Verification of Ethernet based IP/Subsystem in Smart NIC

Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
In Electronics & Communication Engineering
(VLSI Design)

By

Mit Patel
18MECV13

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad - 382 481
May, 2020

Verification of Ethernet based IP/Subsystem in Smart NIC

Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
In Electronics & Communication Engineering
(VLSI Design)

By

Mit Patel
18MECV13

Internal Guide:
Dr. Usha Mehta
Professor,
Institute of Technology
Nirma University

External Guide:
Manan Desai
Engineering Manager,
Intel Technology India Pvt Ltd.

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad - 382 481
May, 2020

Declaration

This is to certify that

1. The thesis comprises my original work towards the degree of Master of Technology in VLSI Design at Nirma University and has not been submitted elsewhere for a degree.

2. Due acknowledgment has been made in the text to all other material used.

Mit Patel
18MECV13

Certificate

This is to certify that the project entitled **"Verification of Ethernet based IP/Subsystem in Smart NIC"** submitted by **Mit Patel (18MECV13)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design, Nirma University, Ahmedabad. The record of work carried out by his under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. Usha Mehta
Internal Guide

Dr. Usha Mehta
PG Coordinator - VLSI Design

Dr Dhaval Pujara
Head, EC Dept.

Director
Institute of Technology

Date :

Place : Ahmedabad

To whomsoever it may concern

This is to certify that **"Mr Mit Pareshbhai Patel (18MECV13)"**, a student of MTech in VLSI Design from **"Institute of Technology, Nirma University"** worked in **"Intel Technology India Pvt. Ltd."** as an intern during **3rd June 2019 to 22nd May 2020**. During this period, he was found regular and had done his project on **"Verification of Ethernet based IP/Subsystem in smart NIC"**, under my supervision.
He has worked with utmost dedication and high level of engineering and analytical competence. We wish him all the best for his future endeavors.

External Guide:
Manan Desai
Engineering Manager
Intel Technology India
Bangalore
Date :

Place : Bangalore

Acknowledgment

"Learn new things is an opportunity, but a chance given to implement what you have learned is a bigger opportunity." For me, developing and evolving of this project has been very critical and joyful throughout out a year. The inner gratification for the accomplishment of the project succesfully would not be complete Without mentioning organization and mentors that helps me to complete this project work throughout year. Their support and motivation made my efforts fruitful.

Foremost, I extend my deepest gratitude to "Manan Desai (Manager)" and "Himanshu Rawal (HiringManager)" for giving me the opportunity to work with their group and guiding me for this challenging project.

I do sincerely appreciate my teammates "Avni Patel", "Heena Mankad", "Sharvil Desai" and "Mohit Rana" for their constant assistance, support and constructive suggestions in the betterment of this project, without which this would not have been possible.

I cordially extend my profound gratitude to thesis committee: "Dr. Usha Mehta" and "Dr. N.M. Devashrayee", for their motivation, valuable advises during project work.

Last but not the least, I would like to take this opportunity to thank my family, friends and colleagues for their help and valuable suggestions from time to time.

Mit Patel
18MECV13

Abstract

Verifying system is very important aspect from the cost perspective, time to market is greatly impacted by it. As silicon-chip is becoming more complex, nowadays it is very demanding that we achieve completeness in verifying design on or before desired time. There are two categories for verification process - the first one is tool based process and the second one is different methods of verification. Demand for proper guidelines of different processes and methods are escalating to achieve good quality of verification. What need to be done is about processes and how to be done is about methods.

During my project work industrial process are followed for verification of single module in the IP. Developing robust, reusable and highly configurable structure is described here. Important aspects and key points while following process of verifying design is described here thoroughly. Impact of configurations provided in the module level structure on the subsystem level structure is explained. Design behaviour under different excitation is studied thoroughly as prerequisite of the verification process. Highly configurable verification structure is developed, to make it reusable. To cover all the corner cases for module, testplan is prepared and maintained the verification process. Also, functional coverage plan is prepared to ensure required coroner cases are covered in the verification process.

During project work, I studied important aspects of subsystem that is mainly focused during verification process. At this level, mainly top level interface related coverage are coded, other required coverage is collected from sub-block. For subsystem coverage tracking HVP is maintained, also all coverage are coded using proper methods, so that only required coverage can be enabled keeping others disabled. To achieve verification closure we need coverage numbers up to the marks, for that we run regression on weekly basis to generate high randomization. This repetitive work is automated using perl automation script. That can generate report for better bucketization of errors and failures. In the later part of thesis sim profiling is described for improving real time of simulation. Thesis also shows different performance testing scenarios generated for the subsystem to eliminate performance bottlenecks.

Table of Contents

List of Figures

List of Abbreviation

NIC	- Network Interface Controller
OSI	- Open System Interconnection
PCIE	- Peripheral Component Interconnect Express
UVM	- Universal Verification Methodology
RTL	- Register-Transfer Level
IP	- Intellectual Property
SoC	- System on Chip
DUT	- Design Under Test
TLM	- Transaction Level Modeling
CG	- Cover Group

Chapter 1

Introduction

This chapter comprises of brief introduction about the project. It includes basic information about smart NIC and Ethernet subsystem. Motivation and objectives of the project are briefed here. During this project verdi tool of Synopsys is used for debugging tests and coverage. Fundamental and some advanced features of verdi are briefed in this chapter. In thorough verification flow, seven levels of verification are critical, these seven levels are discussed at the end of the chapter.

1.1 Ethernet System Architecture

Figure 1.1.1 shows high level Ethernet IP architecture.
It comprises of mainly three subblocks - Address translation unit, Packet processing and Mac Ports. PCIE interface is used at the one of the end points. PCIE is explained in the following chapter. Basically, host interface acts as entry point of the system. PCIE is connected with personal computer on the other side. Any packet coming from personal computer is entered in the ethernet from this host interface. Then packet is sent to packet processor, where all the processing related to the header is done. If we enable address translation unit, then I/O virtualization is applied to the packet address. Then packets are sent tho physical layer. MAC ports provide connection to the physical layer. It may possible that in some case, speed of processing of every block could not be synced, to overcome that FIFO is used at the input and the output side of the subblocks. This ethernet system is a part of networking card, used in personal computer to interact with the network of the computers.

1.2 Motivation

The main motivation for this project is to understand industrial verification flow in detail, learning of various tools those are used to debug failures, understand and prevent the additional factors that affects while verifying the design. UVM - Universal Verification Methodology - is currently the most advanced and broadly used verification methodology in verification domain, to work on UVM is also one of the motivation for this project.

