IP VERIFICATION
AND
IP ENVIRONMENT AUTOMATION

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

HARDEEP SINGH

601762006

Under Supervision of

Dr. Amit Mishra

Assistant Professor

Saransh Mehrotra

Staff Enigeneer (ST Microelectronics)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
JULY, 2019

DECLARATION

I, **HARDEEP SINGH** hereby declare that the work presented in this thesis entitled **"IP VERIFICATION AND IP ENVIRONMENT AUTOMATION"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at the **Electronics and Communication Department, Thapar Institute of Engineering & Technology,Patiala** is an authentic record of work carried out under the supervision of **Dr. Amit Mishra (Assistant Prof.), Electronics and Communication Department. Thapar Institute of Engineering & Technology, Patiala)**and **Mr. Saransh Mehrotra(Staff engineeer, ST Microelectronics)** from June 2018 to July 2019. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date:

HARDEEP SINGH

601762006

Dr. AMIT MISHRA
Assitant Professor
Department of Electronics and Communication Engineering
Thapar Institute of Engineering & Technology
Patiala, Punjab
Date:

STMicroelectronics INDIA PVT. LTD.

Greater Noida, Uttar Pradesh 201308, India

Date: June27, 2019

CERTIFICATE

This is to certify that **Hardeep Singh (601762006),** a student of M.Tech (VLSI), Thapar Institute of Engineering & Technology, Patiala, has successfully completed one year (June 2018 – May 2019) internship program in **STMicroelectronics Pvt. Ltd., Greater Noida**. His title of the dissertation is **"IP VERIFICATION AND IP ENVIRONMENT AUTOMATION".**

During the period of his internship program, he was punctual and hardworking. I wish him every success in life.

Saransh Mehrotra

Saransh Mehrotra

Senior Staff Engineer,

STMicroelectronics Pvt. Ltd.,

Greater Noida, India

Acknowledgment

The satisfaction and euphoria that accompany the successful completion of any task would beincomplete without mentioning the people who made it possible. This humble endeavor bears the imprint of many persons who were in one way or the other helped for thecompletion of this thesis. With deep gratitude, I acknowledge all those guidance andencouragement, which served as a beacon of light and crowned our efforts with success.I would like to thanks **Dr. Alpana Agarwal** for giving such an opportunity of internship at **STMicroelectronics Pvt. Ltd.**I extend my deepest gratitude to **Mr. Rajesh Jeswani**, Head of the verification team, ADG Department, ST Microelectronics for the valuable support he provided me.Iamthankful to our PG coordinator **Dr. Anil Arora** for ensuring that theproject submissions and the presentation dates were well fixed such that it would least affectour work at **STMicroelectronicsPvt. Ltd**. I owe my heartfelt gratitude to my guide **Dr.Amit Mishra**for valuable guidelines, constant assistance, support and constructive suggestionsin the betterment of the thesis, without which this would not have been possible.Also, I would like to express my sincere gratitude to my manager **Mr. Saransh Mehrotra**, staff Engineer STMicroelectronicswho hasalways supported me in taking new initiatives, accomplishing goals and comprehensivelearning. Thanks a lot to my mentors for their valuable guidance during the project work,they have given me valuable advice and support which has helped me in understanding thetechnical aspects of the project.

Last but not the least I thank my family, friends, and colleagues for their timely help andvaluable suggestions.

Hardeep Singh

ABSTRACT

The semiconductor market is a market where the demand of the customer is always increasing to have more and more options available at their hands. Thus to fits more and more options in the chips the complexity of the chips is increasing but, the techniques of verification are not growing as fast as the complexity SoC thus leading to bottleneck of design of cycle. The thesis give a brief description of the previously used techniques and their drawbacks and leads to techniques that are being used in the industries nowadays. In the thesis, the verification at IP level is presented in detail along with technique to automate the verification environment at the IP level. In last the results and future scope are enlisted.

Contents

Table of tables

Table of Figures

1

INTRODUCTION

An SoC (system on chip) is an integrated circuit (also called the "chip") containing all electronic components. These components usually consist of a CPU, memory, I/O ports, and secondary storages. It may contain digital, analog, mixed-signal, and often radio-frequency signal processing systems, depending on SoC's application. As they are integrated on a chip, SoCs consume much less power and use much less area than traditional multi-chip designs with equivalent functionality. i.e. why SoCs arevery common in ASIC applications, mobile computing, etc.

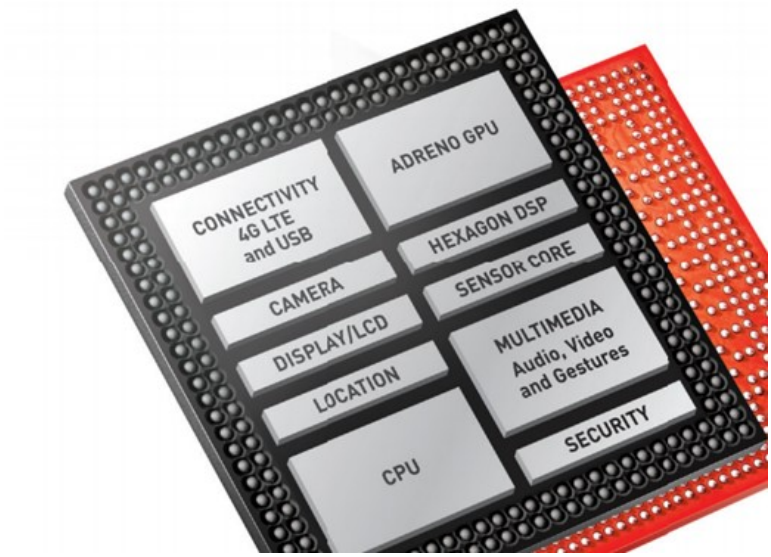


Figure 1.1 SoC (System on Chip)

SoC as compared to the common traditional motherboard-based architecture, which separates components based on their functionality and joins them through a central interfacing circuit board. Motherboard based approach contains replaceable components, whereas all of these components are integrated into a single chip (SoC) as if all these functions are the part the motherboard itself. An SoC will usually integrate a CPU, GPU, memory interfaces, USB connectivity, RAM, ROMS and secondary storage on a single circuit.

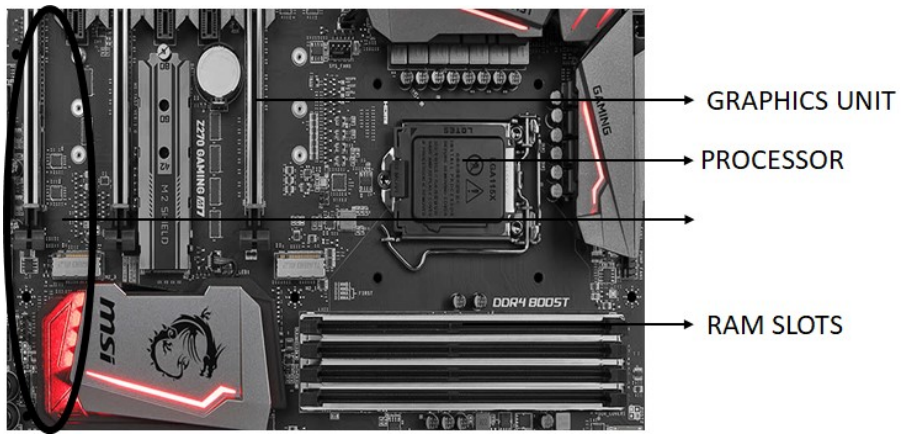


Figure 1.2 Computer motherboard

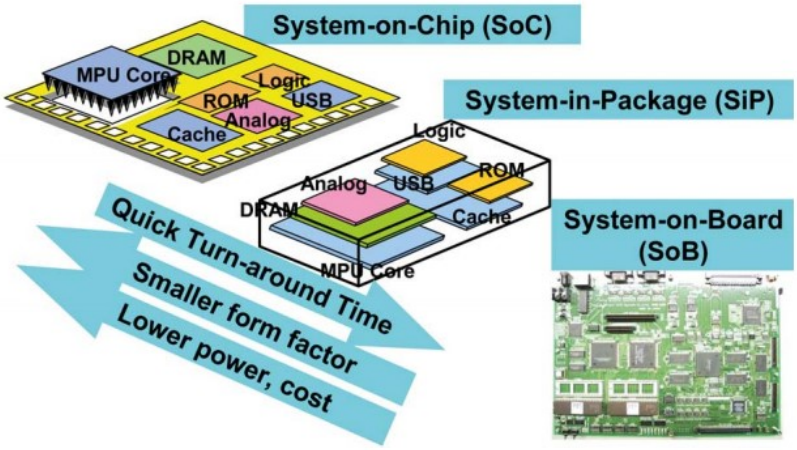


Figure 1.3 Comparision between SoC and SoB

More and more compact integrated SoC designs reduce power consumption, increase performance, reduce the semiconductor die area needed for an equivalent design composed of discrete modules, with the drawback of the higher cost of replaceable components as all the chip is to be replaced for even one faulty module. SoC designs are nearly fully integrated across different component modules. For these reasons, the trend towards tighter integration of components in the semiconductor industry. SoC can be viewed as part of a larger trend towards smaller embedded systems with more hardware acceleration.

How can we make sure that the SoC is performing according to intended properties?

What can be said about its reliability?

What about its security from foreign threats such as hacking?

These questions can be answered using **Verification, Validation, and Testing**. Though these terms sound all the same these all are different and essential steps of SoC development.

1.1 PROCESSES IN CHIP PRODUCTION

The chip also known as SoC(System on Chip) go through various processes before delivering to the customer or even see its way to actual silicon. After the RTL is written for the SoC there are various steps before going to production.

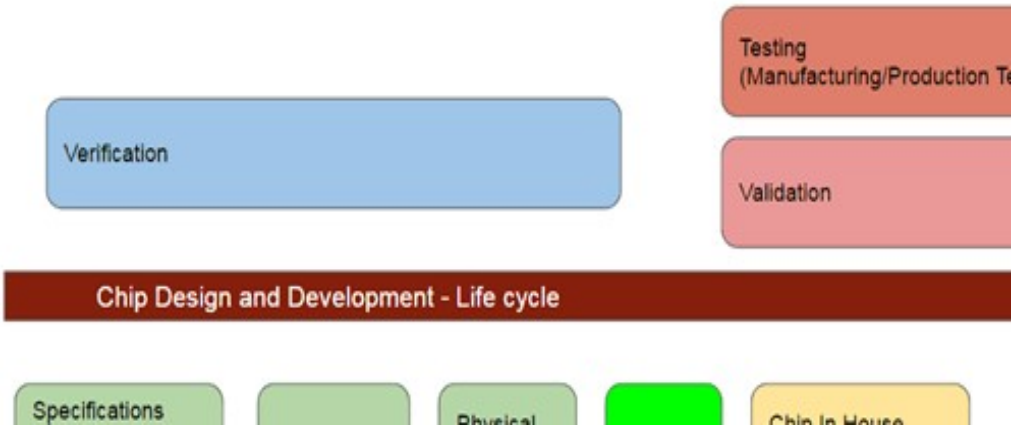


Figure 1.4 SoC Design Cycle

1.1.1 SoC Verification

SoC Verification is a process in which a design is verified against given design implementations before tape-out. This happens alongside the development of the design and starts parallelly with the Architectural Design Document (ADD). The main goal of verification is to verify each and every intended function of the design before the tape out. However, with an increase in design complexities, the domain of verification is also widening so to include much more than verification of SoC functionality. Along with verification, to meet targeted performance and

power consumption, to ensure all the security and safety features of the design and to verify the complexities with multiple asynchronous clocking domains. Simulation of the design code (RTL) is the primary way for verification while many other methodologies show great results such as Formal verification, Power-aware simulations, emulation/FPGA prototyping, static and dynamic checks, etc also are used for verification of the design before tape out. The Verification process is one of the most critical parts of the design life cycle as any bugs in design found after tape-out can lead to the overall increase in the cost of the design process.

1.1.2 SoC Validation

SoC Validation is a process in which the chip(after manufacturing) is validated for all intended functionality in a lab. This is done using the manufactured chip with a test board along with all other components that are part of the system for which the chip was designed. The goal is to validate all user scenario that a customer might have in real life. Validation happens in various steps:-

- Individual features and interfaces of the chip are validated.
- Running real software that stress tests all the features of the design.
- Burnout Test for the Chip.

Some industries that use the term Validation from a broad perspective and defines the activities that take place before and after Silicon manufacturing. Hence Verification is also sometimes referred to as Pre-Silicon Validation and Validation is also known as Post-Silicon Validation.

1.1.3 SoC Testing

Unlike the Verification and Validation, SoC Testing is done on every manufactured chip screening for faults or random defects, and electrical characterization before shipping. The first level of testing before on a wafer before dies are packaged. This testing is known as

Wafer probe testing it screens the bad chips on the basis of the technology and transistor parameters in the wafer form before the die is cut.

So let's say the chip is designed, At what level it will be verified?

How will it be verified?

What functionalities are to verify?

What Methodology is to be used?

What type of Verification scheme is to be used?

At what point a Design is said to be verified?

These questions are to be given by a verification engineer. So let's begin with the level of verifications

1.2 LEVELS OF VERIFICATION

1.2.1 IP (Intellectual property)/Module Level

The IPs RTL is written in a generic way that includes a lot of parameters so that the functionality of the IP can be altered just using the parameters. So the goal of the IP level verification engineer is to verify the IP regressively for every configuration so that RTL can be integrated into a higher (SoC/Subsystem) level. In IP level verification, an IP is verified in its own test environment called IP environment it usually SV/UVM based. The purpose is to verify the functionality of IP. In IP level verification coverage percentage of 100% is expected.

1.2.2 Subsystem Level

In the Subsystem/Platform level[1], the goal is to ensure that all the critical subsystems are verified before integrated to main SoC For e.g. A subsystem is a platform which consists of

Core (processor) along with other bus masters, there is a bus matrix (like NoC) and some slaves. The interaction between masters and slaves is verified at the subsystem level before this subsystem is moved to complete SoC.

1.2.3 SoC Level

Platform and all the non-critical/peripheral IP are integrated into one single code called the SoC (System on Chip) RTL then this RTL code is verified by the SoC verification engineer. The goal of the engineer is to verify the integration of various generic pre-verified IPs in with all the I/Os of the IP. For e.g. A communication IP is verification consist of

- Register test cases
- One/two test cases of transmitter and receiver so that the I/Os of the IP are toggled. The transmitter and receiver test-cases are verified using VIP(verification IP)
- Dma test-cases (if any)
- Interrupt test-case
- Any other test-case based on the functionality of IP

As we saw that IP level mainly focuses on functionality regressively as debugging is easier, At Platform and SoC level the main focus is one integration and basic functionality test-cases.

Now, we can say that every level of verification is necessary and one level can't work without the other level.

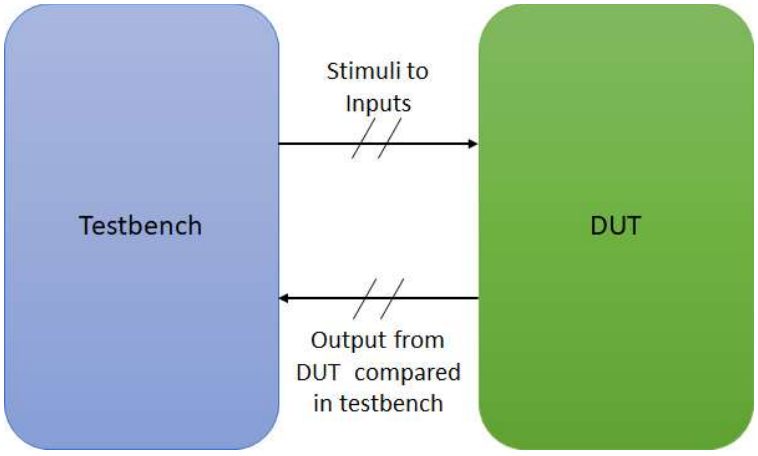
The other thing that differentiates the verification process is the methodology used for verification

1.3 Methodology for verification

1.3.1 Verilog based Verification

Gone are those days when all the inputs were listed in testbench file and given to DUT and the outputs were verified on the based on golden values provided by the RTL test. This type of verification was done with Verilog based verification[2]. The basic structure for the Verilog verification I/O based verification showed below.

