Development and Enhancement of Verification Environment for Complex Designs

by

Raghav Madan

Under the Supervision of
Dr. Sujay Deb
Mr. Nishant Kumar

Indraprastha Institute of Information Technology Delhi
June, 2015

Development and Enhancement of Verification Environment for Complex Designs

by

Raghav Madan

Under the Supervision of
Dr. Sujay Deb
Mr. Nishant Kumar

Submitted
in partial fulfillment of the requirements for the degree of
Master of Technology in Electronics & Communication
Engineering with specialization in VLSI & Embedded Systems

to

Indraprastha Institute of Information Technology Delhi
June, 2015

Certificate

This is to certify that the thesis titled **Development and Enhancement of Verification Environment for Complex Designs** being submitted by RAGHAV MADAN to the Indraprastha Institute of Information Technology Delhi, for the award of the Master of Technology, is an original research work carried out by him under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

June, 2015

Dr. Sujay Deb
Department of Electronic & Communication Engineering
Indraprastha Institute of Information Technology Delhi
New Delhi 110 020

Mr. Nishant Kumar
ST Microelectronics
Greater Noida

Dedicated

To My Mom, Dad and Brother

Abstract

Digital VLSI design verification is a process of checking and verifying the correctness of design functionality with respect to the documented specifications. Functional verification of a complex IP or sub-system, formed by integrating the IPs, accounts for more than half of the time and efforts required for product development. Development of testbench environment for complex designs requires a strategic planning and lot of human efforts. Therefore, in order to dwindle human involvement and efforts, a predefined verification methodology can be followed which provides a structure for generic implementation and development of testbench environment by adopting a standard hardware verification language. Universal Verification Methodology is a recent one that provides a structured implementation of testbench for simulation based verification and also enables reusability of verification components by virtue of SystemVerilog language. A preponderance of testbench development is devoted to self-checking mechanism. Therefore, verification environment development time and engineer$'$s efforts can further be abated with the availability of well-defined and exhaustive self-checking mechanism.

In this dissertation, the development of UVM based verification environment for complex designs along with implementation and performance analysis of different kind of self-checking mechanisms has been presented. This will assist the verification engineers to quickly develop the verification environment with selection of best possible self-checking mechanism as per the design type and complexity. This will also reduce their efforts for checking the functionality of design, checking bus protocol communication with DUV, searching for hidden bugs in the design with the aim of reaching higher coverage goals by taking memory and timing requirement into consideration.

Acknowledgements

I am very grateful to the Almighty for his mercy and blessings. I would like to dedicate my first and foremost token of gratitude to my advisor, Dr. Sujay Deb, for providing me the opportunity to continue my research work in a reputed industrial organisation and giving his invaluable time and guidance. He provided unwavering and immeasurable support throughout the course of my work.

I am extremely grateful to people at STMicroelectronics, Greater Noida, India for permitting me to work in their premises. My colleagues there, have been a great source of encouragement and help. My special thanks are due to Mr. Maninder Singh, Manager, for his continuous support and cooperation. Further, I would like to express my sincere gratitue to Mr. Nishant Kumar for mentoring and always giving me challenging tasks which pushes me to perform to the best of my abilities. This work would not have taken its current shape without his useful comments, critical remarks and constant feedback.

My deepest gratitude and thanks to Prof. R.N. Biswas for giving me the great opportunity of pursuing internship in a prestigeous organisation, STMicroelectronics, in design verification domain.

I express my thanks to all the faculty members and the staff of the Department of Electronics and Communication Engineering for their support.

The unconditional love and encouragement provided by my parents served as a secure anchor during the hard and easy times; thank you. Finally, I would like to thank my friends for being so encouraging and helpful throughout the course of this research.

List of Abbreviations

UVM	Universal Verification Methodology
TLM	Transaction-Level Modeling
VLSI	Very Large Scale Integrated Circuit
IP	Intellectual Property
SBV	Simulation Based Verification
EDA	Electronic Design Automation
SVA	System Verilog Assertion
DUT	Design Under Test
eRM	e Reuse Methodology
AVM	Advance Verification Methodology
URM	Universal Reuse Methodology
OVM	Open Verification Methodology
VIP	Verification Intellectual Property
RVM	Reuse Verification Methodology
VMM	Verification Methodology Manual
UVC	Universal Verification Component
CDV	Coverage Driven Verification
CBV	Cycle Based Verilog
OVA	Open Vera Assertion
OVL	Open Verification Library
VHDL	Very High Speed Integrated Circuit Hardware Description Language
PSL	Property Specification Language
RTL	Register Transfer Language
DUV	Design Under Verification
I/O	Input/Output
HDL	Hardware Description Language
DDR	Double Data Rate

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Verification of Complex Designs

Verification serves as an integral part of any product development cycle. In terms of verification, the complexity of design increases when new functionalities are introduced which adds new modules to the design and increase overall number of interface I/O pins. If a design representing some complex functionality can plug into bigger system in that case it can belong to a category of complex intellectual property (IP). Verification of complex IP is a tedious and time consuming task. The complexity and type of design determines the kind of functional verification that can be used for the particular design. Functional verification of designs can be done either using automated formal tools or by the process of simulations. It′s a general practice to use simulation based verification (SBV) for complex designs as formal tools are incapable of verifying highly complex systems.

Among the various modern SBV approaches, Universal Verification Methodology (UVM) is a standard and most recent one that provides a well established and flexible solution for development of verification environment for complex system verification. Flexibility of UVM lies in the fact that the verification environment developed using it, consists of reusable components and is supported by tools of all major EDA vendors of the industry. UVM provides a complete framework to achieve coverage driven verification that includes automatic test generation, self-checking testbenches and coverage metrics (1). It helps in reducing the complexity and time to develop the verification environment for any complex design. Assertions further enhance the capability of veri-

fication environment by improving debugging process, protocol checking and capturing specific intents of the design. They can also be used as a functional coverage metrics which can help in improving the overall coverage of the design verification. SystemVerilog introduces modern and useful constructs required for the verification and design of any digital system. Moreover, UVM is also based on SystemVerilog, hence, SVA can be used in UVM based testbenches.

1.2 Motivation

The issues related to functional verification of complex designs, which accounts for more than half of the product development cycle, can be taken into consideration by adopting best practices for architecting verification environment that will consume less time and resources. According to latest trend in design verification as shown in figures 1.1 and 1.2, it has been observed that the major issue for both the design and verification engineer is "How to architect the testbench?"

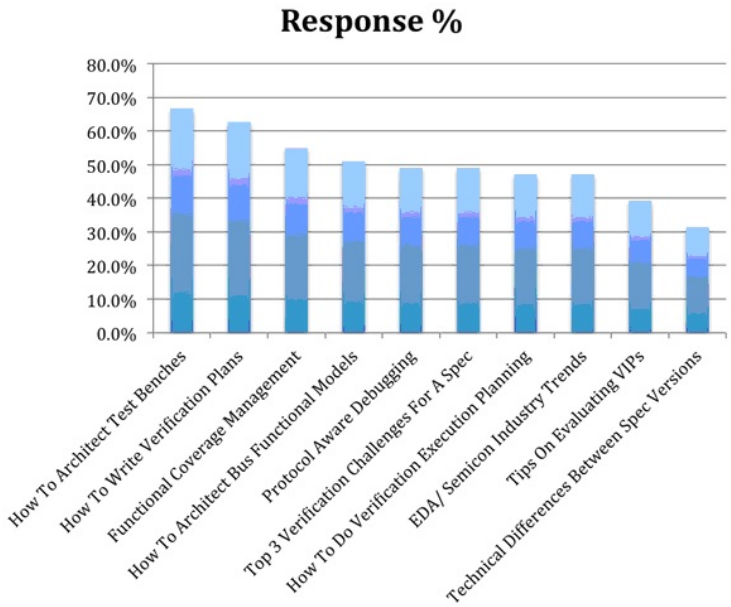

Figure 1.1: Top issues of verification - The Figure shows a survey on various issues of verification, taken from (2).

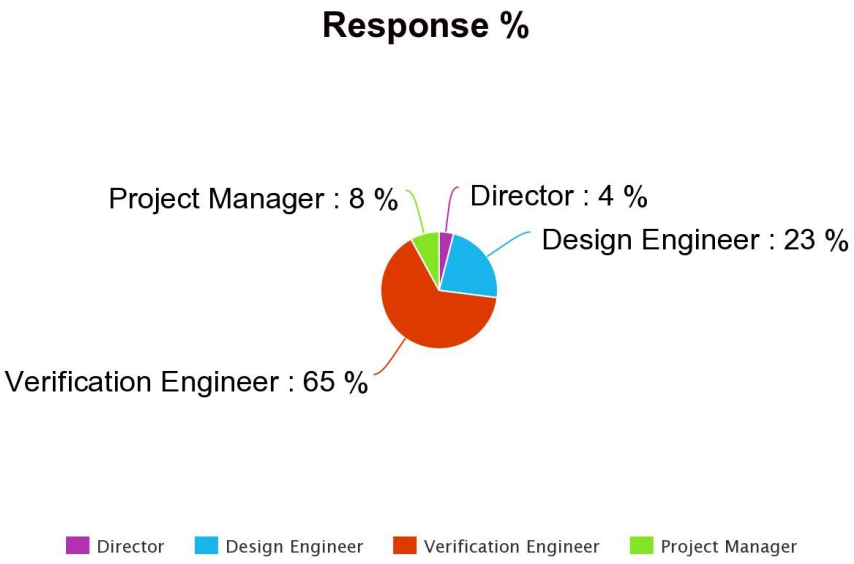

Figure 1.2: People who responded to the survey - The figure shows the percentage of people from different levels involved in responding to the survey, taken from (2).

A methodology is required for applying verification language in a structured and planned way for the development of testbenches. Verification environment developed using standard methodology enables reusability of verification components across different vendors and enforces better communication among verification engineers. As observed from figure 1.3, Universal Verification Methodology is gaining importance and widely being used in the industry. In the similar way SVA (SystemVerilog assertion) is dominantly used among the other assertion languages as shown in figure 1.4.

