UVM for Kmeans IP

[Date]

Liora Huf

Edi Sraiber

Led by: Goel Samuel

Contents

[1 Hardware verification 1](#_Toc52869232)

[1.1 What,why,how? 1](#_Toc52869233)

[2 UVM 2](#_Toc52869234)

[2.1 Structure 2](#_Toc52869235)

[2.2 Coverage 2](#_Toc52869236)

[3 DUT 3](#_Toc52869237)

[3.1 Architectural description 3](#_Toc52869238)

[3.2 Details of parameters that can be varied 3](#_Toc52869239)

[3.3 Description of APB protocol 3](#_Toc52869240)

[4 Implemented Verification environment 4](#_Toc52869241)

[4.1 UVM used classes 4](#_Toc52869242)

[4.2 Ref Model 4](#_Toc52869243)

[5 Test Plan 5](#_Toc52869244)

[5.1 Test Line 1 – Basic Test 5](#_Toc52869245)

[5.2 Test Line 2 – Out as In Test 5](#_Toc52869246)

[5.3 Test Line 2 – Robustness Test 5](#_Toc52869247)

[5.4 6](#_Toc52869248)

[6 Results 7](#_Toc52869249)

[7 Bug Fixes 8](#_Toc52869250)

[8 Conclusion 9](#_Toc52869251)

[9 Bibliography 10](#_Toc52869252)

[10 Appendix A – How to integrate Matlab code to UVM enviroment 11](#_Toc52869253)

# Hardware verification

What is the goal of Verification? The most common answer to this question is “Finding bugs”, but it is only partially correct. The goal of hardware design is to create a device which performs a particular task, based on a design specification. The purpose of hardware verification is to ensure that the devices performs this task successfully, i.e. the device is an accurate representation of the specification. Bugs are only the result of the discrepancy between the device design and the device specification.

Functional design verification has been and continues to be a long pole in the entire design cycle from architecture to tape-out. Many excellent methodologies have emerged to tackle this never-ending dilemma. UVM (Universal Verification Methodology) and UPF (Unified Power Format for Low Power) have now become cornerstones of pretty much all functional design verification methodologies. It is indeed a robust, configurable, transaction level reusable methodology.

## What, why and how?

As mentioned before, Verification is the process in which a DUT is tested to ensure that it performs the tasks described in its specification successfully. In this section, the verification process main features are explained.

### Turning Simulation into Verification

Simulation might be caricatured as the process of poking test vectors into a model of the DUT and observing how that model behaves. A traditional Verilog or VHDL test bench might contain processes to read raw vectors or commands from a file, use those to change the values of the wires connected to the DUT over time, and perhaps collect output from the DUT and dump it to another file. This is fine as far as it goes, but this process does not scale up well to support the reliable verification of very complex systems.

A good verification methodology starts with a statement of the function the DUT is intended to perform. From this is derived a verification plan, broken down feature-by-feature, and agreed in advance by all those with a specific interest in creating a working product. This verification plan is the basis for the whole verification process. Verification is only complete when every item on the plan has been tested to an acceptable level, where the meaning of "acceptable" and the priorities assigned to testing the various features have also been agreed in advance and are continually reviewed during the project.

Verification of complex systems should not be reliant on manual inspection of detailed waveforms and vector sets. Functional checking must be automated if the process is to scale well, as must the collection of verification metrics such as the coverage of features in the verification plan and the number of bugs found by each test. Along with the verification plan, automated checking and functional coverage collection and analysis are cornerstones of any good verification methodology and are explicitly addressed by SystemVerilog and UVM. Checkers and a functional coverage model, linked back to the verification plan, take engineering time to create but result in much improved quality of verification.

All simulation-based verification suffers from the issue that you can never run enough test vectors to exhaustively test the whole design, or even any significant part of a complex design. One way to address this issue is using constrained random stimulus. The use of random stimulus brings two very significant benefits. Firstly, random stimulus is great for uncovering unexpected bugs, because given enough time and resources it can allow the entire state space of the design to be explored free from the selective biases of a human test writer. Secondly, random stimulus allows compute resources to be maximally utilised by running parallel compute farms and overnight runs. Of course, pure random stimulus would be nonsensical, so adding constraints to make random stimulus legal is an important part of the verification process and is explicitly supported by SystemVerilog and UVM.

