UVM for Kmeans IP

[Date]

Liora Huf

Edi Sraiber

Led by: Goel Samuel

Contents

[1 Hardware verification 4](#_Toc54793674)

[1.1 What, why and how? 4](#_Toc54793675)

[1.1.1 Turning Simulation into Verification 5](#_Toc54793676)

[1.1.2 Checkers, Coverage and Constraints 6](#_Toc54793677)

[1.1.3 Test and Coverage 6](#_Toc54793678)

[1.1.4 Verification reuse 7](#_Toc54793679)

[2 UVM 8](#_Toc54793680)

[2.1 UVM Factory 8](#_Toc54793681)

[2.2 UVM Phases 8](#_Toc54793682)

[2.2.1 Build Phases 9](#_Toc54793683)

[2.2.2 Run-time Phases 9](#_Toc54793684)

[2.2.3 Clean-up Phases 9](#_Toc54793685)

[2.3 Structure 9](#_Toc54793686)

[2.3.1 Top block 10](#_Toc54793687)

[2.3.2 Sequence and Sequencer 10](#_Toc54793688)

[2.3.3 Driver 11](#_Toc54793689)

[2.3.4 Monitor 11](#_Toc54793690)

[2.3.5 Agent 12](#_Toc54793691)

[2.3.6 Scoreboard 12](#_Toc54793692)

[2.3.7 Env 12](#_Toc54793693)

[2.3.8 Test 12](#_Toc54793694)

[2.4 Coverage 13](#_Toc54793695)

[2.4.1 Code Coverage 13](#_Toc54793696)

[2.4.2 Conditional Coverage 13](#_Toc54793697)

[2.4.3 Functional Coverage 13](#_Toc54793698)

[2.4.4 FSM Coverage 14](#_Toc54793699)

[3 DUT 15](#_Toc54793700)

[3.1 The K means algorithm 15](#_Toc54793701)

[3.2 Architectural description 16](#_Toc54793702)

[3.3 DUT Parameters 17](#_Toc54793703)

[3.3.1 Mandatory configurations 17](#_Toc54793704)

[3.3.2 Reset 18](#_Toc54793705)

[3.3.3 RAM configuration 18](#_Toc54793706)

[3.3.4 Mandatory registers configuration 18](#_Toc54793707)

[3.3.5 Go signal 18](#_Toc54793708)

[3.4 Optional configurations 18](#_Toc54793709)

[3.4.1 Centroid registers configuration 18](#_Toc54793710)

[3.4.2 Threshold register configuration 18](#_Toc54793711)

[3.5 Description of APB protocol 19](#_Toc54793712)

[3.5.1 Introduction 19](#_Toc54793713)

[3.5.2 Transfers 22](#_Toc54793714)

[4 Implemented Verification environment 27](#_Toc54793715)

[4.1 UVM used classes 27](#_Toc54793716)

[4.2 Ref Model 27](#_Toc54793717)

[5 Test Plan 28](#_Toc54793718)

[5.1 Test Line 1 – Gradual Random Points 28](#_Toc54793719)

[5.2 Test Line 2 – One Iteration Test 28](#_Toc54793720)

[5.3 Test Line 3 – Random Points and Centroids 29](#_Toc54793721)

[5.4 Test Line 4 – Random Constrained Number of Points 29](#_Toc54793722)

[5.5 Test Line 5 – Equal Initial Values Test 29](#_Toc54793723)

[5.6 Test Line 6 – Positive Overflow Test 30](#_Toc54793724)

[5.7 Test Line 7 – Negative Overflow Test 30](#_Toc54793725)

[5.8 Test Line 8 – Full Memory Test 30](#_Toc54793726)

[5.9 Test Line 9 – Fully Random Test 31](#_Toc54793727)

[5.10 Test Line 10 – Out as In Test 31](#_Toc54793728)

[5.11 Test Line 11 – Robustness Test 31](#_Toc54793729)

[5.12 Test Line 12 – Threshold Test 32](#_Toc54793730)

[5.13 Test Line 13 – Isolated Centroid Test 32](#_Toc54793731)

[6 Results 33](#_Toc54793732)

[7 Bug Fixes 34](#_Toc54793733)

[7.1 Negative values bug 34](#_Toc54793734)

[7.2 combinatorical sensitivity list missing item: 34](#_Toc54793735)

[7.3 Fix of controller wrong signal toggle during state machine changes, at 2nd state of empty\_pipe: 35](#_Toc54793736)

[8 Conclusion 36](#_Toc54793737)

[9 Bibliography 37](#_Toc54793738)

[10 Appendix A – How to integrate Matlab code to UVM environment 38](#_Toc54793739)

[10.1 DPI Component Generation Steps 38](#_Toc54793740)

# 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.

Design verification (DV) is a large and complex domain that contains many technologies, languages, and methodologies. The following technologies fall under DV domain:

* UVM (Universal Verification Methodology).
* UPF (Unified Power Format) low-power verification using UPF.
* AMS (analog/mixed signal) verification. Real number modeling, etc.
* SystemVerilog Assertions (SVA) and functional coverage (SFC) languages and
* methodology.
* Coverage-driven verification (CDV) and constrained random verification (CRV).
* Static verification technologies. Static formal verification (model checking),static + simulation hybrid methodology, X-state verification, CDC (clock domain crossing), etc.
* Logic equivalency check (LEC). Design teams mostly take on this task. But the
* DV (design verification) team also needs to have this expertise.
* ESL—Electronic System Level (TLM 2.0) virtual platform development (for
* both software development and verification tests/reference model
* development).
* Hardware/software co-verification (hint: use virtual platform methodology).
* SoC interconnect (bus-based and NoC—network-on-chip) verification.
* Simulation speedup using hardware acceleration, emulation, and prototyping.

In this project, the chosen design verification method was UVM, the main reasons for this choice will be explained in the following sections.

## 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

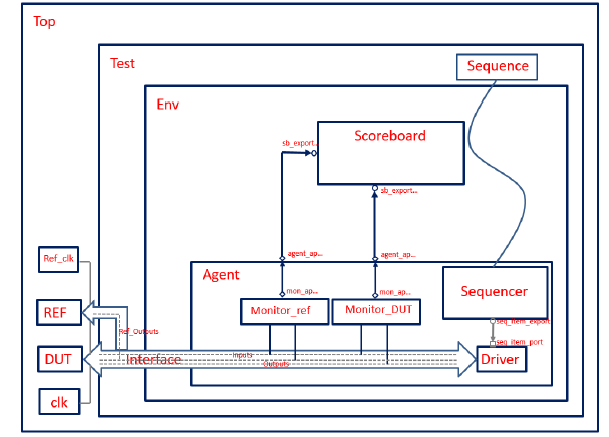


Figure :UVM environment schematic

### 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:

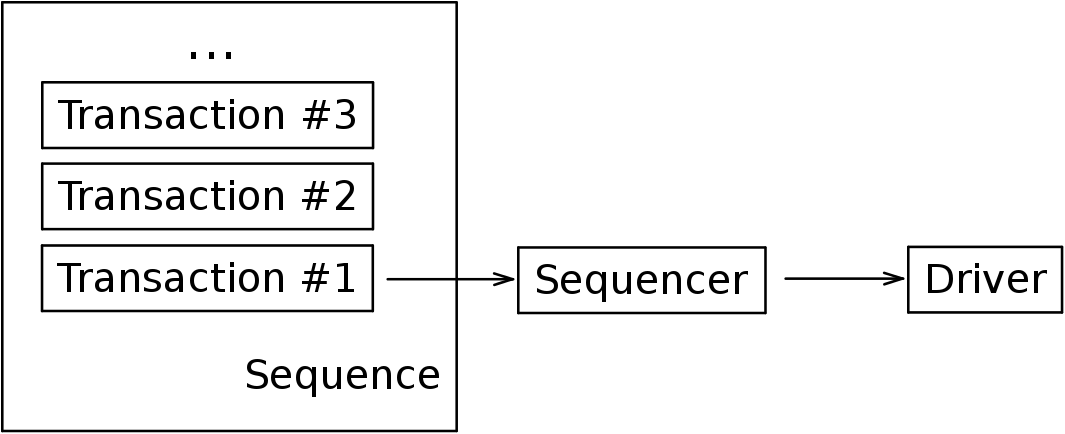


Figure : Relation between a sequence, a sequencer and a driver

### 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, the 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 executed)
* 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

In this section, the DUT architectural description, input parameters and communication protocol are discussed. In order to understand the DUT functionality, is important first to understand the K means algorithm, therefore this sections also includes a brief explanation about this algorithm.

