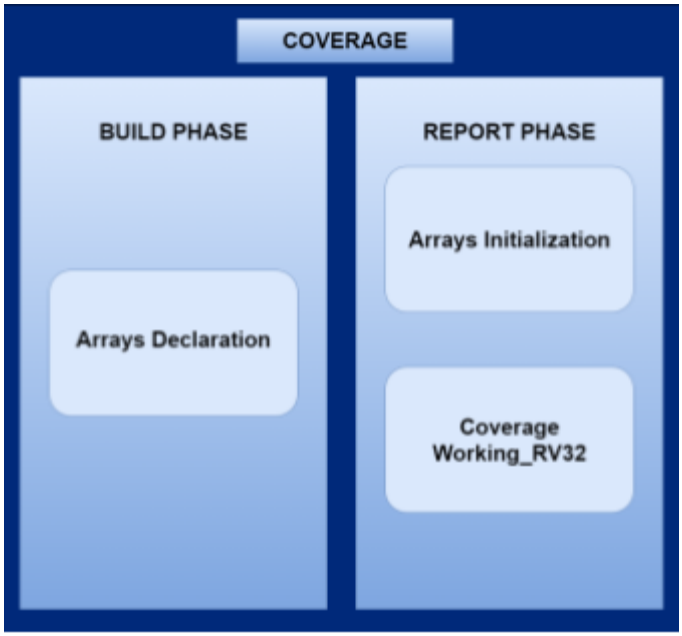

Figure 3.8.1

Figure 3.8.1 depicts the Coverage component's block diagram. The coverage component is executed via two of the inherent UVM component phases, as shown above. All of the necessary array declarations have been carried out in the build phase. As coverage is always executed at the end of the ongoing test, the report phase has been used instead of the run phase. As soon as the report phase begins its execution, the previous coverage report has been invoked and all previous information has been initialized to the arrays. If there is no previous report then all the arrays are initialized with the default value. Another function have been called after the initialization to evaluate the following functional coverages:

1) Instructions
2) Registers
3) Sign of Immediates

The aforementioned functional coverages are opted because they notify the verification engineer how many instructions have been functionally verified, how many times a certain register has been used as a source and destination, and what type of immediate numbers have been generated by the generator.