The best way to approach the verification process is to start with simple directed (non-random) tests to bring up the design, then move to fully random tests to explore the state space in a broad fashion and flush out as many bugs as possible with minimum human effort devoted to test writing. This will typically achieve much less than 100% functional coverage, and the remainder of the verification process is spent defining a series of tests, each of which constrains and shapes the random stimulus is a different way to push the design into interesting corner cases. The state space of a typical design is so vast that random stimulus alone is not enough to explore all the key use cases, yet directed or highly constrained tests can be too narrow to give good overall coverage. Constrained random stimulus is a compromise between the two extremes, but effective usage comes down to making a series of good engineering judgements. The solution is to use the priorities set in the verification plan to direct verification resources to the key areas

### Checkers, Coverage and Constraints

Constrained random verification relies on Checkers, Coverage and Constraints. Each of these "three C's" plays a key role in the verification process and is supported by explicit features of the SystemVerilog language.

Firstly, checkers ensure functional correctness. Nothing is gained by throwing more and more random stimulus into a design to take functional coverage to ever higher levels unless the DUT is being checked automatically for functional correctness. Checkers can be implemented using SystemVerilog assertions or using regular procedural code. Assertions can be embedded within the DUT, placed on the external interfaces, or can be part of the verification environment. UVM provides mechanisms and guidelines for building checkers into the verification environment and for logging reports.

Secondly, coverage provides a measure of the functional completeness of the testing and tells when the goals set out in the verification plan are met, and thus when the simulating has finished. SystemVerilog offers two separate mechanisms for functional coverage collection: property-based coverage (cover directives) and sample-based coverage (cover groups). Both can be used in a UVM verification environment. The specification and execution of the coverage model is intimately tied to the verification plan, and many simulation tools are able to annotate coverage information onto the verification plan document, facilitating tight management control.

Thirdly, constraints provide the means to reach coverage goals by shaping the random stimulus to push the design-under-test into interesting corner cases. Without shaping, random stimulus alone may be insufficient to exercise many of the deeper states of the design-under-test. Constrained random stimulus is still random, but the statistical distribution of the vectors is shaped to ensure that interesting cases are reached. SystemVerilog has dedicated language features for expressing constraints, and UVM goes further by providing mechanisms that allow constraints to be written as part of a test rather then embedded within dedicated verification components. This and other features of UVM facilitate the creating of reusable verification components.

### Test and Coverage

The features enumerated in the verification plan should be captured as a set of coverage statements that together form an executable coverage model. With many simulation tools, the verification plan will include references to the corresponding coverage statements, and as simulation runs, coverage data is back-annotated from the simulator onto the verification plan feature-by-features. This provides direct feedback on the effectiveness of any given test. Holes in the coverage goals can be plugged by writing further tests. The verification plan itself is not part of UVM proper, but is a vital element in the verification process. UVM provides guidance on how to collect coverage data in a reusable manner.

With directed testing, tests are written with the purpose of pushing the design into specific states and exercising specific cases. With constrained random testing, the role of the tests shifts slightly. Although a constrained random test may be written with specific coverage goals in mind, it is not assumed before-the-fact that any particular test will actually test one feature rather than another. The constrained random test is run, and the coverage model is used to empirically measure which features the test did in fact exercise. Tests can be graded after-the-fact using the coverage data, and the most effective tests, that is those that achieve the highest coverage in the fewest number of cycles, can be used to form the basis of a regression test set.

### Verification reuse

UVM facilitates the construction of verification environments and tests, both by providing reusable machinery in the form of a library of SystemVerilog classes, and also by providing a set of guidelines for best practice when using SystemVerilog for verification.

Verification productivity can be enhanced by reusing verification components, and this is an important objective of UVM. Verification reuse is enabled by having a modular verification environment where each component has clearly defined responsibilities, by allowing flexibility in the way in which components are configured and used, by having a mechanism to allow imported components to be customized to the application at hand, and by having well-defined coding guidelines to ensure consistency.

The architecture of UVM has been designed to encourage modular and layered verification environments, where verification components at all layers can be reused in different environments. Low-level driver and monitor components can be reused across multiple designs-under-test. The whole verification environment can be reused by multiple tests and configured top-down by those tests. Finally, test scenarios can be reused from application to application. This degree of reuse is enabled by having UVM verification components able to be configured in a very flexible way without modification to their source code. This flexibility is built into the UVM class library.

