MERL
Accelerating Engineering Innovation

Py-UVM Framework

Initial Draft

Project by:
Py-UVM Team

MERL
Accelerating Engineering Innovation

Table of Contents

Introduction

In this draft we will be discussing the changes incorporated in the initial framework of Py-UVM. After the successful implementation of the initial framework a deep dive session was held between the team members. The session's main finding was that the developed framework did not meet UVM standards. So, the team made the decision to completely restructure the framework while keeping the UVM standards in mind.

The main drawbacks in previous framework were as follows:

1) Frequent use of Queues was observed in many components.
2) Separate functions have been designed to carry out different operations rather than using UVM components' inherent phases.
3) Sequencer and Coverage component was not implemented.
4) Frequent use of For Loop was observed in the Bus Function Model (BFM).
5) Transactions of instruction & data were dealt by BFM rather than UVM Driver component.

Before we go into more details, let's discuss the new features, functionality and changes incorporated in the current framework which are as follows:

1) Transaction Level Modeling is implemented using the UVM paradigm.
2) Unnecessary functions have been replaced by UVM components' inherent phases.
3) UVM Sequencer and Coverage concept are implemented
4) The Bus Function Model (BFM) has been reduced in complexity.
5) Assertions are introduced to terminate the framework for invalid configuration.
6) The data and instruction transfers are handled by UVM Driver.
7) The user configuration file feature is also included.

Now we will be discussing each component of the framework in detail.

Py-UVM Top Framework

In this chapter 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 previous framework worked, consider the block diagram and UVM class hierarchy shown below.

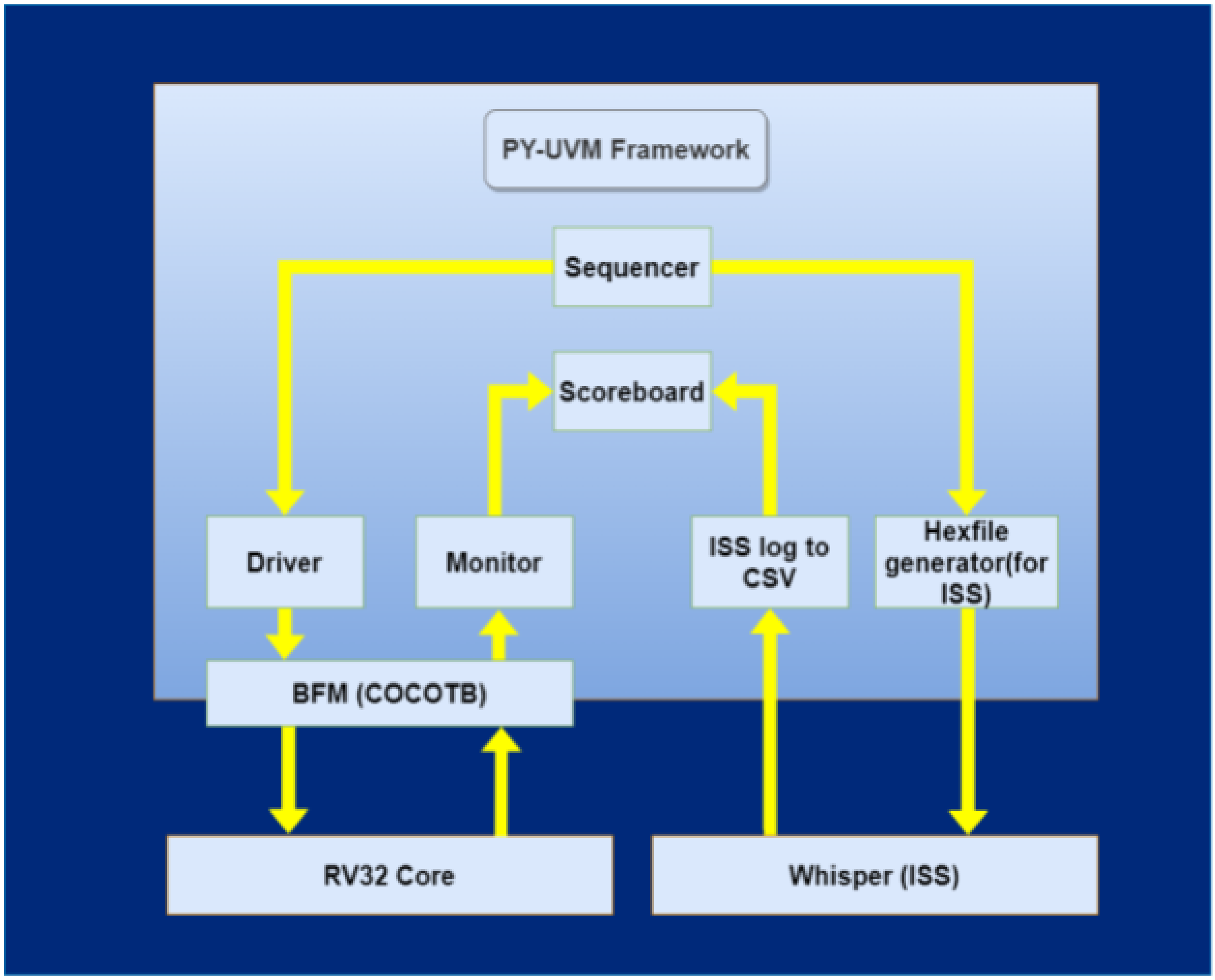

Figure 1.1

Figure 1.1 depicts the block diagram of the initial framework implementation. As shown in the diagram, the following components were initiated in the top environment:

a) Generator (Sequencer for now)
b) Driver
c) Monitor
d) Scoreboard
e) Instruction Set Simulator (ISS)
f) Bus Function Model (BFM)

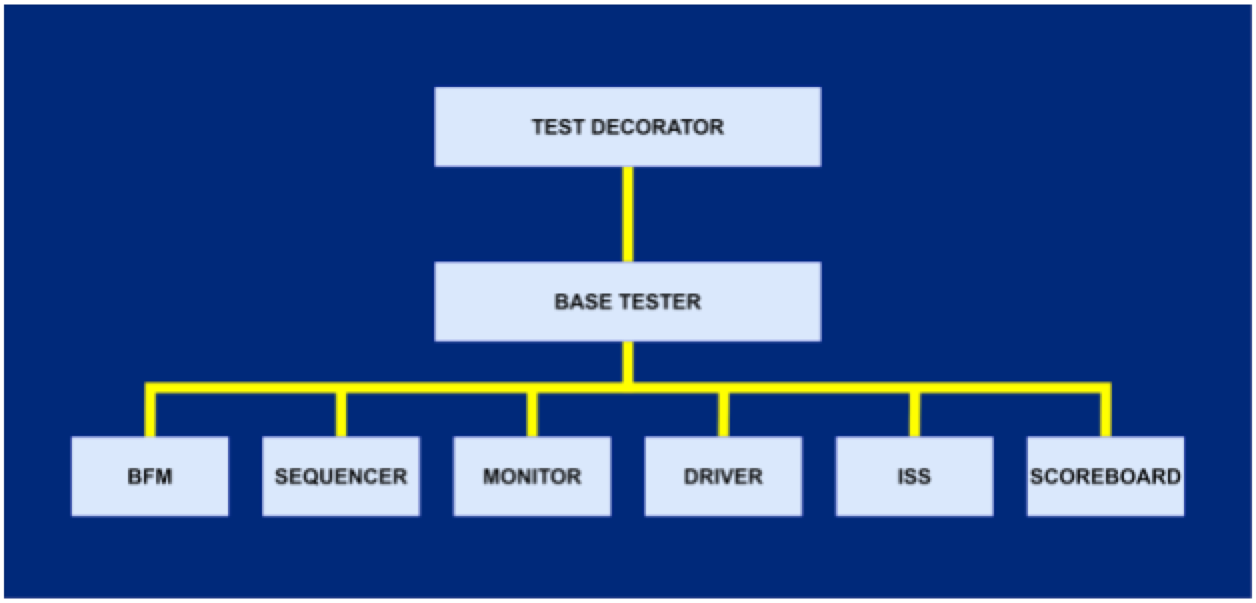

Figure 1.2

Figure 1.2 depicts the framework's hierarchy in which different components are initiated inside Base Tester Class. All of the handlers for the aforementioned components are declared in the Base Tester Class's build phase. As previously stated, one of the initial implementation's drawbacks is that we must design separate functions to carry out different operations. During the run phase of Base Tester Class, all of the functions were called individually in a unique sequence. The details of the Base Tester Class run phase is given below.