Figure 1.5 Verilog based Testbench Environment



1.3.2 SV Based verification

The System Verilog came with the integration of OOPS such as data encapsulation and data polymorphism. The testbench environment became dynamic using classes, with the classes

the inheritance and randomization became easier that led a randomized and constraint randomized environment[3]. The environment basically contained:-

1. Sequencer/Generator:- to packet for DUT
2. Driver:- To drive the sequencer packet on DUT
3. Monitor:- To monitor the Output and response of the DUT and generate an error if needed

These items are wrapped in a wrapper called agent with scoreboard is used for received data and expected data, then a no. of agents are wrapped in the environment, then this environment is included in test-file which is called from testbench top which is connected to DUT using interface. The Representation of Basic Testbench is shown below.The transaction is a class that contains all the data and the data modeling is done on the transaction layer.

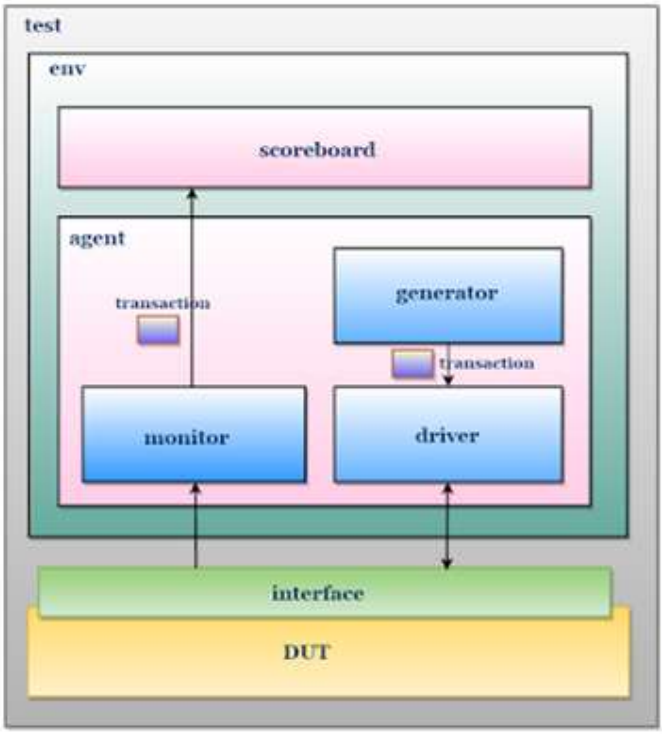


Figure 1.6 System Verilog based Testbench Environment

After SV based testbench various improvements were made on SV based testbench toward a more reusable testbench i.e. a component from testbench of an IP can be used in another testbench with no to minimum change, to minimize to create the basic skeleton of the testbench and streamline the process of running test-cases.

In that process, many methodologies were adopted such as OVM (Open Verification Methodology) then came UVM (Universal Verification Methodology).

1.3.3 UVM based Testbench

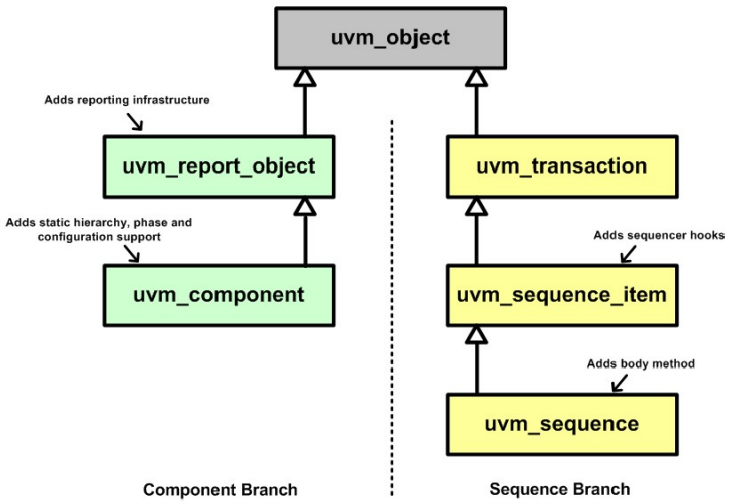


Figure 1.7 Hierarchy of UVM classes

It was based on a similar architecture as SV testbench with additional features.

The UVM[4][5] contained predefined classes that can be used for a specific purpose only like, in the above figure the component branch's uvm_component is only used to create all components like, sequencer, monitor, driver, environment, test, scoreboard, etc.

The sequences that to be driven by the driver are always extended from uvm_sequence.

The UVM also provides the predefined classes extended from uvm_component these classes also contain component specific function or task pre-included in them so the user just has to extend his/her class from the specific class to inherit predefined.

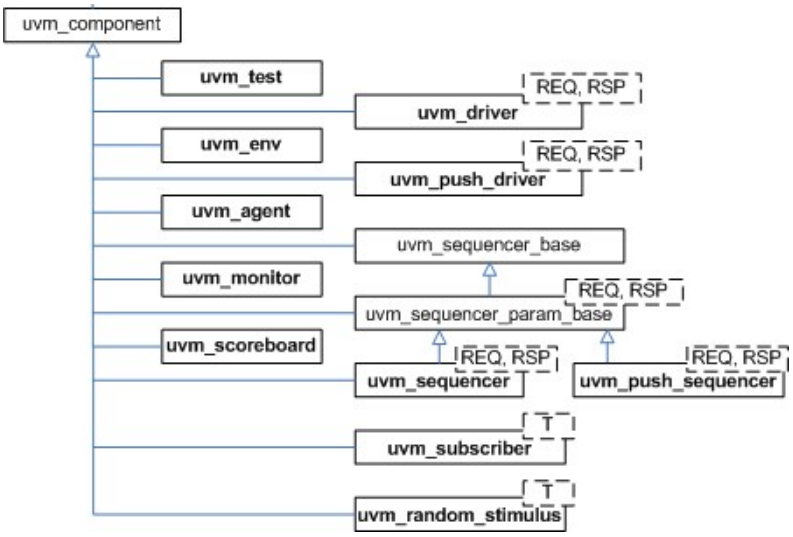


Figure 1.8 UVM Component Hierarchy

The main advantage of UVM is UVM Factory[5]. The UVM Factory is the place where all the classes are created, registered, and override/swapped out if required. The main purpose of it is to enable one type of object to be substituted with a similar object same type without any change in the testbench environment. This method is called as an override, by either type or instance. This feature is very useful for changing behavior or swapping one version of a component with another. To be swapped components must be polymorphically compatible.

1.4 IP Verification

Now the question is what are the various techniques of verifying IP?

There are many techniques to IP verification:-

1. Directed stimuli verification
2. Constraint randomized stimuli verification
3. Formal verification

1.4.1 Direct stimuli verification

In the Direct stimuli[6], the specification list called the Block guide of the IP is read and verification plan is made, according to the verification plan one by one targeted test cases are developed that target the specific functionality of the IP. This ensures steady growth to 100 % functionality coverage.

The Figure shows a comparison between time and coverage in direct stimuli verification.

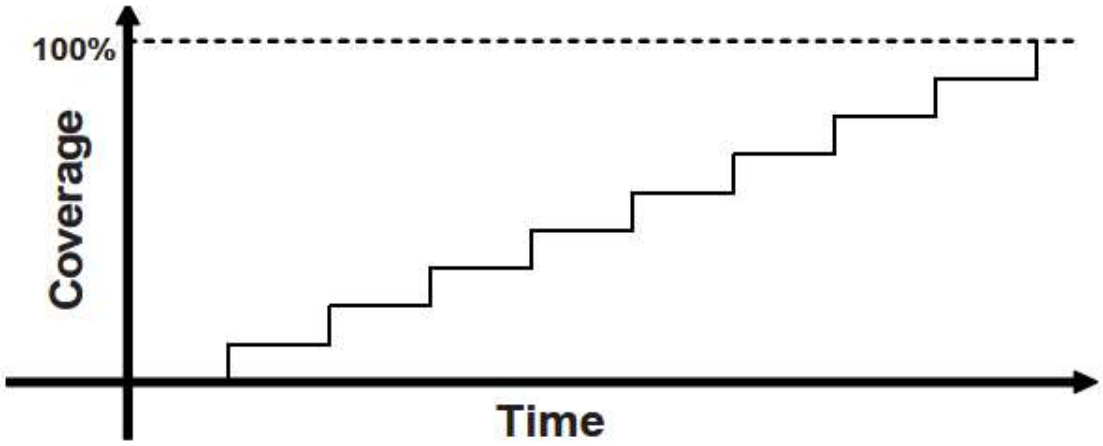


Figure 1.9 coverage vs time in direct stimuli verification

The Direct stimuli verification is good for low complexity designs as the complexity increases the time also increases as there is a directly proportional relationship between them.

So for high complexity designs, this method is time-consuming but there is also the other factor that counts as its drawback that only those bugs are found that are come in contact in the functionality of the IP design other bugs are related to dead code, extra functionality of the code are not found until till alter stage of verification.

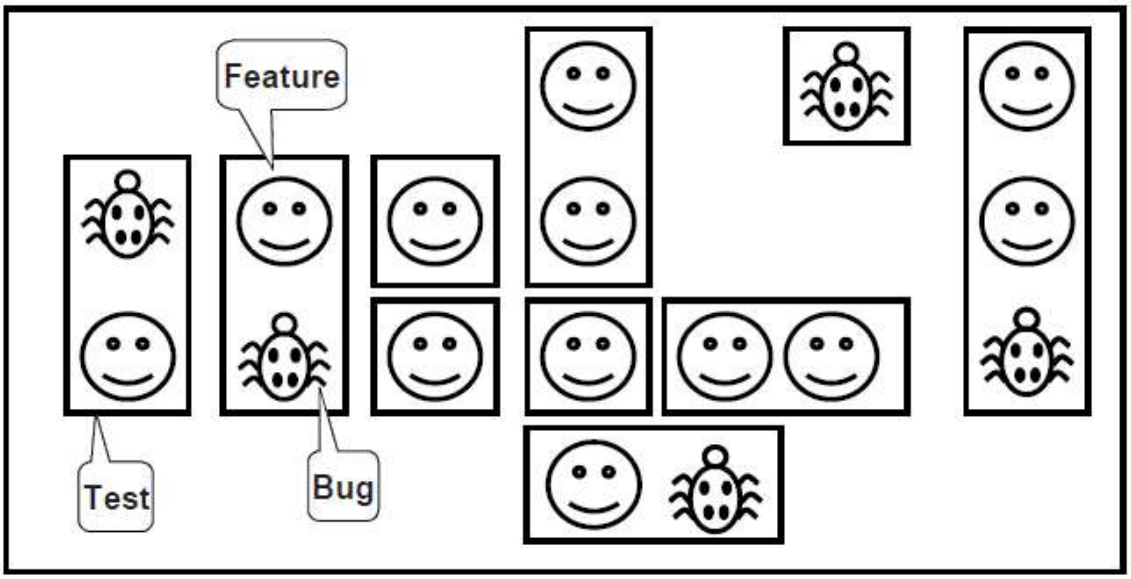


Figure 1.10Area that is verified by direct stimuli verification

The figure shows total IP RTL design as a rectangle and the bugs and the regions that are verified by Direct stimuli verification.

1.4.2 Constraint Randomized verification

What is randomization?

Why we need randomization?

Randomization is a technique by which a test case will assume a totally different value for the used rand declared variables in the test. This done using a seed value which is randomly generated by the simulator on the run, then according to the seed, various values of the variable is set for the test. This means the same test is run using a different value that will produce different results.

The randomization is needed as it provides more coverage in less time. We can't randomize unconditionally, we need some constraints to so the test does not go outside the scope of functionality or violate the protocol so we need set up some according to the spec. list and the protocol that we are using does leading constraint randomized verification[6].

The relationship between the time and coverage does not remain linear but become a rising

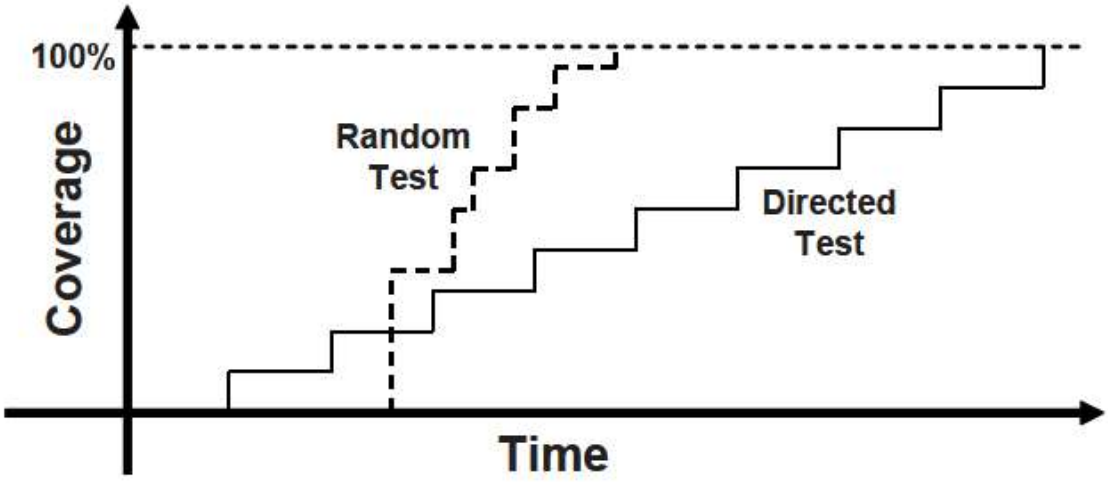


Figure 1.11 coverage vs time in constraint random verification

parabola as shown in the figure.

The circled area is the time take to develop a constraint randomized test. This type of verification can lead to overlapping teat-cases that covered multiple functionalities. The randomized verification leads to greater coverage of the RTL as shown in the figure.

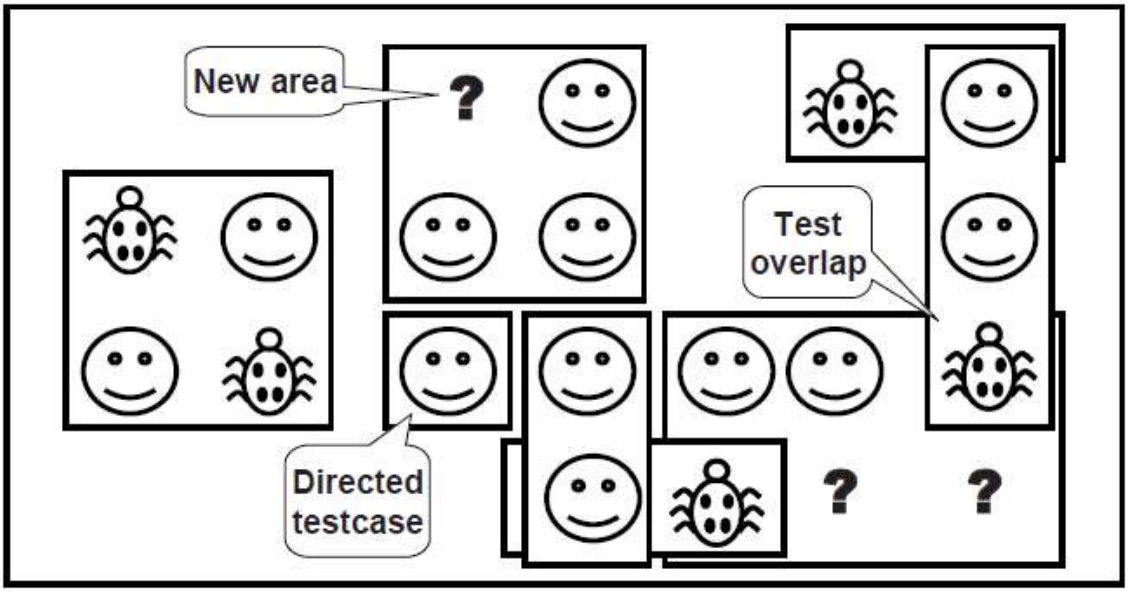


Figure 1.12Area that is verified by constraint random verification

The figure below shows the code coverage comparison between the directed verification and constraint random verification, it shows with multiple runs using random seeds we can increase the coverage and eliminate the loopholes in the RTL.