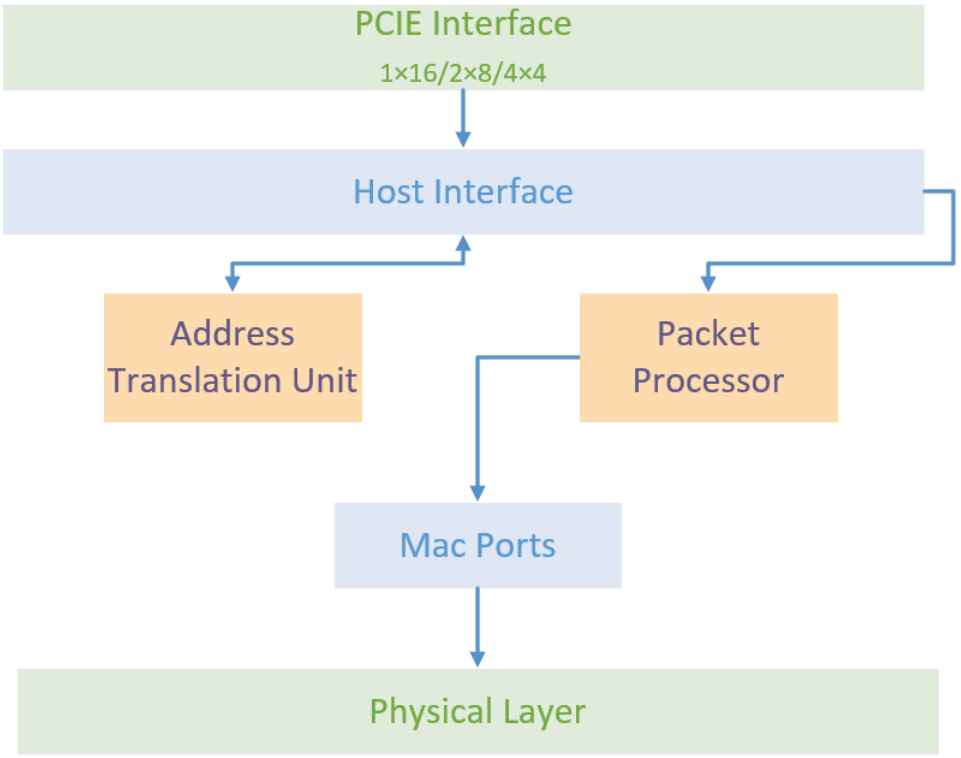


Figure 1.1.1: Ethernet Subsystem

1.3 Objective

The core objective of this project is to carry out functional analysis of design under test to make sure that system will behave correctly as per the given input output relation. It aims to plan and perform thorough functional verification and functional coverage of a system under test. To achieve above goals verification base structure is to be developed with properties like re-usability, effectiveness and robustness for design under test and to be reused in subsystem verification environment.

1.4 Synopsys Debug Tool Verdi [8]

The verdi is a debug tool of Synopsys. It enables in-depth debug for deisgn and verification. It simplifies complex and time consuming debugging process and merge diverse design and verification environment. Verdi uses powerful technology which comprises complex and queer design behavior. [8]
Basic debugging features provided by synopsys verdi:

- To analysie different values at different simulation time, we can dump fsdb file in verdi
- Verdi provides waveform comparision for fast signal database (FSDB) file.

- We can easily trace hierarchy of the signal
- Schematics and block diagrams displays logic of the design and connectivity between blocks.
- We can easily study and understand the operation of the finite state machine using intuitive bubble diagram.

The Verdi provides some advanced debug features also:

- We can automatically trace signal transitions across the clock cycles. That helps in the signal analysis.
- We can generate time-structure combined diagram using verdi, to analyze cause-effet relationship.
- We can debug assertion failure easily in the verdi.
- We can annotate any time stamp in the verdi, to make debug quicker.

SystemVerilog Testbench debug with:

- We can debug even test bench environment with verdi, as it supports full source code. Verdi supports UVM libraries, too.
- We can easily understand the testbench environment, as verdi allows to navigate through all classes, it allows tracing of class relationships and inheritance.
- We can record transactions of any signal. That helps us to understand post simulation verification environment. [8]

1.5 Verification process for different Levels

For some portion of the SoC design, third party source are reused. The major issue with reusing third party source is quality. Hence quality is critical for any intellectual property. Configuarability makes corner cases verification very important. With increasing configuarability of the IP, complexity also increases, which makes quality even more critical for the IP. Thorough verification is very important for the IP, because late bug reporting causes delay in tapeout, which affects time to market window. Verification process for different levels are listed here [6]:

1. Unit-level - This is very critical foundational level, that deals with units and elemets which re root parts in any configurable IP. Verification at this level is very critical, as bugs found later at this level will cause problems at subsequent levels.
2. Module-level – This is user configuration level, based on requirements of the performance and topology configurations will be implemented in the SoC. At this level, IP should be compatible with other functionality of the design. Vendor should verify it first, and results are to be made available for customers.

3. SoC-level – At SoC level, drift verification is done. Drift verification means, vendors verify the SoC release by release.

4. Design flow tool verification – Tools used to configure the IP are verified at this level. Tools can be controlled by either command-line or graphical-user interface or both.

5. Interoperability – Results for AMBA, AXI and other protocols are demonstarted by vendoors at this level. Also, intergartion of EDA tool and verification IP is demonstrated.

6. Customer module and SoC verification – Occurs during all the phases like design, tapeout, debug etc. IP should be implemented, integrated and verified at this level.

7. Customer, system-level user and quality experience - system house and end-customer system life cycle testing are done at this level.

Chapter 2

Literature Survey

Internet comprises of seven different layers, all the seven layer together is called OSI model. Data-link layer includes Ethernet. This chapter gives brief about the OSI model and how packet is formed in different layer. Comparison between ”the current setup where we plug everything into the router” and ”the old setup where long co-axial cable was used” is also discussed. PCIE link is used between Ethernet subsystem and computer. This chapter includes different generation wise transfer rate and throughput supported by PCIE link. Few SoC Verification challenges, process variation testing, some pre-requisites to reduce the verification challenges is discussed here briefly. In the last part of the chapter basics of UVM, UVM inbuilt class hierarchy and standard UVM verification environment is briefed.