Flow Chart

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

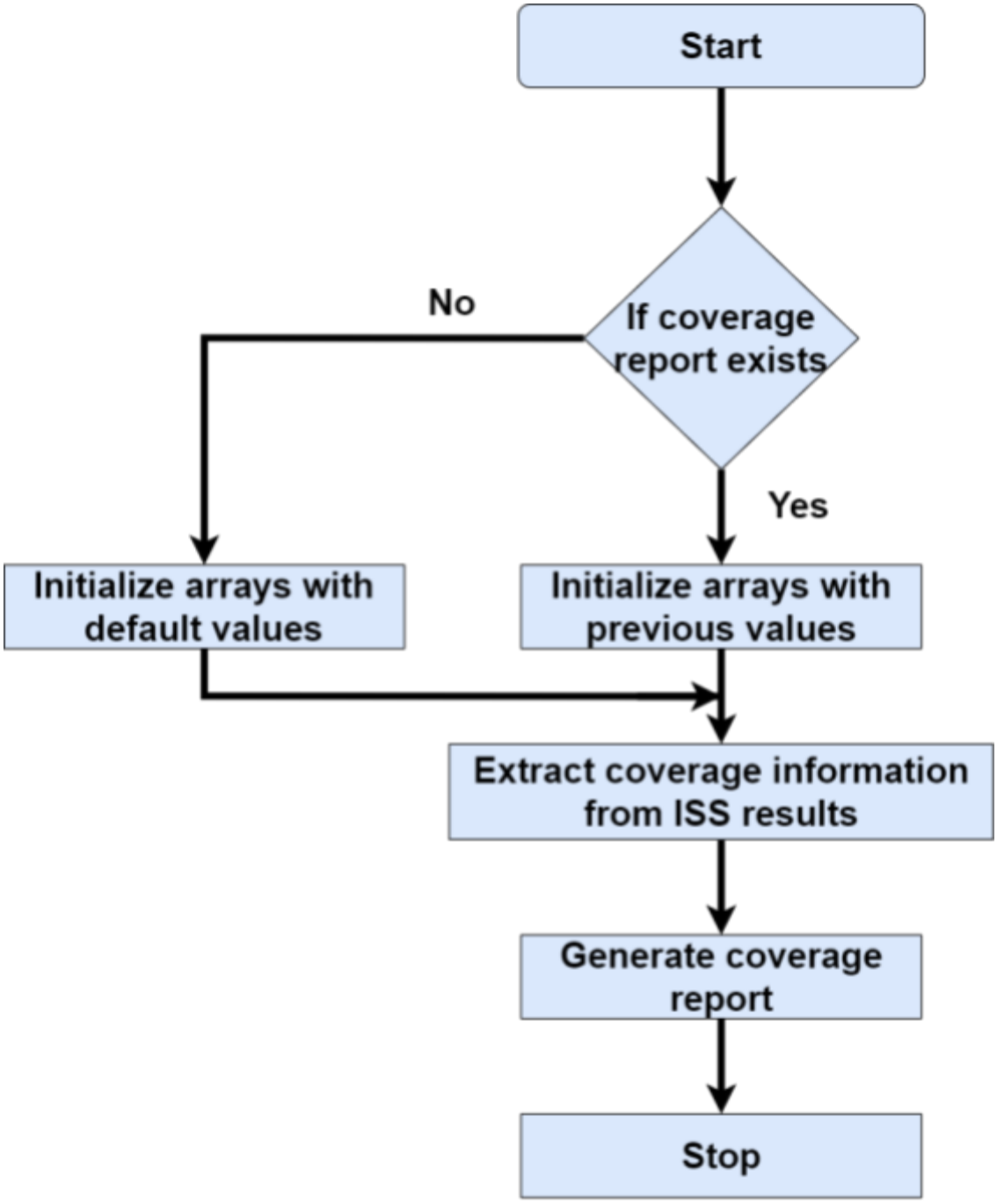
Start
If coverage report exists
No
Yes
Initialize arrays with default values
Initialize arrays with previous values
Extract coverage information from ISS results
Generate coverage report
Stop

Figure 3.8.2

3.9 PY-UVM Top Framework

Implementation

In this heading we will discuss the top environment of the Py-UVM framework. Top environment is responsible for executing the whole framework. To get a better understanding of how the framework worked, consider the block diagram and UVM class hierarchy shown below.