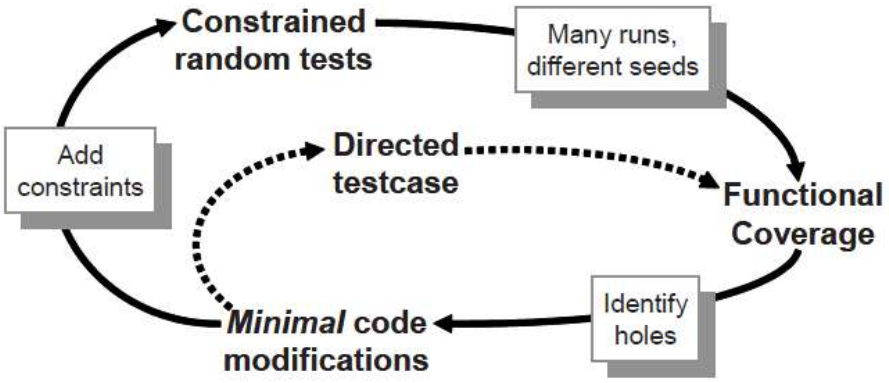


Figure 1.13 coverage cycle comparision of direct stimuli and constraint random verification

1.4.3 Formal verification

This type of verification the input to the IP is provided by the tools randomly itself and the outputs are checked against the written assertions (assertion are written in SVA) SVA language is a subset of System Verilog language[6][3]. The Formal has similar results as that of constraint random verification but the time is taken to setup a testbench is very less. The assertions are written to check the behavior of the design. If the assertion passes then the design is validated according to the given specification but if the assertion fails then it means a counterexample is found i.e. a bug is found. The assertion-based verification is faster than the simulation-based verification like the above two types.

Their many drawbacks of formal verification such as that with the increasing complexity of the IP the assertion-based verification fails to cope. The other drawback is that the assertions are difficult to debug for the complex systems.

So, in the end, I would like to state that no single type of verification is capable for 100% verification of today's complex IP designs so nowadays we use constraint random verification with the assertion to verify the basic functionality of the IP that remains the same throughout

the verification and whose failure means some bug is found. For e.g. Like interrupt signals error signal that is asserted only when functionality failure is detected by the IP RTL, other signals such as a slverr signal that is the asserted when a register is accessed for write that has no_access/ read-only access, etc.

Now we know what methodology and technique we are going to used to verify the IP.

1.5 Literature Review

Han Qi, Zheng Jiang, Jia Wei of Motorola, et al, authors in 2001 with the increase in the complexity of the SoC the verification time was increasing thus needed some standard for IP interface that can be used in every IP. This idea had potential as the IP developed by different companies can be reused as they were made on one IP interface standard, and VIP(verification IP) can be developed for the standard IP interface and can be implemented in every IP. The Motorola developed its own IPS interface which is still used as a peripheral interface slave IP in the SoC. Later own various companies developed their standards. Nowadays AMBA (Advanced Microcontroller Bus Architecture) based bus protocols are used as the connection between different IP.

IEEE computer society et al, authors in 2006 designed hardware description language(HDL) which is simple and effective language. the language is the choice of an overwhelming number of integrated circuit (IC) designers due to its features like Top-down design and hierarchical design along c based syntax. Verilog is a hardware description language (HDL) that was standardized as IEEE Std 1364™-1995. This standard contains all the formal syntax of all Verilog, the formal syntax and semantics of

- standard delay format (SDF) constructs
- simulation system tasks and functions, such as text output display commands
- the programming language interface (PLI) binding mechanism

Joo-Yul Park So-Jin Lee, Ki-Seok Chung, et al, the authors in 2008 discussed the difficulties of the SoC verification as the complexity of the SoC is increasing day by day but the verification techniques are not growing so rapidly thus leading to verification bottleneck. The authors described the way verification of some important mutually interactive IPs at a small level using core before including them to the SoC, thus leading to smaller verification time.

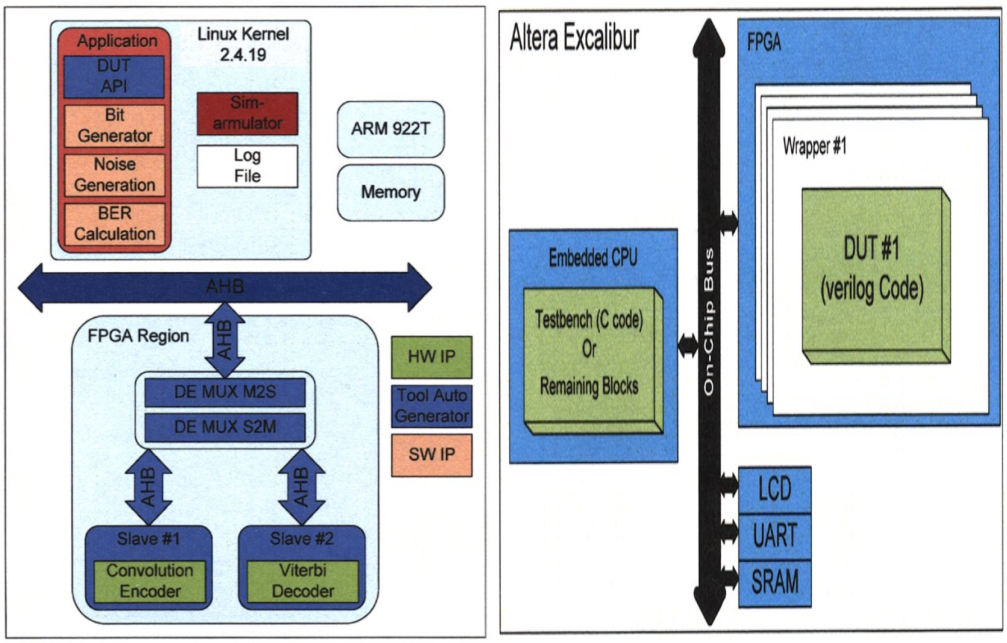


Figure 1.14 Subsystem level verification

This technique was implemented by the authors by using 5 IPs on FPGA. The verification was done using C based testcase using core just like the SoC but with very less time. This technique was further accepted by all industries and nowadays called a Subsystem/Platform level verification.

Purvi D. Mulani et al, the author in 2009 discussed the technique of verification IPs based SV verification environment with the use of SV based assertions. The IP used to demonstrate the method is I2C i.e. Inter integrated communication. The I2C is used to communicate between two different chip. The I2C IP interface is connected to AMBA AHB(advanced high-performance bus) protocol for

programming the IP. The assertions are used for protocol and interrupt checking. Rest verification is done using VCs(verification components contains driver, monitor, scoreboard, etc).

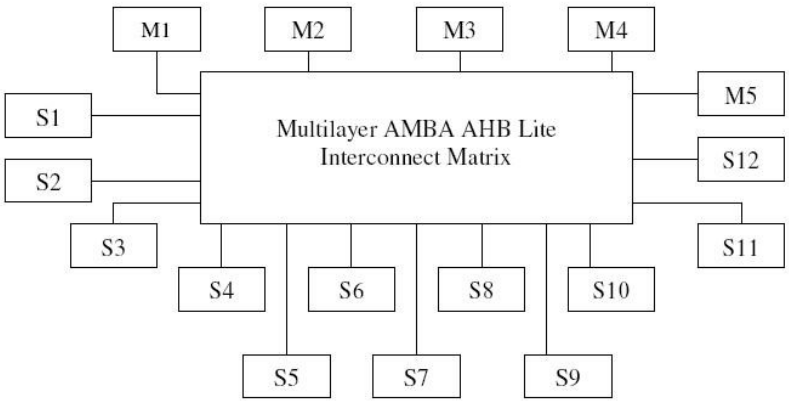


Figure 1.15 AMBA AHB interconnect matrix

The figure shows the interconnect matrix of AHB protocol as the core will be connected to one of the masters and the I2C will be connected to any one of the slave port.

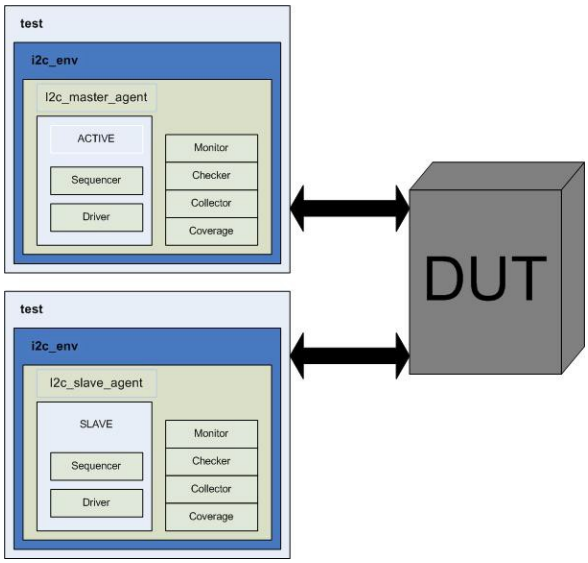


Figure 1.16 I2C systemverilog testbench environment

The figure shows the sv based testbench environment for the I2C. The separate agents are used for transmitter (master agent) and reception (slave agent).

Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, Byeong Min, et al, the author in 2011 discussed

the difficulties in the SV based testbenches basically due less reusability and show the UVM (Universal Verification Methodology) based testbench for the similar IP as discussed above i.e.I2C DUT. The UVM increases the standardization, reusability, increase the quality of verification.

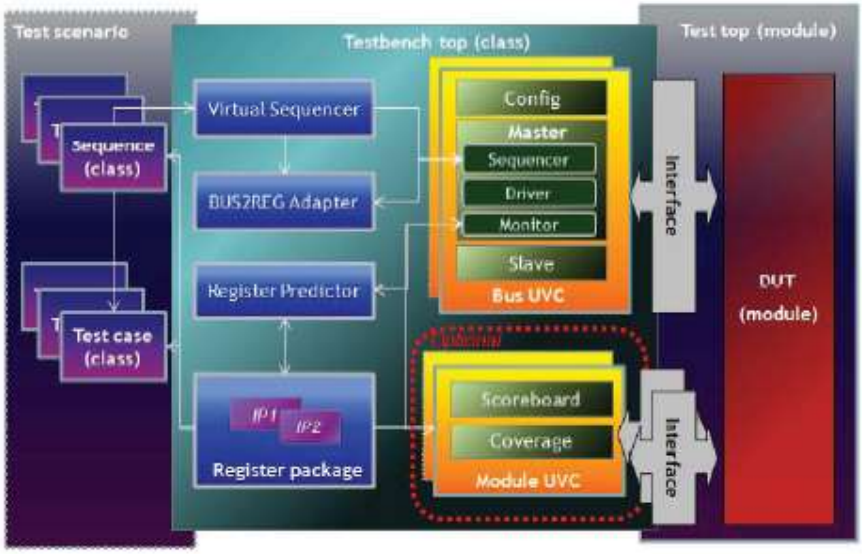


Figure 1.17 UVM Based testbench environment

The UVM based testbench environment is shown in the above figure that contains various component as sequencers, monitors, drivers, ral model adapter, scoreboard, etc.

Juan Francesconi, J. Agustin Rodriguez, Pedro M. Julian, et al, the authors in 2014 shows a FIFO (First Input-First Output) module verified with UVM based testbench with all the basic UVM components with a Functional Coverage collector.

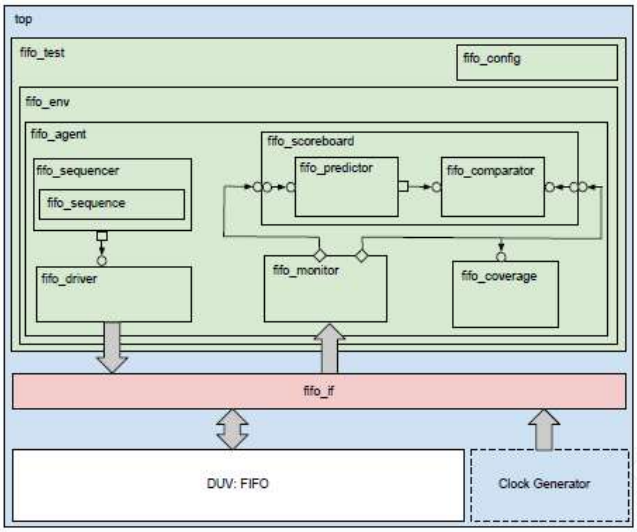


Figure 1.18 Fifo Testbench Layered Structure

UVM provided a complete framework that facilitates the development of the testbenches. The features like the UVM Factory and the UVM communication mechanism i.e TLM (Transaction layer modeling).UVM factory allows us to register a class in the factory and facilities in overriding the class (if necessary) withot changing the testbench structure.

HU Zhaohui, Arnaud PIERRES, et al, the authors in 2012 discussed the SoC verification of the communication IP is very difficult which needs tremendous efforts for developing the testbench for the IP. The represented a way of using the IP environment (that contains the driver, monitor, and scoreboard) as the VIP (verification IP). Basically, VIP acts as receiver if the SoC acts as a transmitter as vice versa thus reducing the efforts of the verification engineer to build up the environment from scratch. The authors further discuss the advantages of using UVM based environment which further extends its reusability. This technique is further used in for creating the VIP of my communication IP on SoC.

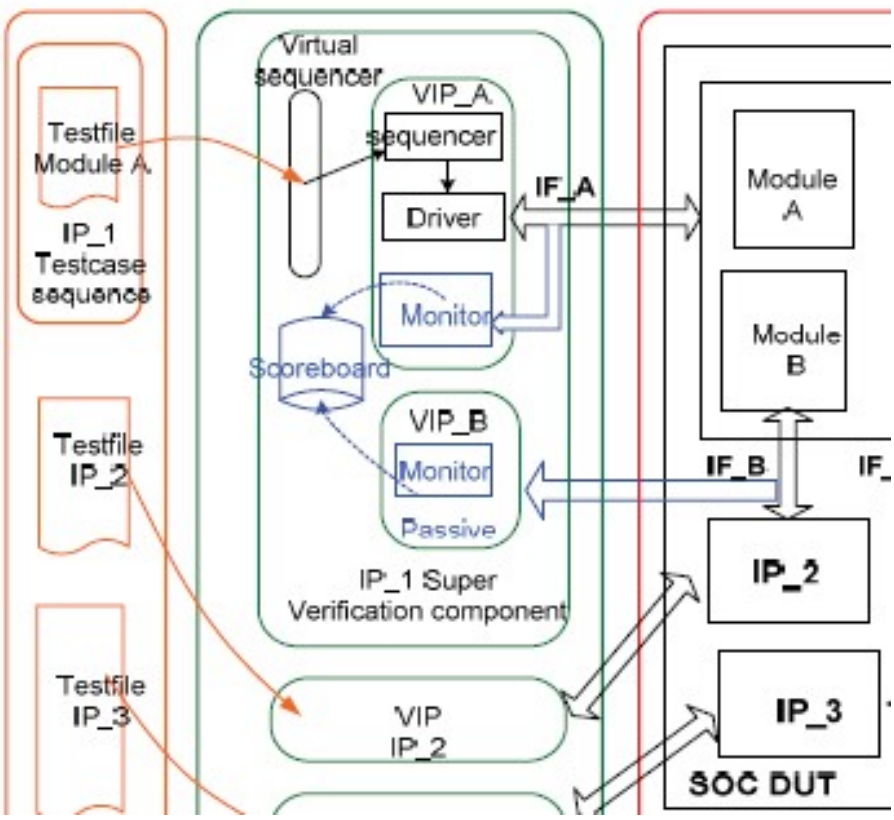


Figure 1.19 Use of UVM based VIP on SoC

The above Figure shows the way of using the IP environment as VIP for testing the IPs at the SoC level. The flow of the test case for SoC IP acting as the receiver:-

- The C based testcase is invoked, it programs the IP as receiver and VIP as a transmitter
- The data sent by VIP is passed to the driver and the driver drives it to the interface of the IP.
- The IP receives the data and makes a packet and send it to the scoreboard
- The scoreboard compares the sent data and the received data thus generating results.