2.1 Fundamentals of Ethernet [3]

OSI stands for “Open System Interconnection”. Telecommunication and computing system are characterized and standardized by OSI model. Inter operability of various communication system using standardized communication protocols is the major goal of OSI model. OSI model divides any system into seven abstract layers. Figure 2.1.1 shows layers of OSI-model.

So, Internet is made up of seven different layers. Data Link layer of the OSI-Model includes Ethernet. Back in 1980s, one long co-axial cable was used to connect all systems. With that kind of the setup only one device could send data over a network at a time, so the system had to be designed around that because in those days, computing hardware was costly. When a computer in the network wanted to send some data, it first had to send a special set of bits on the network, which basically equates to “I want to send something, please be ideal”. This initial sequence of bits is called the preamble. If two devices, at the same time, send the preamble bits then they would detect this conflict and wait for a random amount of time before trying to send data again.
Device can send any information if and only if it has sent preamble successfully. Information that could be sent over network is limited, in order to ensure that every device gets a chance. If any device wants to send more information, then that data should be divided into multiple parts and then those smaller data can be sent over the network. These smaller pieces of the data is called ethernet frame. In this communication of the data, target device should he located

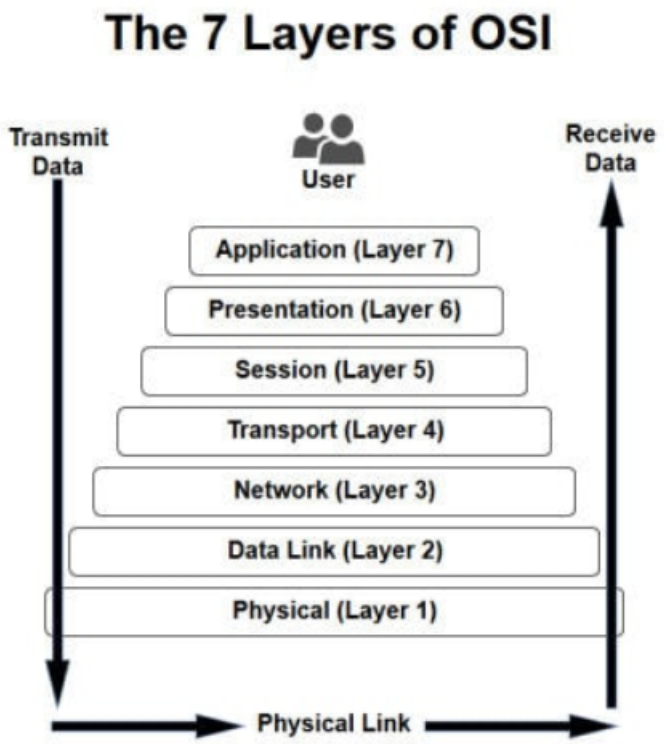


Figure 2.1.1: OSI Model [9]

correctly and carefully, so that information reaches to the correct target device. To locate the target one unique number is used with every devices connected in the network. That unique number is called MAC(Media Access Control) address. This MAC address is unique for any device. It is assigned with rhe vendors. After sending preamble bits, pause bits are sent, theses pause bits are called start delimiter. After the delimiter target MAC address and source MAC address are sent. If data is to be sent to all devices as broadcast, then target MAC address is not sent. After MAC addresses ether type is sent. This is a bit tricky information. It serves two purposes. If the number in this field is less than or equal to 1500, then it is total length of the ethernet frame. If the number in this field is greater than or equal to 1536, the it is type of data contained in the ethernet frame. Nowadays it is used to indicate content of the frame rather than length. After this information checksum is sent. Checksum verifies that data is not corrupted in the transport. These all fields together called ethernet frame. After 12 silenced bits, again information can be sent. If we compare this with present scnario, then it is pretty similar. Today we have intelligent switches and routers to transfer multiple data at the same time, throughout these year ethernet standards are pretty much unchanged.

2.2 Standards of PCIE

PCIE stands for “Peripheral Component Interconnect Express”. It is used to connect personal computers to Ethernet hardware. PCIE is a standard for high speed serial computer expansion

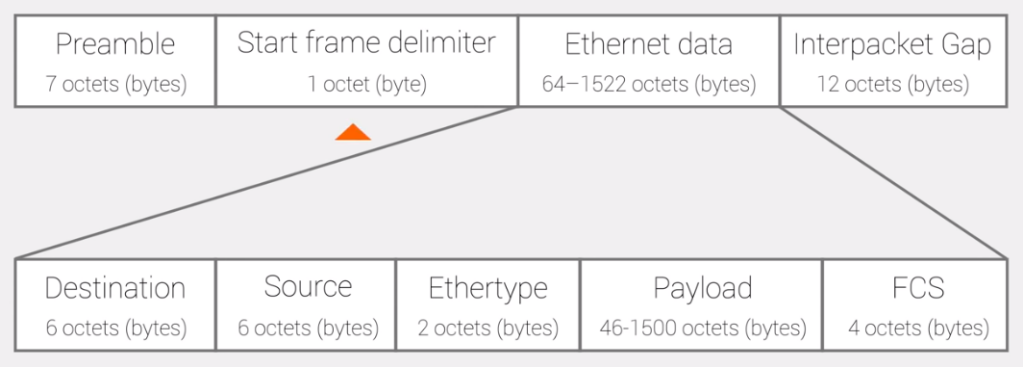


Figure 2.1.2: Ethernet Packet Frame [10]

bus. It also supports I/O virtualization. PCIE has replaced older PCI. Topology used by both these cards are different. In PCI parallel bus is used for data transfer. While in the PCIE point to point communication is done. Hence, PCIE supports full duplex communication. Five different size are available for PCIE - x1, x2, x4, x8, x16. In the physical slot of PCIE card, we can not fit any PCIE card. Slot size should be greater than or equal to size of the card. For example, @16 cannot be put into @8 or @4 slots. Also, we need to take care of number lanes supported by physical slot. It may possible that supported lanes may not be equal to their actual size. Letter "x" tells us the physical dimension, and letter "@" tells us the number of lanes supported by slot. Figure 2.2.1 shows PCIE standards. [11]

PCIE Generation	Transfer Rate (GT/s)	Throughput				
		×1 (MB/s)	×2 (GB/s)	×4 (GB/s)	×8 (GB/s)	×16 (GB/s)
1.0	2.5	250	0.5	1	2	4
2.0	5	500	1	2	4	8
3.0	8	984.6	1.969	3.94	7.88	15.75
4.0	16	1969	3.938	7.88	15.75	31.51

Figure 2.2.1: PCI Express standards

2.3 UVM Base Classes

UVM stands for Universal Verification Methodology. It is derived form OVM(Open Verification Methodology). UVM is highly used verification methodology in the industry. The UVM provides generic functionalities like copy, print, compare, configuring database etc. Every component of a environment has a specific role, like driver class object only deals with driving signals to the DUT, monitor class object only deals with sampling the signal transactions on the interfaces of the design.