Exhaustive and well-defined self-checking mechanism in verification environment reduces engineer′s manual efforts for verifying correctness of design and improves overall coverage of verification. UVM doesn′t provide the detailed implementation of self-checking mechanism and leaves the verification engineer with an option to customize the way of checking the functionality. The special privilege given to verification engineer should be exercised wisely and intelligently. A defined checking mechanism could have further reduced engineer′s efforts for development of testbenches. Customized checking mechanism sometimes forces verification engineers to develop complex checking mechanism even for very small designs and sometimes provides fewer resources for

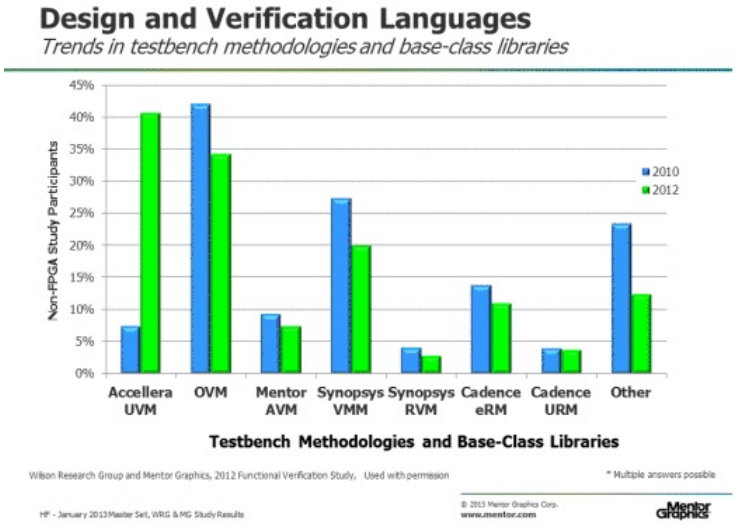

Figure 1.3: Trending Verification Methodologies - The Figure shows a survey on use of various verification methodologies, taken from (3).

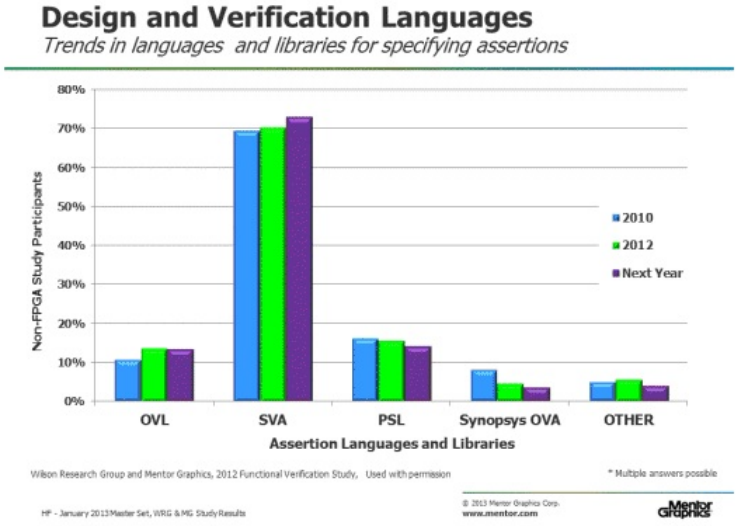

Figure 1.4: Trending Assertion Languages - The Figure shows a survey on use of various verification methodologies, taken from (3).

thorough checking of complex designs. Moreover, deciding location and placement of assertions in the testbench environment becomes an issue for verification engineer.

1.3 Problem Statement

- Development of verification environment for complex IP/Sub-system using most recent and the standard methodology i.e. Universal Verification Methodology (UVM).
- Enhancing capabilities of UVM based verification environment by providing well-defined and exhaustive approaches for self-checking mechanism. The approaches will be implemented on an industrial IP as a case study to analyse their performance and pros/cons on the basis of memory usage, timing requirement and coverage detail.

1.4 Chapter Organisation

In the second chapter, a literature survey of verification languages, methodologies and checking mechanisms has been given. Chapter three provides an overview of UVM, assertion checkers and coverage metrics. The fourth chapter gives a detailed description for development of testbench environment for complex IP/Sub-systems using Universal Verification Methodology. Fifth chapter provides well-defined approaches for self-checking mechanisms that can be used along with UVM based testbench. Then, the approaches will be implemented on an industrial IP and the coverage, implementation and performance related results has been given and analysed in the next chapter. The last chapter concludes the dissertation.

Chapter 2

Literature Survey

2.1 Hardware Verification Languages, Verification Methodologies and Assertion Languages

2.1.1 Hardware Verification Languages

Hardware verification languages are developed to reduce the efforts of engineers for verifying the designs. They introduced various construct for easy development of testbenches. Each existing verification language has its contribution to the development of testbench in a better way (4). OpenVera was introduced by Systems Science, mainly for developing testbenches. It provided facilities like generating random test patterns for hitting specific functionalities of DUT and various constructs for monitoring the variables for checking coverage. The e language was developed by the Verisity as part of its Specman product (5). It also provided various constructs for efficiently writing testbenches. SystemVerilog is a hardware modelling and verification language that provides unique and high level constructs for the development of the verification environment (6). The various interesting features of SystemVerilog are constrained random test case generation which improves the coverage substantially, assertion development to validate specific design intents and provision of constructs for coverage driven verification. Though, SystemVerilog provides various complex constructs to develop a detailed verification environment for complex designs, but still a need arises to use these constructs in a structured and planned manner. Hence, various methodologies are proposed to apply the constructs of hardware verification languages in a structured and planned way.

2.1.2 Verification Methodologies

eRM (e Reuse Methodology) is the first verification methodology, introduced by Verisity in 2002. It provided guidelines for development of verification environment which includes packaging of various features, development of sequences and virtual sequences, scoreboard etc. Mentor Graphics created Advanced Verification Methodology (AVM) in 2006. It provided a framework for component hierarchy and TLM communication to provide a standardized use model for SystemVerilog verification environments. Universal Reuse Methodology (URM) was given by cadence in the year 2007 (7). It supported the development of reusable components using some standard verification language like e, SystemVerilog. The OVM (Open Verification Methodology), introduced in 2008, is jointly developed by Cadence and Mentor Graphics. It provided the first truly open-source and interoperable SystemVerilog class library and methodology. It defines a framework for the development of reusable verification IP (VIP) and tests. It is completely based on SystemVerilog and developed by combining the features of Advanced Verification Methodology and Universal Reuse Methodology. RVM (Reference Verification Methodology) was introduced by Synopsys in the year 2003. It is based on OpenVera. VMM (Verification Methodology Manual) was developed by further enhancing RVM and introduced by Synopsys in 2006. The base libraries for VMM are developed in SystemVerilog language (8). The quick overview of verification methodologies is given in figure 2.1

UVM created by Accellera and was first introduced in the year 2011. Its evolution took place by combining the features of the Verification Methodology Manual (VMM) and Open Verification Methodology (OVM). It was mainly developed to fulfil the needs of today′s highly complex designs and brought standardization in the area of functional verification methodologies. The UVM based verification environment consists of reusable universal verification components (UVC). It provides standard base class libraries which are written in SystemVerilog language (9). It is considered as a methodology of constrained random and coverage driven verification (CDV) which includes automatic testcase generation, self-checking testbenches, and coverage metrics to reduce the time and efforts for verification of complex designs. It is supported by tools of all the major EDA vendors of the industry, i.e. Cadence, Mentor Graphics and Synopsys.

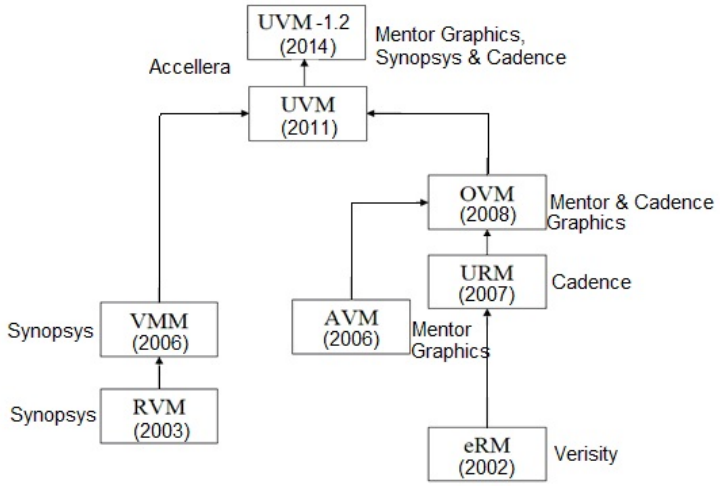

Figure 2.1: Verification Methodologies - The figure shows the developers of various verification methodology and their development year

2.1.3 Assertion Languages

Assertion languages provide special constructs to define the assertion or properties which are used to check the validity or specifying protocol temporal logic aspects of the design. The various assertion languages introduced till date are CBV - Cycle Based Verilog (Motorola, Freescale), e temporal language (Verisity now Cadence), Forspec (Intel), OVA - Open Vera Assertion (Accellera /Synopsys). OVL - Open Verification Library (Verplex, Accellera), PSL - Property specification language (Accellera), Sugar (IBM), SVA SystemVerilog Assertion (Accellera) (10). The OVL is not an assertion language, but it provides a library of commonly used assertions written in Verilog or VHDL (example onehot, signal stability, a request must be acknowledged). Among the given assertion languages, the most widely used are Accellera PSL (Property Specification Language), based on the Sugar language from IBM, and System Verilog Assertion which combines the features of Synopsys OVA, Motorola CBV, and Accellera PSL. Both these assertion languages have been developed further within Accellera. PSL can work with various hardware design and verification languages like Verilog, VHDL, SystemVerilog whereas SVA is used along with SystemVerilog models and testbenches.

2.2 Self-Checking Mechanisms

2.2.1 Functional Checkers

The functional checkers can be developed using lower or higher level of abstraction

2.2.1.1 Functional Checker at Lower Level of Abstraction

At lower level of abstraction the checker can be implemented using Verilog at signal level, these are designed in such a way that all the I/O ports will strictly act as input ports irrespective of the fact that those ports can be connected to input/output ports of the RTL (Register Transfer Language). The signal data can be obtained form the interfaces or by probing the necessary signals directly from the RTL or can get values from the available registers. The probing of the signals can be done using tool specific constructs like $nc_mirror in Cadence NCSim. In the checker, the result of implemented reference model will be checked against the actual result generated by RTL. These checkers can be integrated by instantiating them in Verilog or SV module Testbench (11) in a similar way as DUV instantiation or can be connected using keyword BIND (12). The main drawback of using these checkers is that their development is similar to the development of actual design. But the positive point is that it is not always required to mimic the complete functionality of module, sometimes only partial reference development also solves the purpose. They can be reused from IP to Sub-system level along with proven testcases (11). Hence, provide initial jumpstart in the functional verification coverage of the IP/Sub-system verification. Moreover, they also improve observability of the verification environment. If any of the checkers is not required during simulation then it can be just commented. This will nullify its existence in the verification environment. Hence, they have simple plug in/out feature.