## The K means algorithm

The K means algorithm is an iterative algorithm which divides a given data vector to K different clusters (K is a natural number). Each cluster will be characterized by its “center of mass”, what will be referred in this paper as centroid.

#### The algorithm steps

For a simpler explanation, it can be assumed that K is a constant predefined natural value. First, some symbols need to be defined:

-the cluster number "*i*" centroid

– the group of points in cluster number "*i*"

Upper index “*t*” – iteration or time

#### Initialization step

The first step in the algorithm is to randomly choose centroids for the K clusters. The “time” (“*t*”) for the initialization step will be defined as zero.

#### Classification step

In each iteration(time) of the algorithm, first each point of the input data is assigned to a cluster based on the “distance” from the point to the cluster’s centroid. A point will be assigned to cluster number “i” if the metrical distance between it and the cluster’s centroid is the minimum between the distances from the point to all others cluster’s centroids. To simplify:

\*In case of the distance from two different clusters is the same and is the minimum found, the chosen cluster is the one with the lowest index.

#### Centroids update step

After the classification step, the centroids of each cluster are updated to be mean of all points which belong to it in end of iteration(time) *t*. This is done by verifying if a cluster is empty (in this case the centroid is not changed) and then calculating the mean of all the clusters points:

#### Convergence check step

If the centroids of the next iteration calculated in the step above are close enough to the current centroids, then the algorithm comes to an end. Else, the iteration number(time) is increased by one and a new iteration begins with the assigning step.

##### Algorithm convergence

The k means algorithm assures convergence to a local minimum, i.e. the final centroids values are so that the variance within the clusters is minimized while the intra cluster’s variance is maximized. This minimum variance within the cluster is not always the global minimum that can be reached, the local minimum which was reached by the algorithm depends on the initialization step, specifically on the first values of the centroids.

#### Choosing K

Usually the optimal K is not known before the beginning of the algorithm. Therefore, an error parameter can be defined to help choosing K. The most commonly known error parameter is the clustering error which is defined by:

In this formula, the elements are:

As K increases, the error decreases. For example, if K is as the number of pints in the input vector, the error will be zero. This because its cluster will have just one point which will also be its centroid, but in this case no new information was added by the algorithm.

One suggested method of choosing a natural K so the clustering error is minimized is by gradually increasing K and calculating for each increasement. The process ends when the error reaches a value so that , where is a predefined threshold.

## Architectural description

The purpose of the DUT is to perform the K means algorithm(when K in case of the DUT is et permanently to 8) on the input data set, i.e. given eight initial centroids and data points(the number of data points can vary between 8 and 512), the DUT should calculate the value of the eight final centroids according to the K means algorithm.

The high-level architecture of the DUT is as shown in Figure 3. It is essentially composed of two main modules: the “Register file” and the “K means core”.

The “Register file” interfaces with the CPU host by APB protocol, as APB slave. Besides that, it also stores important data at local registers and interfaces with the second module “K means core”, allowing to read and write to its internal registers.

The “K means core” module is the actual “brain” of the architecture. It is responsible by running the algorithm and when it is done, it throws an interrupt to the CPU host, indicating the algorithm has come to an end.

The data set with which the algorithm is done is stored in a local RAM inside the “K means core module”. In order to do so, every data point is store one by one in this RAM by a process called “Indirect Access”.

The “Indirect Access” process is as its sounds: the CPU can write to the “K means core” local RAM only though a mediator, in this case, through the “Register File.

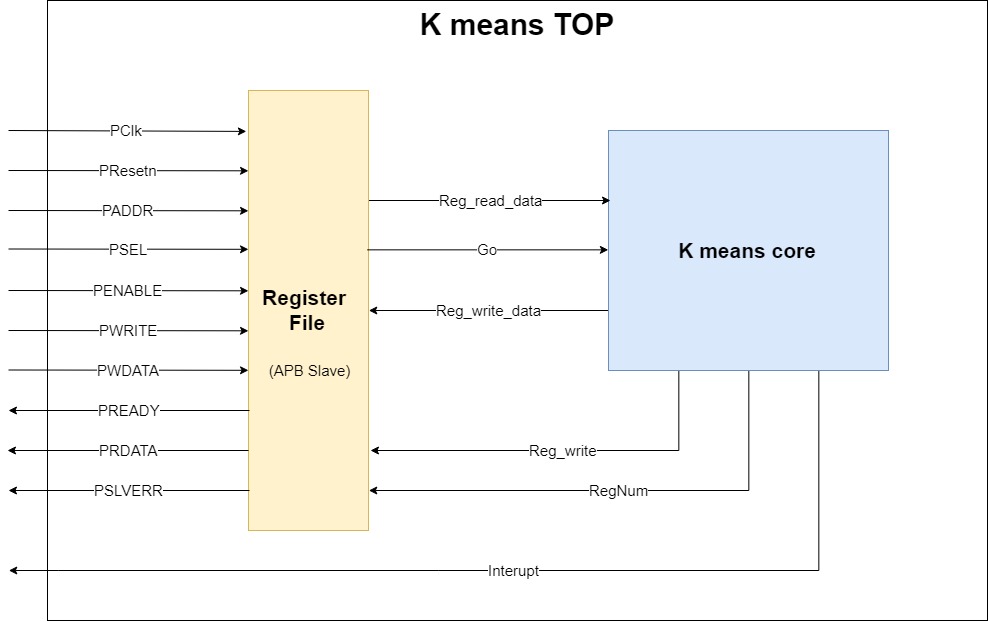


Figure : K means TOP block diagram

### K means core

The proposed architecture for the k means core block is as described in Figure 15.This block is responsible for running the k means algorithm. It receives the input data points from the register file block by indirect access, as well as the initial centroids. The block output (to the register fil block) is the final centroids value after the algorithm’s end and an interrupt indicating the calculation has been finished.