UVM is a wrapper prepared around system verilog. Different virtual classes are defined in the UVM to standardize the methodology. These virtual classes are UVM inbuilt libraries. Theses base classes have some virtual methods and tasks defined in it. Verification environment is developed by extending theses library classes.
There are three main UVM base classes.

1. UVM Object
2. UVM Component
3. UVM Transaction

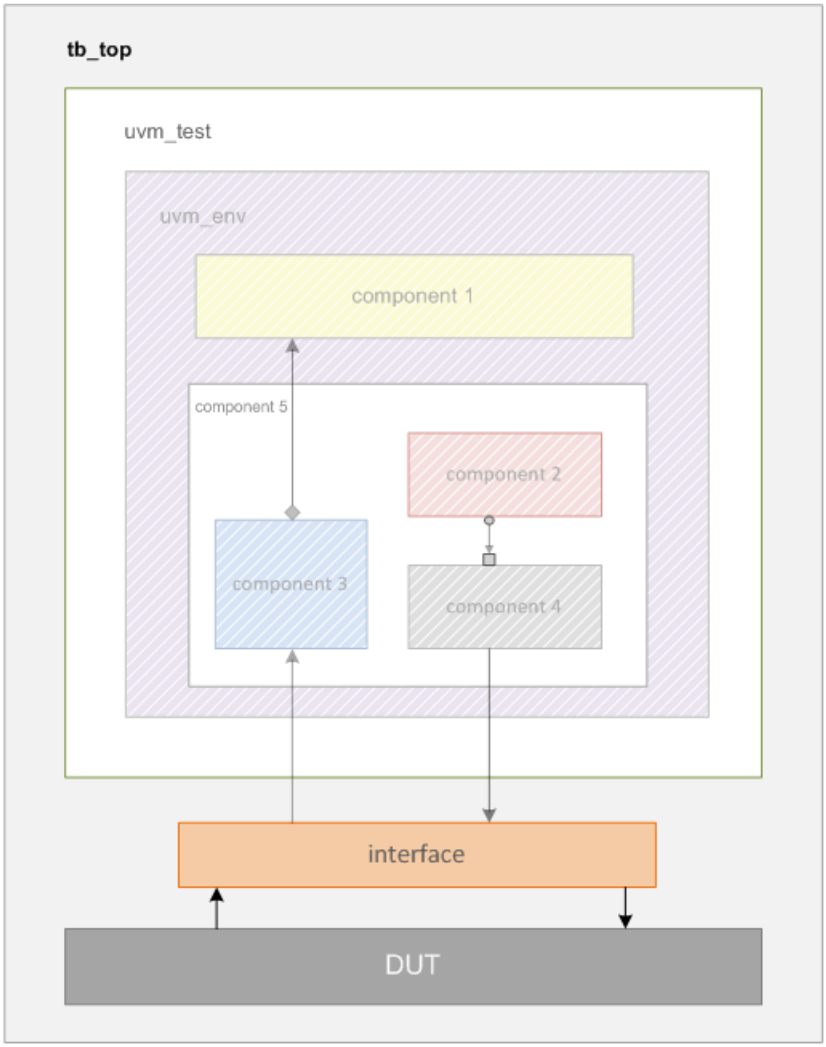


Figure 2.3.1: Basic UVM Testbench [7]

Figure 2.3.1 shows, top level view of testbemch. That is a holder of every verification components. In the top module clock is generated. Then clock is given to the interface module.

Interface module is having definition of all virtual interfaces used in the verification environment. All virtual interfaces are provided to the environment by configuration db. By using configuration database, we can propagate any object values to the other components in the hierarchy. [7]

2.3.1 UVM Test

A testcase is a pattern to verify specific feature or scenario of the design. Development of testcases depends on test plan. Test case is developed as per the target feature or scenario from the test plan. Test class is extended from uvm_test. Instead of writing test cases for each and every scenarios of the test plan, uvm environment is used. This environment can contain all verification components, hence we just need to reconfigure the environment using environment knobs to generate different scenarios to verify different features. We can provide some tweak knobs like enable/disable bit for agent, default sequence for the test, enable/disable bit for coverage model etc.

UVM class hierarchy is shown in figure below:

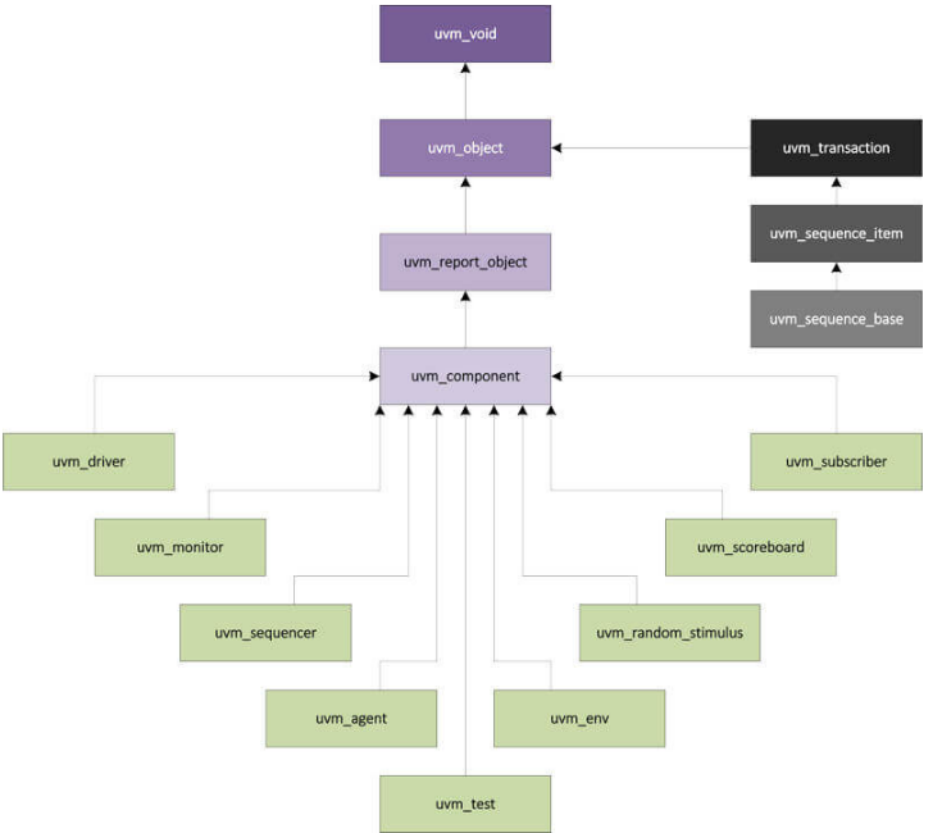


Figure 2.3.2: UVM Class Hierarchy [7]