J.-Y. Park, S.-J. LeeInstitute of VLSI Design, Hefei University of Technology, et al, the authors in 2015 discussed the IP verification is becoming more complex as the IP is becoming more and more complex thus the time of verification increases leading to costlier chips. The constraint random verification cannot succeed in achieving 100% code coverage and functional coverage without any feedback. So the authors propose that coverage as feedback to constraint randomized verification.

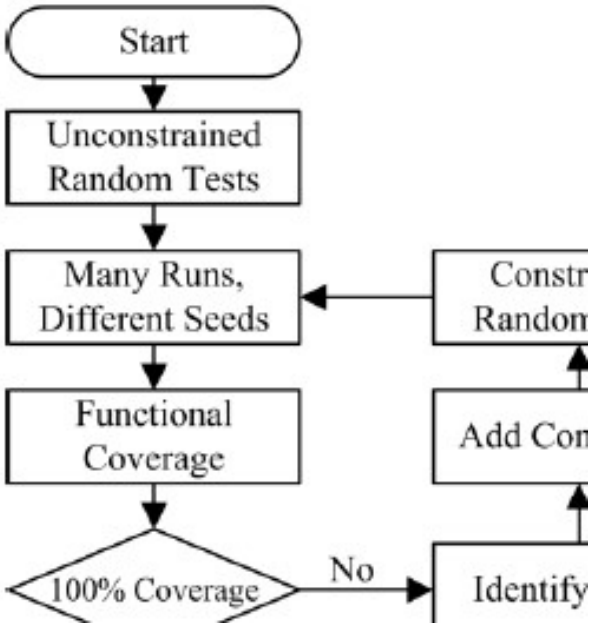


Figure 1.20 Flowchart representation of coverage driven constraint random verification

As the figure shows the coverage used as feedback to make a coverage based driven testbench. Thus new testcases are developed based on the feedback of coverage. Thus reducing the time for achieving 100% code and functional coverage.

Rohit Srivastava, Gaurav Gupta, et al, authors in 2013 discuss the difficulties in the IP verification environment creation from scratch which takes a time of 80 – 100 working man hours just to connect and configure the testbench environment. Thus the authors enlist the difficulties that are to be addressed:-

- Collect already available testbench components
- Create template components for new interfaces
- Connect verification IPs to make testbench
- Connect various components with the design under test
- Configure the verification IPs

- Generate constrained random stimulus
- Run sample test

The automation script takes some inputs from the paper for generating the testbench environment. The automation script tackles with every problem stated in the paper and reduce the environment generation time to a couple of minutes.

Srikant Kumar Mohanty, Suchismita Sengupta, et al, the authors in 2015 discusses the problem stated in the above paper i.e. automation of connection and configuration of the VIP for the testbench environment. The authors state the method of connecting the VIP after the creation of the testbench directory architecture of NoC IPs. This is achieved using Perl language and giving manual inputs in the CSV format.

My automation script tackles the problems that are not included in the above paper and improves their method by eliminating the creation of the CSV files for input and extend this approach to every IP.

2

IP VERIFICATION

The IP that was allotted to me was a communication IP.

2.1 Specification of IP

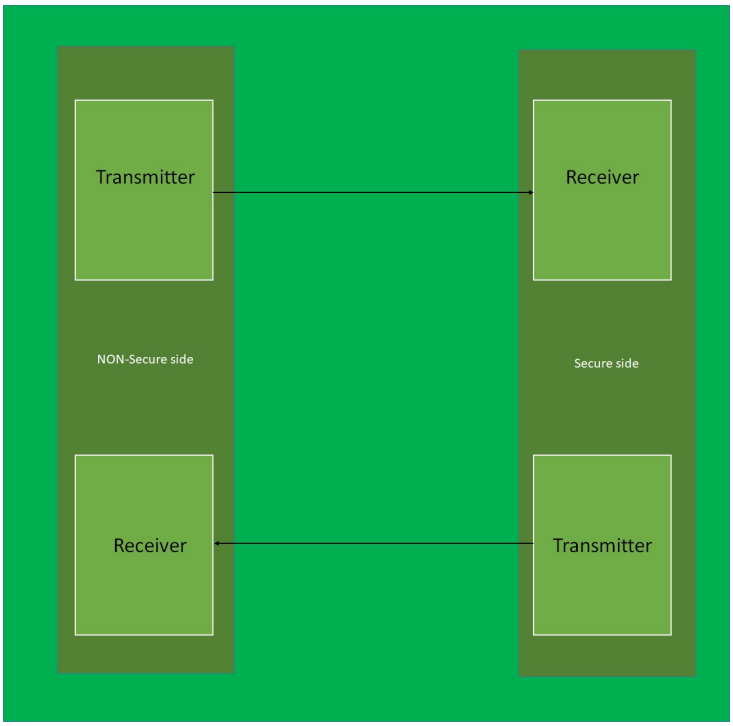


Figure 2.1 Block Diagram of communication IP

The function of Communication IP supports communication between the secure and non-secure areas of the SoC. The IP contains one receiver and transmitter on the both with some checking features disabled on the non-secure area. The task of the IP is facilitated to run the interrupt routine whenever the interrupt is being risen by any IP. Every IP has a protocol that is used to access the register and IP it can be any of the AMBA Protocol such as APB, AHB for peripherals and AXI for Core and NoC, etc.

The flow of the Interrupt:- interrupt risen by any IP in the non-secure area -> Core (interrupt related to the security of SoC)-> comm. IP -> Secure Core -> interrupt executed -> Acknowledgement to IP->

Non-secure core

So as shown above task of the IP is to collect the interrupt information from the core and the give to secure core for execution but in between IP also perform its own check on the interrupt to prioritize the interrupt that has to delivered first to the secure core.

So as it was a new IP and was being verified for the first time its all environment was to be setup from scratch.

The IP always include three things RTL code, Port XML (contains the information of the I/O ports), Register XML (contains the information for the registers in the IP) along with Block guide which lists the specification of the IP and integration guide that contain the information of the I/Os.

Like the First step towards anything is planning so in IP verification the first step is planning the test cases that target the specific functionality of the IP after reading the block guide. This can be done using various verification planning tools. In STMicroelectronics, we used cadence vplanner for planning along with it we also make a document named verification guide where we list the complete information of the verification environment.

The IP testbench needs number components that play an important part in the verification.

2.2 COMPONENTS IN IP VERIFICATION

2.2.1 VIP

After planning, the first task of the IP verification engineer is to setup the environment for the testbench. As at the IP level, there is no core present for sending/ receiving the instruction to IP So we need an approach to send the instruction by driving every signal of the protocol and observe the IP functionality. But this task will be tedious, difficult and buggy. So we need a

master which performs that sends the program data to the IP on a specific protocol that change from IP to IP and a monitor is needed to monitor/observe the responses of the IP.

This task is done by VIP (verification Intellectual property)[4][5][7] these VIPs are programmed to send /receive data to IP on a particular protocol that is supported by the IP, So we need to program the data in VIP and it takes care of every signal in the protocol. This paves the way for reusable testbench. VIP itself contains all the components of the environment so we have to include this environment to IP's testbench.

So according to the IP's supported protocol, the VIP is inserted to the IP testbench. The VIP can be supplied by 3^{rd} party vendors companies that develop VIP or it can be in house developed VIP. After insertion of the VIP environment, The testbench is compiled-clean when the compile is cleaned then we move to the next step.

2.2.2 RAL Model

If you have like 10 to 100 registers you can write every test-case remembering the registered address, but if the register count increases to like 500 to 1000 registers then it looks like an impossible task[4]. As the number of lines increases the mistake in the code increases. As each register has a name attached to it then we can make defines for the registered address with the same name and use in the test-cases but it still is difficult as the register set can be changed in next release of IP thus leading to redevelopment of register test-cases. So the answer to this problem is the RAL model.

RAL (register abstraction layer) model is used to create an object-oriented model of the DUT hardware registers. It is collections of the classes extended from UVM uvm_reg class when these classes are extended carefully, it assists in the read and writes of the DUT registers.

2.2.2.1 Hierarchy of the RAL model

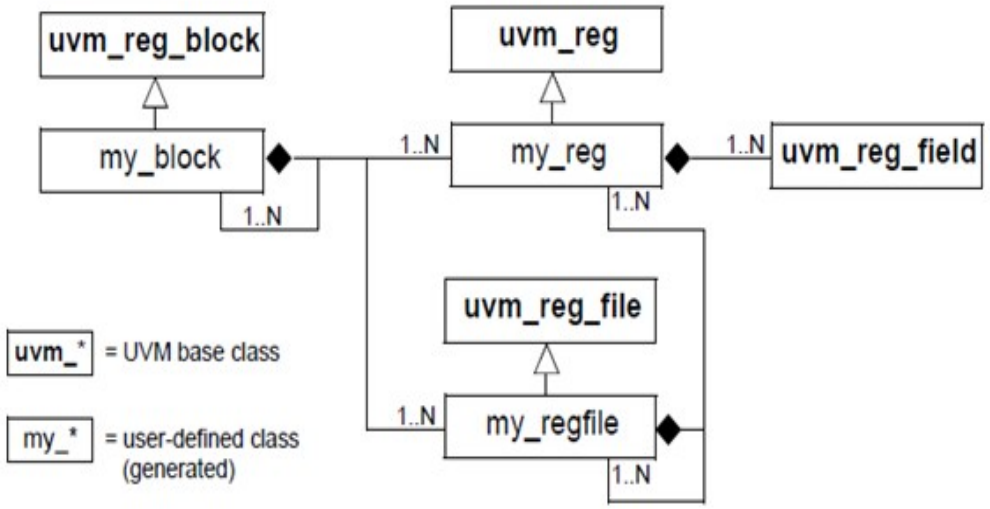


Figure 2.2Hierarchy of the RAL model

As seen in the figure above all the classes are extended from UVM base classes. The top RAL model is a user-defined class extended from uvm_reg_block, this top class contains the instances:-

1. User-defined reg_class extended from uvm_reg representing a specific register and it register fields that are extending from uvm_reg_field and all the information like access type, reset value, etc. is also given in this class.
2. The user-defined reg_file is nothing but a file containing all the user-defined reg_class.

Suppose the DUT block diagram containing 2 registers CONFIG with ADDR, DS, OE as register fields and INTRPT with STATUS and MASK register fields and 2 RAMs memories called TBL and BFR[8].

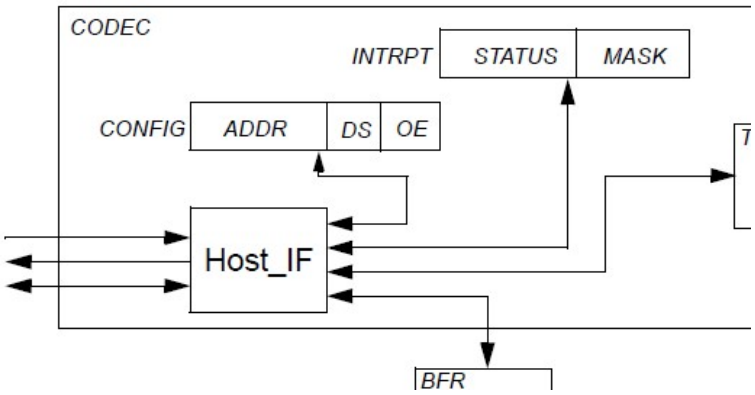


Figure 2.3 Hardware description of registers

The representation of these hardware registers is shown in below figure.

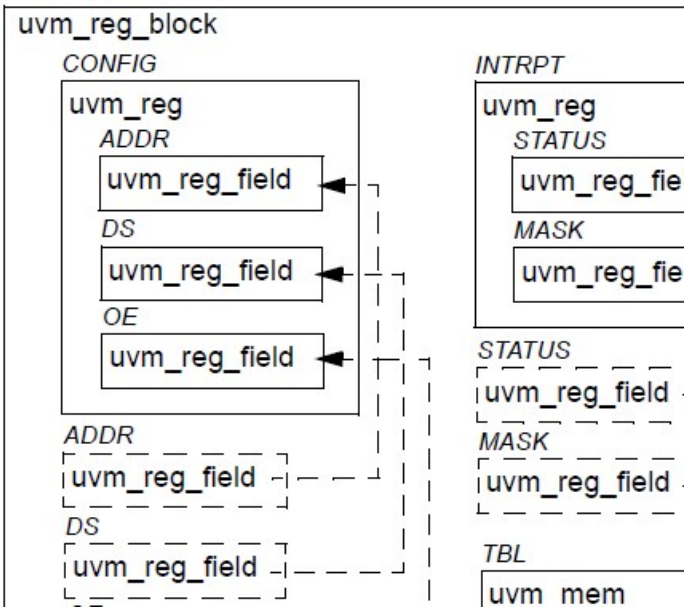


Figure 2.4 Ral representation of above Hardware registers

So if we need to access the register CONFIG is to be accessed that is contained in mine_block then we will write like

mine_block.CONFIG.write();

mine_block.CONFIG.read();

Rest is taken care of by the RAL model as what signals are to be asserted on the Interface so that the transaction can be completed. Now we will see how the RAL model work?

2.2.2.2 How the .write()/.read() is converted to bus level?

Block diagram of the RAL model[8]

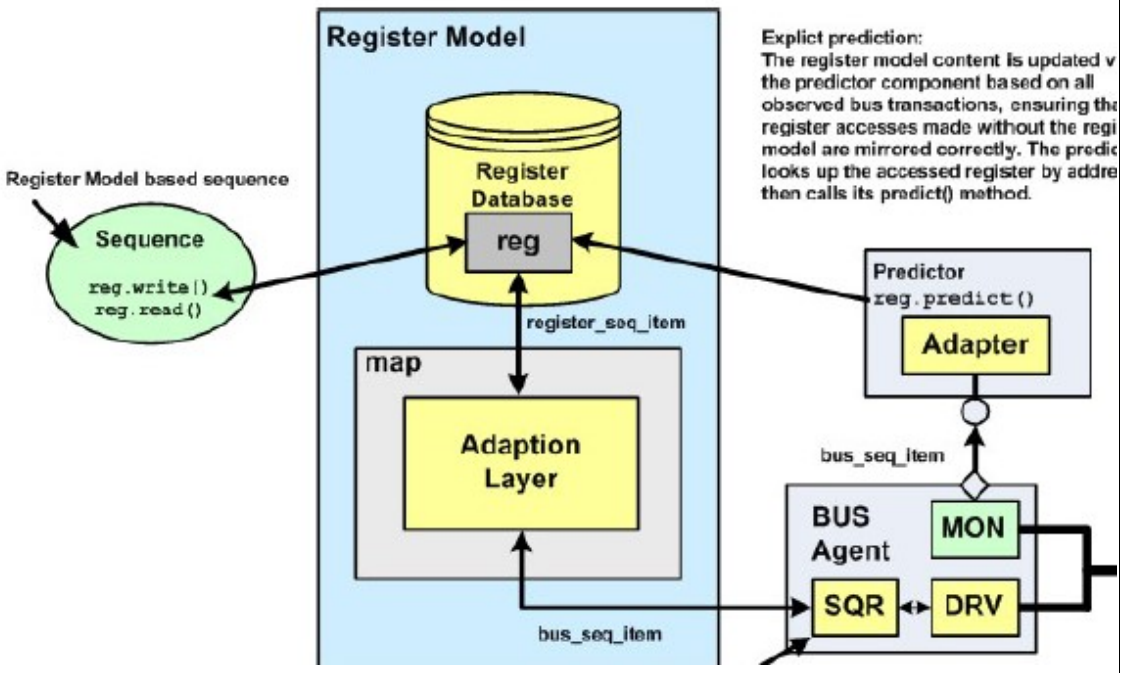


Figure 2.5 Flow diagram of wrtie/read operation in Ral Model

The Block diagram of the RAL model is shown it contains many components each having its own purpose. We will see the working of the RAL model as well as the working of individual components.