In order to generate random data, required instructions and log them in the form of arrays, we initially called the function from the sequencer component. The generated arrays were then fed into the Bus Function Model (BFM) using a function from the driver component. As previously stated, one of the drawbacks of the initial implementation was that the driver component is not used for transaction, so in this case the driver component was only used for stimulus transfer. Internally, BFM component functions are called within the driver component, which executes in a for loop until all the stimuli are transferred to the Design Under Test (DUT). Throughout the process, while the driver and BFM are running, one of the monitor component functions, which is responsible for capturing transfer stimulus into the DUT, is running in parallel. Following a successful stimulus transaction, another two functions of the monitor component are called, which are in charge of capturing the DUT's results and converting them into a log file. The same generated arrays of instructions and data are fed into the Golden Model (Whisper ISS) by calling the ISS component function to validate the result generated by the DUT. Finally, the scoreboard component function is called, which informs the user about the test's status.

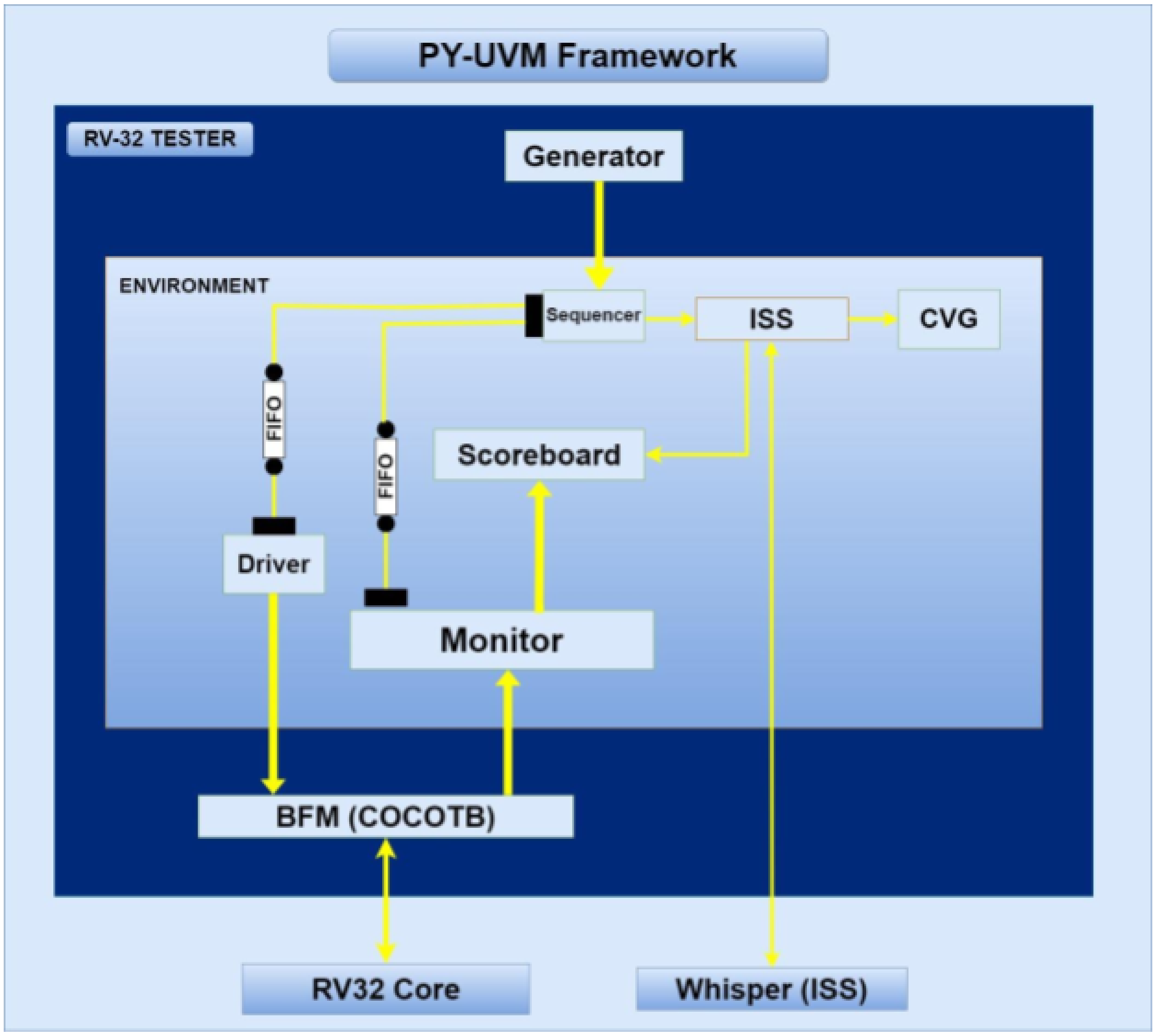

Figure 1.3

Figure 1.3 depicts the block diagram of the updated 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