2.3.2 UVM Environment

A uvm_environment is a container of various uvm_componenet of test bench. If we directly instatiate uvm_componenets in the test class, then we need to make changes in test class, every time there will be some change in the way of uvm_components are connected with each other. Besides, uvm_test class in not reusable as those are relied on the particular scenarios. Hence uvm_components are not directly instantiated in the test class, they are instantiated in the uvm_environment. To provide high controllablity and configurablity to uvm_environment, various knobs are to be used. it provides loose coupling between test class and environment. [7]

Chapter 3

IP verification

SoC comprises of many IPs. Major issue while reusing IPs for SoC design is quality of the source. With increasing complexity of the IP, its quality becomes more critical, and verification of the IP becomes difficult. We need to make sure that every corner cases of the highly configurable IPs are verified thoroughly.
During the project work, I followed below steps while verifying the Ethernet subsystem IP.

- Detailed study of design under test
- Prepare verification test plan
- Prepare coverage points
- Prepare architecture for verification environment
- Implementation of tests
- Functional coverage
- Review

Above steps are followed one by one, however verification is not a straight forward process, we might need some re-spinning in the process.

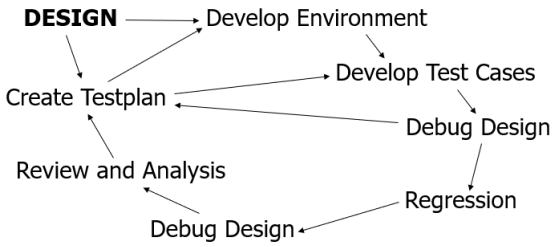


Figure 3.0.1: Verification Process

3.1 Basic tenets of UVM [1]

Encapsulation is supported by UVM, UVM supports OOPS concepts. Hence, we can define our own coding standards and methodology to be followed throughout the verification. We can prepare coding template for the class to be implemented later by using virtual class and virtual methods. That is how we can provide encapsulation to our environment. By default, every function is encapsulated in the uvm_component and uvm_object classes. Those are virtual classes having declarations of the virtual methods and tasks within them. For establishing communication between classes, we can use TLM ports supported by UVM.
Types of TLM ports:

- Get port
- Put port
- Analysis port

Get port and put are by default blocking ports, analysis port is by default non-blocking one. The uvm_sequences are used to generate transaction in the verification environment. The uvm_sequence is a class extended from uvm_object. It includes body task in the class definition. We need to register uvm_sequence with uvm_sequencer, that act as container of the uvm_sequence. This duo is connected with the uvm_sequence using get port, to transfer generated stimuli.

3.2 Detailed study of design under test

Specifications of DUT is understood as the first step of verification. Thorough study and understating of the DUT behaviour is very important, before starting actual verification process. Any misunderstanding in the DUT behaviour could cause false verification. That forces us to change structure of verification environment. Once sturcture of the environmet is finalized, it should not be changed, to avoid any additional cost and delay in tapeout. False verification also causes holes in the verification. Here, holes means any unverified feature or corner cases of DUT. System holes can also affect market value. Hence proper thorough understanding of the design in very important as the first step of verification of any design.
During my project, I used memory aligner as DUT. Memory aligner is shown in figure 3.2.1 As name suggests, Memory aligner is used to align different size of memory. It has total of 1024 bytes of memory storage. At a time, we can give input data of size 8, 16 or 32 byte to the aligner. As shown in figure 3.2.1, credit_av gives total available empty memory in the aligner. It will be in 2D words, if credit_av gives 128, it means (128*2*4) 1024 bytes are empty in the aligner. If we try to write data of size greater than the available memory, than that data will be dropped. Memory aligner follows FIFO principle, it gives first out, that is written in the aligner first. Data_e informs aligner to take available data on the 32 byte Data bus. Valid values for data_e are : 8, 16, 32, depends on the size of the input data. Credit_co informs aligner to provides credits from the available credits to the data. Valid values for credit_co are : 1, 2, 4. Value-1 for 8 bytes of data, value-2 is for 16 bytes and value-3 is for 32 bytes. At the right hand

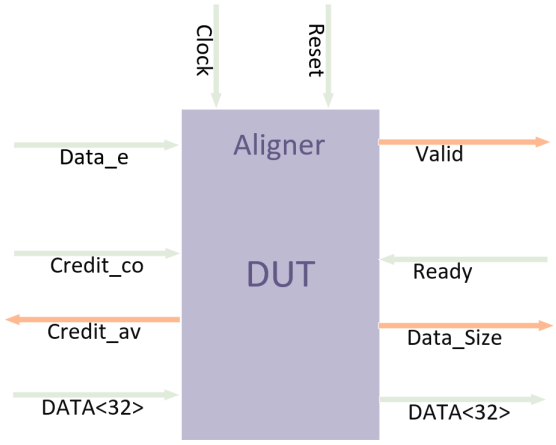


Figure 3.2.1: Memory Aligner

side of DUT, output signals are shown, valid signal informs data available at the output data bus is valid for the current clock cycle. Ready signal informs aligner to send next data on the output bus, on the output bus only 32 or 16 bytes of data will be available. Data_size shows the size of valid data available on the Data bus. So, memory aligner is a typical multidimensional FIFO with depth 1024 that accepts the data of 8, 16 and 32 bytes and gives the output data of 32 bytes.

3.3 Verification plan

Test plan is prepared to keep track of features and scenarios to be verified for the design. With preparing test plan, coverage plan is also prepared to have list of cover points for which functional coverage is to be implemented. So, that tests and functional coverage can be developed simultaneously. Test plan includes all the scenarios and corner cases for the design, it also includes negative scenarios, to verify behaviour of the design under wrong or erroneous stimuli. Test plan and coverage plan is prepared in the excel sheet. Along with scenarios to be tested, which signal of the design to be randomized, how to generate specific stimuli, what checks we need to do, etc are maintained in the spread sheet.