1. The RAL make a ral_block model in the testbench which represents all the hardware registers in the DUT. We can say that the Hardware registers of DUT are replicated in hierarchical in the testbench side.
2. In the sequence we write block.reg.write()/block.reg.read() it does two functions:-
   I. The RAL model updates the write value in the real block model (Desired copy) and then sends this to adapter (adaption layer).
   II. The adaption layer contains 2 functions reg2bus and bus2reg i.e. convert the reg (RAL) transaction in the bus agent sequence item and vice versa.
3. The reg2bus converted sequence item is given to Sequencer and then follows the usual flow.
4. The bus sequence item is then tapped and the through monitor than given to predictor.

5. The predictor asks the adapter to convert the bus sequence item using bus2reg the result is mirror value.
6. The Mirror value is then compared with the desired copy if the difference is found then the uvm_error is thrown.

The write operation is shown below.

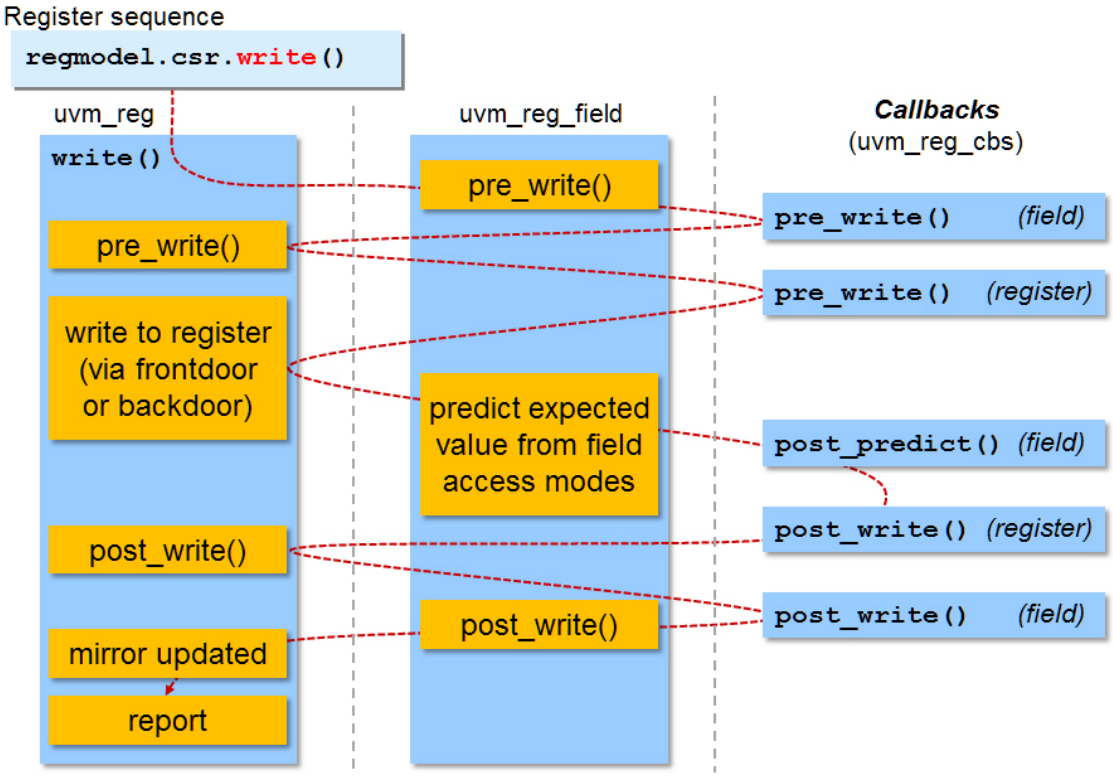


Figure 2.6 Flow of .write() task

2.2.2.3 Now after understanding the RAL model, the question arises that how is the RAL model is made?

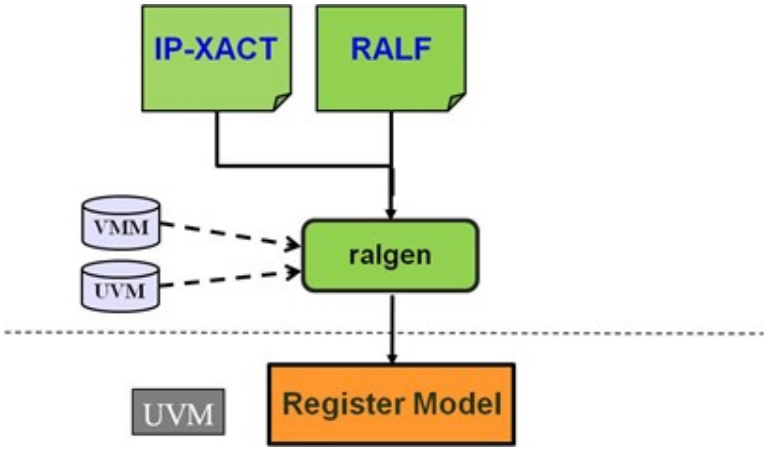


Figure 2.7 Tool flow for RAL model generation

As told above we get a register XML that contains all the register that register XML is fed to a ralgen tool that generates the RAL block register model.

The major advantage of the RAL model is that the basic register access cases can be written into the RAL sequence that can be used to do all register tests on all the RAL model. The verification engineer just has to debug if any error happens everything else is done by the RAL model sequences.

2.2.3 INTERFACE

The interface is[5][6] basically a collection of all the wires connected between the DUT and the Testbench side. It also contains additional information such as direction and timing information of the wires between the DUT and testbench.

Its advantages of Interface: -

- It allows grouping signals together so that it can be represented as a single port, the single port handle is passed all over the testbench.
- The signal declared in the interface can be passed across the modules/components.
- Any further changes in the DUT I/Os can be easily accommodated in the interface by addition or deletion of signals with any major changes in the testbench.

The interface is passed in the testbench as a virtual interface as it helps the classes to drive or change the values of the interface signals while the interface is not even being part of that class or module

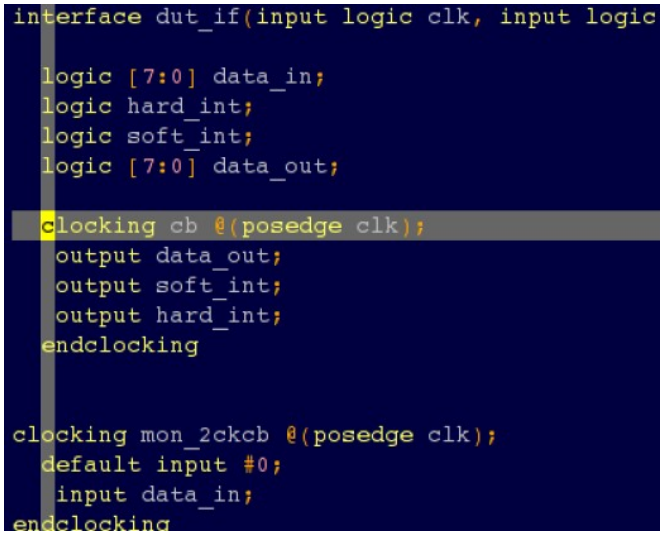


Figure 2.8 Snapshot of Interface of my IP

.

2.2.4 UVM SEQUENCE ITEM

The sequence-item is extending froma uvm_sequence_item[9][8][10] base class, uvm_sequence_item derived fromthe uvm_transaction class which further drove from the uvm_object. Sequence item consists of data fields required for data modeling.The purpose of extending the user-defined sequence_item from uvm_sequence_item is that all the functions such as print, copy, clone, etc. are predefined in the base class. The sequence items data fields are randomized from sequences with some constraints on the randomized fields thus leading to constraint randomized testbench. Therefore data fields in sequence item should generally be declared as rand and can have constraints defined.

```
class ip_sequence_item  extends uvm_sequence_item;
  rand  logic [7:0]  tx_data;// for data_in
  rand  logic [31:0]  w_data;// for bus protocol
  rand  logic [2:0]  chip_sel;
  rand  logic [1:0]  tx_fadd;
        logic [7:0]  rx_data;//data out
  // ---------------------------------------------
  // Constraints
  // ---------------------------------------------
  constraint tx_data { tx_data >= 8'h00; tx_data <= 8'h55; } //maximum data_in support by DUT
  // ---------------------------------------------
  // Register with uvm utils
  // ---------------------------------------------
  `uvm_object_utils_begin(ip_sequence_item )
    `uvm_field_int(tx_data         , UVM_ALL_ON+ UVM_DEC)
    `uvm_field_int(w_data        , UVM_ALL_ON)
    `uvm_field_int(rx_data         , UVM_ALL_ON)
    `uvm_field_int(chip_sel         , UVM_ALL_ON)
    `uvm_field_int(tx_fadd         , UVM_ALL_ON)
  `uvm_object_utils_end
  // ---------------------------------------------
  // constructor of class
  // ---------------------------------------------
  function new(input string name= "ip_sequence_item" );
    super.new(name);
  endfunction : new

endclass : ip_sequence_item
```

Figure 2.9 Snapshot of the sequence item of my IP

The figure shows the user-defined sequence_item class. All the inputs are declared rand, all the data fields also registered to the factory. So that the inbuilt functions can be used by the user-defined data fields.

2.2.5 UVM SEQUENCE

A sequence generates a series of sequence_item's and sends to the driver via sequencer, Sequence is written by extending the uvm_sequence[9][8][10]. A uvm_sequence is derived from an uvm_sequence_item and it is parameterized with the type of sequence_item, this defines the type of item sequence will send/receive to/from the driver.

```
class ip_base_seq extends uvm_sequence #(ips_transaction);
  `uvm_object_utils(ip_base_seq)

 logic [31:0] read_value;
 `uvm_declare_p_sequencer(ip_sequencer)
 function new(string name="ip_base_seq");
   super.new(name);
 endfunction
virtual task pre_start();
 if(!uvm_config_db#(ral_model)::get(get_sequencer(),"","reg_model",r
 `uvm_fatal("reg_model"," not able to get reg model in sequence lib"
 end
 else begin
```

Figure 2.10 Snapshot of uvm_sequence of my IP

2.2.6 UVM SEQUENCER

The sequencer[9][8][10] controls the flow of request and response sequence items between sequences and the driver.Sequencer and driver use TLM Interface to communicate transactions.uvm_sequencer and uvm_driver base classes have seq_item_export and seq_item_port defined respectively. User needs to connect them using TLM connect method. The sequencer can be written by extending the uvm_sequencer parameterized with seq_item type.

```
class ip_sequencer extends uvm_sequencer #(ip_sequence_item
  `uvm_component_utils(ip_sequencer)

  function new(input string name = "ip_sequencer", uvm_comp
    super.new(name, parent);
  endfunction : new
```

Figure 2.11 Snapshot of uvm_sequencer

2.2.7 UVM DRIVER

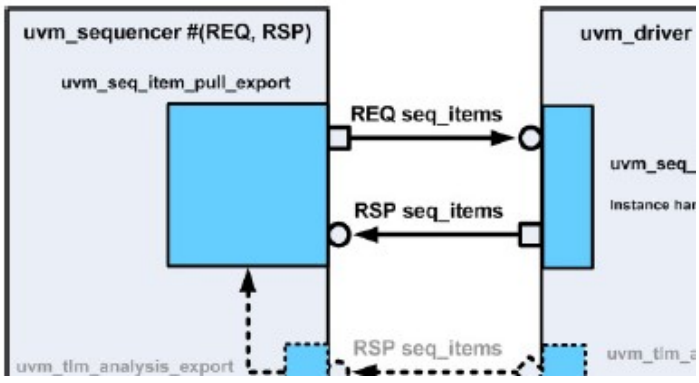


Figure 2.12 Connection between both driver and sequencer

The driver[9][8][10] is a block whose role is to convert the high-level code into BFM (Bus Functional Model). The driver can be used in a push or pull mode i.e. either the driver can demand sequence from sequencer or sequencer can push sequence to the sequencer, by default the driver works in pull mode. The transaction is observed and evaluated by another

uvm_component, the monitor, thus driver's task is just sending a transaction to DUT through an interface. The user-defined driver is extended uvm_driver which is further extendedfrom uvm_component. TLM port isused for communication between sequencer and driver as shown in the figure.The uvm_driver class is a parameterized class and it sequence_item as a parameter.

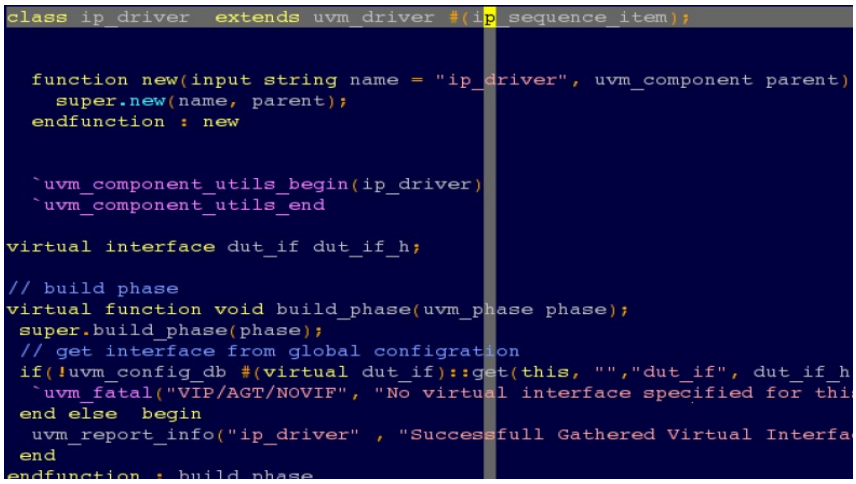


Figure 2.13 Snapshot of uvm_driver of my IP

2.2.8 UVM MONITOR

The user-defined monitor[9][8][10]is to observe the interaction of every transaction of the DUT with the Testbench.The purpose of the monitor the interface signals and, if the signals are not working according to the protocol, the monitor shouldthrow an error.The monitor is a passive part of the agent, as it used is to extract signal information and translate it into sequence-item form. Data collected by monitor send to the scoreboard to be evaluated. Each UVC (universal verification component) have its own monitor. Soall the monitors will collect transactions from the different virtual interface based on the UVC and use their own analysis ports to send those transactions to the scoreboard.

```
class ip_monitor extends uvm_monitor;
  ip_sequence_item config_data;
  uvm_event begin_listen = uvm_event_pool :: get_global("begin_listen"
  uvm_event stop_listen  = uvm_event_pool :: get_global("stop_listen");
  uvm_analysis_port #(ip_sequence_item) monitor_to_sc_port;

 `uvm_component_utils_begin(ip_monitor)
  `uvm_field_object(config_data,  UVM_ALL_ON)
 `uvm_component_utils_end

 virtual dut_if dut_if_h;

 function new(string name, uvm_component parent=null);
   super.new(name, parent);
 endfunction : new

 function void build_phase(uvm_phase phase);
 if(!uvm_config_db #(virtual dut_if)::get(this, "","dut_if", dut_if_h)
 `uvm_fatal("VIP/AGT/NOVIF", "No virtual interface specified for this a
 end
 else  begin
 uvm_report_info("ip_monitor" , "Successfull Gathered Virtual Interface
```

Figure 2.14 Snapshot of uvm_monitor of my IP

2.2.9 UVM AGENT

The user-defined agent[9][8][10] is extended from uvm_agent, which is further extended from uvm_component.An agent contains three component a sequencer, a driver and a monitor. An agent usually doesn't have a run phase, as there is no code in this block but there arebuild phase and connect phase.We will createthe sequencer, the driver and the monitor in the build phase. The connect phase may be used in Environment so as to push for more reusability of the UVC component. Same approach is used this testbench.