2.2.1.2 Functional Checker at Higher Level of Abstraction

At higher level of abstraction i.e. transaction level, these checkers are implemented using hardware verification languages (SystemVerilog, SystemC). The development of checker can be done without using any methodology or using some standard methodology. In development using Universal verification methodology, the method generates the need to develop complex monitors which collect the data signal at the ports and

pass them to scoreboard after converting them into transactions. The scoreboard component is developed using a predictor and a comparator. The predictor implements the Reference Model of the module and the monitor generates and send transaction to it; it takes the operation fields, evaluates the input and generates a new predicted transaction. The comparator synchronizes the transaction with the DUV′s output coming from another monitor with the predicted transactions, compares them and generates the error messages if the transaction values are not same (13). During IP/Sub-system level verification, the signal ports of module UVC (universal verification component) which are connected to internal RTL signals are usually disabled and connected together to perform end-to-end checking. The internal UVCs which are found in the middle of the DUV can be enabled as passive agents. This allows the module level checkers and scoreboard to perform hop-by-hop checking, which helps the verification engineer to pin point the failure when debugging a failed system testcase (14). The major advantage of this approach is that it allows reusability of checker. The module UVC when reused from module to IP/Sub-system level shows great integration challenges. Though, at all the levels of verification, the testbenches are developed in order to support plug-and-play kind of integration. However, it is found that the current UVM guidelines lacked the framework to support this feature seamlessly from module level testbenches to the IP/Sub-system testbench.

2.2.2 Protocol and Specific Design Intent Checkers

These can be developed in following ways:

2.2.2.1 Implementing Assertions in Verification Environment

In order to check temporal aspect of bus protocol communication with design, it is possible to use assertion statements along with UVM environment. As UVM environment is developed using SystemVerilog, hence, SVA (SystemVerilog Assertions) can be used for this purpose. The placement of assertions check in UVM is a major issue. The assertions can be immediate or concurrent. The class based environment does not allow concurrent assertions to reside in it. Hence, these assertions can be written in SystemVerilog interfaces. But using larger number of assertions in interface overcrowds them. To overcome this, paper (15) presented practical suggestions for encapsulating SVA-based protocol checkers in such a way that they can be instantiated into the UVM

verification component interface and illustrated some suitable methods to allow the assertions to be aware of configuration and factory generated values in the class-based verification component. On the other hand, immediate assertions checks can be used in class based universal verification components (UVC) such as monitors and scoreboards. This checker mechanism enhances UVM capability for protocol checking and improves debugging.

2.2.2.2 Embedding Assertions in the RTL

Assertions can also be directly embed into the RTL. This checking method requires great understanding of internal functionality and knowledge of RTL. Verification engineers are usually not well versed in working with RTL (16). If the assertions are embed into the RTL then they can actively monitor a design functional behaviour and ensure its correctness. They detect design errors at their source; greatly increasing the observability and decreasing debugging time.

2.2.2.3 Assertion Binding Across RTL Modules

The checking mechanism allows user to bind set of assertions across the modules to verify their functional correctness. These set of assertions are connected to modules using bind keyword. Manually writing these assertions requires lot of human efforts and skills. To overcome such limitations, commercially available auto assertion synthesiser tools can be used, which automatically generate high quality assertions using simulation activity. A commercially available tool, named BugScope (17) , takes RTL and testcases as input and generates assertions. These assertions can be bind across the respective modules while performing simulation based verification using UVM.

Among the methods discussed above, the first set of two methods is responsible for functional checks of the design whereas the last set of three methods is based on assertions and is responsible for protocol checks as well as captures specific design intent. The methods in above two sets can be used in combination to obtain the improved and exhaustive verification outcome.

Chapter 3

Overview of UVM, Assertion Checkers and Coverage Metrics

3.1 About UVM

UVM provides the framework to achieve coverage driven verification which include automatic test case generation, self-checking testbenches, and coverage metrics. It reduces the efforts for development of large number of testcases, verifying the correctness of design manually and reaching higher coverage goals. This method provides the standard organised way of thoroughly verifying designs. It allows the development of test cases in constraint-random fashion which helps in reaching some scenarios even before their consideration. Therefore, provides early reach to verification goals. By using UVM, sources of coverage can be planned, observed, ranked, and reported at the various levels. UVM based testbench consists of reusable verification environments called universal verification components (UVCs). A verification component is complete configurable verification environment which is ready-to-use for interface protocol, module of a design, or a complete system. Each verification component, given by UVM, follows an orderly consistent architecture that consists of a complete set of elements for stimulating, self-checking, and collecting coverage information for a specific protocol or a design. The verification environment developed using UVM components is applied to the device under test (DUT) to verify implementation of the protocol or design architecture.

3.1.1 UVM Class Library

UVM class library is set of well-defined classes that provide quick development of reusable verification components and environment (9). The hierarchy of complete set of base classes is shown in figure 3.1 . It mainly consists of following classes:

The uvm_object class: It is the base class from which the other derived classes are extended. All the data and hierarchical classes are extended directly or indirectly from uvm_object class. It mainly defines the methods for basic operations like create, copy, compare, print, pack/unpack and record. Hence, these are visible to all the subsequent classes.

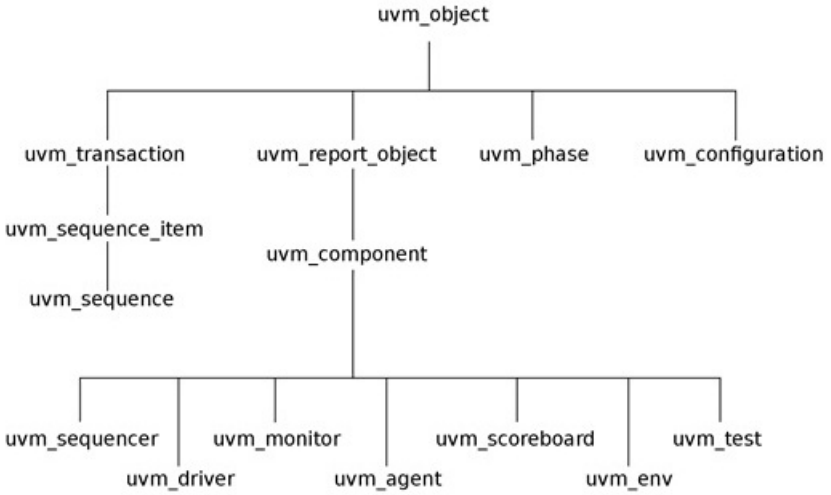

Figure 3.1: UVM Class Library - Figure shows the hierarchy of UVM classes (9)

uvm_component class: It is derived from the uvm_object_class and all the uvm reusable components like sequencer, driver, monitor, etc are extended from this class. Inheriting all the methods of uvm_object, uvm_component also adds hierarchy, phasing, configuration, transaction recording and factory interface.

uvm_transaction class: It is derived from uvm_object class. Further, uvm_sequence_item is derived from uvm_transaction and consequently uvm_sequence class is derived from uvm_sequence_item class. The uvm_transaction class is the root or the base class for all the UVM transactions. Uvm_transaction class inherits all the features of uvm_object class and enhance it by providing support for notification events, transaction recording and timestamp properties. As a further advancement in UVM libraries, now this class is no more in use as a base class for user defined transactions. Now, uvm_sequence_item

is used as a base class for user-defined transactions. It added basic functionality for objects to operate in the sequence mechanism. The uvm_sequence class is derived from uvm_sequence_item class and provided necessary interfaces to create streams of sequence items or other sequences.

3.1.2 UVM Phases

The various user defined UVM components executes their behavior in strictly ordered manner using pre-defined phases (18). These phases are defined by their virtual methods, which can be override by their derived components in order to obtain component-specific behavior. These methods, by default, do not perform any task. Other than uvm phases, there is one more important function called new() which act as a constructor. It is the first function that is called during the allocation of component. It is parametrised and includes two parameters, i.e. a string name parameter which contains the name of the component and other one is for the reference to the parent of the component.

All the UVM components call their common phases, which include functions and tasks, simultaneously. These components are always synchronized with respect to the common phases. The phases are called in the same order as shown in the figure 3.2. The descriptions of various phases are given below:

- Build_phase(): It is the first common phase of all the uvm components which is called in top-down fashion. The method for this phase allows configuration and creation of child components for the respective individual component. As discussed, build_phase() is executed in top-down manner, hence, build_phase() method of child is called after the configuration calls of parent are completed.

- Connect_phase(): This phase is executed after the build_phase(). Now, as all the components and sub-components are created in the previous phase, hence connection between those components is required. The connect_phase() makes the TLM connections between the components, assign pointer references and also make virtual interface assignments. The connection between the components is done in the bottom-up fashion, therefore, connect_phases() methods of all the components are called in bottom-up manner.

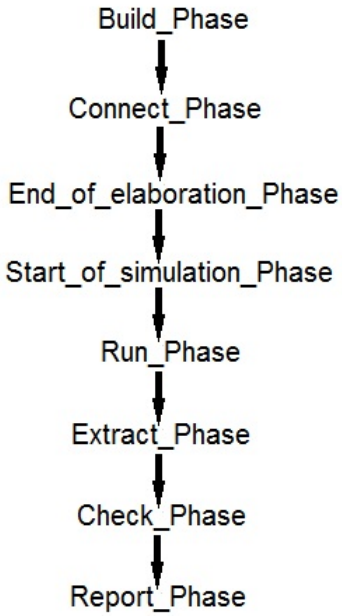

Figure 3.2: UVM Phases - Figure shows the order of execution of UVM phases

- End_of_elaboration_phase(): The end_of_elaboration_phase() is executed after the execution of all the connect phases phases. It mainly ensures that all the connections and references are properly made in the previous phase. UVM completes all the TLM bindings just before this phase so that, in this phase, all port bindings can be established and examined.