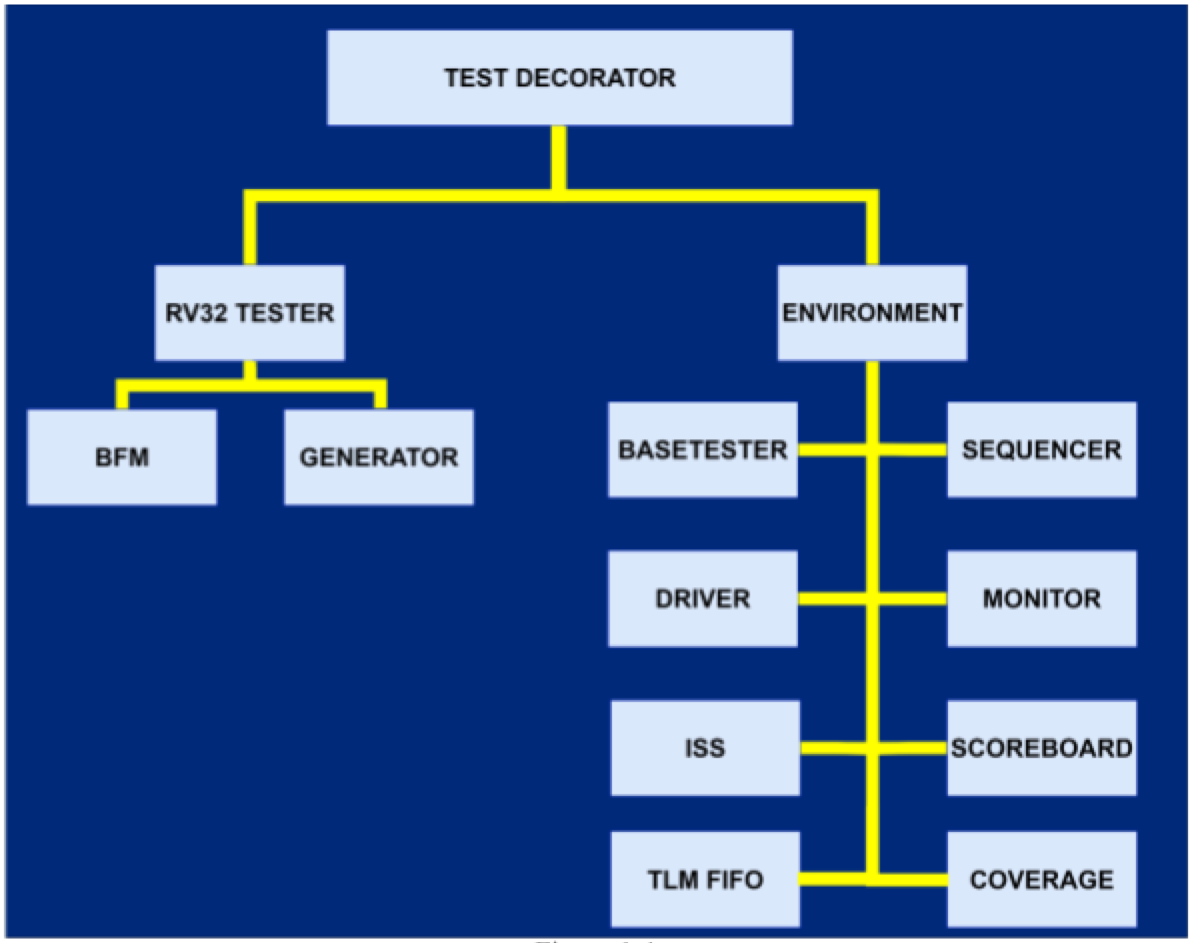

Figure 1.4

Figure 1.4 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. The adoption of Sequencer, Coverage and TLM concepts, as previously stated, is one of the advancements in the updated framework. TLM is an acronym for Transaction Level Modeling. It serves as FIFO, which eliminates the use of Queues in the framework.

The following phases are inherent in UVM components:

1) Build Phase
2) Connect Phase

3) End of Elaboration Phase
4) Start of Simulation Phase
5) Run Phase
6) Extract Phase
7) Check Phase
8) Report Phase
9) Final Phase

One of the most significant benefits of employing UVM Component inherent phases is that all phases in each component will execute in parallel and all the UVM components inherent phases mentioned above run sequentially. This minimizes the complexity and eliminates the requirement to call distinct functions in a certain sequence to carry out framework execution.

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. As previously stated, one of the advancements in the framework is the adoption of Coverage. As a result, the coverage component is executed during the report phase.

Hence, this gives a high-level description of the updated framework. The details of the modifications made in each component are discussed below.

Generator

Prior to discussing generators in more detail, the sequencer designed in the initial Py-UVM framework will be discussed first.

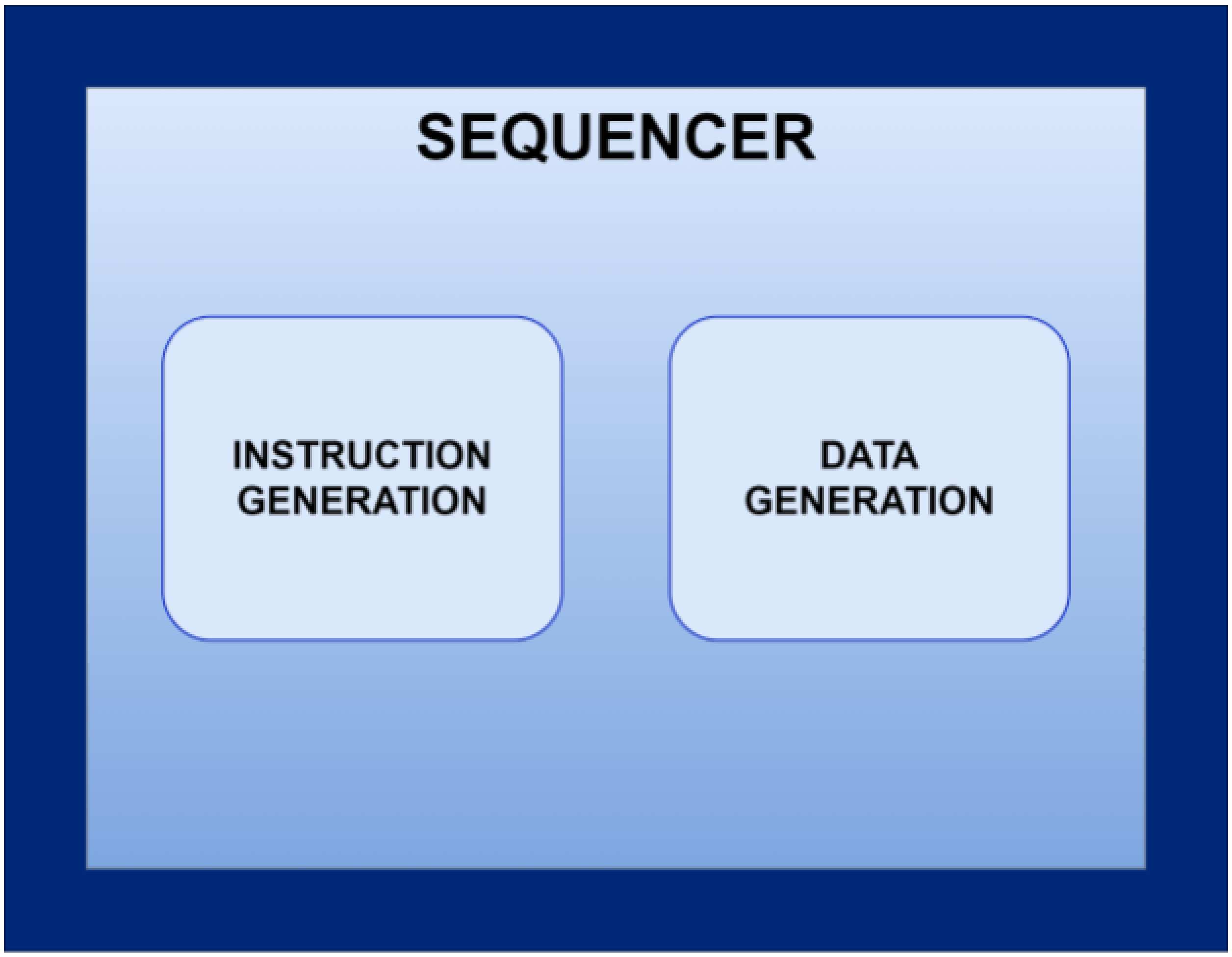

Figure 2.1

The sequencer shown in Figure 2.1 was responsible for the generation of instructions and random data points. But soon after the discussion with the team members it was realized that sequencer is named incorrectly. It should be named a generator which will be generating the stimuli for the DUT. As mentioned earlier in the drawbacks of the initial framework, sequencer was inherited by the advanced UVM concepts but none of these concepts/phases were coming into use and to tackle this issue we had to create our own function to run it from the top. The concept of creation of our own function was against the UVM framework standards.