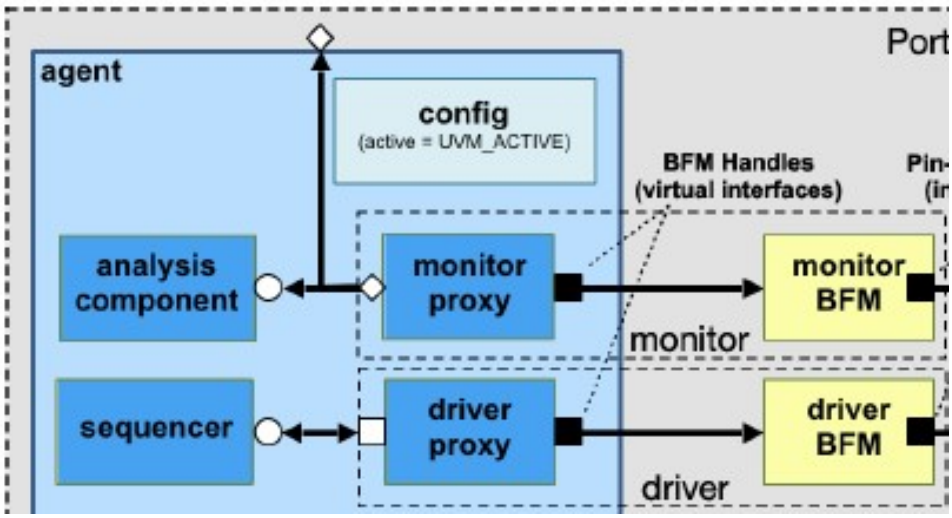


Figure 2.15 UVM Agent in active configuration

```
class ip_agent  extends uvm_agent;
  ip_driver                          ip_driver_h    ;
  ip_sequencer                       ip_sequencer_h ;
  ip_monitor                         ip_monitor_h   ;
  protected uvm_active_passive_enum          is_active      ;
  `uvm_component_utils_begin(ip_agent  )
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_AL
  `uvm_component_utils_end
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_report_info("ip_agent" , " In Build Phase " , UVM_HIGH
    ip_monitor_h   = ip_monitor::type_id::create("ip_monitor_h
    if (is_active == UVM_ACTIVE)
    begin
      ip_driver_h    = ip_driver::type_id::create("ip_driver_h
      ip_sequencer_h = ip_sequencer::type_id::create("ip_seque
    end
  endfunction : build_phase
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
```

Figure 2.16 Snapshot of uvm_agent of my IP

2.2.10 UVM SCOREBOARD

The scoreboard[9][8][10] is a crucial element in a self-checking environment, it verifies the proper operation of a design at a functional level. This component is the most difficult one to write, it varies from project to project and from designer to designer.In our case, we decided to make the prediction of the DUT functionality in the monitors and let the scoreboard compare the prediction with the DUT's response. But there are designers who prefer to leave the prediction to the scoreboard. So the functionality of the scoreboard is very subjective. In the agent, we created two monitors, as a result, we will have to create two analysis exports in the scoreboard that 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 testbench and the DUT both agree in the functionality and it will return an "OK" message.But we have a problem: we have two transaction streams coming from two monitors and we need to make sure they are synchronized. This could be done manually by writing appropriated write() functions but there is an easier and cleaner way of doing this by using UVM FIFO.

2.2.11 UVM ENVIRONMENT

The ENV[9][8][10]is a very simple class that instantiates the agent and the scoreboard and connects them together. The user-defined environment is derived from uvm_env, uvm_env is inherited from uvm_component. The environment is the container class, It contains one or more agents, as well as other components such as the scoreboard, top-level monitor, and checker.

```
class [redacted]_env extends uvm_env;
`uvm_component_utils(msc_env)

  function new(string name = "[redacted]_env",
    uvm_component parent=null);
    super.new(name,parent);
  endfunction : new

  reset_agent                    reset_agent_h;
  ip_virtual_sequencer          vs;
  ip_agent                      ip_agent_h;
  ral_sys_IP                    ip_reg_model;
  bus_ral_adapter             bus_adapter;
  ip_bus_sequencer              sequencer;
  ips_env #(virtual ips_if#(.IPS_ADDR_WIDTH(14))) ipsenv;

  virtual function void build();
    super.build();
    uvm_report_info(get_full_name(), "i am  in build of env",UVM_LOW);
    ipsenv = ips_env #(virtual ips_if#(.IPS_ADDR_WIDTH(14)))::type_id::create("ipsenv",this);
    vs                = ip_virtual_sequencer::type_id::create("vs",this);
    sequencer               = ip_bus_sequencer::type_id::create("sequencer",this);
    reset_agent_h     = reset_agent::type_id::create("reset_agent_h", this);
    ip_agent_h        = ip_agent::type_id::create("ip_agent_h", this);
    ip_reg_model      = ral_sys_IP::type_id::create("ip_reg_model");
    bus_adapter       = bus_ral_adapter::type_id::create("bus_adapter");
    ip_reg_model.build();
```

Figure 2.17 Snapshot of uvm_env of my IP

2.2.12 UVM TEST

At last, we need to create one more block: the test[9][8][10]. This block will derive from the uvm_test class and it will have two purposes: Connect the sequencer to the sequence and defines the test scenario for the testbench. We might be wondering why are we connecting the sequencer and the sequence in this block, instead of the agent block or the sequence block.The reason is very simple: by specifying in the test class which sequence will be going to be generated in the sequencer, we can easily change the kind of data is transmitted to the DUT without messing with the agent's or sequence's code. The user-defined test is derived

from uvm_test, uvm_test is inherited from uvm_component. The UVM testbench is activated when the run_test() method is called, the global run_test() task should be specified inside an initial block. The test case also contains uvm_table_printer and uvm_report_server.

The uvm_table_printer is used to print the hierarchy of the uvm component at the end of the elaboration phase. the hierarchy is can be printed in the tree, line or table form.

The uvm_report_server is used to process all of the reports generated by a uvm_report_handler in the report phase and generate the result at the end of the simulation by showing PASS/FAIL.

```
class ip_base_test extends uvm_test;
  `uvm_component_utils(ip_base_test)

  ip_env           ip_env_h;
  uvm_table_printer printer;
  uvm_report_server server;

  function new(string name = "ip_base_test",
    uvm_component parent=null);
    super.new(name,parent);
    printer = new();
  endfunction : new


  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
```

Figure 2.18 Snapshot of uvm_test of my IP

2.2.13 UVM TOP

The top block[9][8][10] will create instances of the DUT and of the test bench and the virtual interface will act as a bridge between them. 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.

2.2.14 Concept of Virtual sequence

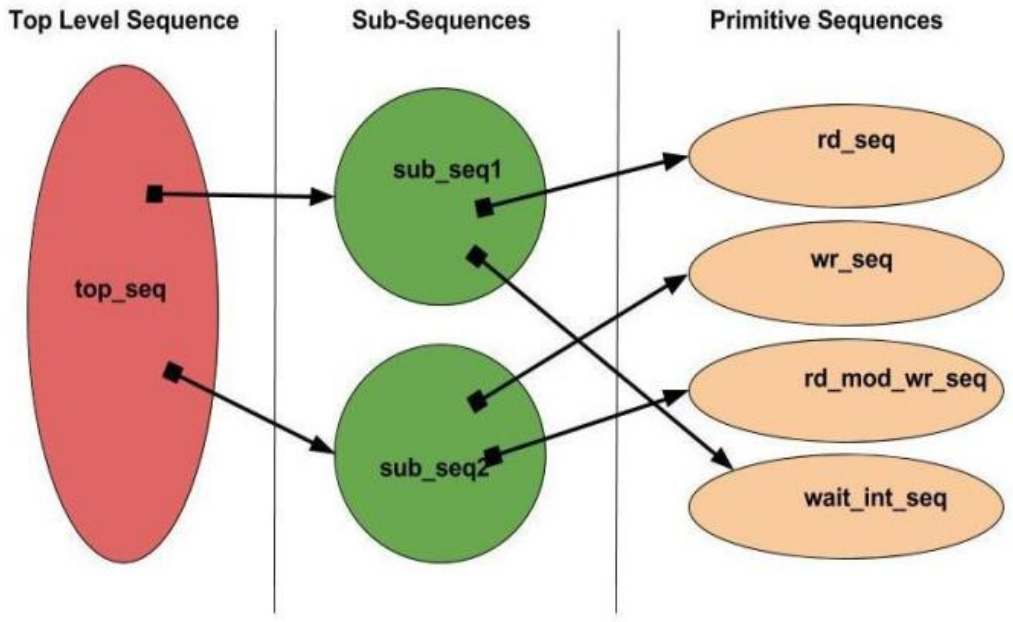


Figure 2.19 Virtual sequence Hierarchy

Suppose there are three layers of virtual sequences[9][8]. The primitive sequences are individual sequences extended from uvm_sequence. The top sequence and mid-level Sequences practically does not send any sequence_item to the testbench Driver. Their task is to set the trigger for relevant sequences or Primitive Sequences.A Top Sequence is also called Virtual Sequence. It synchronizes between the different Sub-Sequences. For example, in my DUT there are more than one interface ports, so there is the same number of agents for each interface port. Virtual Sequence determines which Agent's Sequence will start on the sequencer. Virtual Sequence acts like a Controller on the sub-cases that generate data for the DUT.

2.2.15 Concept of the virtual sequencer

As explained above every different sequence extended from different sequence_item are to be run on different sequencers[9][8]. The direct use of the sequencers in the testbench will lead to ambiguity as the number of base sequences increases thus leading to more errors and

less reusability. So virtual sequencer class is used which contains the handle of all the sequencers in the testbench. These handles are connected to the original sequencers in the connect_phase function of the top environment. Thus we need to declare only the virtual sequencer in the sequencer and we just probe the sequencer handler on which the sequence is to be run. This increases the reusability of the testbench. Suppose there are some changes in the DUT a new sequencer is to be added/removed it can be easily done by taking the handle of the sequencer in Virtual sequencer and connecting the sequencer in the connect phase of the top environment[9][8].

```
class ip_virtual_sequencer extends uvm_sequencer;
  reset_sequencer      reset_sequencer_h;
  buss_sequencer        bus_seqr_h;
  ip_sequencer        ip_sequencer_h;

  `uvm_component_utils_begin(ip_virtual_sequencer)
    `uvm_field_object(reset_sequencer_h, UVM_ALL_ON)
    `uvm_field_object(bus_seqr_h, UVM_ALL_ON)
    `uvm_field_object(ip_sequencer_h, UVM_ALL_ON)
  `uvm_component_utils_end

  function new (string name = "ip_virtual_sequencer", uvm_component
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH)
  endfunction : new
```

Figure 2.20 Snapshot of virtual_sequencer

2.3 The working TESTBENCH Block diagram for a testbench is shown below

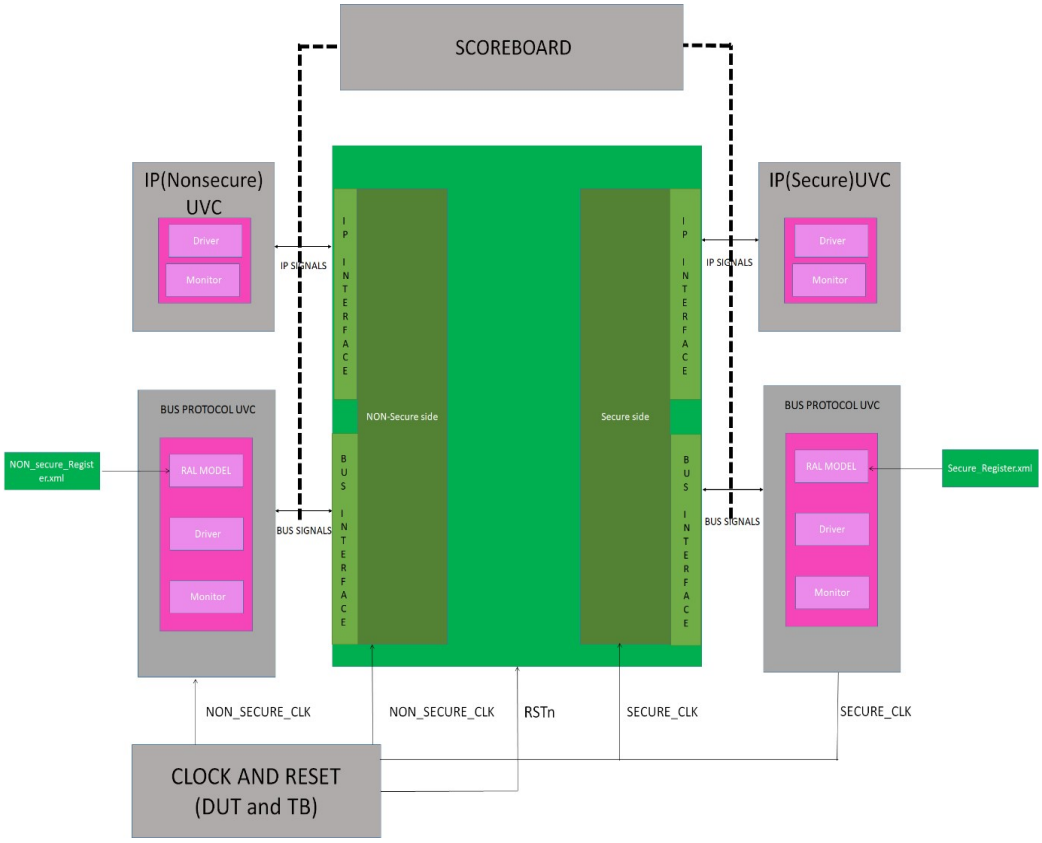


Figure 2.21 IP level testbench environment

The basic approach in the verification model to integrate the VIP i.e. Bus protocol UVC environment in the top environment of the Testbench along with other required components such as the environment for the IP signal interfaces. So basically in my testbench in the Top environment, 1 Bus UVC environment and 3 more agents are included that are a non_secure agent (containing the driver and monitor for non_secure interface), a secure agent (containing the driver and monitor for secure interface) and Reset agent (used to provide reset to the DUT and the TB).

Upon generating and simulating the testcases for a different scenario that regressively test functionality of the IP. We move past to negative testing it includes the functionality such as interrupt generation i.e. we provide IP wrong input and verify whether the interrupt is generated or not. There are many other functionalities that are needed to be verified so that we may now the limit of the IP where it fails. Suppose there is a FIFO in the IP of length 4 what will happen if the data is being continuously filed but not emptied out, want to know what will happen to data at last stage of FIFO

whether the data is overwritten or after the length the data is not being filed in the FIFO.

How will we know that the total code is verified?

This answer is given by coverage report of the DUT. The coverage report is generated by passing a switch to generate a coverage report while performing regression.

2.4 Coverage

The coverage report is classified as code and functional coverage.

2.4.1 Codecoverage

The code coverage is automatically generated by the tool and the report is shown to the user for the analysis. The code coverage can be divided into 5 categories.

2.4.1.1 Toggle CoverageIt gives the information about the change of signals and ports of the IP module and included sub-modules that are included in the coverage report. It measures activity in the signals, such as unconnected signals, signals that remain at single value throughout the simulation, or signals that change assumes values states then values it can assume.Block coverage: -It reports whether a particular block of code is being hit in the regression runs.

2.4.1.2 Branch coverage: -It reports whether individual branches in the block are hit in the regression runs.

2.4.1.3 Statement coverage: - It reports a number of the individual statements that are hit in the code during regression runs.