- Start_of_simulation_phase(): It is executed after end_of_elaboration_phase(). The start_of_simulation_phase() allows component initializations to be done so that everything is properly initialized before the components start their run phase.

- Run_phase(): This phase is executed after start_of_simulation phase. It is the only phase which is defined as task whereas all other phases are defined as functions. The reason for defining it as task is that it is time-consuming phase which executes till the end of simulations. It allows the execution of various sequences and tests.

- Extract_phase(): It is executed after the run_phase() and is used to extract simulation results at the end of a simulation run. It can be used to collect information regarding assertion-error count, coverage and also used to extract the signals,

register values, internal variable values, statistics or other information from components. It is defined as a function and executed in bottom up fashion.

- Check_phase(): This phase allows the checking of simulation results extracted in previous phase and validating the data and determining the overall simulation outcome. Again this phase is also defined as a function and executed bottom-up order.

- Report_phase(): This is the last phase which is used to report output in the form of files or by displaying it on the screen. This phase is again a function and called in bottom-up order

3.2 Assertion Checkers

Assertions are the properties that are used to check the data at signal level for correct behaviour. SystemVerilog assertions bring a number of innovations that are useful in simulation based verification (19). Assertions can be used to check specific design intents or can be developed for describing standard bus protocols. A set of assertions that describe some standard bus protocol can be packaged to create a reusable verification IP unit. The assertion verification IP includes a complete set of checks and coverage points for a particular standard protocol.

SystemVerilog provides two type of assertions:

1. Immediate assertions: This type of assertions is independent of time and triggered at any particular instant during the simulation process.

   Example:

   ```
   EQUALITY_CHECK: assert (a==b)
   $display(PASS)
   else
   $error("FAIL");
   ```

2. Concurrent assertions: These assertions are time dependent and runs in parallel with other processes during the simulation

   Example:

```
ADDRESS_STABILITY_CHECK: assert property(@(posedge clk) req|=> $stable(addr))
else
$error("Address is not stable when request is high");
```

It illustrates a concurrent assertion that verifies whether address(addr) remain stable when request(req) is high, the condition is checked on each positive edge of the clock

Both immediate and concurrent assertions perform a test on some aspect of the design. After the test is completed, pass or fail statements can be executed. The severity level (importance) of a pass or fail message can be set by calling special reporting system tasks: $fatal - run-time fatal, $error - run-time error, $warning - run-time warning, $info - run-time informational message.

SystemVerilog also provide availability of cover property and assume property statements which are similar to concurrent assertions in terms of defining some property. It provides one more kind of statement called expect statement that checks the occurrence of some specified activity. Assertion statements make use of sequences and properties in order to describe the complex temporal behaviour of design.

Example of assertion defined using sequences and properties:

```
sequence request
    Req;
 endsequence

sequence acknowledge
    ##[1:2] Ack;
endsequence

property handshake;
 @(posedge Clock) request |-> acknowledge;
endproperty

assert property (handshake);
```

SystemVerilog also provide availability to control the assertions using $assertkill, $assertoff, and $asserton system tasks. The recent modification in SystemVerilog in-

troduced in 2012 adds a new $assertcontrol system task. This task can enable, disable or kill the assertions based on the assertion type or directive type (assert, assume, cover, expect) (20). The task can also enable or disable action block execution of assertions and expect statements. The syntax of $assertcontrol is shown below along with the help of an example.

```
Control Type: LOCK=1, UNLOCK=2, ON=3, OFF=4, KILL=5
Directive Type: CONCURRENT=1, IMMEDIATE=2, D_IMMEDIATE=12, EXPECT=16
Assertion types:  ASSERT=1, COVER=2, ASSUME=4

$assertcontrol(OFF, CONCURRENT, COVER|ASSUME, 0);
```

3.3 Coverage Metrics

Design verification process requires coverage metrics to evaluate its progress. Coverage metrics provides a number which describes the degree of completion of verification activity. This plays a major role to get a clear picture on how well the design has been verified and also to identify the uncovered areas in verification. The coverage metrics are mainly of two types:

3.3.1 Code Coverage

Code coverage provides a quantitative measurement that describes the degree to which the source code of a DUT and the verification testbench has been accessed from the available designed test scenarios. It is further divided into following type of coverages:

1. Statement coverage: This gives information on how many statements are covered in the simulation, by excluding lines like module, endmodule, comments, timescale etc.

2. Block coverage: A group of statements which reside inside the begin-end, if-else, case, wait, loops etc. forms a block. Block coverage provides the information that whether these blocks are covered or not during the simulations.

3. Conditional/Expression coverage: It provides information on how well variables and expressions are evaluated in conditional statements.

4. Branch/Decision coverage: In Branch coverage, conditions like if-else, case and the ternary operator (?:) statements are evaluated in both true and false cases.

5. Toggle coverage: It gives information that how many times signals on ports are toggled during the simulation.

6. FSM coverage: It tells, whether the simulation could reach all of the states in a given state machine.

Code coverage is useful in finding coverage holes in verification. A coverage hole is defined as a section of code that has not been exercised or verified. On the other hand, high code coverage doesn′t necessarily signify that a design is bug-free or the verification efforts are completed thoroughly.

3.3.2 Functional Coverage

Functional Coverage provides the information about the thoroughness of the implementation of the verification plan. It also provide detail about how much of the design specification have been exercised. Functional metrics are always defined by the verification engineer by understanding the design specifications. In SystemVerilog, these can be defined in the following ways:

1. Covergroups: It is a defined by the user. Once the definition is written, it can be used multiple times by creating its instances in different contexts. It is also created using the new() operator similar to the object of the class. It can be defined in a module, class, package, interface or program.

   A covergroup can mainly contain following constructs:

   - Coverage points : A coverage point can be a variable or an expression.
   - Cross coverage : Coverage group can also specify cross coverage between two or more coverage points or variables.

2. Assertion coverage/cover property: Assertions are directly related to functionality of design. Therefore, the assertion coverage will provide the information about how much the design intents, specified by the user on the basis of documented specifications, has been covered.

Chapter 4

Testbench Architecture: Development of UVM Based Verification Environment for Complex IPs

The testbench mimics the environment in which design is actually going to reside after its development. The development of testbench environment includes designing of testcases, sequences, sequencers, drivers, monitors, interfaces, checking mechanism, coverage metrics, etc. In this section, a detailed method for the development of testbench architecture using SystemVerilog-UVM has been presented for complex designs. Here, the testbench will be designed for complex IPs residing in multi-bus protocol environment. It provides the clear architectural separation between development and integration of various elements in top-level testbench. The main elements of top-level testbench are:

4.1 UVM Package

The package contains all the reusable verification components and testcase sequences developed using UVM corresponding to some standard bus protocol or design specific interfaces. The UVM package containing all the components corresponding to standard bus protocol can also be called as its VIP (Verification IP) . The detailed development

of package or VIP is stated in the subsequent sections:

4.1.1 Transaction

It is a bunch of data signals that together represent a smallest data transfer in the testbench environment. It is extended from uvm_sequence_item class and registered in the common factory by the macro ‘uvm_object_utils . The signal variables are followed by rand keyword that allows the randomization of transaction. It enables checking and modelling of design at higher level of abstraction.

4.1.2 Sequencer

It is a component shown in figure 4.1 which is responsible for taking the sequences and forwarding them to driver by maintaining synchronisation. It is extended from uvm_sequencer and parametrized to a chosen transaction class. It is simplest and self-sufficient component of UVM that rarely require any human intervention or further modification.

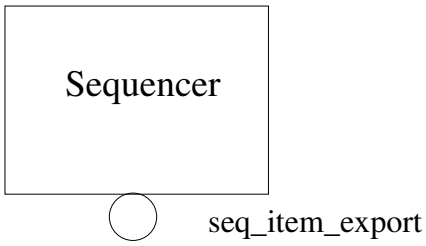

Figure 4.1: UVM Sequencer - The figure shows the uvm_sequencer component

4.1.3 Virtual Interface

In order to write-on/read-from interfaces through the UVM user defined tasks, an instance of the interface is required to be created virtually in that particular class. The interface is required to be first set in the top level testbench where it is instantiated and the same will be get in the required class using uvm_config_db:get() method in its build function.

4.1.4 Driver

Driver (shown in figure 4.2) takes the sequences from the sequencer through in-built TLM port named as seq_item_port. The communication between sequencer and driver is performed by using get_next_item() and item_done() methods in the driver. The user defined driver class is extended from uvm_driver base class. The main purpose of driver is to convert the transaction level data into signal level and passing it to interface which are virtually get into the driver. This signal level data will ultimately be driven to DUT.

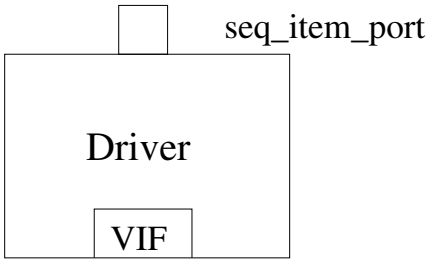

Figure 4.2: UVM Driver - The figure shows the uvm_driver component

4.1.5 Monitor

Monitor performs reverse operation of driver. It collects the signal level data from the interface and converts it into transaction level. The user defined monitor class is extended from the uvm_monitor base class. The transaction formed by monitor can be driven to other components like scoreboard, checkers, and coverage components through uvm_analysis_ports as shown in figure 4.3.

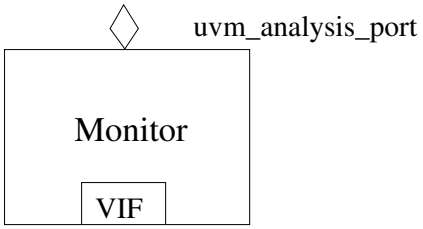

Figure 4.3: UVM Monitor - The figure shows the uvm_monitor component

4.1.6 Agent

The agent packs the sequencer, driver and monitor in a single unit to enable reusability as shown in figure 4.4. In agent, sequencer is connected to driver through TLM ports. Driver and monitor are connected to virtual interface. The analysis port is created for monitor in the agent to read the observed transactions. The agent can be configured either as active or passive. Active agent brings all three enclosed components in active state whereas passive agent only enables the monitor in active state. Each agent corresponds to some set of DUV interface and is usually mimicking some standard bus protocol. There can be any number of agents in UVM environment.