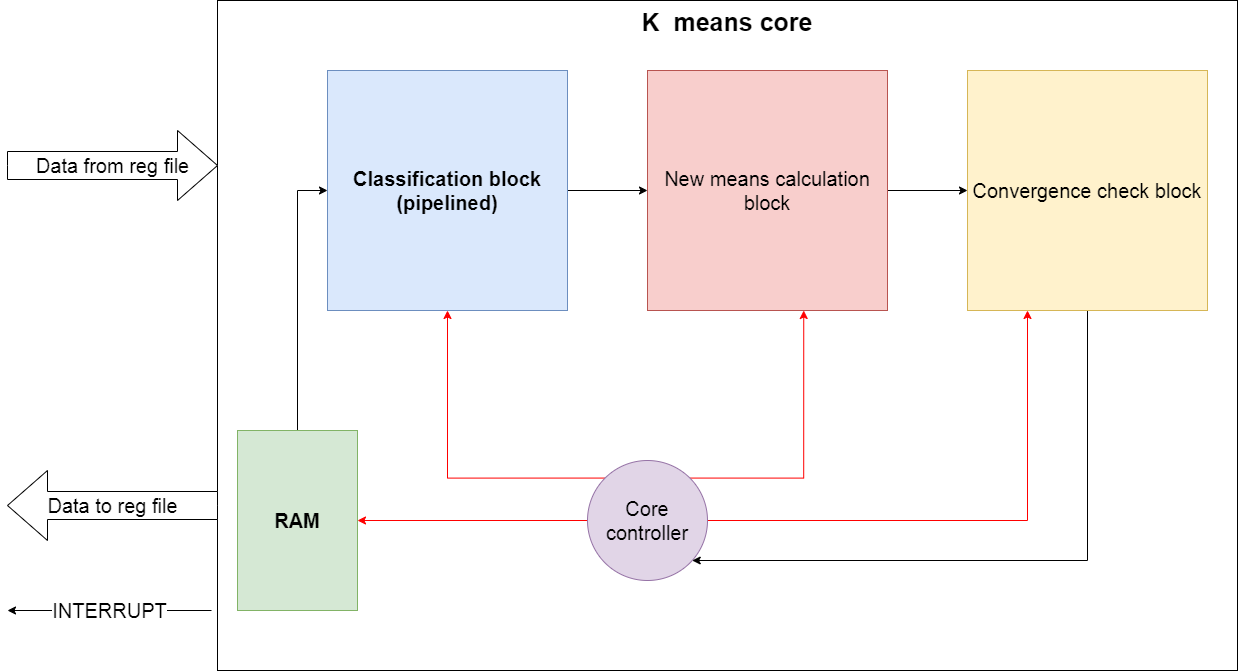


Figure 15:K means core top block diagram

The k means core block is composed of four main components:

* RAM – memory which is used to store the input data points.
* Classification block - this subblock is responsible for the classification step of the k means algorithm. It will start running only after all the input data points are stored in the RAM and the initial centroids are stored within local register of the block and it will run for each data point every iteration of the algorithm.
* New means calculation block – this block is responsible for the centroids update step of the algorithm. It will start running only after the classification block has finished classifying all input data points stored in the RAM. This block will run for as many times as the number of centroids, in the case of this IP, eight times.
* Convergence check block – after the “new means block” calculations are done, this block will be responsible for the “convergence check step” of the algorithm. In case convergence was achieved, it will inform the controller of so.
* Core controller - this is a state machine, which will control the k means core block by sending control signals to each of its internal blocks. It will receive form the register file the value of the registers and

## Input data characteristics

The every data in the DUT is a seven dimension point. Every data point coordinate or centroid coordinate in the DUT is represented by fixed point representation with 13 bits: 1(the MSB) to determine the sign of the number(in two’s complement convention),2 for the integer part of the number and 10 for the fractional part of the number. Therefore, every data coordinate could be in the dynamic range [-3.999,3.999]

The data points are stored in the DUT as matrix of 512X7, i.e. it has maximum 512 points with 7 coordinates each, so in order to represent the accumulator’s results, in the worst case where all data points enter the same accumulator, the accumulator maximum value per coordinate will be as high as:

In order to represent this value, 22 bits will be needed: 1(the MSB) to determine the sign of the number, 11 for integer part of the number and 10 for the fractional part of the number.

## DUT Parameters

This chapter is a user guide of the DUT, in which there are basic instructions of how to use the DUT as well as which parameters are mandatory and which are optional and which values they can receive

### Mandatory configurations

The following configurations are mandatory, i.e. in case the user chooses not to perform them, the IP functionality may not be correct.

### Reset

Before starting to use the DUT, it is needed to assert low the reset signal (PResetn, this is an active low signal) for at least half a clock cycle. This is needed also between two consecutive uses of the IP.

### RAM configuration

Before setting the “Go” register to 1, at least 8 data points need to be written to the IP’s RAM. The maximum Ram capacity is of 512 data points, therefore insertion of more than 512 data points may cause unexpected behavior.This is the reason that the input data set has to have between 8 and 512 data points.

### Mandatory registers configuration

Before setting the “Go” register to 1, the following registers need to be configured (not necessarily in this order):

* First ram addr – this register must be configured to the first ram address in which the user wrote data.
* Last ram addr - this register must be configured to the last ram address in which the user wrote data.

As mentioned before, the maximum Ram capacity is of 512 data points, therefore the parameter “First ram addr” should be between 1 and 512. The parameters “Last ram addr” shall therefore be set to the sum of the parameter “First ram addr” and the number of points chosen by the user.

### Go signal

After making the configurations described above, in order to instruct the DUT to start its function, the user must write the value ‘1’ to register named “Go\_reg”.

## Optional configurations

The following configurations are optional.

### Centroid registers configuration

The centroid initial values can be configure by writing these values(in the data form used by the DUT, i.e. fixed point number, MSB is sign bit, then 2 bits for integer part and 10 bits for fractional part) to registers “Cent\_X\_reg”(X is an integer between 1 and 8),before the “Go\_reg” is configure to ‘1’. In case these registers are not configure, all centroid initial values will be zero.

### Threshold register configuration

The threshold value used for convergence check of the algorithm may be configure by user. It can be configured by writing the desired threshold value (in the data form used by the DUT, i.e. fixed-point number, MSB is sign bit, then 2 bits for integer part and 10 bits for fractional part) to register “Thresh hold”. In case this register is not configure, the threshold value will be zero.

## Description of APB protocol

As mentioned before, the commination protocol used by the DUT is the APB protocol, which is explained in this section.

### Introduction

The Advanced Peripheral Bus (APB) is part of the Advanced Microprocessor Bus Architecture (AMBA) protocol family. This protocol is a single master multi slave and set guidelines for transactions between the master and its low-bandwidth peripherals, the slaves. The APB protocol signal transactions are only related to the rising edge of the clock and every transaction takes at least two cycles. It can be used to provide access to the programmable control registers of peripheral devices. Furthermore, the APB is a low-cost interface that is optimal for minimal power consumption.

The figure bellow (Key to timing diagram conventions) explains the timing diagrams in the following sections. Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Figure : Key to timing diagram of APB protocol

The signals which are part of APB protocol are listed and described in the table below:

|  |  |  |
| --- | --- | --- |
| Signal | Source | Description |
| PCLK | Clock source | Clock. The rising edge of PCLK times all transfers on the APB. |
| PRESETn | System bus equivalent | Reset. The APB reset signal is active LOW. This signal is normally connected  directly to the system bus reset signal. |
| PADDR | Master | Address. This is the APB address bus. It can be up to 32 bits wide and is driven  by the peripheral bus bridge unit. |
| PSELx | Master | Select. The APB bridge unit generates this signal to each peripheral bus slave.  It indicates that the slave device is selected and that a data transfer is required.  There is a PSELx signal for each slave. |
| PENABLE | Master | Enable. This signal indicates the second and subsequent cycles of an APB  transfer. |
| PWRITE | Master | Direction. This signal indicates an APB write access when HIGH and an APB  read access when LOW. |
| PWDATA | Master | Write data. This bus is driven by the peripheral bus bridge unit during write  cycles when PWRITE is HIGH. This bus can be up to 32 bits wide. |
| PREADY | Slave | Ready. The slave uses this signal to extend an APB transfer. |
| PRDATA | Slave | Read Data. The selected slave drives this bus during read cycles when  PWRITE is LOW. This bus can be up to 32-bits wide. |
| PSLVERR | Slave | This signal indicates a transfer failure. APB peripherals are not required to  support the PSLVERR pin. This is true for both existing and new APB  peripheral designs. Where a peripheral does not include this pin then the  appropriate input to the APB bridge is tied LOW. |

Table : APB signal description

The PADDR, PWRITE, PWDATA signals are common among all the slaves, however there are as many PSEL signals as slaves, and for each slave one PRDATA from it to the master. The following shows the block diagram between master and slave of APB:



Figure : APB block diagram

#### Operating states

The figure bellow describes the operating states of the protocol:



Figure : APB operating states

The state machine operates through the following states:

**IDLE** - This is the default state of the APB.

**SETUP** - When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, PSELx, is asserted. The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.