Sample test plan is shown in figure 3.3.1
Following three types of the testing scenarios are included in the plan.

- Valid data traffic
  - Write random number of data to aligner with fixed size
  - Write random number of data to aligner with random size
  - Write and read random number of data simultaneously
  - Generate overflow, write data even when empty slots are not available.
  - Generate underflow, read data even when aligner is empty.

Scenario description	What to generate	what to check
Basic Alive Test	Generate Traffic Generate Reset and Clear and Clk	Credit availble Valid Pin Output Data and Size
Overflow Test	Genarate Traffic Make Enable pin high Generate the data with credit_co > credit_av	Output Data and Size Valid Pin Credit availble
Back Pressure Test	Generate Traffic Make Enable pin low Generate the data with credit_co > credit_av	Data Output and Data size Valid Pin Credit availble
Reset Test	Make enable low and generate the Data with proper Data_e and Credit_co Generate Reset and Clear Make enable high and generate the Data with Proper Data_e and Credit_co	Output Data and Size Valid Pin Credit available after transaction
Fifo Flush within traffic	Make enable low and generate the Data with proper Data_e and Credit_co Generate Clear Make enable high and Generate the data with proper Data_e and Credit_co	Output Data and Size Valid Pin Credit available after transaction
Under run Test	Generate 5 packets and try to read more than 5 packets Make enable high	Output Data and Size Valid Pin
Over run Test	Generate the data with credit_co > credit_av Make enable pin low	Output Data and Size Valid Pin Credit available after the transaction
Negative Tetsing	Generate 8 Byte of data with Data_e - 11/f and Credit_co - 1 Make Enable Low/High	Output Data and Size Valid Pin Credit available after the transaction

Figure 3.3.1: Sample test plan

  - Multi cycle scenario - Write data and make aligner full, then wait for till credits are not available and then again start writing data

- Interrupt/Reset
  - Generate random number of the data and start writing them to aligner, before it ends generate reset signal
  - Generate random number of the data and wait till it ends, and after resetting aligner again start writing random data

- Invalid data traffic
  - Generate erroneous data with injecting intentional error in the one the signal of aligner and write invalid combinations

3.4 Prepare architecture for verification environment

In this step, basic structure of the environment is decided. Architecture of the environment contains verification components and connections. Basic uvm architecture contains virtual interfaces, uvm drivers, uvm monitors, virtual uvm sequences and scoreboards. We can not code whole structure for every scenarios in the test plan. Hence, architecture is finalized in such way that it is useful for every cases. Basic steps to decide testbench environment structure:

- Keep number of agents in the environment exactly equal to the number of DUT's interfaces
- Prepare uvm transaction class, to finalize uvm sequences
- Finalize uvm base components like drivers, monitors and scoreboards.
- Finalize flow of base test, to check basic sanity flow of the design.

Finalized test bench environment is shown figure 3.4.1.

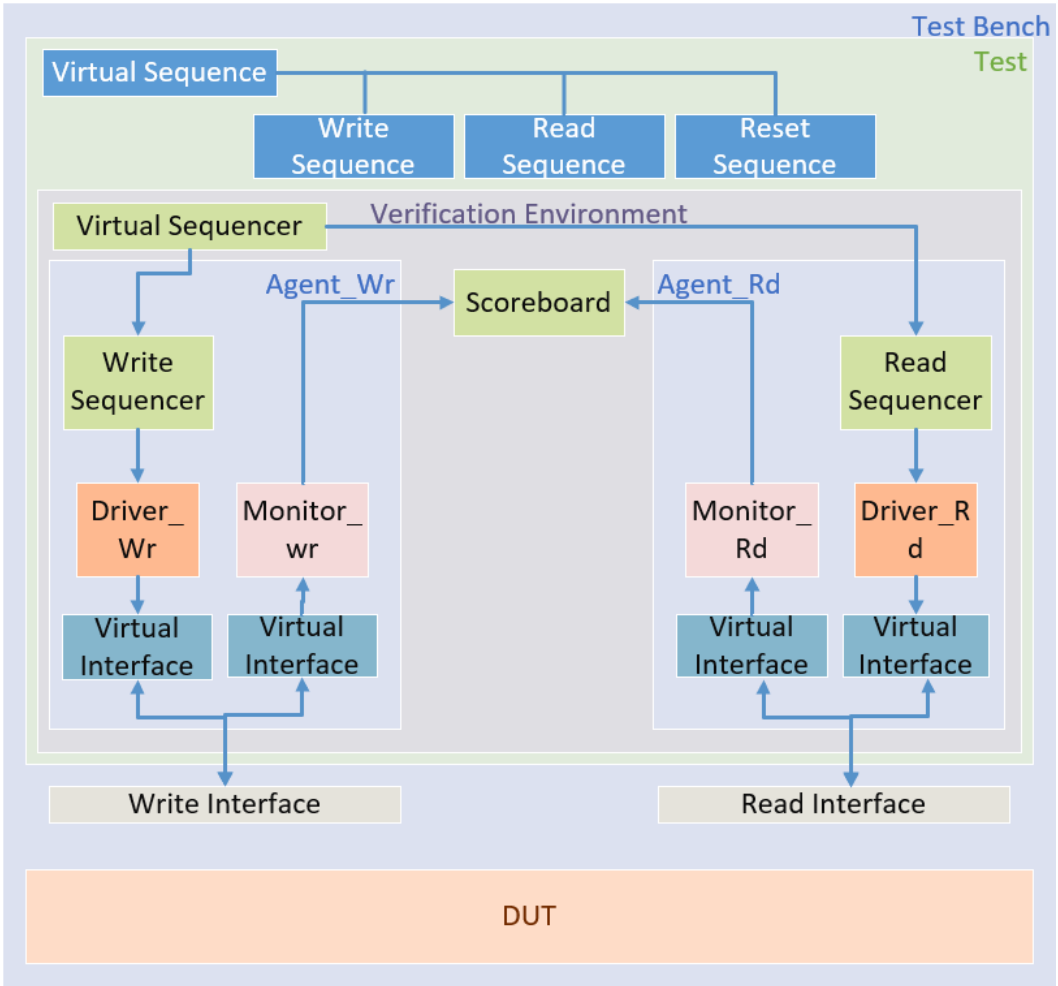


Figure 3.4.1: Test Bench Architecture

3.5 Implementation of tests

After finalized test bench structure, one base test that verifies basic traffic scenario is coded. To develop base scenario, all the base and virtual sequences are coded. Virtual sequences are a