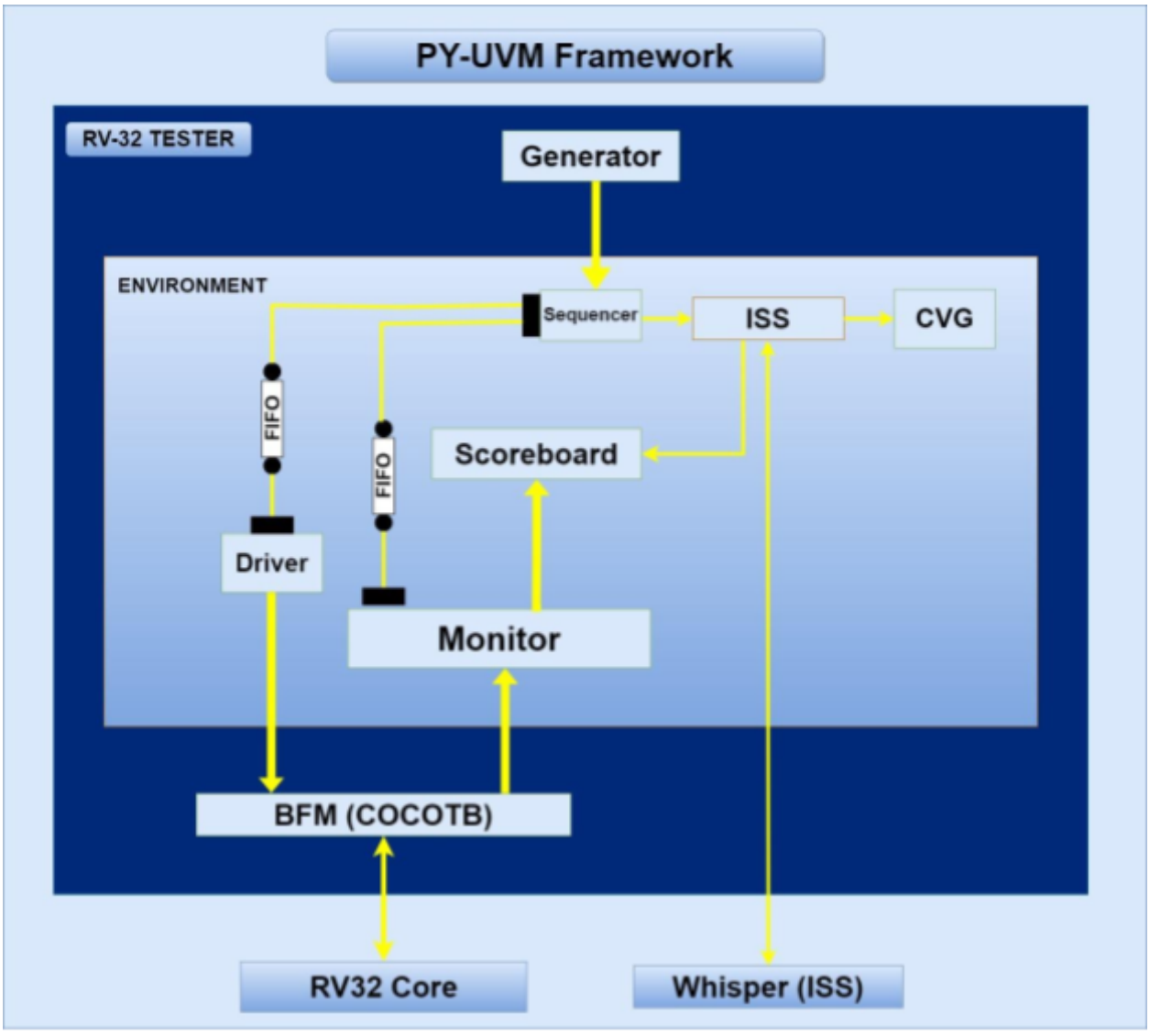

Figure 3.9.1

Figure 3.9.1 depicts the block diagram of the framework implementation. As shown in the diagram, the following components were initiated in the top framework:

a) Generator
b) Sequencer
c) Driver

d) Monitor
e) Scoreboard
f) Instruction Set Simulator (ISS)
g) Bus Function Model (BFM)
h) Coverage (CVG)
i) TLM FIFOs