**ACCESS** - The enable signal, PENABLE, is asserted in the ACCESS state. The

address, write, select, and write data signals must remain stable during

the transition from the SETUP to ACCESS state. Exit from the ACCESS state is controlled by the PREADY signal from the slave:

• If PREADY is held LOW by the slave then the peripheral bus remains in the ACCESS state.

• If PREADY is driven HIGH by the slave then the ACCESS state is exited and the bus returns to the IDLE state if no more transfers are required. Alternatively, the bus moves directly to the SETUP state if another transfer follows.

### Transfers

Each transfer consists of two cycles: one for the SETUP state and another for the ACCESS state. There are three types of transfers: write transfers, read transfers and error response transfers. In addition, write and read transfers can be with or without wait states, that are SETUP states which follow an ACCESS state instead of going to IDLE STATE.

#### Write Transfers

##### Write Transfers without wait states

A write transfer without wait states consist of two clock cycles: in the first (the SETUP STATE) the signals: address (PADDR), write data (PWDATA), write (PWRITE) and select (PSEL) are asserted. PADDR is asserted to the desired address where the data is supposed to be written, PWDATA is asserted to the desired data to be written, PWRITE is asserted HIGH and PSEL is asserted HIGH only for the specific slave which the write command is for, the rest of the PSEL lines are driven LOW. These signals remain unchanged through the second cycle.

In the second cycle (the ACCESS state) the slave sets the enable signal (PENABLE) HIGH. The ready signal (PREADY) is set HIGH by the slave in order the informed the master that the slave is ready to receive the data, which is latched by the slave in the rising edge ending the second clock cycle. After this last clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) meaning that the transfer is over.

In Figure 7 there is an example of write transaction with no wait states can be seen, with the first cycle of the transfer being from T1 to T2 and the second cycle from T2 to T3.



Figure : APB write transfer with no waits

##### Write Transfers with wait states

The first cycle of the transfers is the as the transfers without wait states. During the ACCESS state, when PENABLE is HIGH, the transfer can be extended by driving the PREADY LOW. The signals PADDR, PWRITE, PSEL, PENABLE and PDATA remain unchanged from the end of the first cycle (SETUP state) until the data is latched by the slave, which occurs at the first rising clock edge after the slave sets the PREADY signal HIGH. After this clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master meaning that the transfer is over (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) .

In Figure 7: APB write transfer with no waits Figure 8 an example of write transaction with wait states can be seen, with the first cycle of the transfer being from T1 to T2,two wait states occur from T2 until T4 and the last cycle of the transfer from T4 to T5 , in which the slave sets the PREADY signal HIGH and at the end of this cycle the data is latched by the slave.



Figure : APB write transfer with wait states.

#### Read Transfers

##### Read Transfers without wait states

A read transfer without wait states consist of two clock cycles: in the first (the SETUP STATE) the signals: address (PADDR), write (PWRITE) and select (PSEL) are asserted. PADDR is asserted to the desired address where the data is supposed to be read, PWRITE is asserted LOW and PSEL is asserted HIGH only for the specific slave which the write command is for, the rest of the PSEL lines are driven LOW. These signals remain unchanged through the second cycle.

In the second cycle (the ACCESS state) the slave sets the enable signal (PENABLE) HIGH. The PRDATA signal is set by the slave according to the data in stored in the desired address(the address which is set in PADDR signal) and the ready signal (PREADY) is set HIGH by the slave in order the informed the master that the slave is ready to send the data. The data in PRDATA signal is latched by the master in the rising edge ending the second clock cycle. After this last clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) meaning that the transfer is over.

In Figure 7: APB write transfer with no waitsFigure 9 an example of write transaction with no wait states can be seen, with the first cycle of the transfer being from T1 to T2 and the second cycle from T2 to T3.



Figure : APB read transfers with no wait states

##### Read Transfers with wait states

The first cycle of the transfers is the as the first cycle of transfer without wait states. During the ACCESS state, when PENABLE is HIGH, the transfer can be extended by driving the PREADY LOW. The signals PADDR, PWRITE, PSEL and PENABLE remain unchanged from the end of the first cycle (SETUP state) until the data is latched by the master, which occurs at the first rising clock edge after the slave sets the PREADY signal HIGH. After this clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master meaning that the transfer is over (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) .

In Figure 10 an example of read transaction with wait states can be seen, with the first cycle of the transfer being from T1 to T2,two wait states occur from T2 until T4 and the last cycle of the transfer from T4 to T5 , in which the slave sets the PREADY signal HIGH and at the end of this cycle the data is latched by the master.



Figure :APB read transfer with wait states.

#### Error response

Some APB peripheral offer a way of indicating that an error occurred during a transfer with the PSLVERR signal. Errors can occur both in read and write transfers, and the signal PSLVERR is only considered valid during the last cycle of an APB transfer, when PSEL, PENABLE, and PREADY are all HIGH.

It is recommended, but not mandatory, that you drive PSLVERR LOW when it is not

being sampled. That is, when any of PSEL, PENABLE, or PREADY are LOW.

Transactions that receive an error, might or might not have changed the state of the

slave. This is peripheral-specific, and either is acceptable.

When a write transaction receives an error, this does not mean that the register within the slave has not been updated. Read transactions that receive an error can return invalid data.

There is no requirement for the slave to drive the data bus to all 0s for a read error.

APB slaves are not required to support the PSLVERR pin. This is true for both

existing and new APB peripheral designs. Where a slave does not include this pin

then the appropriate input to the master is tied LOW.

##### Error response in a write transfer

When there is an error in a write transfer and the slave in the transfer has an active PSLVERR signal, during the last cycle of the transfer (when PSEL, PENABLE, and PREADY are all HIGH) PSLVERR is driven HIGH, informing the master about the error in the transaction. These can be seen in Figure 11:



Figure : APB error in write transfer

##### Error response in a write transfer

When there is an error in a read transfer and the slave in the transfer has an active PSLVERR signal, during the last cycle of the transfer (when PSEL, PENABLE, and PREADY are all HIGH) PSLVERR is driven HIGH, informing the master about the error in the transaction. These can be seen in Figure 12:



Figure : APB read in write transfer

# Implemented Verification environment

This section contains a detailed explanation of the verification environment.

## UVM used classes

The overall UVM structure of the implemented verification environment is as described in section ‎2.3. In this section only the classes specifically change to fit this specific project.

### Transaction

The transaction used by the verification environment was named *Kmeans\_transaction*. It is child class of the UVM built in class *uvm\_sequence\_item*  and has the following variables and constrains:

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Type | Constrain | Purpose |
| Centroids | Rand logic [8][91] |  | Used to randomly generate initial centroid values for DUT and ref model |
| Num\_points | Rand int | 8<num\_points<512 | Used to randomly generate the number of points used as input points on the DUT and on the ref Model |
| Data\_points | rand logic [512][91] |  | Used to randomly generate data points values for DUT and ref model |
| Threshold | rand logic [12:0] | threshold[12:8] == 5'd0  (so the threshold is small) | Used to randomly generate threshold value for DUT and ref model |
| first\_point\_index | rand logic [13] | first\_point\_index <= 512 - num\_points;  first\_point\_index >=1; | Used to randomly generate the RAM index where the first data point will be stored |
| last\_point\_index | Rand logic [13] | last\_point\_index == num\_points + first\_point\_index - 13'b1 | Used to randomly generate the RAM index where the last data point will be stored |

### Sequence

The sequence class used by the verification environment was named *Kmeans\_in\_sequence*. It is child class of the UVM built in class *uvm\_sequence .*In this class, there is a variable called *num\_txs*  which is set to be the number of transactions the Testbench will produce and send,i.e. the number of actual tests performed.

Also in this class, in a loop which runs *num\_txs*  times, a *Kmeans\_trasaction* is instantiated and the build in function *randomize* is called in order to generate all variables explained in the previous section(section ‎4.1.1).Also in this loop, the build in function *start\_item* is called with the previous instantiated *Kmeans\_trasaction* as parameter, in order to “send" it to the driver through the sequencer. After the driver finishes using the transaction sent to it, the sequence class than calls the build in function *finish\_item* with the same *Kmeans\_trasaction* as parameter, in order to end this transaction.