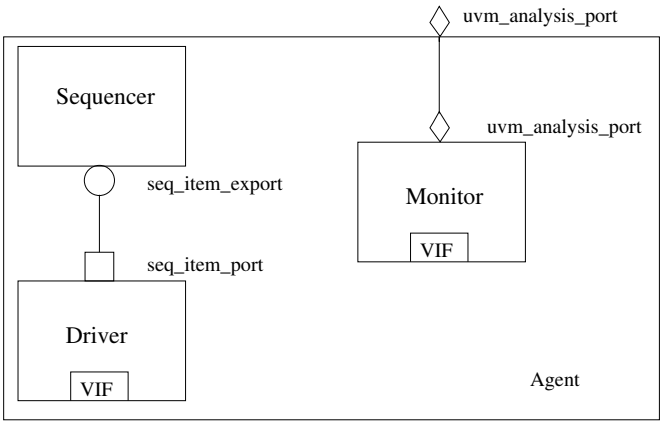

Figure 4.4: UVM Agent - The figure shows the uvm_agent component

4.1.7 Scoreboard

It is used to check the correctness of design functionality(given in figure 4.5). The user defined scoreboard class is extended from uvm_scoreboard base class. The architecture of scoreboard is customised by the engineer who develops the verification environment. It is used to verify the functionality of design or part it.

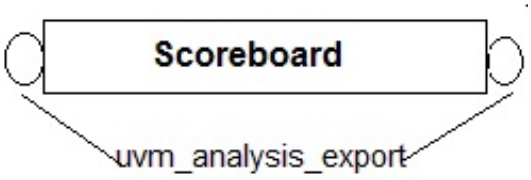

Figure 4.5: UVM Scoreboard - The figure shows the uvm_scoreboard component

4.1.8 Virtual sequencer

A virtual sequencer is a sequencer that is not connected to a driver itself, but contains handles for sequencers in the testbench hierarchy. It is an optional component for running virtual sequences. It is considered optional because it does not need driver hookup, instead call other sequences which run on real sequencers. Virtual sequencer contains reference handle for sequencers of each agent.

4.1.9 Environment

Environment is a top level component that contains scoreboard, virtual sequencer and all the agents as shown in figure 4.6. It connects agents to respective scoreboard. Depending on the complexity of design there can be any number of scoreboard and agents in the environment. User defined environment class is extended from uvm_env base class.

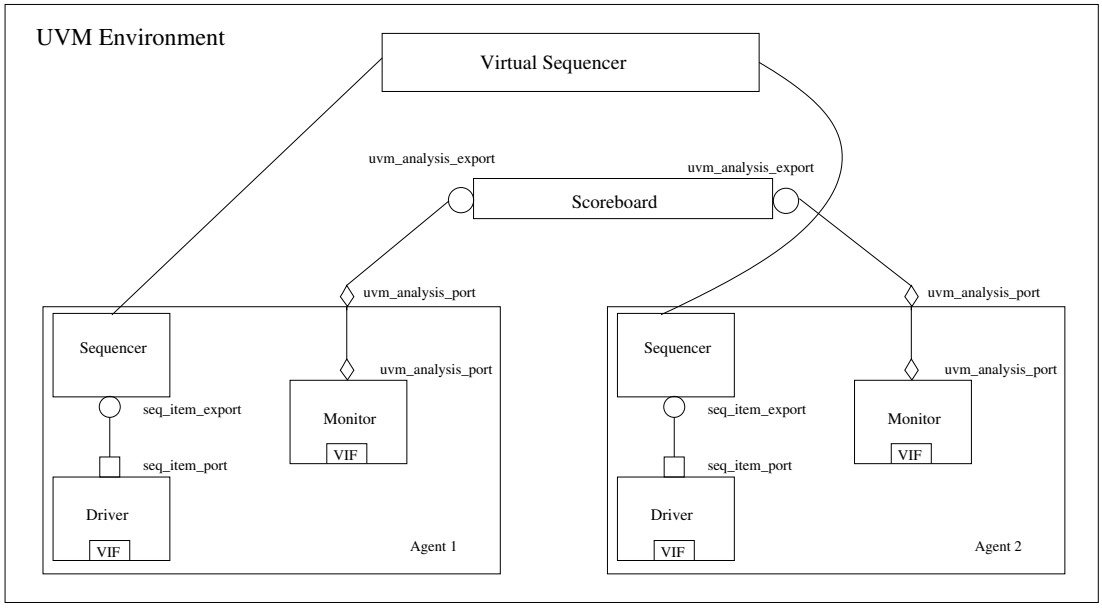

Figure 4.6: UVM Environment - The figure shows the uvm_env component

4.1.10 Sequence library

It consist of all the reusable sequences listed for specific design as shown in figure 4.7. These sequences can also be reused among different tests. A sequence defines the logical

order of transactions. Sequences are extended from uvm_sequence class. These are also registered in the common factory by macro ‘uvm_object_utils. The complete sequence is defined in body() task and the transaction is constraint randomised using macro ‘uvm_do_with.

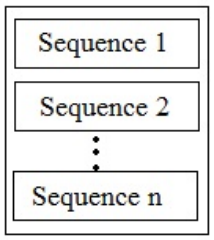

Figure 4.7: UVM Sequence Library - The figure shows the uvm sequence library containing various test sequences

4.1.11 Virtual Sequence Library

It contains list of all the sequences used for complex designs (shown in figure 4.8) which consist of multiple bus protocols. Each interface corresponds to different bus protocol takes data in orderly manner. For instance, a DUT containing programmable registers that are configured thought bus_protocol_A and normal operation is done through bus_protocol_B. So, the sequences corresponding to register configuration should be generated before generating sequences for operating DUT in normal mode. Virtual sequence library contains the sequences that defines the order of two or more complex sequences which can be passed to same or different sequencers using p_sequencer and the ‘uvm_do_on .

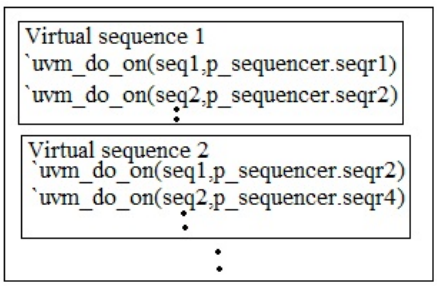

Figure 4.8: UVM Virtual Sequence Library - The figure shows the uvm virtual sequence library containing various complex sequences developed using sequences defined in sequence library

4.1.12 Test Library

It contains list of tests. Each test is corresponding to one of the virtual sequence defined in virtual sequence library. Test initiates respective virtual sequence which is designed to hit the particular functionality. User defined tests are extended from uvm_test library.

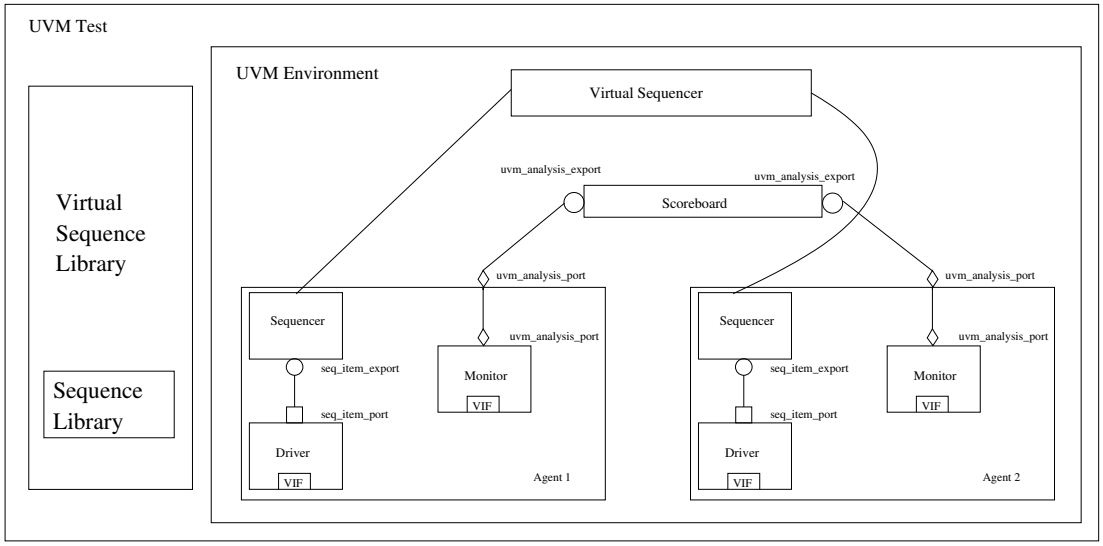

Figure 4.9: UVM Package - The figure shows the block diagram of components present in UVM package

In case of multi-protocol each bus protocol can have its own VIP which is defined using a package as stated above. When multiple VIPs are integrated together then virtual sequencer of different protocols are controlled by top level virtual sequencer defined in top level environment that maintains synchronization among the various lower level sequencers. The detailed view of multi-protocol environment is shown in figure 4.10

4.2 Interfaces

It a built-in construct of SystemVerilog language which is used for the clustering of ports. The complex designs usually consist of multiple I/O ports which correspond to one or more bus protocol. For each set of I/O ports which corresponds to some standard bus protocol, one can define an interface. These interfaces are instantiated

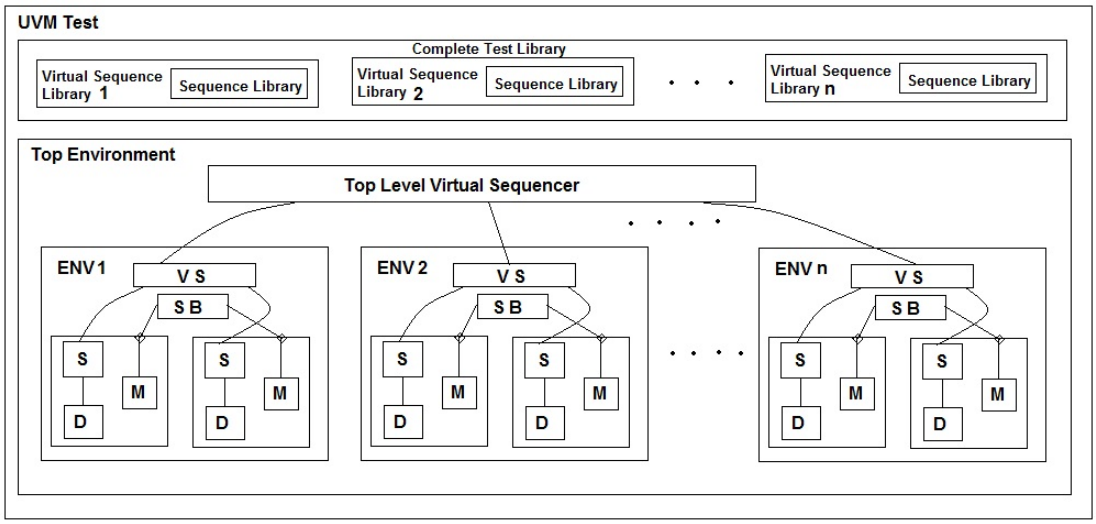

Figure 4.10: Multi-Protocol UVM Package - The figure shows the block diagram of components present in Multi-Protocol UVM package

in a testbench and each one can be accessed as a single item. As shown in figure 4.11, four differnt interfaces are be defined corresponding to bus protocols.