2.4.1.4 Expression coverage: -It reports when a conditional code was hit during the regression runs

2.4.1.5 FSM coverage: - It reports the state of the FSM that is hit during the regression runs.

2.4.2 Function Coverage

It is reported on user-defined coverage points, assertions, or covergroup statements. The function coverage basically covers all the functionalities of the IP. The Code coverage just give the idea that whether the certain code is executed or not but the function coverage gives report whether the functionality is verified the the verification engineer. Thus this coverage is to manually developed by the verification engineer.

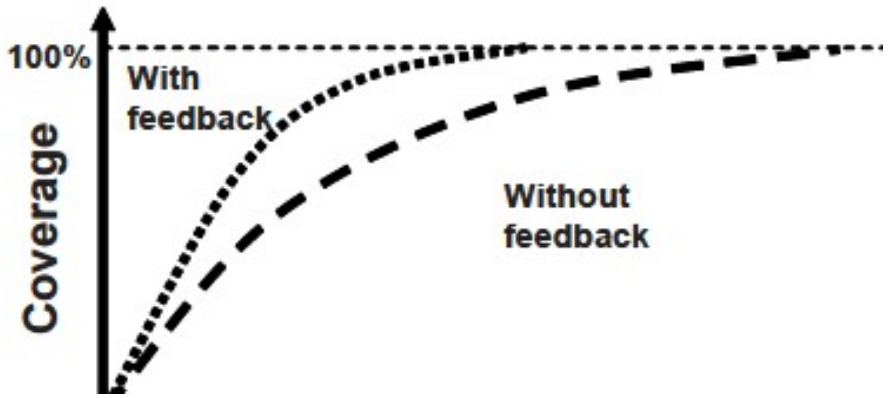


Figure 2.22 The comparision between coverage with and without feedback

The current functional coverage acts asa feedback[10] to the testbench code, with the feedback to the user, the testbench side sequences can be changed to increase functional coverage in the next iteration. Thus leading to newer scenarios leading to coverage-driven verification.

After the report is generated by the tool for both Code and Functional coverage, the user analyzes the report and make changes in the testbench code to hit the code and make refinement in a report by excluding the certain unnecessary ports/signals (with the insight of the designer). After all these processes that final coverage report is generated.

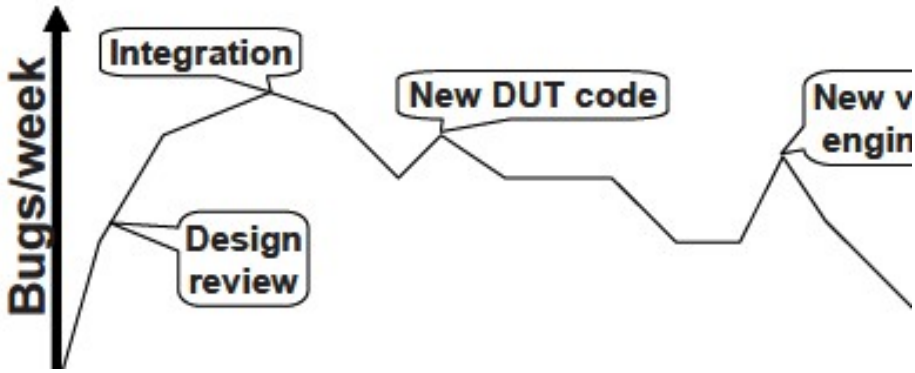


Figure 2.23 Life cycle of IP in verification

The figure shows the timeline of the IP from designing to verification plotted against the Bugs per week. This figureshows the timeline of the tape out of the IP.

3

IP ENVIRONMENT AUTOMATION

The environment used in verification is shown as shown in the figure.

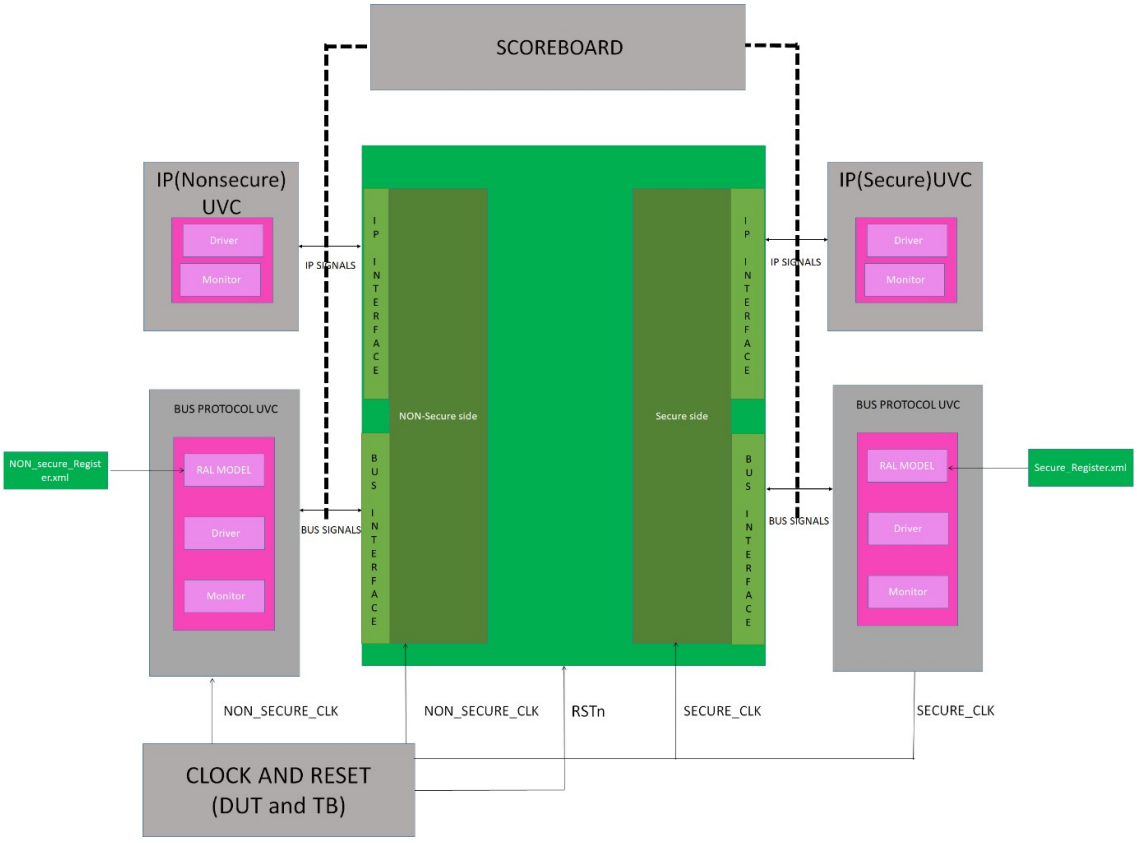


Figure 3.1IP level testbench environment

The environment usually contains [11] 3 UVC (Universal verification component): -

- Reset and clock UVC
- Bus protocol UVC
- IP specific UVC as shown in figure

As from the previous chapter, we know the first step of test-case development is register test-case development, the requirement of the register test-case development. For the register test-case development only two UVCsare required i.e. bus protocol UVC and reset and clock UVC.

3.1 Problem Statement

For the DUT verification the environment we need to create the testbench from the scratch i.e. from testbench top module. The creation of test-bench and including 2 UVC to do register test-cases took a lot of time and efforts, sometimes even takes 80- 100 man hours[11]so that basic reset register test-case can be done. Another problem was that every person uses his/her own method for creation of the testbench so it leads disparity between the flow.

3.2 Idea of Automation

The basic idea of automation was to create all the testbench so that it is capable of performing register test-cases with some/no user interference [11].

3.3 Testbench directory structure

The directory structure for the IP verification uses the standard structure with some more directories as vm_lib and vmanger according to the ST Microelectronics standard.

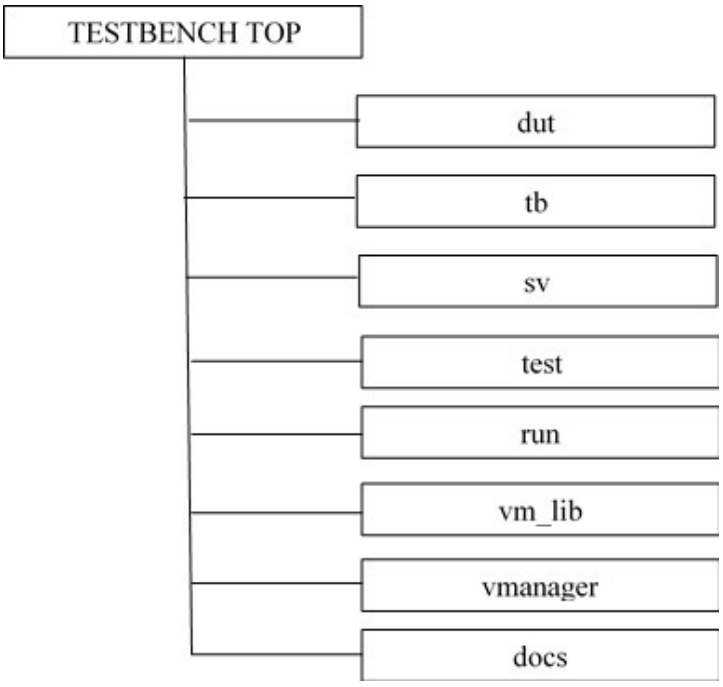


Figure 3.2 Directory structure of IP level testbench

3.3.1 *README:*This file gives details of all directories and steps to run the simulation and regression.

3.3.2 *Docs:*This folder contains a verification plan, design doc, and all other related documents.

3.3.3 *SV:*This folder has all the top level verification environment files. It has top-level environment. Virtual sequencer, scoreboard and configuration file.

3.3.4 *Vm_lib:*This folder has all UVCs sub-directory.

- Bus_protocol_uvc:
- reg_seq:

3.3.5 *Tb:*This folder contains testbench. We have instantiated STA1_MAILBOXHSM_APB in this file. The input signals which are needed to be initialized, are initialized here.

3.3.6 *Test:*This folder contains test cases to verify STA1_MAILBOXHSM_APB.

3.3.7 *Dut:*This directory contains all the design files.

3.3.8 *Run:*It contains all the run scripts which are necessary to launch the simulation. It also stores the simulation results (coverage data, waveforms, etc.).

3.3.9 *Vmanager:*This folder contains the regression setup and link of vplan.

3.4 Environment creation

The creation of the script based environment takes a little different approach to create a compile error free testbench so that the user can extend the testbench for functionality verification. The testbench top directory is the name of IP suffixed by the "_tb". In testbench, all the sub-directories for the IP testbench are created.

3.4.1 Creation of the testbench top module

The testbench top module has the following things: -

- Contains the instance of the VIP UVCs, a virtual interface, reset and clock UVC, DUT.
- The instantiation of all the signals in the DUT is also done in the testbench top module.
- Connection of the interface signals to both VIP and the DUT signals
- The initial block which calls the particular test using run_test
- The initial block used to setting the handle interface, reset and clock UVC for all the testbench.

This all the process used to be done the user him/herself.

In the automation, we use the ports.XML and register.XML provided by the RTL Designer, port XML is used to create the testbench top, the ports.XML contains all the information about the top signals and parameters of the DUT top. The information provided by the designer is extracted from XML and presented it in the Testbench top.

```
wire clk_ip;
wire [`MAX_WADDR-1:0] addr
wire sel_ip;
wire enable_ip;
wire write_ip;
wire [`MAX_WDATA-1:0] wdat
wire [`MAX_WSTRB-1:0]  str
```

Figure 3.3 wire instantiation of the bus protocol interface

The signals of the DUT top is initiated in testbench with the same name by declaring wire.

Then these signals are connected to the DUT.

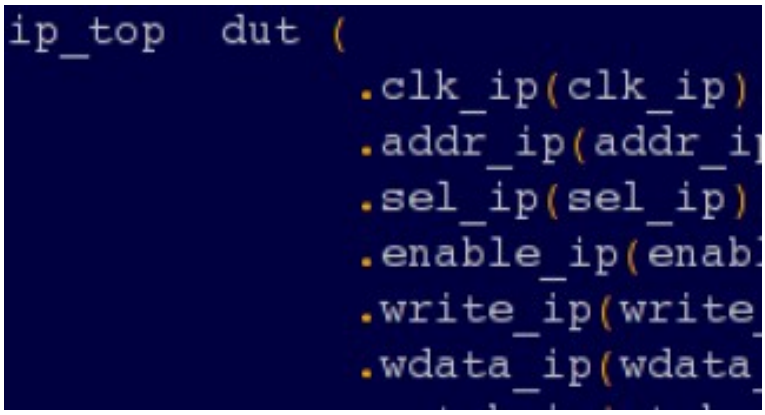


Figure 3.4 Instantiation of Dut signals

Along with this, the virtual interface is created which remains the same for a particular bus protocol. Then the names of that signals are taken from the user in config_file is used as the input to the script. The user specifies the signals corresponding to the interface so that they can be connected the tb_top. The frequency of the clock is also taken from the user in the config_file so that it can be passed to the clock and reset block while initiation in the top module. Then the handle of the interface and the clock and reset block is set in the initial block along with the run_teat function for the calling of the test.

3.4.2 Generation of the RAL model

The RAL model files are created from the ralgen tool[4]. The ralgen takes register.xml as an input, the XML file's path is taken in config_file in the starting. Before running the ralgen command the ralgen tool is sourced on the terminal by the script then the command for the ralgen is passed to the terminal. As a result, the ralgen generate the ral_model.sv file in a specified path.

3.4.3 Generation of all SV files

All the files that are used in the IP register test-case development are generated by the script in the sv folder according to the testbench top signals and the virtual interface which is passed by the testbench top.

3.4.4 Inclusion of the RAL model

The RAL model file handle is created in the sv files so that RAL model can be used on the bus protocol UVC. The RAL model files are initiated in the top_env then is set for all the environment so that can be used by hierarchical sv files.

3.4.5 Creation of the reg_seq_lib

The reg_seg_lib is created in the vm_lib sub-directory the reg_seq_lib contains all the predefined and some user-defined sequences that are used by the ral model for the register test-cases.

3.4.6 Creation of bus protocol UVC

The bus protocol VIP depending on the bus protocol is either linked/imported in testbench top (for AMBA-based bus protocols) or downloaded from the sync.

3.4.7 Test directory creation

The test classes are created for every register test-cases that are extended from the base test class that contains uvm_printer and uvm_reporter instances for the reporting of the pass/fail of the test.

3.4.8 Vmanger

As the vmanger contains the scripts that are used to run regression on the vmanger. The other script developed by me also included streamlining the regression setup. The regression setup includes '.group' the file that contains the names of all the test that are to run in regression which earlier was hand-made by the user. But the included script takes the vplanner.csv as input and create the '.group' so that user has to update the test-cases only in one place i.e. vplan and the all the testbench aligns itself to vplan.

3.4.9 Run

The run folder contains various run scripts to run the standalone test-cases in IP verification.

After all this work the script generates a compile error-free environment for the user.

3.5 Advantages of the scripts

- The script cuts short the 80-100 man hours for environment creation to 10-15 mins. Thus reducing the manually efforts 5-10 hours to perform register test-cases.
- The script itself includes the VIP and generate the RAL model and include both of them in the testbench on its own.
- The script provides compile error-free environment for the user.
- The script aligns all the different testbench to one hierarchical flow.
- As we know the reduced verification time means reduced cost of verification thus reducing cost and increasing profit.

4

RESULTS AND FUTURE SCOPE

4.1 Work Done

- The verification of the communication IP was done in the internship tenure. The verification document was made that contained the basic idea of the testbench environment along with it the verification plan was prepared that contained the categories and the number of testcases that were to be developed. After planning the next step included the building of the testbench environment hierarchy from the scratch followed by the inclusion of the VIP and RAL model. The IP testcases were developed for the verification of the functionality of the IP. The IP testcases included the following testcases:
    - Register testcases:- these are the basic testcases that used to check the basic read/write access of the register set
    - Functionality testcases:- these testcases contain the verification of the functionality of the IP
    - Negative testcases:- the negative testcases that used to stress the IP by sending the wrong input and expecting the error from the IP
- Along with the development of the testcases, certain bugs were found in the IP functionalities that were reported to the designer.
- The functional coverage was written for every necessary functionality that was written in the block guide of the IP.
- The coverage report was generated for the code coverage by the tool by running all the testcases in the regression with randomized values and the coverage was reported to the curtained IP RTL designer.
- After the Bugs were cleared by the designer, the coverage report was shown to the designer and then with the acknowledgment of the designer certain exclusions were made to coverage report. These exclusion are saved in the refine file.