# UVM

SystemVerilog is a language (HDL) just like Verilog, having its own constructs, syntax and features. In the other hand, UVM is a framework of SystemVerilog classes from which fully functional testbenches can be build.

The primary advantage of the UVM is that this methodology specifies and lays out a set of guidelines to be followed for creation of verification testbenches. This fact ensures testbench uniformity between different verification teams, cross-compatibility between IPs and standalone environment integration, as well as flexibility and ease of maintaining testbenches.

Every verification environment has similar components like drivers, monitors, stimulus generators and scoreboards. UVM provides a build in base class for each of these components with standardized functions to instantiate, connect and build the test bench environment.

## UVM Factory

A factory is a commonly used concept in object-oriented programming. It is an object that is used for instantiating other objects. The UVM Factory is mechanism

There are two ways to register an object with the UVM factory. In the declaration of class A, one can invoke the `uvm\_object\_utils(A) or `uvm\_component\_utils(A) registration macros. Otherwise, the `uvm\_object\_registry(A,B) or `uvm\_component\_registry(A,B) macros can be used to map a string B to a class type A [3]. The UVM factory provides a variety of create methods that allow the user to instantiate an object with a particular instance name and of a registered type

## UVM Phases

UVM Phases are a synchronizing mechanism for the environment. Phases are represented by callback methods, a set of predefined phases and corresponding callbacks are provided in uvm\_component. The Method can be either a function or task.

Any class deriving from uvm\_component may implement any or all of these callbacks, which are executed in a particular order.

The UVM Phases are:

* build
* connect
* end of elaboration
* start of simulation
* run
* extract
* check
* report

The run phase is implemented as a task and remaining all are functions.

### Build Phases

The following phases belong to this category: build phase, connect phase and end\_of\_elobaration phase.

Phases in this categorize are executed at the start of the UVM testbench simulation, where the testbench components are constructed, configured and testbench components are connected.

All the build phase methods are functions and therefore execute in zero simulation time.

### Run-time Phases

The following phases belong to this category: start of simulation and run phase. The run phase will get executed from the start of simulation to till the end of the simulation. The run phase is time-consuming, where the testcase is running

### Clean-up Phases

The following phases belong to this category: extract, check, report and final phase.

In these phases the results of the testcase are collected and reported. For example: the number of error’s during the simulation is reported.

## Structure

The UVM structure can be seen in Figure 1 below. Each of the components seen in this figure will be explained in this section

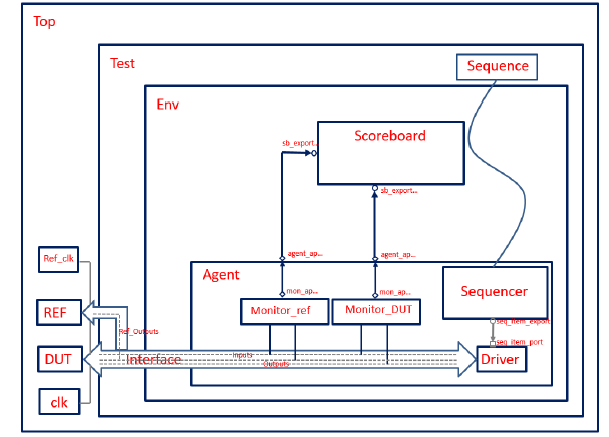


Figure 1:UVM environment schematic

### Top block

In a normal project, the development of the DUT is done separately from the development of the testbench, so there are two components that connects both of them:

* The top block of the testbench
* A virtual interface

The top block will create instances of the DUT,the Reference model and of the testbench. It will also declare the virtual interface,which will act as a bridge between the Test component and the DUT/Reference Model.

The interface is a module that holds all the signals of the DUT. The monitor, the driver and the DUT are all going to be connected to this module.

This block will be a normal SystemVerilog module and it will be responsible for:

* Connecting the DUT and Reference Model to the test class, using the interface defined before.
* Generating the clock for the DUT.
* Registering the interface in the UVM factory. This is necessary in order to pass this interface to all other classes that will be instantiated in the testbench. It will be registered in the UVM factory by using the uvm\_resource\_db method and every block that will use the same interface, will need to get it by calling the same method.
* Running the test.