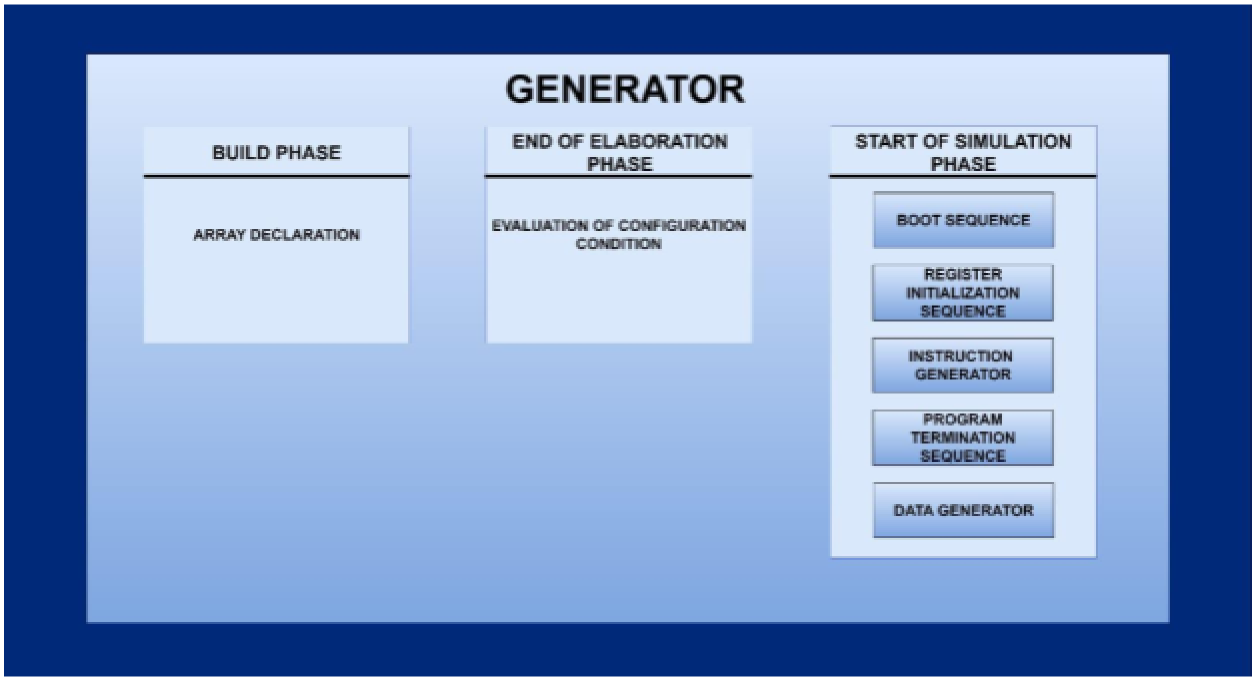
GENERATOR
BUILD PHASE
ARRAY DECLARATION
END OF ELABORATION PHASE
EVALUATION OF CONFIGURATION CONDITION
START OF SIMULATION PHASE
BOOT SEQUENCE
REGISTER INITIALIZATION SEQUENCE
INSTRUCTION GENERATOR
PROGRAM TERMINATION SEQUENCE
DATA GENERATOR

Figure 2.2

We will now be discussing the generator. In the initial Py-UVM framework, we didn't execute the sequencer component (now named generator) using the UVM component's inherent phases. In the updated framework the execution of the generator was done using UVM three phase, as can be shown in Figure 2.2. To pass various arrays to the various functions of the generator, arrays are constructed during the build phase. End of elaboration phase work is to make sure the configurations added by the user in the configuration file is correct or not. Lastly, when the start of simulation phase is executed, different functions inside this phase work according to the sequence depicted in Figure 2.2. Initially the boot sequence is generated, after that the sequence to initialize the registers, user defined test instruction and program termination instructions are generated. After the successful generations of instructions random data points are generated to carry out the smooth execution of the generated program. All of the above-mentioned stimuli are stored into arrays which are passed to the sequencer component inside the verification environment during the run phase.

Bus Function Model (BFM)

Bus Function Model (BFM) is a type of protocol used in verification to communicate with the DUT. 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.

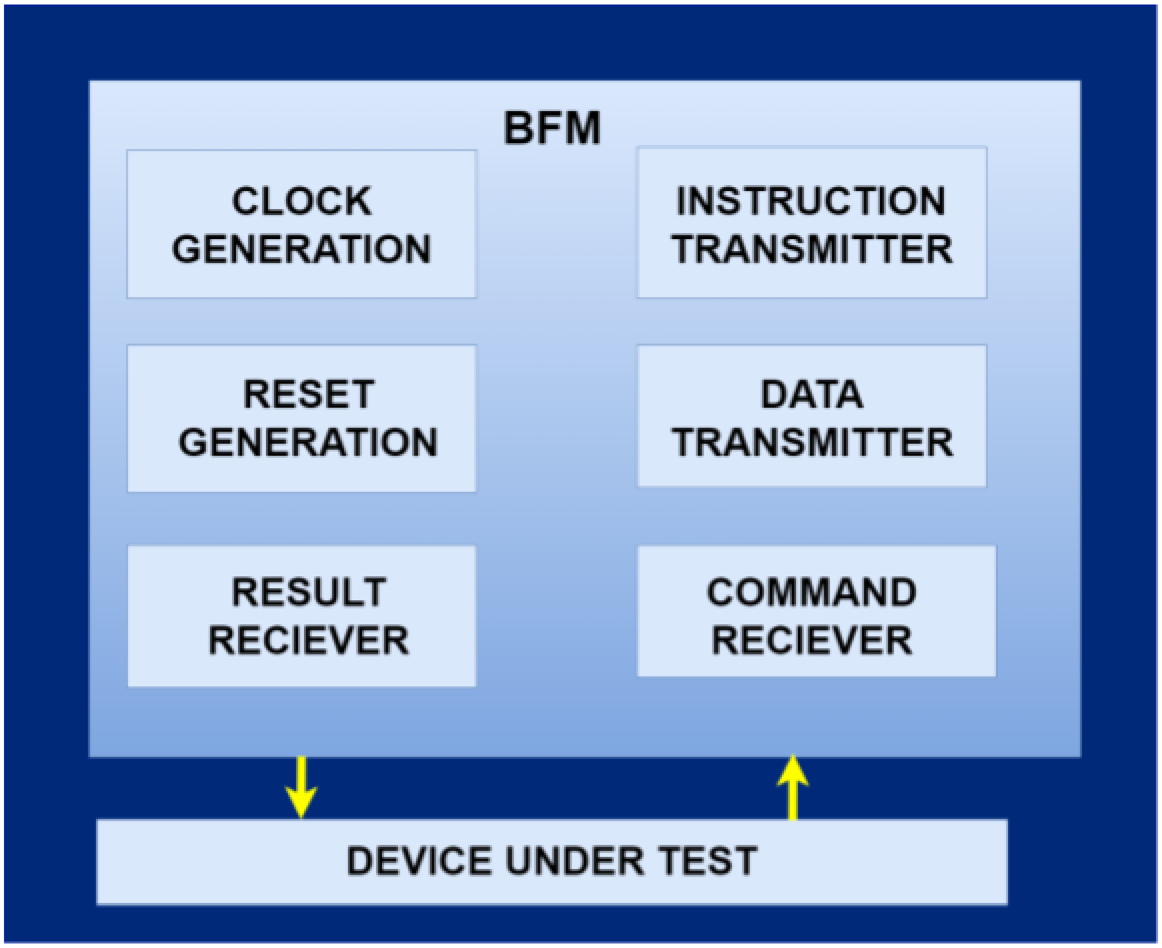

Figure 3.1

Figure 3.1 depicts a BFM model for the RV32 Single Cycle Core. This model employed nearly six functions. Two of the functions are responsible for generating Clock and Reset. The other functions are used for data and instructions transmission and others are used to get results and commands from the DUT.

Six functions were employed, as previously stated. The clock function generated a clock for the DUT, while the reset function provided a reset to the DUT. For loops are used to store the instructions and data transmitted by Driver in a queue. Following the initialization of the queue, the instruction transmitter and data transmitter functions are called one by one to transmit the stimuli to the DUT.