### Driver

The Driver class used in the environment was named *Kmeans\_driver* and it is child class of the UVM built in class *uvm\_driver*. As the driver is responsible to send the transaction(*Kmeans\_trasaction*) to the DIT and the Ref model, it implements the following tasks:

* *send\_APB\_transaction* – This task receives three parameters: *address, write* and *data*. According to this parameters, it toggles the DUT virtual interface signals(APB signals) according to section ‎3.5.
* *k\_means\_calculation* – This task receives a *kmeans\_transaction*. It then configures the DUT centroids registers by using the task *send\_APB\_transaction* eight times*,* each time using one centroid from the *kmeans\_transaction*  as data for the *send\_APB\_transaction task.* Then this task configures the *first\_ram\_addr\_reg* and *last\_ram\_addr\_reg* of the DUT by using the *send\_APB\_transaction* task where the *data* parameter is the variable *first\_point\_index* and *last\_point\_index*  accordingly.

Next, this task write all data points stored in the *kmeans\_transaction* variable *data\_points* from *first\_point\_index* to *last\_point\_index* to the DUT RAM, again, using the *send\_APB\_transaction.*

Finally, the task will send a GO signal to the DUT (in order to start the DUT calculations) by using the *send\_APB\_transaction*, where the parameter *data*  is ‘1’ and the *address* is the address of the “Go register”(GO\_reg)

* *k\_means\_ref\_calculation* - This task receives a *kmeans\_transaction*. In this task the *kmeans\_transaction* fields *first\_point\_index*, *last\_point\_index* and *threshold* are directed inserted into the Ref Model virtual interface(named *vrefif*). Then the every data point in the *kmeans\_transaction* field *data\_points* is inserted into the Ref Model virtual interface field *matrix* in a loop, duo to the fact that these fields structure is different (*data\_points* is logic [512][91] and *matrix* is logic [12:0] [0:3583]). Similarly, the *kmeans\_transaction* field *centroids* is inserted into the Ref Model virtual interface field *in\_centroids.*

### Scoreboard

The Scoreboard implemented class contains:

* Two *uvm\_analisys\_export* (one for the DUT centroids and one for the Ref Model centroids)
* Two *uvm\_tlm\_analisys\_fifo* (one for the DUT centroids and one for the Ref Model centroids)
* Virtual functional named *compare centroids*.

This class uses the uvm\_tlm\_analisys\_fifo and uvm\_analisys\_export in order to get the results from the DUT and Ref Model (the eight centroids of each of them).

In is *Run* task, the Scoreboard class calls the function *compare\_centroids* in order to determine if a test run failed or passed. This function determines the result by comparing the centroids od the DUT and the Ref Model, if the overall difference between all coordinates off all centroids is smaller than 16 times the value of the threshold. The pass/fail condition was derived from the functionality of the DUT and Ref Model, which consider a centroid converged if the absolute value of the distance between it and the last iteration centroid is smaller then the threshold value.

## Ref Model

The Reference Model used to check the DUT results was written using Matlab. It is Matlab function named RefModel.m. This functions tries to imitate the DUT functionality, performing the K Means algorithm(explained in section ‎3.1) in software level instead of hardware level.

The RefModel.m function receives five input parameters:

* Point input matrix with 512 rows and 7 columns, where each row represent a point in the DUT numeric representation model, i.e. each row is a point with 7 dimentions, each dimension is a fixed point number with 13 bits(MSB is the sign bit, the following two bits represent the integer value and the rest ten bits represent the fractional part)
* Initial centroid matrix with 8 rows and 7 columns, where each row represents an initial centroid value in the DUT numeric representation model.
* Input threshold value in the DUT numeric representation model
* First point index
* Last point index

These parameters are used in the following way: the Reference model function uses the Point input matrix as the DUT uses its RAM, its read the points values from “First point index” until “Last point index” into another matrix, named point matrix , which will be used to run the algotirhm.

Then the function interactively does the algorithm steps using the point matrix, the “input centroid matrix” and the “input threshold”. In each iteration the following steps are followed:

* An accumulator 8x7 matrix is initiated to zero, where each cell in this matrix has the following format: 22 bits, MSB being the sign bit, 11 bits for the integer part and 10 bits for the fractional part.
* An accumulator counter 8x1 matrix is initiated to zero. Each row I n the accumulator counter represents a counter for the same row index accumulator in the accumulator matrix.
* The function calculates for every point in the point matrix the distance between it and all centroids in the “Input centroid matrix”.
* The minimum distance and closets centroid for each point in the point matrix are found using Matlab’s built in function *min*.
* Every point in the point matrix is then added to the accumulator corresponding its closets centroid and corresponding accumulator counter rises accordingly.
* A convergence counter is initiated to zero.
* New centroids values are calculated by dividing each accumulator by its counter.
* The Manhattan distance, which is the sum of the absolute distance of each coordinate , of each new centroid and “old centroid”(the centroid used in the beginning of the iteration to classify the point matrix) is calculated and compared to the “input threshold value”.

If this distance is smaller than the “input threshold value”, then one its added to the convergence counter.

* The convergence counter is compared to 8: if it is 8 then the function end, else it begins another iteration using as initial centroids the new calculated centroids.

The following figure summarizes the Reference Model in a flow chart:

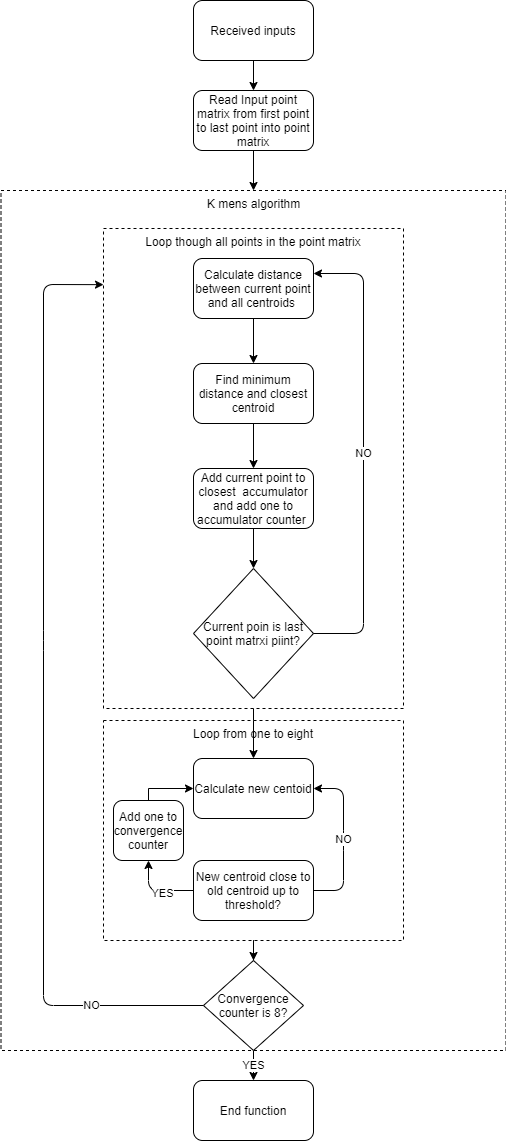


Figure :Reference Model flow chart

# Test Plan

In this section, the test plan for the verification of the DUT is explained. Each Test Line in this section was build and run. These test lines purpose is to test the main purpose of the DUT, not the communications protocol.

Therefore, in order to verify the functionality of the communication with the DUT, an early test was done in which all registers of the DUT Reg File were written to and read from. This test was successful, leading to the conclusion that the communication protocol with the DUT(as described in section ‎3.4.2) works correctly.

For each test, the pass/fail criteria is as described in the Scoreboard class paragraph, section ‎4.1.4.