### Sequence and Sequencer

The first step in verifying a RTL design is defining what kind of data should be sent to the DUT. While the driver deals with signal activities at the bit level, it doesn’t make sense to keep this level of abstraction far away from the DUT, so the concept of transaction was created.

A transaction is a class object, usually extended from uvm\_transaction or uvm\_sequence\_item classes, which includes the information needed to model the communication between two or more components.

Transactions are the smallest data transfers that can be executed in a verification model. They can include variables, constraints and even methods for operating on themselves. Due to their high abstraction level, they aren’t aware of the communication protocol between the components, so they can be reused and extended for different kind of tests if correctly programmed.

An example of a transaction could be an object that would model the communication bus of a master-slave topology. It could include two variables: the address of the device and the data to be transmitted to that device. The transaction would randomize these two variables and the verification environment would make sure that the variables would assume all possible and valid values to cover all combinations.

In order to drive a stimulus into the DUT, a driver component converts transactions into pin wiggles, while a monitor component performs the reverse operation, converting pin wiggles into transactions.

After a basic transaction has been specified, the verification environment will need to generate a collection of them and get them ready to be sent to the driver. This is a job for the sequence. Sequences are an ordered collection of transactions, they shape transactions to our needs and generate as many as we want. This means if we want to test just a specific set of addresses in a master-slave communication topology, we could restrict the randomization to that set of values instead of wasting simulation time in invalid values.

Sequences are extended from uvm\_sequence and their main job is generating multiple transactions. After generating those transactions, there is another class that takes them to the driver: the sequencer.

The sequence englobes a group of transactions and the sequencer takes a transaction at the time from the sequence , sending it to the driver.

The following figure demonstrates the relation between the sequence, sequencer and driver:

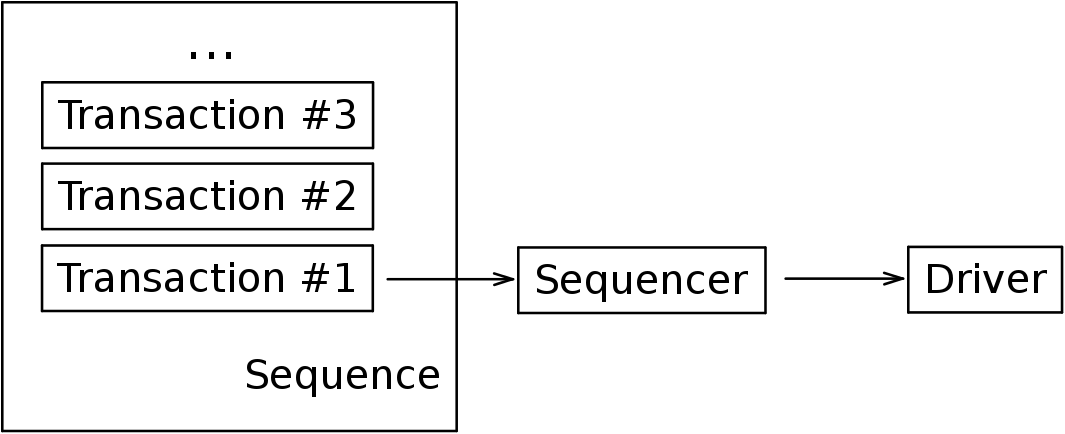


Figure 2: Relation between a sequence, a sequencer and a driver

### Driver

The driver is a block whose role is to interact with the DUT. The driver pulls transactions from the sequencer and sends them repetitively to the signal-level interface. This interaction will be observed and evaluated by another block, the monitor, and as a result, the driver’s functionality should only be limited to send the necessary data to the DUT.

### Monitor

The monitor is a self-contained model that observes the communication of the DUT with the testbench. At most, it should observe the outputs of the design and, in case of not respecting the protocol’s rules, the monitor must return an error.

The monitor is a passive component, it doesn’t drive any signals into the DUT, its purpose is to extract signal information and translate it into meaningful information to be evaluated by other components. A verification environment isn’t limited to just one monitor, it can have multiple of them. In the case of ths project, the eviroemnt will have two monitors: one for the DUT and one for the Reference Model.

The monitors should cover the outputs of the DUT/Reference Model in order to later send them to the scoreboard.