The command receiver function's main objective is to capture the stimulus transmitted to the DUT via BFM and compare it to the stimulus generated by the generator. Similarly, the result receiver function was in charge of capturing the result from the DUT and storing it in a queue, which was later converted into a log file for future use by other components. Frequent usage of queues and for loop affects the execution time of the framework

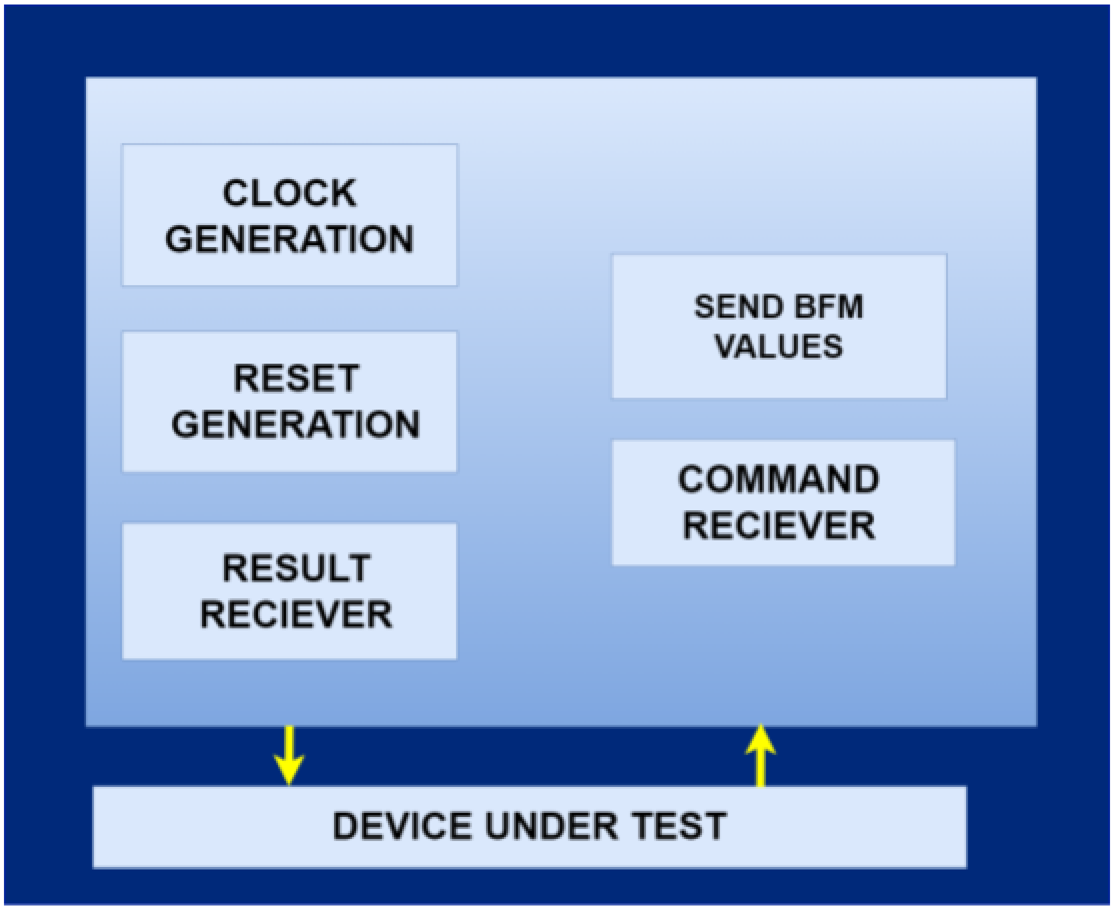

Figure 3.2

The modified block diagram of the BFM is shown in figure 3.2. As previously stated, one of our early framework's flaws was that we use so many queues and conditions rather than just make a simple BFM interface to DUT. Hence all the queue elements and loops have been removed from the BFM.

The modified BFM contains five functionalities. As previously indicated, two of the functions are in responsibility 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.
Five functions were employed, as previously stated. The clock function generated a clock for the DUT, while the reset function provided 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 and compare it to the stimulus generated by the generator. 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.

Sequencer

Earlier, we thought that the generator was our sequencer, but after a literature analysis and a deep dive session with team members, we decided that the sequencer is not built in the same way as the sequencer is made in UVM. Following that, we decided to recognise the sequencer components for greater use in the PY-UVM framework.

Sequencer is one of the Py-UVM framework components inherited from the UVM component. The main objective of a sequencer component is to pass the stimuli that are generated by the generator in a sequence to the DUT.

Currently, the sequencer extends with a uvm component, which allows us to use uvm phases. The build phase is one of them, while the run phase is the other. A Generator handler is declared in the build phase, along with two separate blocking put ports of TLM FIFO.

The generator generates the stimuli in the Start of Simulation Phase. As previously stated, all run phases of UVM components execute in parallel; therefore, the sequencer gets the stimuli array from the generator via a database. The Sequencer extracts the individual stimulus from the array and sends it to two TLM FIFO ports.

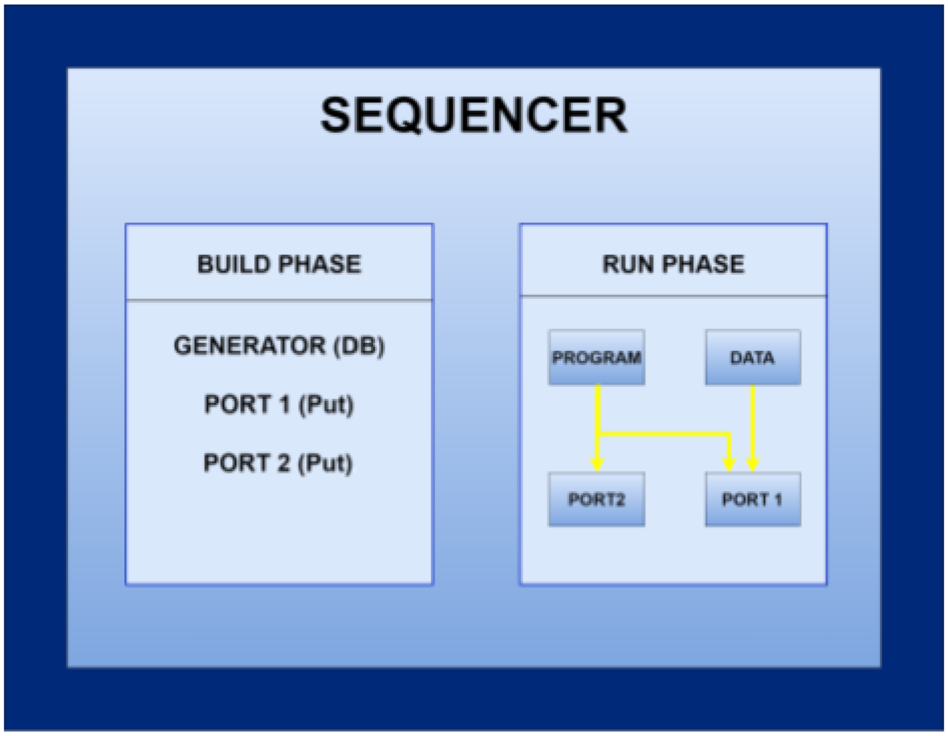

Figure 4.1

Driver

Driver is one of the Py-UVM framework components inherited from the UVM component. The Driver component's main function is to transfer the stimulus generated by the generator / tester to the DUT via a specific protocol.