- Then 100% coverage is achieved with the applied refine file.

4.1.1 Screenshot of the simulation and results

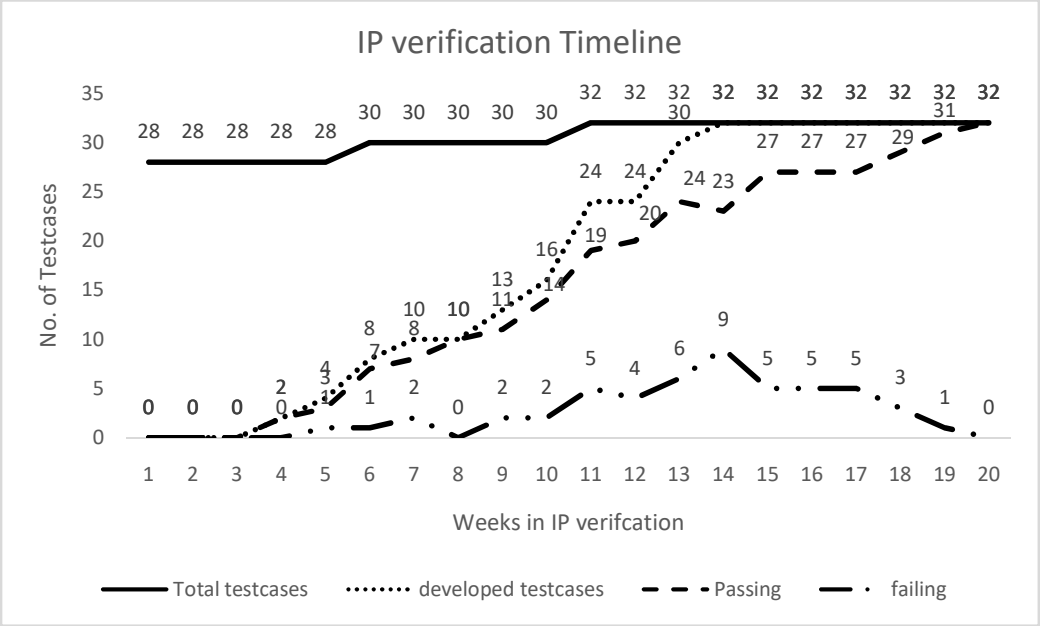


Figure 4.1 Testcases vs weeks in IP verification

The above figure shows the timeline of the IP verification. It shows the graph between the weeks an the total no. testcases. As the figure shows through week 1 to 3 no testcases were developed, week 1 is used to read the blockguide and the integration guide provided by the designers. In the week two verification plan and verification methodology document were created along with the creation of the IP environment that was carried to week 3. From week 4 test-case development started with the register test-cases. The bugs were caught and were reported to the designer then in week 15 all the proposed testcases were developed and failing testcases were due to bugs found in the verification. Then in the week 16-17, the functional coverage was written. In week 18, the bugs were fixed and some changes were made by the designer testcases were modified and all the testcases were passing in the standalone mode in week 20.

After week 20 the regression was launched with the coverage enabled and the coverage was observed and the final code coverage 100% were achieved with some exclusion. The failing

testcase waveform is the approximate match of the figure 2.23 that depicts the life cycle of IP verification. Hence the simulation results matches the theoretical expectations.

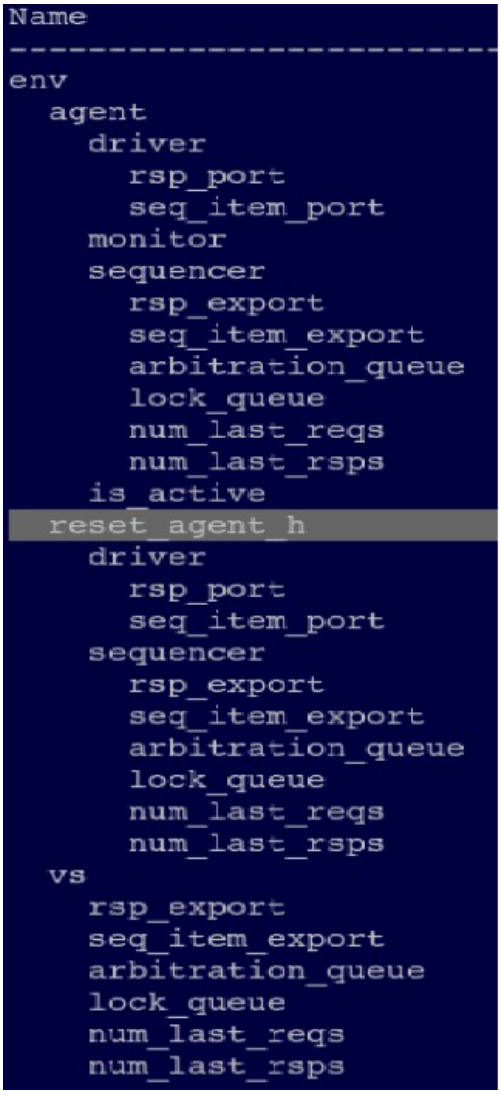


Figure 4.2 Hierarchy of simulation IP

The figure shows the hierarchy of the IP which printed by the uvm_printer at the end of the elaboration phase. It shows the hierarchy of all the components that are built during the build phase.

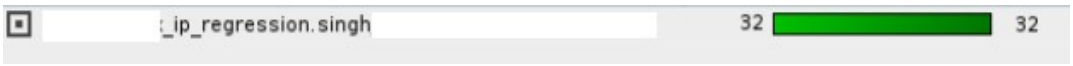


Figure 4.3 Regression results

The above figure shows the regression that was launched with all test-cases that were developed for the IP verification of the communication IP.

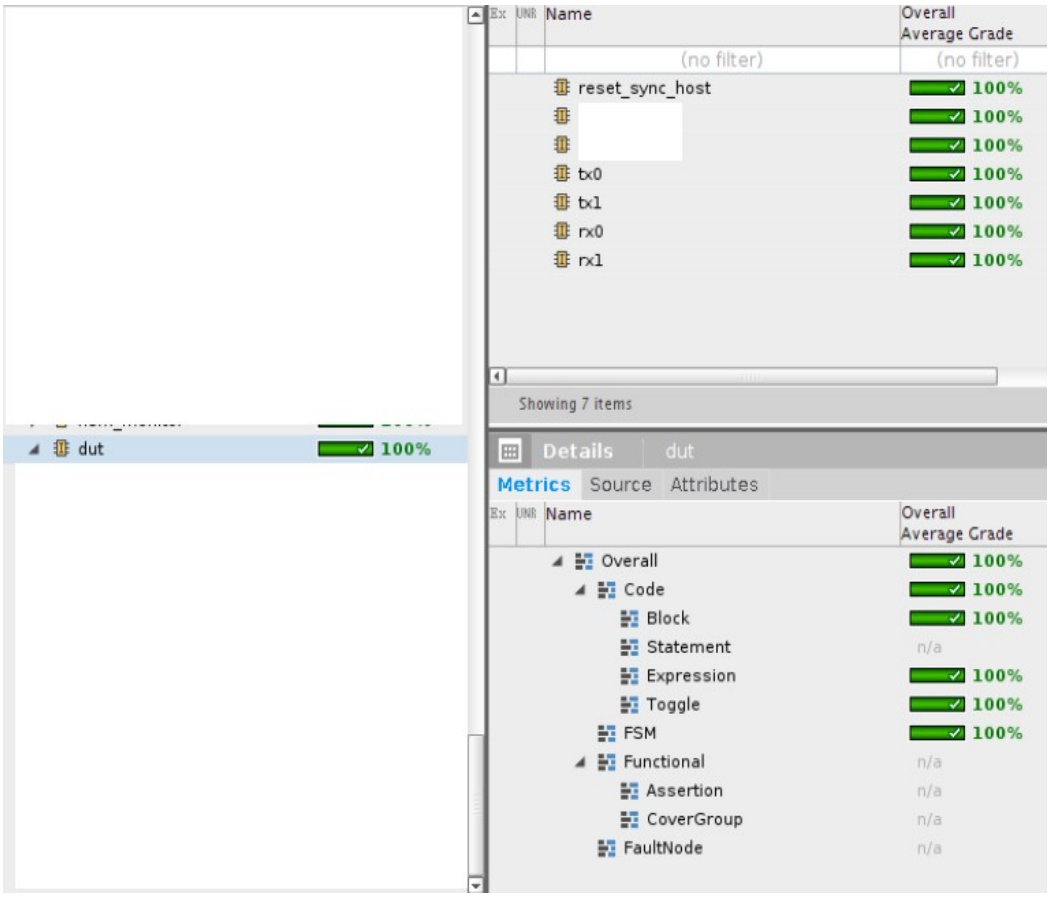


Figure 4.4 Coverage of IP level verification

The regression was also launched with the coverage enabled, the results of the coverage that are shown in the above figure are achieved by applying some exclusion through refine file.

The results of the automation of IP verification environment are tabulated below

Table 1 Comparision between the automated environment and manual environmrnt generation time

Environment	Automated Environment	Manually generated
Input format creation time	15 mins	Not applicable
TB TOP generation	-	4-5 hours
Verification Env. Creation	-	15-20 hours
Compilation error-free environment	-	10-15 hours
Some adjustments	1 hour (if any)	5 hours
Basic register testcase	5 mins	1-2 hours

As shown in the table the time for generating the automated environment is drastically decreased.

4.2 Future Scope

The verification nowadays is just meant to verify the designs for their functional correctness. But with the changing time, the new techniques are developed in which real-life scenario will be run as simulation so to judge the working of the chip in real life. For e.g. there is a digital twin for every accident that is occurring in Germany which will be used in verification of the chips that will be used the upcoming vehicles so as to eliminate/reduce the mistakes that were committed by the previous chips. It would correct to say that the chips will be verified the mistakes that have been committed by previous generation chips.

4.2.1 IP Verification

In the IP verification, the IP environment lacks scoreboard as all the checking is done in the sequences and the monitor so the scoreboard will be added into the testbench, this will be done along with the release of next version of IP.

In further IP verification for different IP a parameterized architecture will be used to increase the reusability of the various testbench UVCs.

4.2.2 IP Environment Automation

In the IP environment automation is currently supported for a few Bus protocol so the automation will be extended for different Bus protocol. The automation of the IP environment now only supports bus protocol signal, it will be extended to support some generic functional port of the IP.

In the next, the release of the automated environment the all parameterized class will be used. This will further increase the reusability of the UVCs that are developed by different users.

5

References

[1] Y.-J. Oh and G.-Y. Song, “Simple Hardware Verification Platform using,” 2011.

[2] I. C. Society, “IEEE Standard Verilog Hardware Description Language”.

[3] “Testbench.in,” 2015. [Online]. Available: http://testbench.in/SV_00_INDEX.html.

[4] G. Allan, M. Baird, R. Edelman, A. Erickson and M. Horn, Universal Verification Methodology UVM Cookbook, Mentor ASiemens Business.

[5] I. C. Society, “IEEE Standard for Universal Verification Methodology Language Reference Manual,” 2017.

[6] CHRIS SPEAR Synopsys, Inc., SYSTEMVERILOG FOR VERIFICATION A Guide to Learning the Testbench Language Features, Springer, 2008.

[7] ARM, “AMBA 3 APB Protocol,” 2004.

[8] K. Shimizu, “Cluelogic,” 2016-2017. [Online]. Available: http://cluelogic.com/category/uvm/.

[9] J. Francesconi, J. A. Rodriguez and P. M. Juli´an, “UVM Based Testbench Architecture,” 2014.

[10] W. Ni and X. Wang, “Functional Coverage-Driven UVM-based UART IP Verification,” 2015.

[11] R. Srivastava, G. Gupta, S. Patankar and N. Mudgil, “Automatic Test Bench Generation and Connection in Modern Verification Environments:Methodology and Tool,” 2013.

[12] J.-Y. Park, S.-J. Lee and K.-S. Chung, “A Novel SoC Platform Based Multi-IP Verification and Performance,” 20008.

[13] Y.-N. Yun, J.-B. Kim, N.-D. Kim and B. Min, “Beyond UVM for Pratical SoC Verification,” 2011.

[14] S. K. Mohanty, S. Sengupta and S. K. Mohapatra, “Test Bench Automation to overcome Verification Challenge of SOC Interconnect,” 2015.

[15] H. Zhaohui, A. PIERRES, H. Shiqing, C. Fang, P. ROYANNEZ, E. P. SEE and Y. L. HOON, “Practical and Efficient SOC Verification Flow by Reusing IP Testcase and Testbench,” 2012.

[16] T. W. B. Silva, D. C. Morais, H. G. R. Andrade, F. C. A. Nunes, E. U. K. Melcher1, A. M. N. Lima and A. V. Brito, “A Distributed Functional Verification Environment for the Design of System-on-Chip in Heterogeneous Architectures,” 2018.

[17] G. S. Rashmi VS and S. Bhamidipathi, “A methodology to reuse random IP stimuli in an SoC functional verification environment,” 2015.

Hardeep Singh

ORIGINALITY REPORT

11%	7%	6%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	colorlesscube.com Internet Source	3%
2	en.wikipedia.org Internet Source	1%
3	www.verificationguide.com Internet Source	1%
4	Submitted to CSU, San Jose State University Student Paper	1%
5	Pedro Araújo. "Development of a Reconfigurable Multi-Protocol Verification Environment Using UVM Methodology", Repositório Aberto da Universidade do Porto, 2014. Publication	1%
6	staff.ustc.edu.cn Internet Source	1%
7	www.vlsiencyclopedia.com Internet Source	<1%

Rohit Srivastava, Gaurav Gupta, Sarvesh

8	Patankar, Nandini Mudgil. "Chapter 34 Automatic Test Bench Generation and	<1%

8	Patankar, Nandini Mudgil. "Chapter 34 Automatic Test Bench Generation and Connection in Modern Verification Environments: Methodology and Tool", Springer Nature, 2013 Publication	<1%
9	Ashok B. Mehta. "ASIC/SoC Functional Design Verification", Springer Nature, 2018 Publication	<1%
10	Submitted to Institute of Technology, Nirma University Student Paper	<1%
11	Submitted to VIT University Student Paper	<1%
12	research.ijcaonline.org Internet Source	<1%
13	Juan Francesconi, J. Agustin Rodriguez, Pedro M. Julian. "UVM based testbench architecture for unit verification", 2014 Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA), 2014 Publication	<1%
14	www.openfoam.com Internet Source	<1%
15	Submitted to Rochester Institute of Technology Student Paper	<1%

16	www.inderscienceonline.com Internet Source	<1%
17	Submitted to Loughborough University Student Paper	<1%
18	www.specman-verification.com Internet Source	<1%
19	N. Z. Muhammad, A. Harun, S. A. Z. Murad, A. B. Jambek, M. N. M. Isa, S. N. Mohyar, R. C. Ismail, H. F. Hawari. "RTL platform validation of digital intellectual property (IP) of real time clock (RTC) for customized wireless microcontroller unit", AIP Publishing, 2018 Publication	<1%
20	Marcela Šimková, Zdeněk Přikryl, Zdeněk Kotásek, Tomáš Hruška. "Chapter 12 Automated Functional Verification of Application Specific Instruction-set Processors", Springer Science and Business Media LLC, 2013 Publication	<1%
21	Submitted to Charotar University of Science And Technology Student Paper	<1%
22	Submitted to Yonsei University Student Paper	<1%