### Agent

The purpose of the agent module is to connect the both monitors, the sequencer and the driver.An agent doesn’t require a run phase, there is no simulation code to be executed in this block but there will be a connect phase, besides of the build phase.

The Agent component will construct the monitors, the sequencer and the driver in the build phase. It will also need to create two analysis ports, these ports will act as proxies for the monitors to be connect to an external scoreboard through the agent’s ports.

After it has constructed the components mentioned before, the Agent has to make the connections between them. Using the concept of TLM ports, it can connect each port to its destination.

### Scoreboard

The scoreboard is a crucial element in a self-checking environment, it verifies the proper operation of a design at a functional level. In the case of this project, the same inputs are given to the DUT and the Reference Model, and their outputs are monitored by the monitors. The scoreboard them receives this outputs and core them.

In the other hand, there are designers who prefer to leave the prediction to the scoreboard. So the functionality of the scoreboard is very subjective.

In the agent, two monitors were created, as a result, two analysis exports have to be created in the scoreboard, which are going to be used to retrieve transactions from both monitors. After that, a method compare() is going to be executed in the run phase and compare both transactions. If they match, it means that the Reference Model and the DUT both agree in the functionality and it will return an “OK” message.

### Env

The env is a very simple class that instantiates the agent and the scoreboard and connects them together.

### Test

At last, one more block is created: the test. This block will derive from the uvm\_test class and it will have two purposes:

* Create the env block
* Connect the sequencer to the sequence

The fact that the sequencer and the sequence are connected in this block, instead of the agent block or the sequence block, is because by specifying in the test class which sequence will be going to be generated in the sequencer, the kind of data is transmitted to the DUT can be easily changed, without any change in the agent’s or sequence’s code.

## Coverage

In traditional directed verification methodology, thet testcase pass/fail results are used to measure the verification status (functional correctness) & code coverage (which determines how much design code is exercised by the test scenarios generated by the Testbench).

However verification coverage comes into various different types :

* Code Coverage (which lines of code are exercised)
* Condition Coverage (weather certain expressions and sub-expressions in code evaluate to true or false)
* Functional coverage(how much design functionality has been exercised/covered by the testbench or verification environment)
* FSM Coverage (which states and possible state transitions are exercised)

### Code Coverage

It specifies that how much deep level the design is checked. There are sub parts of the code coverage that will be discussed bellow.

#### Statement/Line Coverage

This is the easiest understandable type of coverage. This is required to be 100% for every project. From N lines of code and according to the applied stimulus how many statements (lines) are covered in the simulation is measured by statement coverage. Lines like *module*, *endmodule*, *comments*, *timescale*, etc are not covered.

#### Block/Segment Coverage

The nature of the statement and block coverage looks somewhat same. The difference is that block which is covered by begin-end, if-else or always, those group of statements which is called block counted by the block coverage.

### Conditional Coverage

Conditional coverage will report the true or false of the branch like if-else, case and the ternary operator (? :) statements. In these statements the execution is depending upon the implementation of stimulus. The default branch in case statement in RTL is not exercised mostly because the Design guidelines insist to mention all the branches of the case statement.

### Functional Coverage

It works on the functional part of the stimuli's implementation. Functional coverage will check the overall functionality of the implementation.

### FSM Coverage

It is the most complex type of coverage, because it works on the behavior of the design. In this coverage we look for how many times states are visited, transited and how many sequence are covered. That is the duty of FSM coverage.

This coverage has three mains parts: stage coverage, transition coverage and sequence coverage.

#### State coverage

It gives the coverage of number of states visited over the total number of states. Suppose the design FSM has N number of states and state machines transecting is in between only N-2 states, then coverage will give alert that some states are uncovered. It is advised that all the states must be covered.

#### Transition Coverage

It will count the number of transition from one state to another and it will compare it with the total number of transitions. The total number of transition is nothing but all possible number of transition which is present in the finite state machine.

#### Sequence Coverage.

In some FSM there are many sequences of states possible. This coverage purpose is to check which sequences have been covered in the test and which have not. Stimulus should be such a way that all the possibilities must be covered.

# DUT

TBD

## Architectural description

TBD

## Details of parameters that can be varied

TBD

## Description of APB protocol

TBD

# Implemented Verification environment

TBD

## UVM used classes

TBD

## Ref Model

TBD

# Test Plan

Description of all scenarios to be defined : which parameters will be exercised.