static container, that contains all the sequences. All uvm sequences are instantiated in the virtual sequence. Use of virtual sequence gives advantage of loose coupling, over the use of direct uvm sequences in the test class. test class is extended from uvm component , that has predefined phases like - build phase, connect phase, run phase etc. In the build phase virtual sequence and environment are instantiated. All the configurations of environment for the particular tests are done using uvm config db. Environment is also uvm component. In the build phase if the environment required agents are instantiated. Similarly, in the build phase of agents drivers and monitors are instantiated. All the TLM port connections are done in the connect phase of the environment. After implementing base test class, as per the prepared test plan test classes are coded extending from this base test class.
One base test is prepared first. In the test file, all virtual sequences are instantiated. Virtual sequences have different multiple uvm sequences, each sequences are developed to generate particular stimuli for the DUT. UVM sequences also contains TLM get port, using this get port, generated stimuli are sent to uvm driver. Connection between UVM driver and UVM sequences are done in the connect phase of the environment file. UVM driver are connected to the DUT through virtual interface. UVM monitors are also developed to continuously monitor the values of the DUT interfaces, those sampled values are then sent to uvm scoreboard using TLM analysis port. Scoreboard are developed with all the required checks those are required to ensure the given I/O functionality of the DUT.

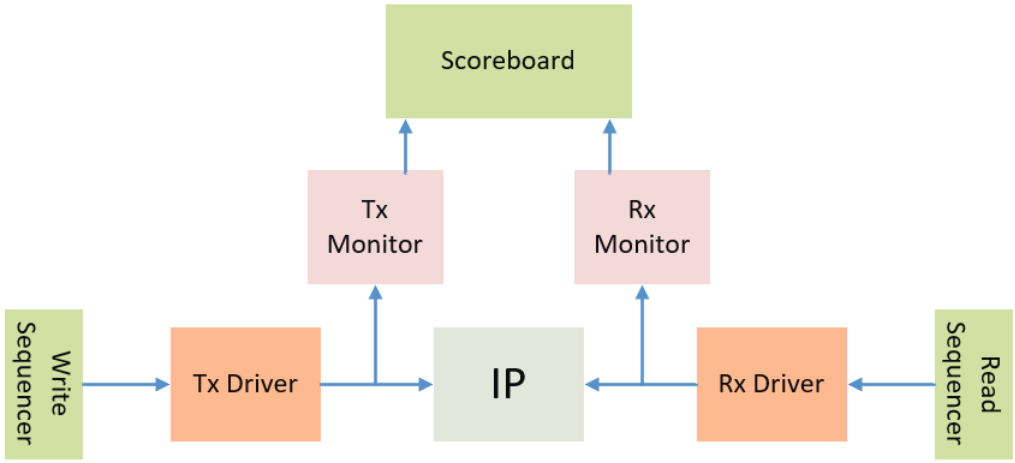


Figure 3.5.1: Verification Componenets

In negative testing, sequence generates erroneous stimulus and inject in the DUT through driver. Due to the erroneous stimulus, actual and expected values won't be matched in scoreboard and scoreboard will shout in this case, however this error is intentionally injected, hence we have suppressed the error in this case using uvm report checker. Figure 3.5.2 shows code snippet for the demoting uvm error.

```
class error_demoter extends uvm_report_catcher;
  function new(string name="my_error_demoter");
    super.new(name);
  endfunction

  function action_e catch();
    `uvm_info(get_type_name(), "UVM_CATCHER", UVM_HIGH)
    if(get_severity() == UVM_ERROR && get_message() == "----------Actual Packet is not recieved----------"
)
      `uvm_info(get_type_name(), "UVM_CATCHER_INSIDE", UVM_HIGH)
      set_severity(UVM_INFO);
    return THROW;
  endfunction : catch
endclass : error_demoter
```

Figure 3.5.2: UVM Demoter

3.6 Functional Coverage

Functional coverage is used to keep track on the progress of the verification process. System verilog provides covergroup and coverpoints to code the functional coverage. For corner cases, functional coverage are coded to check if it is verified or not. Functional coverage for covering values of the credit_av signals of the aligner. That coevarge shows if scenarios meets aligner full and empty conditions ever. Some cross coverage is also implemented to check scenarios like, reading white aligner is empty and writing while aligner is full, etc.
Traditionally, verification quality is measured with the code coverage. How thoroughly HDL code is exercised is reflected in code coverage. Code execution is traced using code coverage tool - verdi. Verdi provides some coverage metrics for code coverage, that includes line, block, fsm states, conditions, branches, toggling of bits, event etc. However there some limitations with code coverage. we can not say, design is perfectly verified only on the basis of code coverage. To ensure the verification process closure functional as well as code coverage is achieved up to the mark. Functional coverage is all about, how many corner testcases, negative scenarios, basic scenarios are covered during regression run. Functional coverage also allows combinations of two or more different fields. For example, reading while FIFO is empty, writing while FIFO is full. These examples say, we need to cover read operation when empty flag is high, and cover write operation when full flag is high. These are example of cross coverage, basically combination of different values of different fields. Functional coverage also allows relationships, ”OK, I’ve covered every state in my state machine, but did I ever have an interrupt at the same time? When the input buffer was full, did I have all types of packets injected? Did I ever inject two erroneous packets in a row?”
Systemverilog provides following features for coverage:

- It provides point coverage for single value, multiple values. Values can also be in the form of expression.
- It provides Cross coverage for combinations of different fields.
- Automatic bin can be generated, or we can manually define bins that can have single value, multiple values, sequences, toggling etc.

```
covergroup aligner_cg
    credits : coverpoint credit_av {
        bins emtpty = {128};
        bins full = {0};
    }
    write_data : coverpoint data_e {
        bins write = {1,3,7};
        illegal_bins invalid_com = {0,2,4,5,6};
    }
    write_op : coverpoint valid {
        bins write = {1};
        bins ideal = {0};
    }
    read_op : coverpoint raedy {
        bins read = {1};
        bins ideal = {0};
    }
    read_empty : cross read_op, credits {
        bins read_em = binsof(credit_av) intersect {128};
    }
    write_full : cross write_op, credits {
        bins write_f = binsof(credit_av) intersect {0};
    }
engroup
```

Figure 3.6.1: Sample Functional Coverage

Planning for functional coverage includes documentation of all coverpoints required to be covered while verifying the system. Planning of functional coverage is started basically with the planning of test cases. Execution of test cases are observed by functional coverage. Functional coverage is code written to track whether important values and sequences are ever generated at particular interface. One of the factors used to observe the verification progress is functional coverage, hence it is very important aspect to any verification approach. We can say that all scenarios from testplan has successfully verified, when we achieve 100% coverage number for functional coverplan. Coverage written to examine the value within single object is called point coverage. Coverage written to examine the combination of two or more different fields is called crossing of coverage. Verdi automatically calculates code coverage. Code coverage is related to the number lines in the code, being executed during the simulation. 100% coverage can be achieved for code coverage. However, it is not meant that functionally we achieve complete verification.