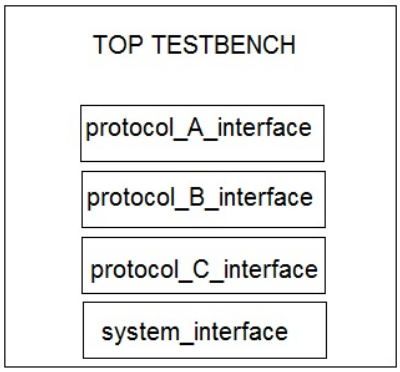

Figure 4.11: System verilog interfaces - The figure shows the block diagram of interfaces in top testbench

4.3 Design Under Verification

It is a complex design for which the verification environment is being developed. The design can be written in any of the Hardware Description Language i.e. Verilog or

VHDL. The development of testbench environment does not require any prior knowledge of the HDL used for the development of design.

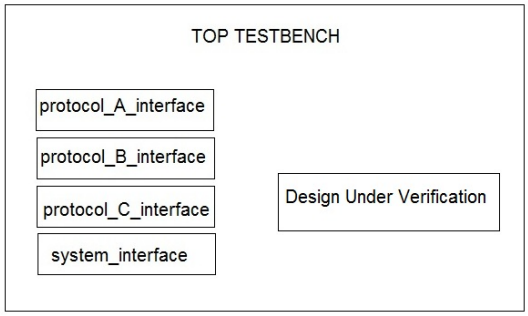

Figure 4.12: DUV in Top Testbench - The figure shows the block diagram Top testbench after addition of DUV

4.4 Clock and Reset Generators

The clock generators are used to generate the clock that handles and synchronizes the communication of DUT and the verification environment. Clocks are generated in top testbenches as shown in figure 4.13. Similarly reset signal generation is also defined in top testbenches.

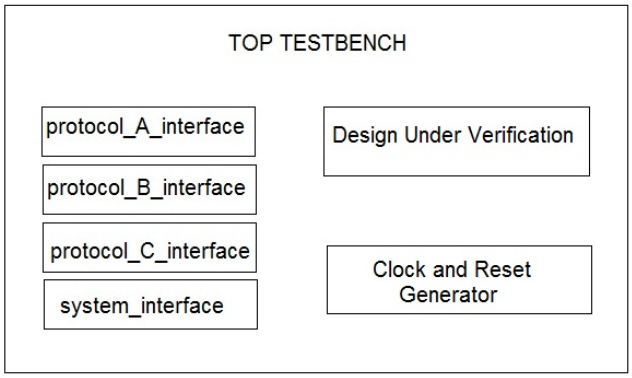

Figure 4.13: Clock and Reset generator block - The figure shows the block diagram Top testbench after addition of clock and reset generator block

The complete UVM based testbench architecture of multi-protocol complex Ip is shown below in figure 4.14:

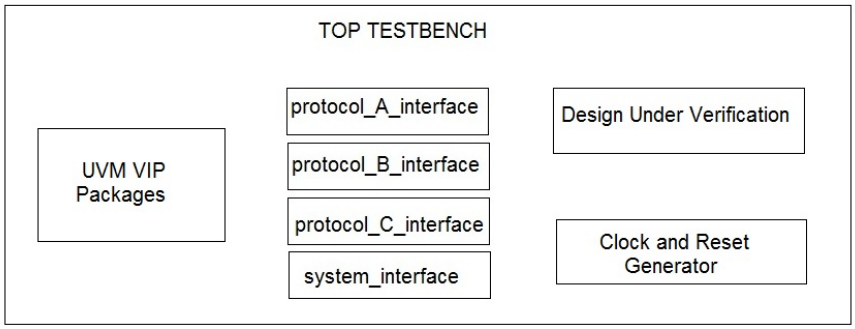

Figure 4.14: Multi-Bus Protocol Top Testbench - The figure shows the complete block diagram Multi protocol Top testbench

It has been observed that UVM provides a well-defined framework for driving the stimuli to DUV and it leaves the verification engineer with an option to customize the way of checking the functionality. The special privilege given to verification engineer should be used very wisely and intelligently. In next chapter, we will discuss various techniques for checking the correctness of design and bus protocol functionality.

Chapter 5

Proposed Architectures for Self-Checking Mechanism

The testbench development flow and creation of reusable verification components, provided in the previous chapter, gives a brief overview of development of testbench architecture for complex designs. It also provides a framework to develop automatic self-checking testbenches. The internal architectural details for self-checking mechanism is required to be defined by the user. Among the various UVCs, scoreboards and monitors are the part of a self-checking mechanism that gives a provision for functional and data checks. The self-checking mechanism also includes concurrent assertions for protocol timing checks which can be developed using SVA (SystemVerilog Assertion) language. At System on Chip (SoC) (21)or higher level of verification using UVM, all the internal module level UVCs are disabled (14), other than scoreboard, monitors and protocol checks. Therefore, a self-checking mechanism is an important part of the overall verification environment which needs to be remain active during simulation process. Hence, its performance should be taken into consideration while developing testbenches. The available information doesn′t provide a clear and detailed view for implementation of a self-checking mechanism in UVM testbench which may lead to ineffective implementation and poor performance. In this chapter, we will propose few detailed approaches for the development of generic self-checking mechanism based on the SystemVerilog (SV) and UVM. So that verification engineer can easily develop the checker mechanism by taking following factors into consideration:

1. Availability of resources: memory usage (compilation memory, elaboration memory)
2. Timing requirement: compilation time, elaboration time
3. Type of design model requirement: cycle accurate model, data accurate models
4. Level of abstraction: signal level or transaction level
5. Level of verification: module level, IP level, sub- system level or system level (21)
6. Re-usability

The above analysis for development of checker mechanism will greatly help the verification engineer to develop the most important part of verification environment in simple, structured and defined way.

5.1 Approach 1: Development of Checker Mechanism using Verilog/System Verilog

In this approach, the checking mechanism is developed using the SystemVerilog language. It is developed without using any methodology, but still it can be integrated with the UVM based verification environment as UVM is also developed using the SystemVerilog language. Fig 5.1, represents the self-checking mechanism developed using this approach.

During the development process, the first step is to decide the signals around which the checking mechanism is required to be developed. Then these signals will act as an input to the checker. Inside the checker, the signal data will be collected by the method of sampling. The sampling of signals is usually performed at the clock edge along with some protocol specific restrictions like occurrence of request and grant signals. The sampled signal level data might be converted to transaction level. This type of checker can be developed in two ways: timing/cycle accurate or data accurate

In case of timing/cycle accurate checker, timing/cycle accuracy can be maintained by avoiding various queues within the checker. The sampled data, possibly after converting into transaction level, will be directly sent to reference model which can be as detailed as the design itself. The output of reference model must be in alignment

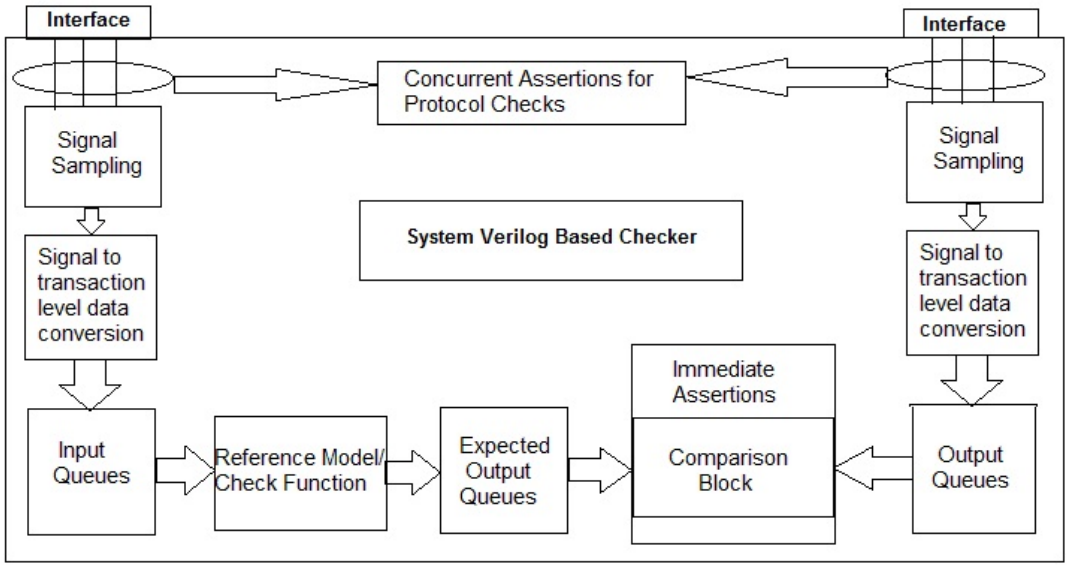

Figure 5.1: SystemVerilog Based Checker - The figure shows the internal architecture of SystemVerilog based self-checking mechanism

with the sampled output data of interface in terms of timing. The development of comparison block is simpler as it just checks the equality of data with respect to time. The SystemVerilog concurrent assertions and immediate assertions can be used on any intermediate signals of this checker.