         How is equivalence defined

         Description of Assertions to test APB interface

## Test Line 1 – Basic Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values
3. Initial Centroid values

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 2 – Out as In Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values
3. Initial Centroid values

These randomly generated parameters should be sent to the DUT and the REF Model. After verifying that the outputs given by the DUT and the REF Model for the mentioned input shall are equivalent (every output centroid presented by the DUT is also presented by the REF Model), use the received outputs as input for a new run.

The results expected for the new run are:

1. The output centroids are equal to the input centroids
2. Convergence reached within one iteration

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 3 – Robustness Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values
3. Initial Centroid values

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

This test line will produce multiple (at least 10.000 runs) tests which will be run in series without breaks.

## Test Line 4 – One Iteration Test

In this test line, the following parameters will be randomly generated:

1. Eight data values

These eight values will be used both as points values and initial centroid values. The pass criteria of this test line is to verify that in all runs convergence is reached in one iteration and final centroids are equal to initial centroids.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 5 – Threshold Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values
3. Initial Centroid values
4. Convergence threshold value (within a constrain of TBD percent)

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 6 – Isolated Centroid Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values
3. Initial Centroid values

Where there one of the following additional constrains:

1. One of the centroids is constrained to be far away from the all the data points. Verify its values does not change (no points are assigned to it)
2. All of the centroids, except from one, are constrained to be far away from the all the data points. Verify their values does not change (no points are assigned to it)

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

In addition, in case ‘a’ the isolated centroid value should not change, and no data points shall be assigned to it.

Similarly, in case ‘b’ all data points should be assign to the non-isolated centroid, while the others centroid values shall not change.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 7 – Full Memory Test

In this test line, the following parameters will be randomly generated:

1. 512 points values
2. Initial Centroid values

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 8 – Positive Overflow Test

In this test line, the following parameters will be randomly generated:

1. Initial Centroid values

Besides, 512 data points will be generated and set to have maximum allowed value.

These generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

It is expected that one final centroid(the one with biggest initial value) should received the maximum allowed value and the rest should remain the initial value.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 9 – Negative Overflow Test

In this test line, the following parameters will be randomly generated:

1. Initial Centroid values

Besides, 512 data points will be generated and set to have minimum allowed value.

These generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

It is expected that one final centroid (the one with smallest initial value) should receive the minimum allowed value and the rest should remain the initial value.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

## Test Line 10 – Equal Initial Values Test

In this test line, the following parameters will be randomly generated:

1. Number of points
2. Points values

Besides, a single value will randomly generated and it will be used as initial values for all centroids.

These randomly generated parameters should be sent to the DUT and the REF Model. The outputs given by the DUT and the REF Model for the mentioned input shall them be compared, being equivalent if every output centroid presented by the DUT is also presented by the REF Model.

This test line shall produce overall ten tests where each test the randomly generated parameters shall have different values.

# Results

TBD

 Results of tests/scenarios run – bug identification

         Results of coverage

         Results of assertions

# Bug Fixes

TBD

## Negative values bug

While building the verification environment, a “sanity check test” done in order to verify if the UVM environments works, the results from the DUT indicated a bug. This bug was apparently connected to the DUT inability to recognize negative values. This bug was fized by the following steps:

* 1. Fix sign representation of variables:

During the calculation, each data point vector to 7 coordinates which shall be represented in fixed point and signed (TBD See reference to chapter blab la in DUT chapters).

* + 1. The variable type of those coordinates were represented in unsigned (default of type in system Verilog is unsigned unless stating "signed" in the type, i.e. signed + type.
    2. The reason for the bug was since it was believed that the compiler will fit to 2's complement when arithmetic operations are being done, yet it did not happened and after diving in a debug process it came up.
    3. The solution was simple in this case and a "signed" syntax was added accordingly for each parsed coordinate process, it shall be noted that as a concatenated vector, the sign does not hold meaning since it matters in coordinate resolution.
    4. The file "accumulator\_adder.sv" changed, as explained above.
  1. Fix 2's complement representation of numbers:
     1. In the summation process of points to form the nominator of the next developed centroids for each iteration, each coordinate holds 22 bits per coordinate(21 + 1 for sign), when each point hold 13(12 + 1 for sign).