## Test Line 1 – Gradual Random Points

In this test line, the following parameter will be set:

1. Centroids one to eight will be set to values 1 to 8(respectively).
2. Threshold value will be one (only threshold LSB will be one).
3. There will be ten data points.

This test will be run overall ten times, where in each run the only parameters which will be changed are the input data points

At first, eight of the ten data points will be set to have values 11 to 18 in their left most coordinate. The ninth data point will be set to have the value 7 in the third coordinate third from right) and the last data point will be randomly generated.

Then the second run of the test will set not only the last data point randomly, but also the ninth data point.

As the test continues, in each run there will be one more data point which will be random, meaning, in the third run the test will set three data points randomly, in the fourth run there will be four random generated data points and so on until the tenth run, in which all data points will be randomly generated.

These 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.

## Test Line 2 – 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 3 – Random Points and Centroids

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

1. Eight Points values
2. Eighth 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 4 – Random Constrained Number of Points

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

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

This test will be run overall ten times, where in each run the constraint over the *Number of Points* parameters will change.

Where at first the *Number of Points* parameter will be constrained to be between 8 and 58.In the second run this parameter constrain will change so that it can receive every integer between 8 and 108, and so on until in the tenth run the *Number of Points* parameter will be constrained to be between 8 and 512

At each run, 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.

## Test Line 5 – 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 be 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.

## Test Line 6 – 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 receive 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 7 – 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 8 – 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 9– Fully Random 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 10 – 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 11 – 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 12 – 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.

# Results

## Test Line 1 – Gradual Random Points

TBD

## Test Line 2 – One Iteration Test

TBD

## Test Line 3 – Random Points and Centroids

TBD

## Test Line 4 – Random Constrained Number of Points

TBD

## Test Line 5 – Equal Initial Values Test

TBD

## Test Line 6 – Positive Overflow Test

TBD

## Test Line 7 – Negative Overflow Test

TBD

## Test Line 8 – Full Memory Test

TBD

## Test Line 9– Fully Random Test

TBD

## Test Line 10 – Isolated Centroid Test

TBD

## Test Line 11 – Robustness Test

TBD

## Test Line 12 – Threshold Test

TBD

# Bug Fixes

In this section, bugs found and fixed during the verification processed are presented.

## 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 fixed 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 (as explained in section ‎3.3)

* + 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 files "accumulator\_adder.sv" and “convergence\_check\_block.sv” were 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)( as explained in section ‎3.3).
     2. 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.
     3. 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.
     4. The file "distance\_calc.sv" changed, as explained above.

## Combinatorical sensitivity list missing item:

In convergence\_check\_block, as explained in chapter ‎3.2, the new calculated centroids of each iteration are check for convergence by comparing them to the last iteration centroids values(one by one), in the case convergence was not reached, this new centroids are used in the next iteration as the algorithm centroids.

In the convergence check module, there is a sensitivity list used to take one old centroid from the 8 and compare it to the correspondingly new centroid which come as input from prior module (new means calculation block).

This sensitivity list was not covering the change values of relevant inputs for this supposedly combinatorial representation of always, the following fix has been done to fix it:

Before: "always @(cent\_num) begin".

After: "always @\* begin".

This solves the bug.

## Fix of controller wrong signal toggle during state machine changes, at 2nd state of empty\_pipe:

The controller has signal for enabling the accumulators of pipe3 of classification block,the signal is in charge for determining when to sample data point, which comes as input from RAM to classification block.

The read of data points is split for filling the pipe, reading points, and empty the pipe, during the 2nd state of empty pipe(there are 2 separate states following), there is a need to pull down the enable so at next state, which is "calculate new means", there would be no sampling of any more data points.

There was a bug which we sampled one more data point since we pulled down the signal only at the "calculate new means" state, therefore, pulling it down one state/cycle before, remove the bug.

A code representation of the bug:

Before, empty\_pipe2 state consist the following code line: " accumulators\_en\_r <= 1".

After, the line was replaced by: " accumulators\_en\_r <= 0".

# Coverage Results

Coverage was made for the test line called Robustness(for more see section ‎5.11), because it is the most “random” test line(every input for the DUT is randomly generated) and is has the biggest number of tests.

The results for the coverage are show in this section.

## Code Coverage

TBD

## Conditional Coverage

TBD

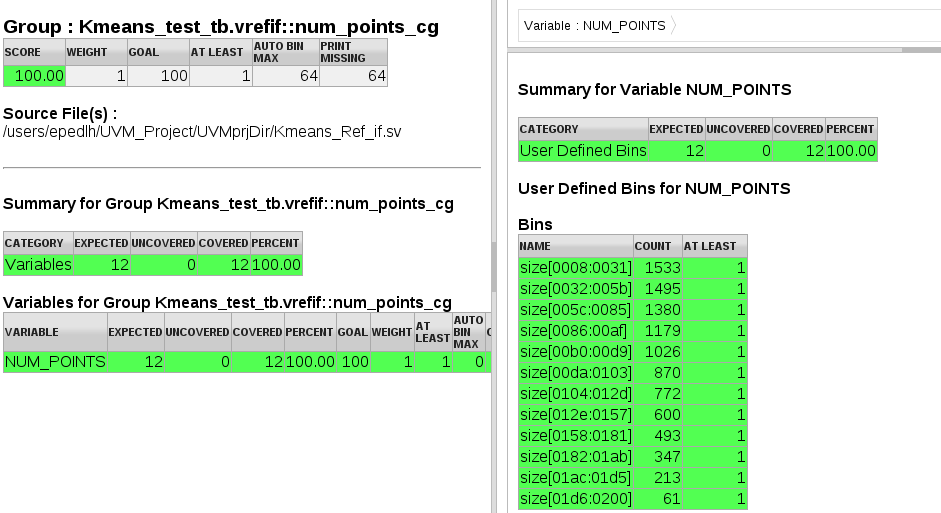
## Functional Coverage

In order to check the functional coverage of the test line, the following cover groups were made:

* NUM\_POINTS - This covergroup samples the number of data points randomly generated at each test, checking to see that all value for this variable is uniformly distributed between 8 and 512.
* DATA\_VALUE - This covergroup samples the value of each one of the seven coordinates of all data points randomly generated at each test, checking to see that all values for this variable are uniformly distributed between all the possible values.
* CENTX\_VALUE - This covergroup samples the value of each one of the seven coordinates of centroid X, randomly generated at each test, checking to see that all values for this variable are uniformly distributed between all the possible values.

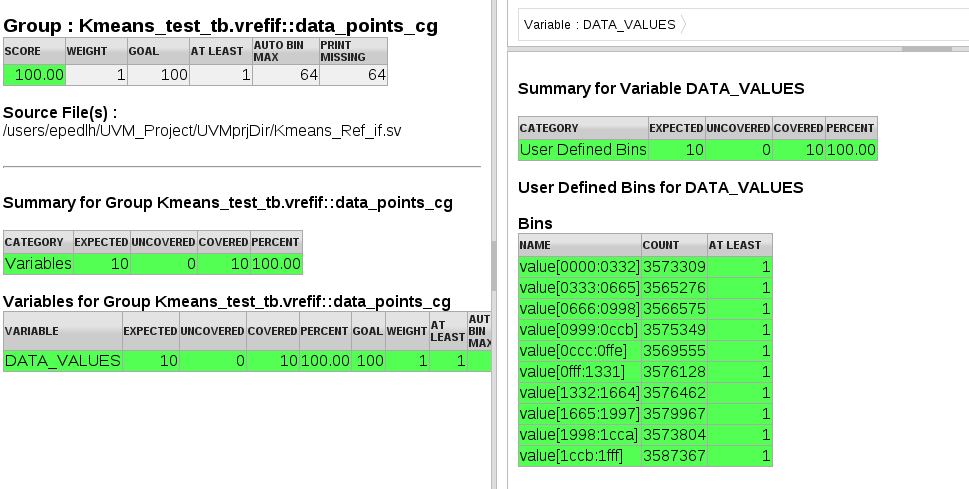
### NUM\_POINTS

As it can be seen form the figures bellow, the number of points in all the tests was uniformly distributed between 8 and 512.