In case of data accurate checker, the sampled data will be optionally converted to transaction and then pushed to the respective input and output queues. The data in input queue will be popped out and passed to the reference model or check function. The check function is to be developed as per the description of design specifications. This can also be abstract model of design. The outcome of check function is stored in another intermediate queue named as expected output queue. After this, the data will be popped out from both i.e. expected output queue and the output queue. The popped data will be passed to comparison block that decides the correctness of implemented design, and raises the error flag if data is not same. The SystemVerilog concurrent assertions for checking protocol communication can be applied to the signals which act as input to the checker and immediate assertions can be used on any intermediate data of the checker.

The checker described above can be integrated with the UVM based verification

environment by instantiating it in the SV top testbench (11) and mapping its ports with the interfaces or internal signals of the Register Transfer Language (RTL).

5.2 Approach 2: Development of Self-Checking Mechanism using UVM.

In this approach, the checker mechanism is developed using UVM and SystemVerilog language. This type of checker is shown in figure 5.2.

In this, functional checker consists of following two UVCs: monitors and a scoreboard. The main purpose of monitors is to read the signal level data at the interface and convert it to transaction level. In this, atleast two monitors will be used. One will read the data at input interface and other will read at output interface. Inside monitors, UVM analysis ports are implemented through which the transactions are taken out of it. Data is written on the analysis port using write(transaction) function. The

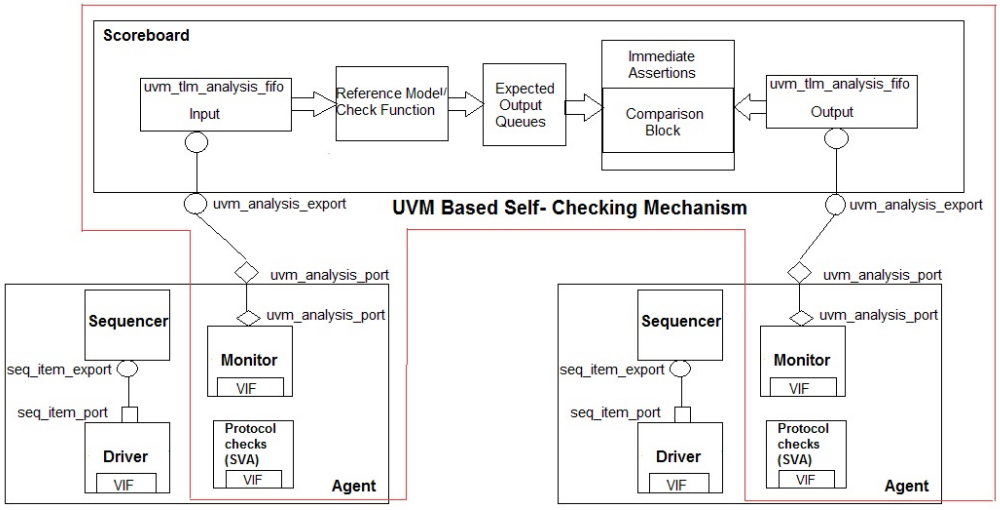

Figure 5.2: UVM Based Self-checking Mechanism - The figure shows the internal architecture of UVM based Self-checking Mechanism

transactions are passed to scoreboard in which uvm_analysis_exports are implemented that are connected to uvm_tlm_analysis_fifos. Hence, the transactions coming from the monitor are directly fed into FIFOs (First In First Out). The scoreboard consists of predictor and a comparator (13). The predictor is basically a check function or reference model. The first transaction is taken out from an input analysis FIFO and passed

to predictor for processing. The valid transaction coming out of predictor will let the transaction to be popped out from the output queue. Now, these two transactions will be compared in the comparator. The concurrent assertions can be used for checking protocol communication at interface only. We have separately created a component in which interfaces are virtually created, therefore, concurrent assertions can be written in it. Immediate assertions can be used in scoreboard as well as monitors. This type of checker can work on transaction level data only. Developing timing/cycle accurate checker using this approach is a tricky task as sampling of data and processing/ comparison of data is done in different sections. Hence maintaining timing/cycle accuracy would become difficult for verification engineer.

5.3 Approach 3: Development of Self-Checking Mechanism using UVM: A Modified Version

This is modified version of approach 2 which is applicable to development of verification environment at top level or where re-usability of verification environment is not required. In this approach, the monitors which are present in active agents can be removed as shown in figure 5.3.

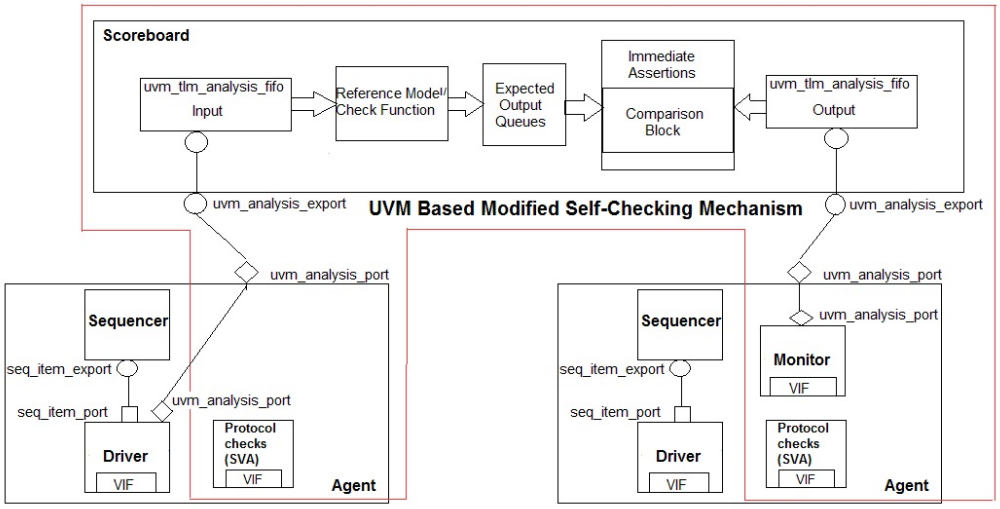

Figure 5.3: UVM Based Modified Self-Checking Mechanism - The figure shows the internal architecture for modified version of UVM based Self-checking Mechanism

The transactions which are forwarded from sequencer to the driver and then driven to interface after converting into signal level can also be directly driven to the scoreboard at transaction level only. This will save time in developing one of the complex monitors. This approach hinders the vertical re-usability of checker mechanism (22).

During higher level of verification from module to IP and IP to SoC, internal agents are enabled in passive mode, i.e. sequencer and driver are disabled. Only the monitor is enabled because the task of sequencing and driver is already fulfilled by adjacent modules. If this approach is followed at all levels of verification, then the existence of internal agents will be nullified as the monitor is not present (which is the only source of their existence). Hence inputs to scoreboard will remain undefined. Therefore, this approach is preferred only at top level verification. Moreover, concurrent assertions for protocol communication checks have to be written in some other component where interface information is directly available. For this purpose, a separate component is created in which virtual interfaces are called that allows the development of concurrent assertions using them.

Chapter 6

Results

The various approaches defined in the previous chapter are implemented on a complex DDR4 (Double Data Rate 4) arbiter IP.

The main functional blocks of the complex IP are:

- input transaction filter
- input transaction arbiter
- packet splitter
- priority arbiter.

It mainly consists of 4 major independent functionalities which are verified individually. All the self-checking approaches, defined in previous chapter, are implemented on all the four functionalities and then the verification environment of individual functionality is reused in verification of complete Complex IP. The experimental environment for single block is given in figure 6.1 .

6.1 Coverage Analysis

The main purpose of coverage analysis is to show the significance of assertions checks as a part of self-checking mechanism. The self-checking mechanisms consist of assertions which can be used as a functional coverage metrics. Each of the functionality is associated with unique set of assertions in the self-checking testbench irrespective of the

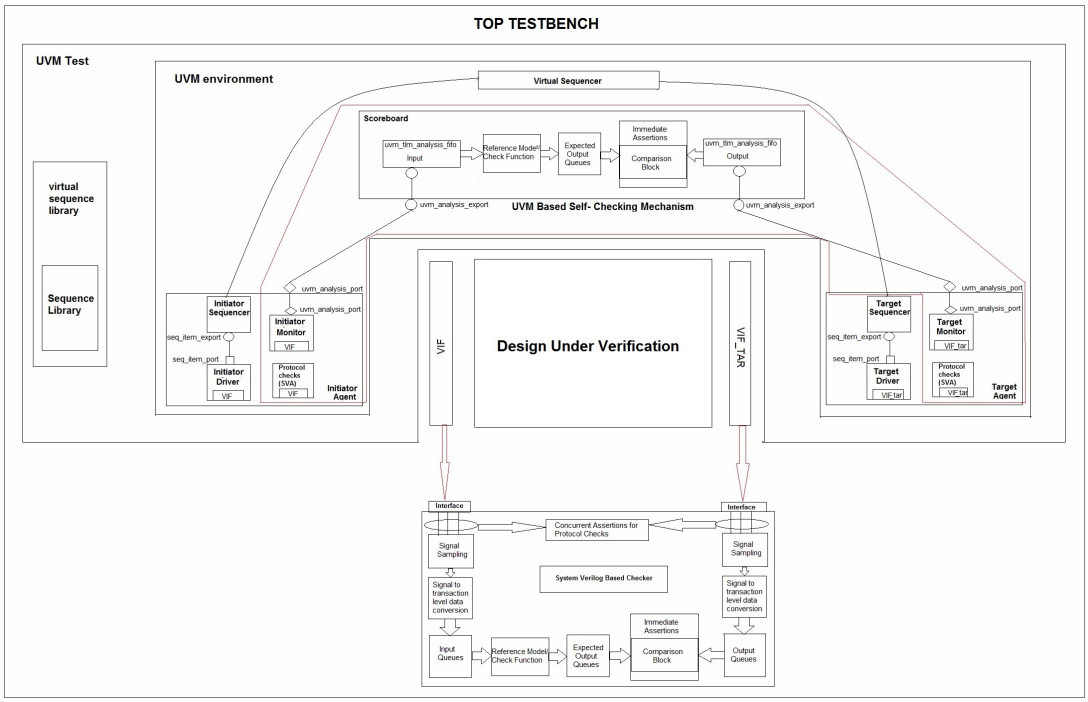

Figure 6.1: Experimental Setup - The figure shows the experimental environment for single Block

approaches described in previous chapter. The approaches only describe the placement and location of assertions in the different manner.