As previously stated, one of the shortcomings of our initial framework was that the stimuli transactions were not handled by the driver component. The driver component was only in charge of transferring the stimuli array to BFM from the sequencer. To transfer the stimuli array to BFM, we designed two functions, instruction transmitter and data transmitter, as shown in Figure 5.1 below:

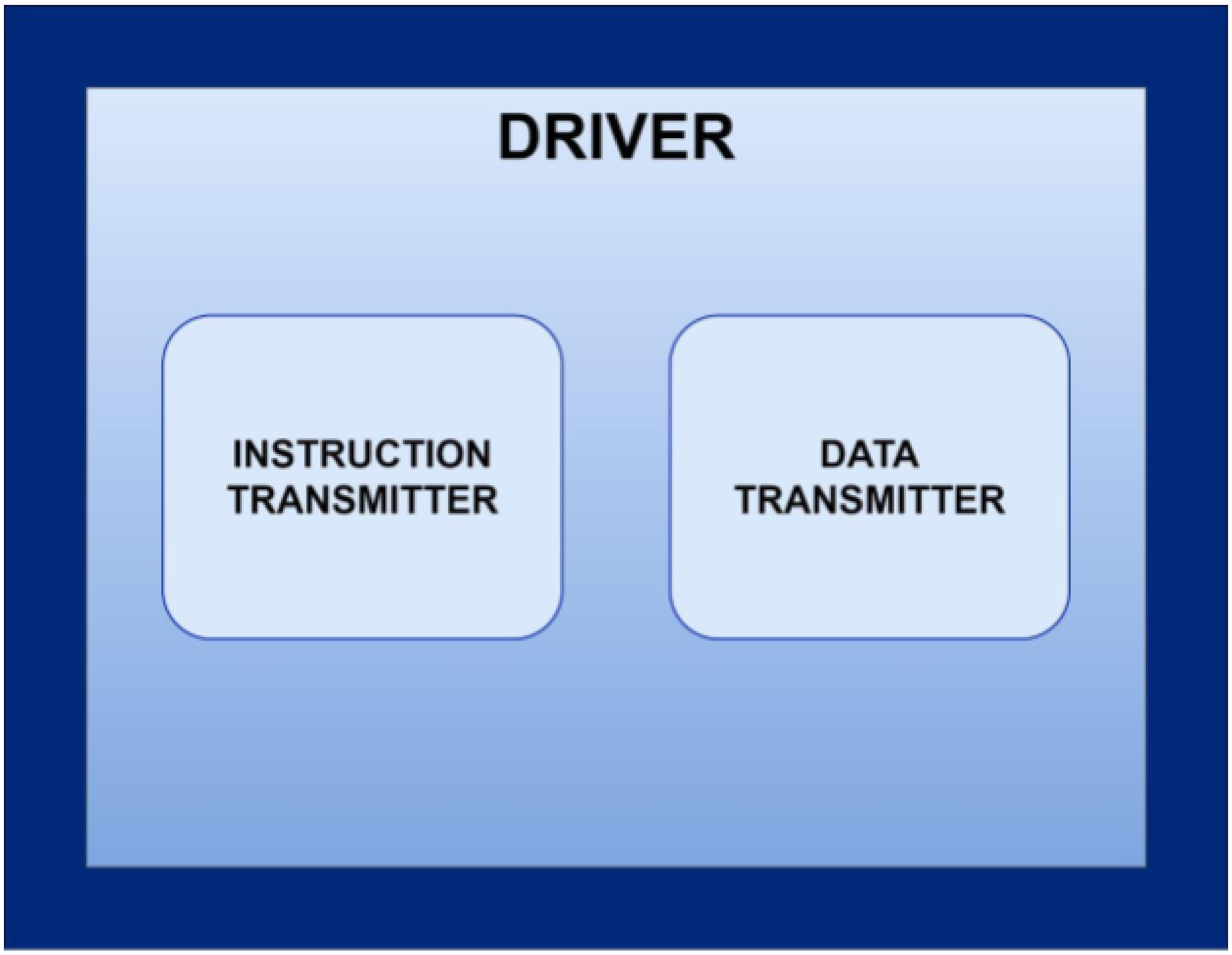

Figure 5.1

Both functions were called internally in another function in the driver component called run driver. The run phase of Base Tester Class invoked the run driver function.

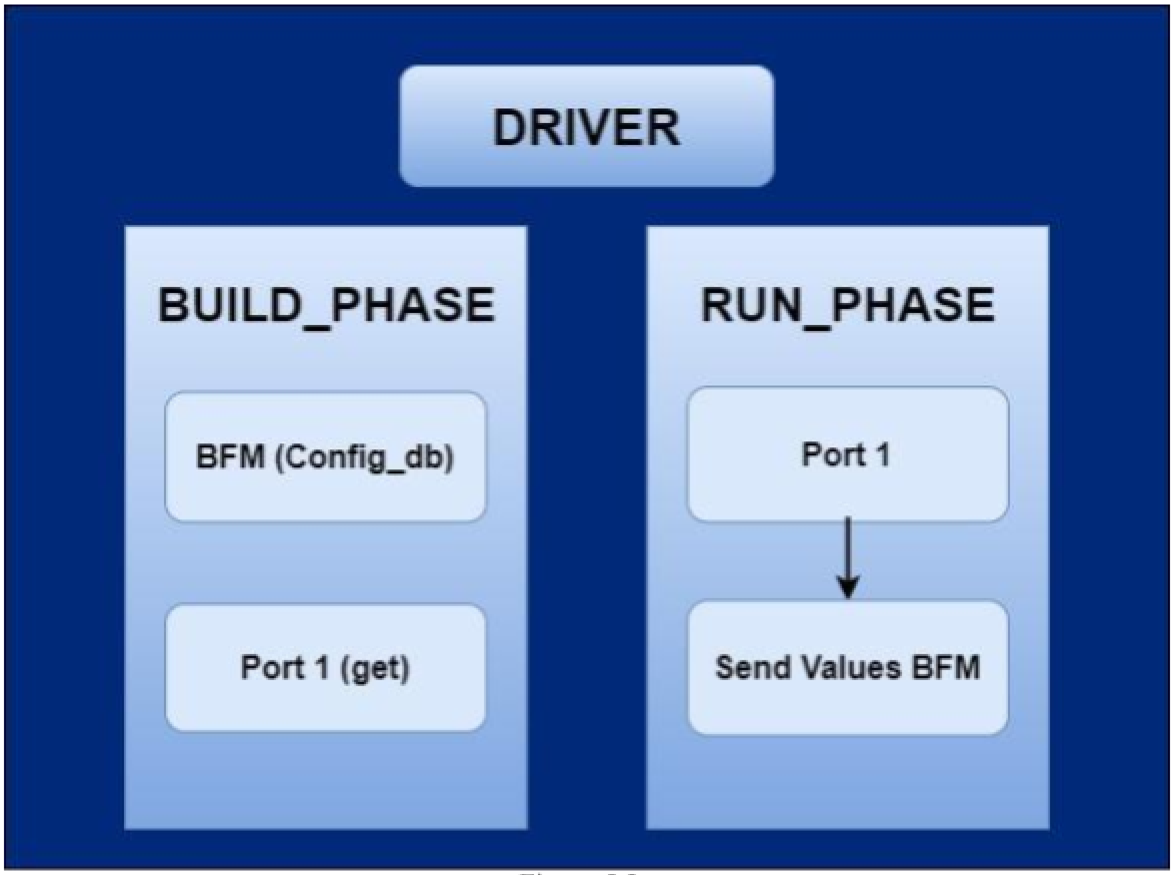

Figure 5.2

The modified block diagram of the driver component is shown in Figure 5.2. As previously stated, one of our early framework's flaws was that we did not employ inherent phases of the UVM component to execute our driver component. In the above figure it is visible that two of the phases have been used to carry out the execution of the driver component. In the build phase an handler of BFM is declared along with the blocking get port of TLM FIFO. As previously stated, all run phases of UVM components execute in parallel; therefore, when the sequencer provides the stimulus on the put port of TLM FIFO, the driver receives it via the get port and passes it to one of the function of BFM, which transmits the stimulus straight to the DUT.

Monitor

The sixth 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.

As was already mentioned, one of the drawbacks of our first framework was the frequent usage of queues in the Monitor components as well as the separate functions that were created to carry out various activities rather than utilizing the inherent phases of UVM components.