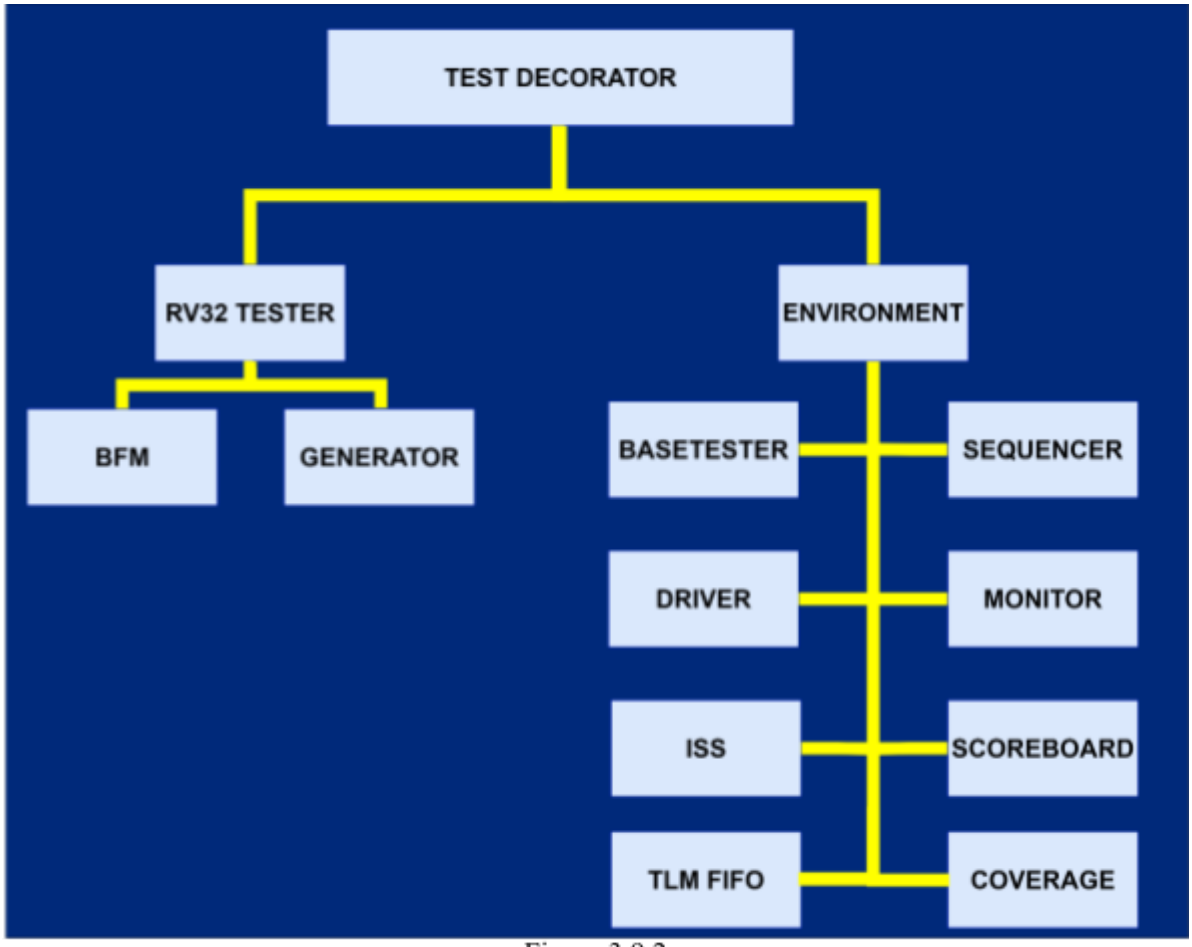

Figure 3.9.2

Figure 3.9.2 depicts the framework's hierarchy in which different components are initiated inside the Environment and RV32 Tester. The reason behind breaking the hierarchy of the framework is to increase its reusability. To validate different designs using the above framework the user is required to modify the Tester Class only. Two of the framework components BFM and Generator are initiated inside the RV32 Tester which will vary according to Design Under Test (DUT). The handlers of both of the components are declared in the RV32 Tester Class's build phase.

As shown in the figure above, the Environment component inherited from UVM Environment has the following components. The handlers of each of the components shown above are declared during the

build phase of the Environment Component Class. Since the RV32 Tester inherits from the Base Tester Component, it overrides a Base Tester component in Environment.

When the build phase of all the components is completed, the connect phase of the Base Tester component is executed, which connects the Sequencer, Driver, and Monitor components to the TLM FIFO's. At the Start of Simulation phase, the Generator generates stimuli as the connection between the components is established. As soon as the generator completes its execution, the run phase of the Sequencer, Driver, and Monitor components commences. The Sequencer transfers the stimulus to the FIFOs, whereas the Driver & Monitor receives the stimulus from FIFOs. The Driver passes the stimulus to the DUT via BFM, whereas Monitor passes the stimulus to the assertion logic. The Monitor starts collecting the result generated by DUT as soon as the stimulus transaction is completed.

During the Extract phase, the same generated stimuli are fed into the Golden Model (Whisper ISS) to validate the DUT's result. After the execution of the Golden Model, Scoreboard is executed at the check phase, where the results of both the DUT and the Golden Model are compared. Finally the coverage component is executed during the report phase.

The preceding document covers all the aspects of the Py-UVM framework for RISC-V Single Cycle Core which was designed by the MERL-DSU Verification Team.