Functionality	No. of Assertions	No. of Testcases	Code Coverage
Block 1	9	15	18%
Block 2	14	20	15%
Block 3	11	13	17%
Block 4	16	21	21%
Overall	50	69	71%

Table 6.1: Coverage Analysis

Table 6.1 provide the analysis of coverage for complex IP verification due to assertion checks that are part of our proposed checker mechanisms. It presents the number of assertion checks developed corresponding to each block having unique functionality. In order to hit these assertions the required numbers of corresponding designed test cases are mentioned in the next column. The subsequent column presents the individual code coverage for each block when all the assertions corresponding to that block has been exercised by the test cases designed for it. The last row presents the total number of assertion, testcases and overall code coverage due to all the assertions.

It has been observed from the table that large amount of code can be covered by designing the testcases for hitting the developed assertions. So, developing assertion checks as a part of self-checking mechanism greatly help in reaching higher coverage goals at the early stages of verification. Hence, assertion checks add value to coverage driven approach followed by the UVM.

6.2 Implementation Analysis

The implementation analysis of checker mechanisms is done on the basis of type of design model that can be included in the checker, level of abstraction, level of verification and reusability. It has been observed that approach1 allows both cycle accurate or data accurate models to be implemented at signal level as well as transaction level whereas other two approaches only allows data accurate models to be implemented at transaction level. Though, first method provides more flexibility but the later ones provide early development as they can use various automated features of UVM. Among the

two later UVM based approaches, approach3 further reduces efforts for development of checker by eliminating monitors from the active agents. The first two approaches can be implemented on IP, Sub-system and System level verification but the approach3 can only be implemented on top or system level verification because approach3 becomes non-reusable due to elimination of UVM monitor components. The complete summary and comparison of implementation analysis for three different approaches is given in Table 6.2

	Approach1	Approach2	Approach3
Design model that can be developed	Both cycle accurate and data accurate	Data accurate	Data accurate
Level of abstraction	Signal level and transaction level	Only transaction level	Only transaction level
Level of verification	All the levels of verification	All the levels of verification	Only top or system level verification
Reusability	Yes	Yes	No

Table 6.2: Comparitive Implementation Analysis of Three Different Approaches

6.3 Performance Analysis

The performance analysis is done on the basis of resource utilization i.e. memory usage (compilation memory, elaboration memory) and timing usage: (compilation time, elaboration time). The verification environment is developed for each block using different approaches and each approach is reused individually to verify complete IP in each case. The IP consists of 4 complex blocks. The compilation and elaboration details of testbench for each block are given in Tables, 6.3, 6.4, 6.5 and 6.6 .The variation in the results is mainly due to difference of the self-checking mechanism as the base testbench is same for all the approaches.

	Approach1	Approach2	Approach3
Compilation Detail			
Memory Usage	31.1M	33M	31.5M
Time Usage	2s	5s	3s
Elaboration Detail			
Memory Usage	114.9M	113M	112.7M
Time Usage	9s	8.6s	7.7s

Table 6.3: Performance Details for Block 1

	Approach1	Approach2	Approach3
Compilation Detail			
Memory Usage	65.31M	70.6M	67.15M
Time Usage	14.2s	20.5s	17.3s
Elaboration Detail			
Memory Usage	241.09M	239.7M	238.9M
Time Usage	18.9s	17.6s	16.8s

Table 6.4: Performance Details for Block 2

	Approach1	Approach2	Approach3
Compilation Detail			
Memory Usage	46.65M	50.2M	47.9M
Time Usage	7.03s	10.6s	8.5s
Elaboration Detail			
Memory Usage	172.39M	170.99M	170.1M
Time Usage	13.6s	12.8s	12.1s

Table 6.5: Performance Details for Block 3

	Approach1	Approach2	Approach3
Compilation Detail			
Memory Usage	52.24M	57.98M	55.5M
Time Usage	11.2s	13.72s	12.18s
Elaboration Detail			
Memory Usage	193.03M	192.1M	191.8M
Time Usage	15.92s	14.9s	14.2s

Table 6.6: Performance Details for Block 4

By doing performance analysis of different approaches for self-checking mechanism, we have observed that the compilation time and memory usage of approach 1 is much less than the approach 2 and approach 3 in all the 4 blocks. On the other hand, there is no much difference seen in the elaboration memory and time usage among the three approaches. This clearly states that compilation requirement for UVM based self-checking mechanism is much larger than that of SystemVerilog based self-checking mechanism. It is also observed that among the two UVM based approaches, the compilation time and memory usage of approach 3 is less that approach 2 due to elimination of monitors of active agents.

For performance analysis of self-checking mechanism at IP level, only approach 1 and combination of approach 2 and approach 3 will be considered for comparison purpose because it has been observed from the implementation analysis that approach 3 is not reusable and can be used only at top level. The complete performance analysis for the IP, using each approach, is given in Table 6.7. The IP level comparison shows the significant difference in the results. Keeping both the above scenarios into consideration, it can be suggested that if the deadlines are approaching rapidly, UVM based Self-checking mechanism can be avoided. But it is always preferable to use UVM based approach, if time permits, as it reduces human efforts greatly during the development of verification environment

	Approach1	Approach2 & Approach3
Compilation Detail		
Memory Usage	270.3M	295.2M
Time Usage	64.9s	76.5s
Elaboration Detail		
Memory Usage	781.67M	777.4M
Time Usage	110.2s	108.6s

Table 6.7: Performance Details for Complete IP

Chapter 7

Conclusion and Future Work

The verification environment development process discussed in this work is generic and applicable to all the complex designs at the various verification levels. The features and order of development of verification components provides the basic idea of developing the complete testbench architecture for complex designs using UVM methodology. This will ultimately assist the verification engineer during the verification process.

In this dissertation, we have also discussed various self-checking approaches that can be a part of the UVM based testbench and provided their coverage,implementation and performance related analysis. All these approaches provides an internal architectural and detailed view for the self-checking mechanism which will help the engineers to develop testbenches in a better way by keeping various factors into consideration. The coverage analysis discusses the importance of assertions in self checking mechanisms whereas implementation and performance analysis provides the comparison of three approaches. This analysis will allow the verification engineers to choose the best approach among the available approaches that suit their verification needs and available resources.

In brief, the dissertation provides:

1. The development process of verification environment for complex designs using UVM methodology.

2. Three different approaches for self-checking mechanisms. Among these one is defined using SystemVerilog and other two are based on UVM.

3. The detailed analysis of the approaches are provided by implementing them on an industrial IP and its modules.

As discussed earlier, UVM provides a framework to achieve coverage driven verification that mainly consist of automatic test generation, self-checking testbenches and coverage metrics. The main focus of our work was on Self-checking testbenches. In future, I will try to bring other two features, i.e. automatic test generation and coverage metrics, into consideration. The detail research on other two features will further enhance the capabilities of UVM based verification environment.

Bibliography

[1] **Universal Verification Methodology(UVM) 1.1 Users Guide**. http://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.1.pdf. UVMGuide, 2011. 1

[2] http://www.arrowdevices.com/blog/ip-design-functional-verification-top-trends-for-2015/. 2, 3

[3] http://blogs.mentor.com/verificationhorizons/blog/2013/08/12/part-9-the-2012-wilson-research-group-functional-verification-study/. 4

[4] **Design and Verification Languages**. http://www.cs.columbia.edu/techreports/cucs-046-04.pdf. 2004. 6

[5] **IEEE Draft Standard for the Functional Verification Language 'e'**. *IEEE 1647/D8+3, December 2010*, pages 1–490, Dec 2010. 6

[6] **IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language**. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013. 6

[7] http://paradigm-works.com/wp-content/uploads/Migrating-AVM-URM-to-OVM-CDNLive-SV-2008.09-Pap.pdf. 7

[8] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. *Verification methodology manual for SystemVerilog*. Springer Science & Business Media, 2006. 7

[9] **Universal Verification Methodology(UVM) 1.1 Users Guide**. http://accellera.org/images/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf. UVM 1.1 class reference, 2011. 7, 13

[10] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-based verification*. Springer Science & Business Media, 2006. 8

[11] Hu Zhaohui, Arnaud Pierres, Hu Shiqing, Chen Fang, Philippe Royannez, Eng Pek See, and Yean Ling Hoon. **Practical and efficient SOC verification flow by reusing IP testcase and testbench**. In *SoC Design Conference (ISOCC), 2012 International*, pages 175–178. IEEE, 2012. 9, 33

[12] David Rich. **The missing link: the Testbench to DUT connection**. *Fremont, CA: Design and Verification Technologies Mentor Graphics*, 2013. 9

[13] Juan Francesconi, J Agustin Rodriguez, and Pedro M Julian. **UVM based testbench architecture for unit verification**. In *Micro-Nanoelectronics, Technology and Applications (EAMTA), 2014 Argentine Conference on*, pages 89–94. IEEE, 2014. 10, 33

[14] **Maximize Vertical Reuse, Building Module to System Verification Environments with UVM e**. http://events.dvcon.org/2013/proceedings/papers/01P_25.pdf. DVCON, 2013. 10, 30

[15] Mark Litterick. **SVA Encapsulation in UVM**. 2013. 10

[16] **A Practical Guide to Deploying Assertions in RTL**. https://www.cadence.com/rl/Resources/conference_papers/2.8Paper.pdf. Texas Instruments,CDNLive, 2007. 11

[17] **Assertion Synthesis: Enabling Assertion-Based Verification For Simulation, Formal and Emulation Flows**. Nextop Software. 11

[18] Hannibal Height. *A practical guide to adopting the universal verification methodology (UVM)*. Lulu. com, 2010. 14

[19] **Who Put Assertions In My RTL Code? And Why?** http://www.sutherland-hdl.com/papers/2015-SNUG-SV_SVA-for-RTL-Designers_paper.pdf. at SNUG Silicon Valley - Synopsys Users Group Conference, by Stuart Sutherland. 16

[20] **Keeping Up with Chip**. http://www.sutherland-hdl.com/papers/2012-DVCon_SystemVerilog-2012_paper.pdf. 2012. 18

[21] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, and Byeong Min. **Beyond UVM for practical SoC verification**. In *SoC Design Conference (ISOCC), 2011 International*, pages 158–162. IEEE, 2011. 30, 31

[22] Mark Litterick. **Pragmatic Verification Reuse in a Vertical World**, 2013. 35