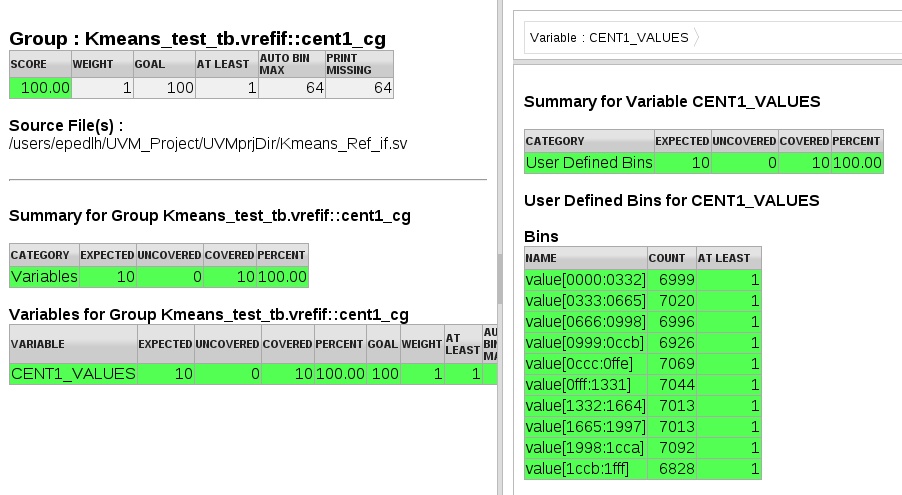
### DATA\_VALUE

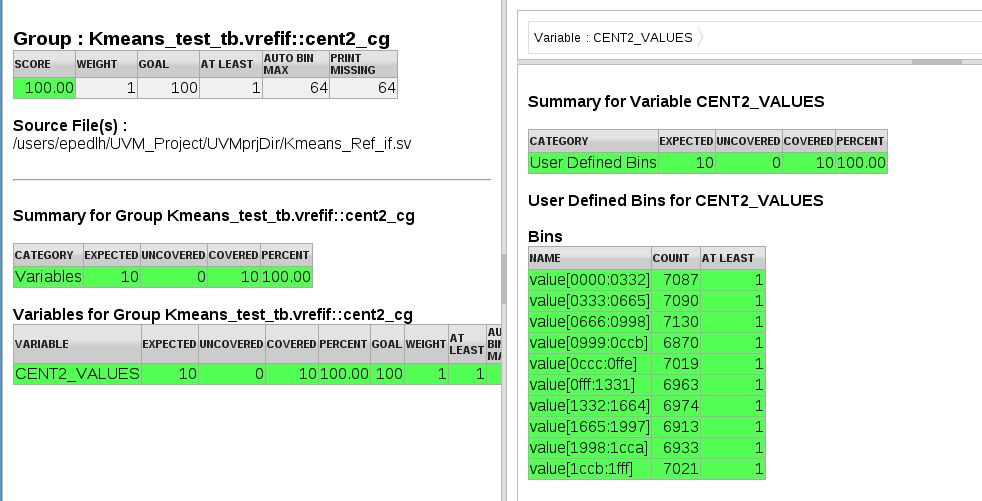
As it can be seen form the figures bellow, for all data points, the values of their coordinates were normally distributed between all the possible values in all the tests.

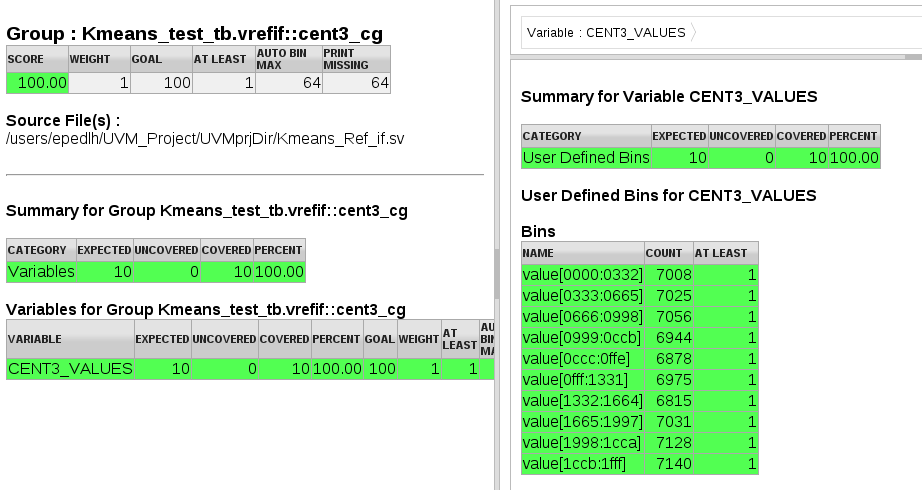


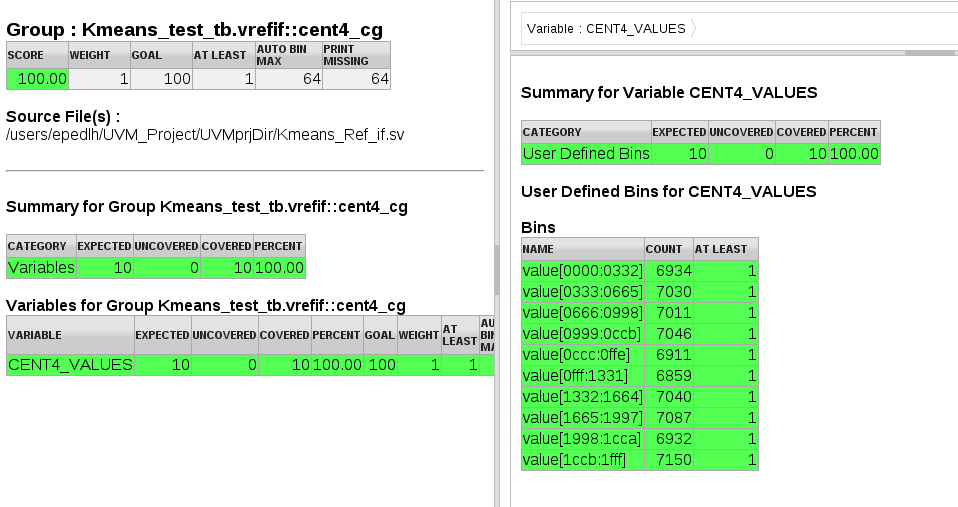
### CENT\_VALUE

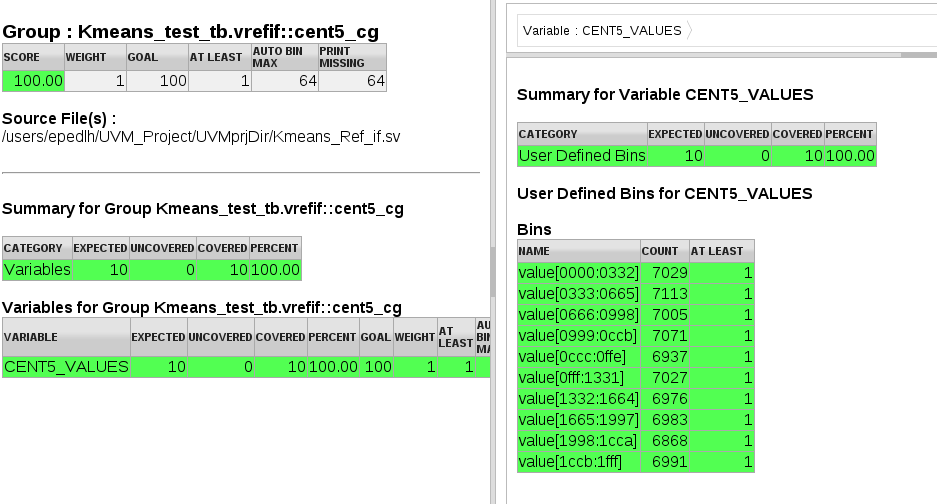
As it can be seen form the figures bellow, for each centroid the values of its coordinates were normally distributed between all the possible values in all the tests.