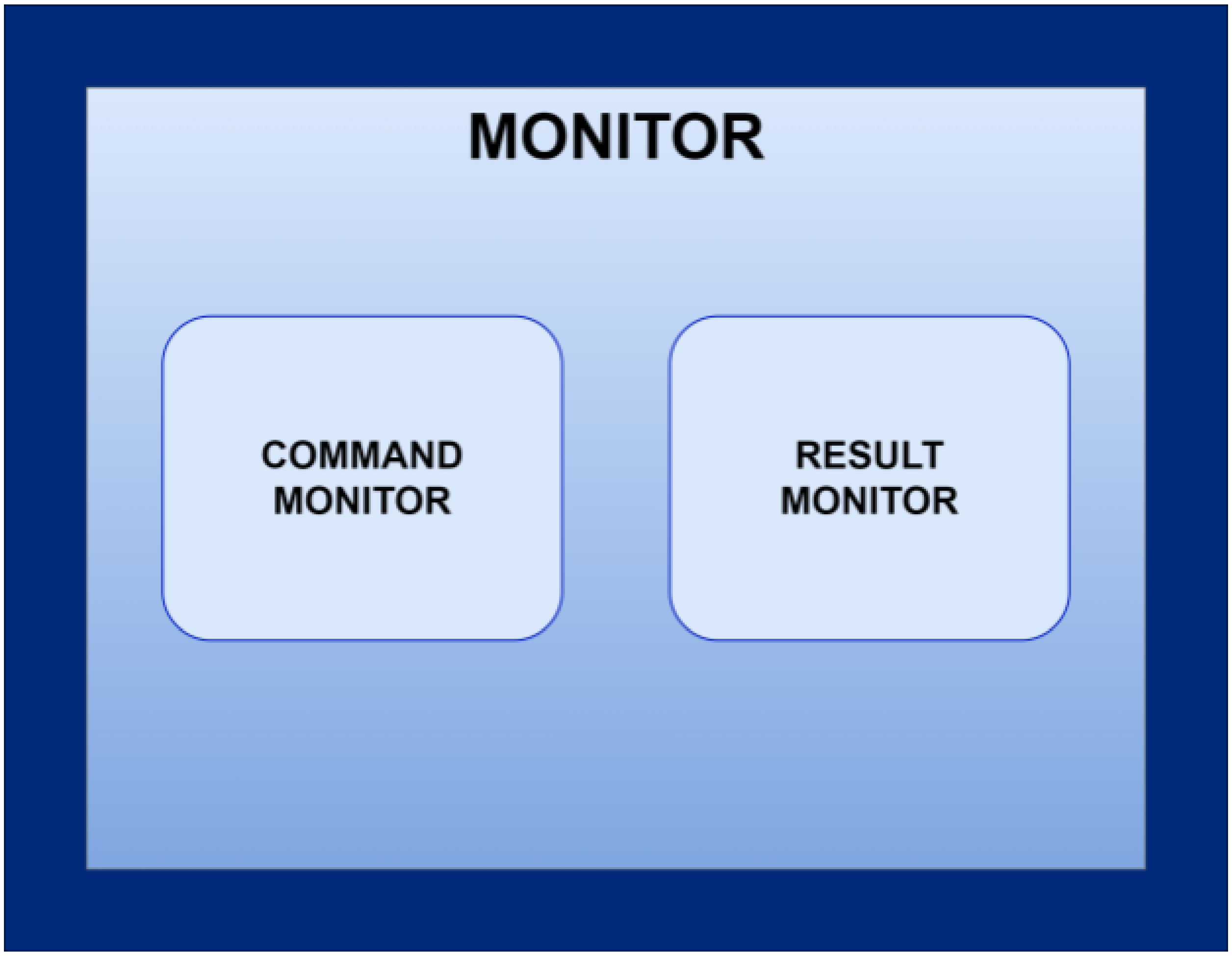

Figure 6.1

Figure 6.1 depicts the Monitor component's two distinct functions. The command monitor function's primary purpose is to capture 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.

Both functions were called internally in another function in the monitor component called run_monitor. The run phase of Base Tester Class invoked the run monitor function.

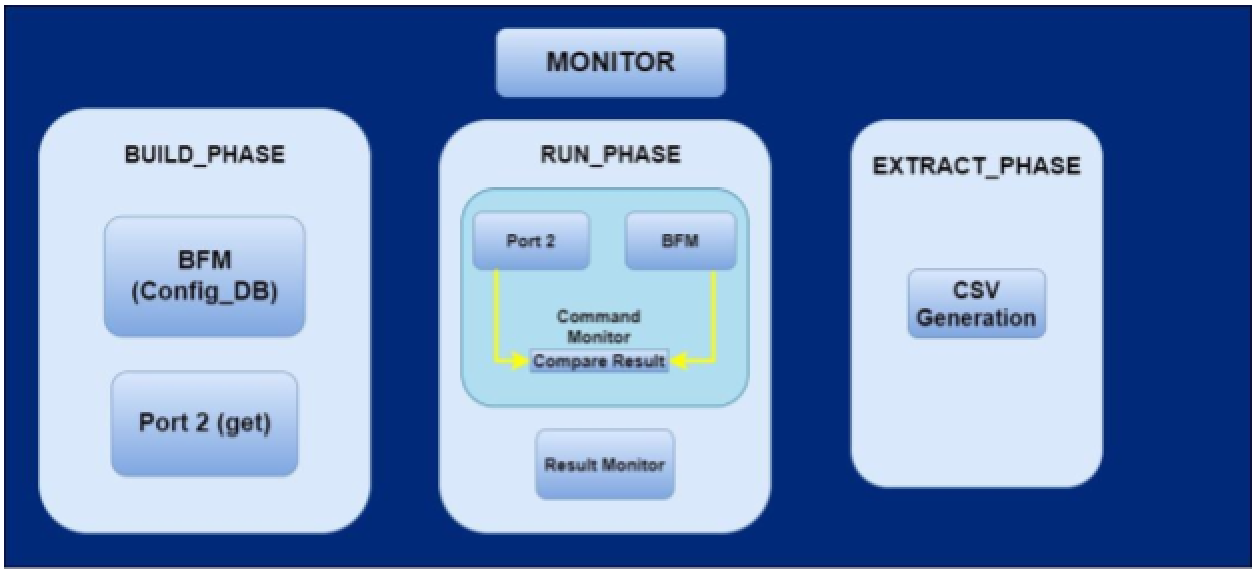

Figure 6.2

The modified block diagram of the monitor component is shown in Figure 6.2. As mentioned, one of our early framework's flaws was that we did not employ inherent phases of the UVM component to execute our monitor component. 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 an handler of BFM is declared along with the get port of TLM FIFO rather than using queues. As previously stated, all run phases of UVM components execute in parallel; therefore, when the sequencer provides instructions on the put port 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.

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 RISCV-based architecture.

As previously stated, one of our early framework's shortcomings was that we did not use the UVM component's inherent phases to execute our ISS component, therefore the function of the ISS component was called during the run phase of a Base Tester Class.

The only update to an ISS component is that all of his functionality has been executed in the extract phase of the UVM component.