See reference to chapter blab la in DUT chapters.

* + 1. When performing arithmetic operations to sum, a negative number represented in 2's complement with 13 bits, wasn’t handled to fit for the operation to be summed to 22 bits number.
    2. The fix was to handled transform the number to its absolute value, then creating the same value in 2's complement representation in 22 bits, then perform arithmetic operations to sum.
    3. The file "distance\_calc.sv" changed, as explained above.

# Conclusion

TBD

# Bibliography

1. <https://verificationguide.com>
2. https://www.chipverify.com/uvm

# Appendix A – How to integrate Matlab code to UVM environment

The following steps should be taken in order to integrate a Matlab function to a System Verilog code. In the case of this report, the Matlab function is used a the Reference Model for the UVM environment.

This is done by exporting a MATLAB function as a component with a direct programming interface (DPI) for use in a System Verilog code.

In order to do so, the following Matlab libraries must be installed:

1. Matlab Coder
2. Matlab HDL Verifier

## DPI Component Generation Steps

1. Write a Matlab function. The some Matlab internal functions are not supported by the DPI generator, therefore after trying to run the generator function, the function code may have to be changed.
2. Write a second matlab function called: build\_dpi. In this function there must be only the *dpigen* Matlab function only.

The *dpigen* receives two mandatory parameters:

* 1. the name of the function intended to be transformed into a DPI component
  2. A flag named *args* followed by the function’s(the function intended to be transformed into a DPI component ) arguments types.

For example, in the case of this project, the Reference Model function receives the following inputs:

1. A 512 by 7 matrix of fixed points numbers, where the integer part is represented by 2 bits, the fractional part is represented by 10 bits and it is signed.
2. A 8 by 7 matrix of fixed points numbers, where the integer part is represented by 2 bits, the fractional part is represented by 10 bits and it is signed.
3. One fixed point number, where the integer part is represented by 2 bits, the fractional part is represented by 10 bits and it is signed.
4. One fixed point number, where the integer part is represented by 13 bits, the fractional part is represented by 0 bits and it not is signed.
5. One fixed point number, where the integer part is represented by 2 bits, the fractional part is represented by 10 bits and it is signed.

Therefore, the depigen command in the case of this projects if the following:

dpigen -args {fi(zeros(512,7),1,13,10,'RoundingMethod','Floor'),fi(zeros(8,7),1,13,10,'RoundingMethod','Floor'),fi(zeros(1,1),1,13,10),fi(zeros(1,1),0,13,0),fi(zeros(1,1),0,13,0)} refModel3.m -rowmajor -launchreport -FixedPointDataType BitVector

Where the named of the function intended to be transformed into a DPI component is *RefModel3.m* .

The used command in this case had additional optional flags for the *dpigen* function, in order to use the Matlab type fixed point type *fi* and how to ”pack” the arguments which are matrixes ( these flags *are -rowmajor -launchreport -FixedPointDataType BitVector*)

For more on the *dpiden* function and optional flags, refer to : <https://www.mathworks.com/help/hdlverifier/ref/dpigen.html>

The *dpigen* function generates a System Verilog DPI component shared library from the chosen MATLAB function and all the functions that the function written in previous steps calls. The generated libraries are:

* 1. .dll for shared libraries if the *build\_dpi* function ir run on Microsoft® Windows® systems
  2. .so for shared libraries on Linux® systems if the *build\_dpi* function ir run on Microsoft® Windows® systems

1. In order to integrate the DPI component in UVM environment, the build\_dpi function must be run on Linux systems(duo to the fact that a .so file is needed). Therefore, created a Matlab folder within the project files folder in a Linux system containing the function intended to be converted to DPI component and the *build\_dpi* function.
2. Run the *build\_dpi* function. The function will create the needed libraries and .sv files within the path *Matlab\_folder*/codegen/so/*function\_name*, where the *Matlab\_folder* is the named of the folder created in the previous step and the *function\_name* is the name of the function intended to be converted to DPI component.
3. In the UVM TBD file, include the dpi generated files:

Include *Matlab\_folder*/codegen/so/*function\_name\_dpi.sv*

Include *Matlab\_folder*/codegen/so/*function\_name\_dpi\_pkg.sv*