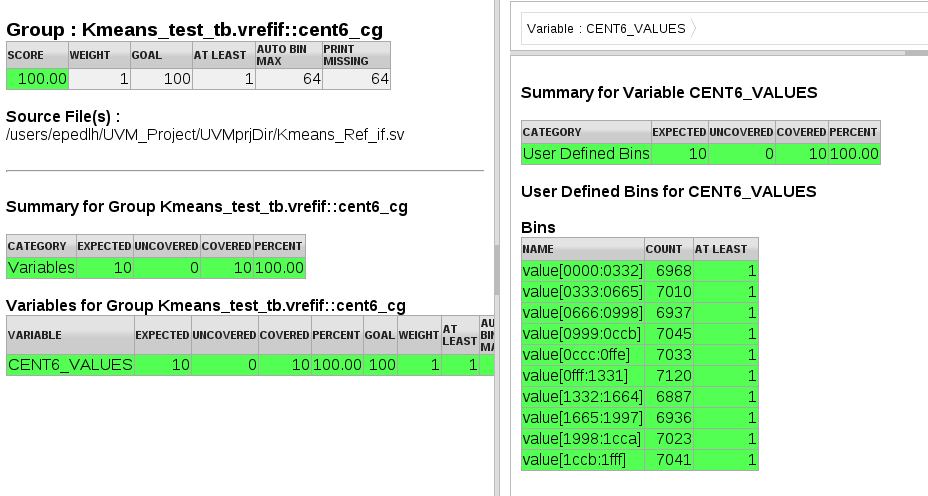


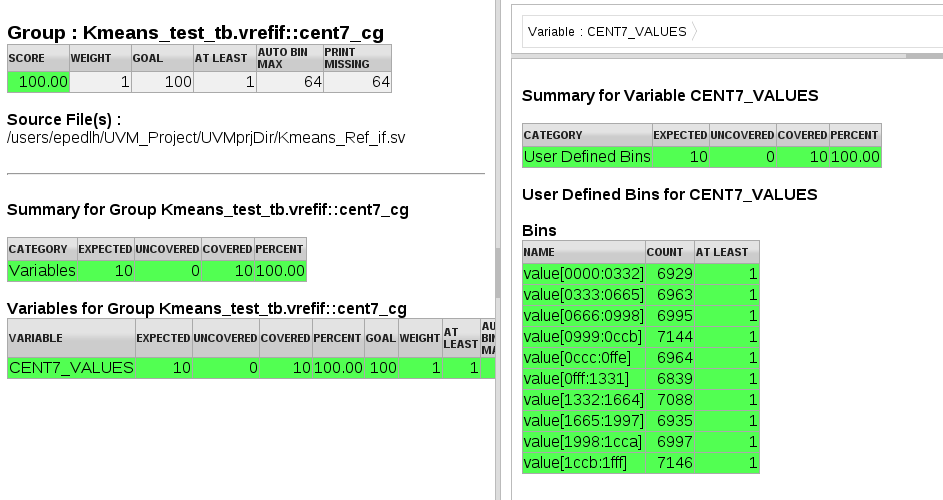


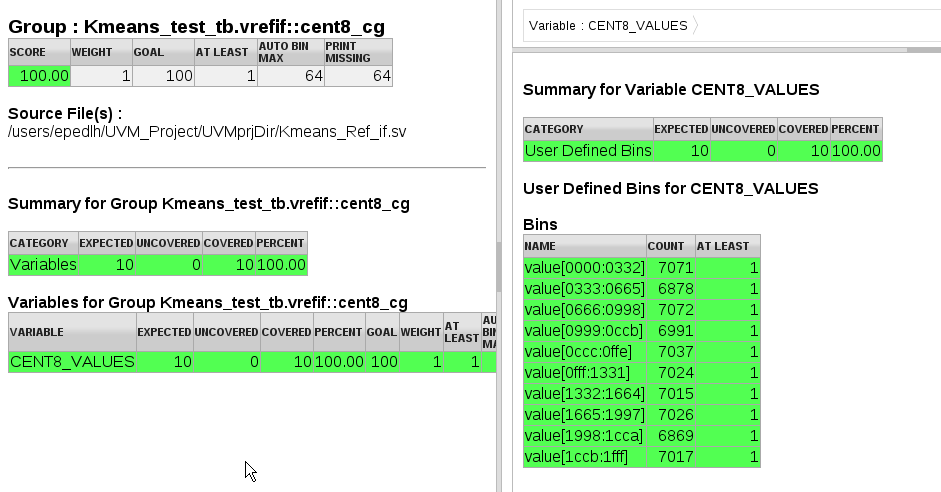






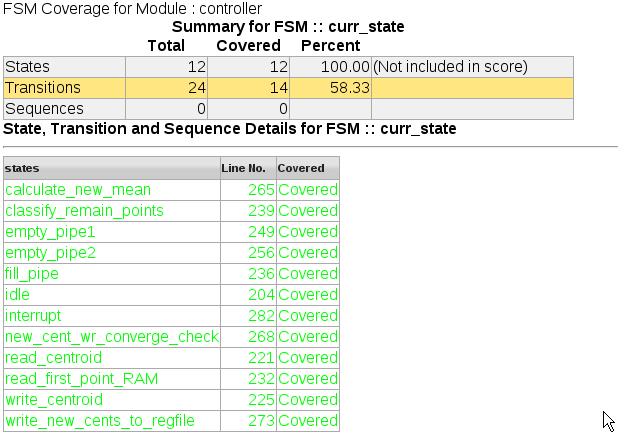






## FSM Coverage

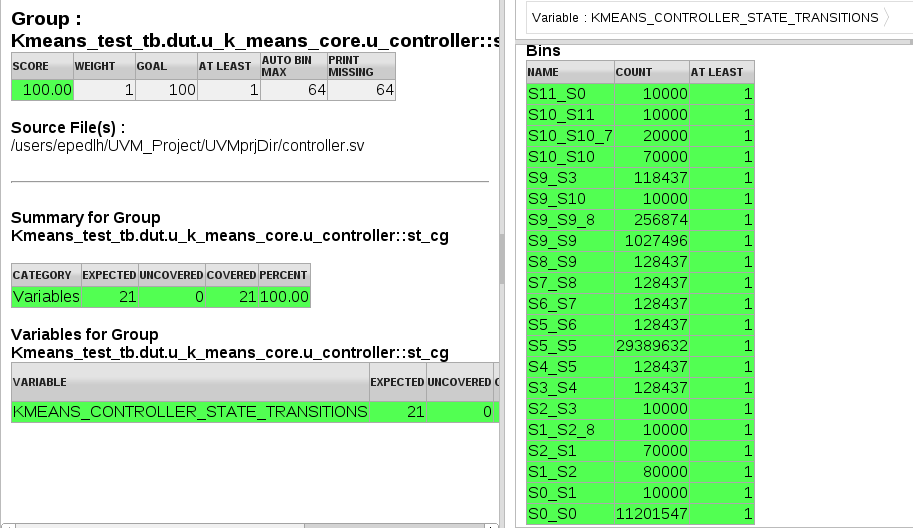
The build in FSM coverage has a low result duo to the fact that it check all transitions for the FSM, even the one which are not possible. Therefore, a covergroup for the legal transitions was build. This covergroup and its results are presented in this section



### Kmeans controller FSM

The covergroup for the valid transitions was written according to the transitions and control signals in the following figure:





# 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 testbench 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*