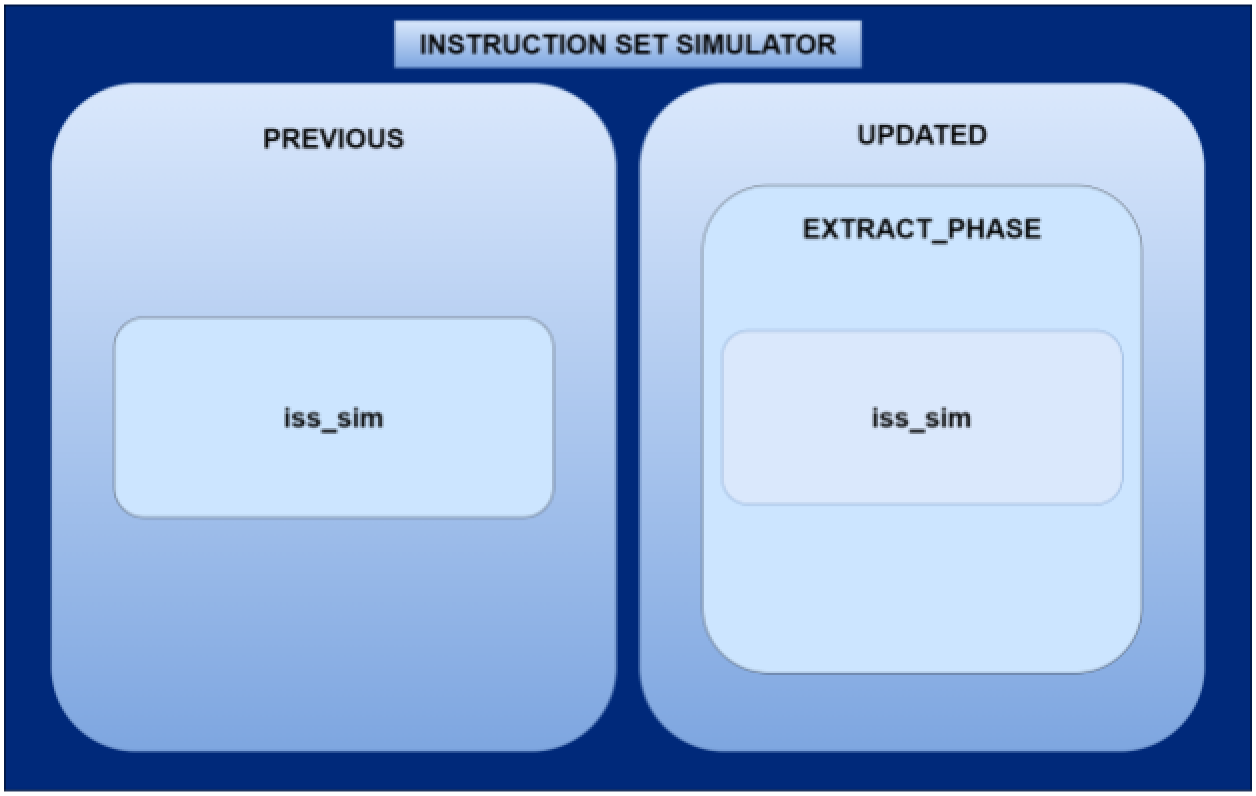

Figure 7.1

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 (ISS).

As previously stated, one of our early framework's shortcomings was that we did not use the UVM component's inherent phases to execute our scoreboard component, therefore the function of the scoreboard component was called during the run phase of a Base Tester Class.

The only update to a scorecard component is that all of his functionality has been executed in the check phase of the UVM component.

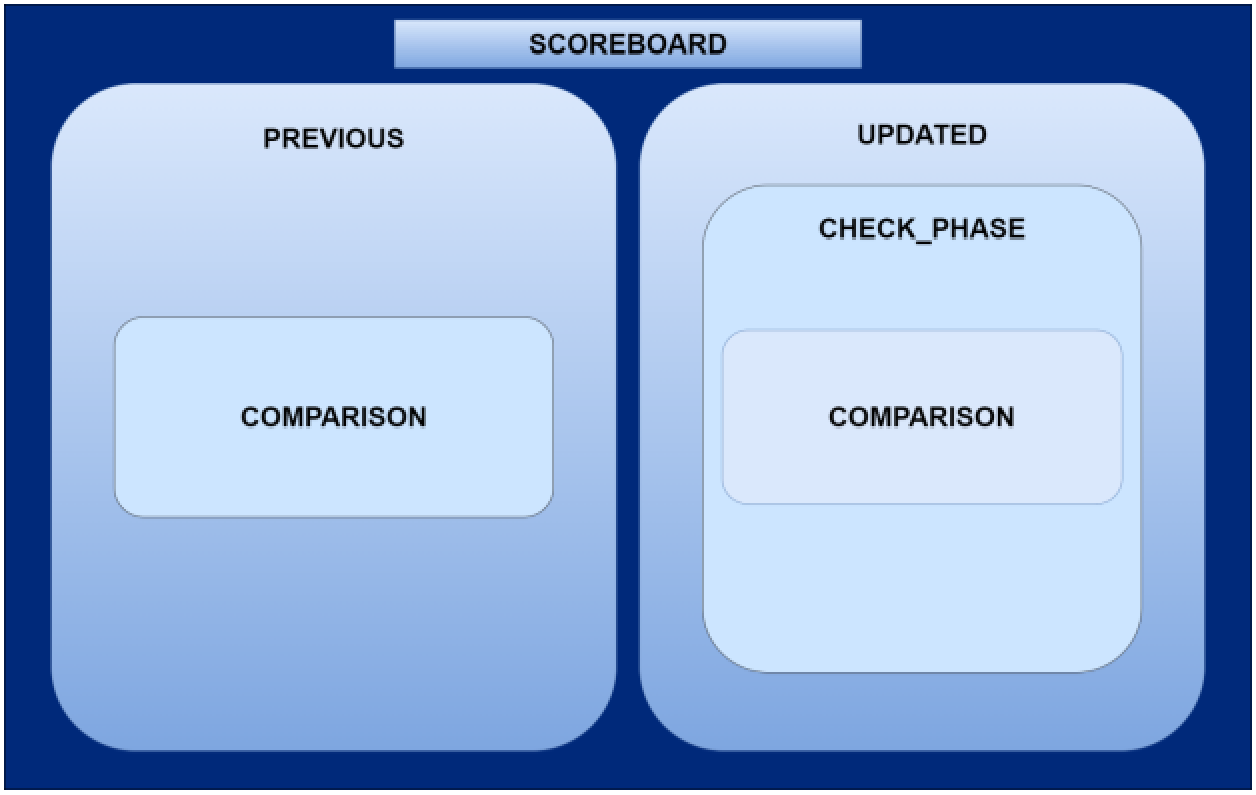

Figure 8.1

Coverage

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.

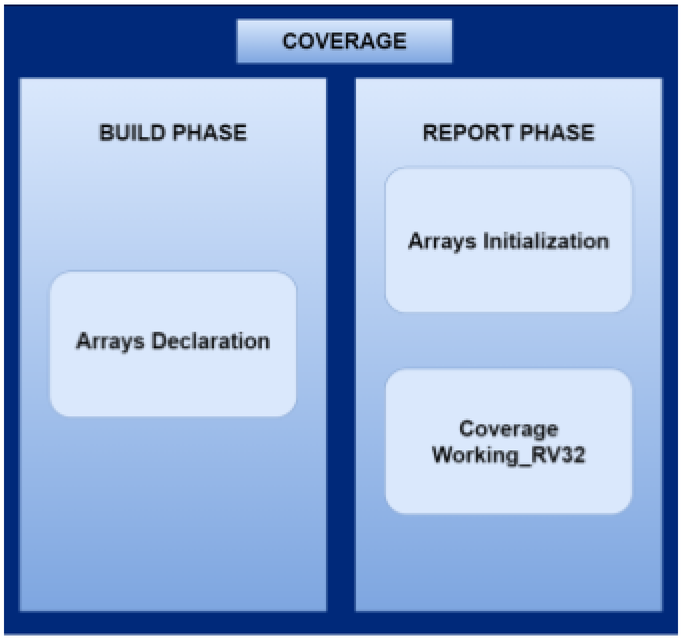

Figure 9.1

Figure 9.1 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.