On the other side, functional coverage is code written to examine some of the corner cases. It is totally dependent on the coverage plan. It may possible that coverage plan has some hole. In

that case we can achieve functional coverage 100%, however code coverage in that case can be lower. Hence, either code coverage or functional coverage alone will not help us for verification closure. It is required to achieve both coverage metrics up to the mark. Figure 3.6.2 shows four places where functional coverage points are added. [5]

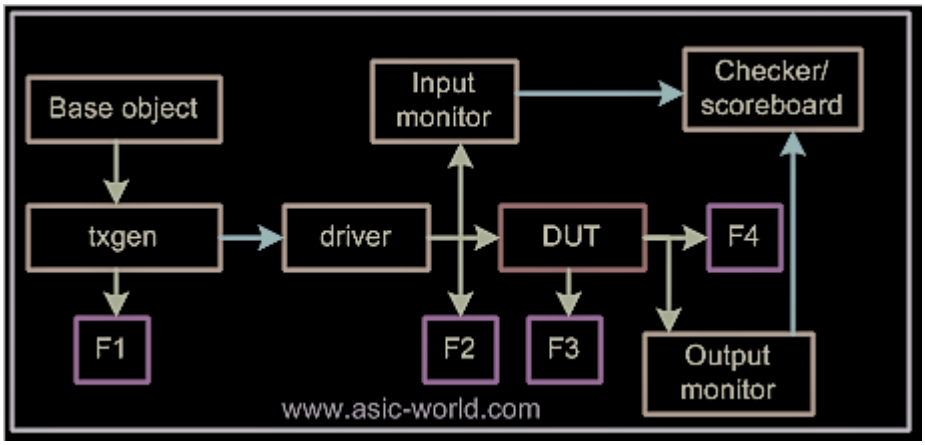


Figure 3.6.2: Test Bench Architecture

- F1 - Class codded with these set of coverage points is instantiated very near to the randomization and before the driver. In some cases, driver is not sending randomization object to the DUT, coverage sampled with this condition are not acceptable.
- F2 - Class codded with these set of coverage points is instantiated inside a input monitor. It has functional coverage on stimulus that DUT is being driven with.
- F3 - Class codded with these set of coverage points is instantiated standalone. Internal states of DUT, like FSM states, or some registers are monitored in this class.
- F4 - Class codded with these set of coverage points is instantiated inside a output monitor. It has functional coverage on output of the DUT.

3.7 Automation using Scripting

In the developed test cases, there are two types of cases coded - to generate directed scenario and to generate some random scenario. For rigorous verification random test cases are run multiple time to generate various random scenario. One perl script is developed to analyse the regression result. This script runs as post simulation process to over regression result to extract failures and some important information.
Following information are extracted from the results using perl script:

- How many times one error is being repeated
- Failure rate of every feature and scenarios

- Total failure rate of all scenarios
- Detailed list of passed and failed tests
- Errors causing failures
- Pointer to the systemverilog file for failure
- Pointer for log file
- Original commands of each failure

---	Date	---	Tue Dec 3 21:57:40 2019	---	---	---	---
***	***	***	***	***	***	***	***
***	Report	Summery	***	***	***	***	***
	ErrorType	Occurance					
	NO ERROR	24					
	UVM_ERROR	2					
***	***	***	***	***	***	***	***
***	Report	Summery	***	***	***	***	***
	TestName	Failing Occurance	Passing Occurance	Passing %			
	Base_Test	2	24	92%			
***	***	***	***	***	***	***	***
	Total	26					
	Passing %	92%					
	Failing %	8%					
***	***	***	***	***	***	***	***
Sr.No	Test_Name	Seed	Result	ErrorType	Error	File_Path	Log_file
1	Base_Test.1	3a4e65b6	FAIL	UVM_ERROR	UVM_ERROR -@ 308762560 ps - Error Text - Scoreboard error.	</nfs/...../uvm_scoreboard_core.sv(34)>	/nfs/...../Regression.test.1
2	Base_Test.2	34a29e9c	FAIL	UVM_ERROR	UVM_ERROR -@ 211512640 ps - Error Text - Scoreboard error.	</nfs/...../uvm_scoreboard_core.sv(34)>	/nfs/...../Regression.test.14

Figure 3.7.1: Report generated from script

Chapter 4

Verification of Subsystem

Subsystem comprises of multiple IPs. Efforts required in the verification of subsystem is reduced by reusing IP verification environment. We reconfigure the subIP environment to make it compatible for subsystem. At subsystem level, we do not need thorough verification of all IPs, instead we need to check interfaces between all IPs. Our main focus for verification at this level is on connections and interfaces.

4.1 Subsystem Verification Flow [2]

4.1.1 Top Level View

During subsystem verification, first step is to study subsystem I/O functions. A detailed study of subsystem behaviour under different stimuli and its structure is done. Any misinterpretation of the behaviour can cause re-spinning in the later stage, that can affect verification cost and time.

4.1.2 Verification Plan

Detailed plan for the Verification of subsystem is done here. As a plan for verification functionalities to be verified are listed. We identified IP components that can be reused for the verification of subsystem. So that development time at subsystem level is reduced. Few sub-IP sequences and coverage models are reused here. At subsystem level, main focus is to be verified the interconnections between the sub-IPs, rather than verifying functionality of each block or sub-IPs deeply. Also, interface transactions between subsystem and TOP module is verified at subsystem level.

4.1.3 Verification environment

Environment for verification of subsystem contains sub_IP verification components as well as some specially coded components in subsystem architecture. For verification of subsystem proper methods is to be selected that is best for the rigorous verification of subsystem. Combining IP verification components to make one whole subsystem environment is done first before

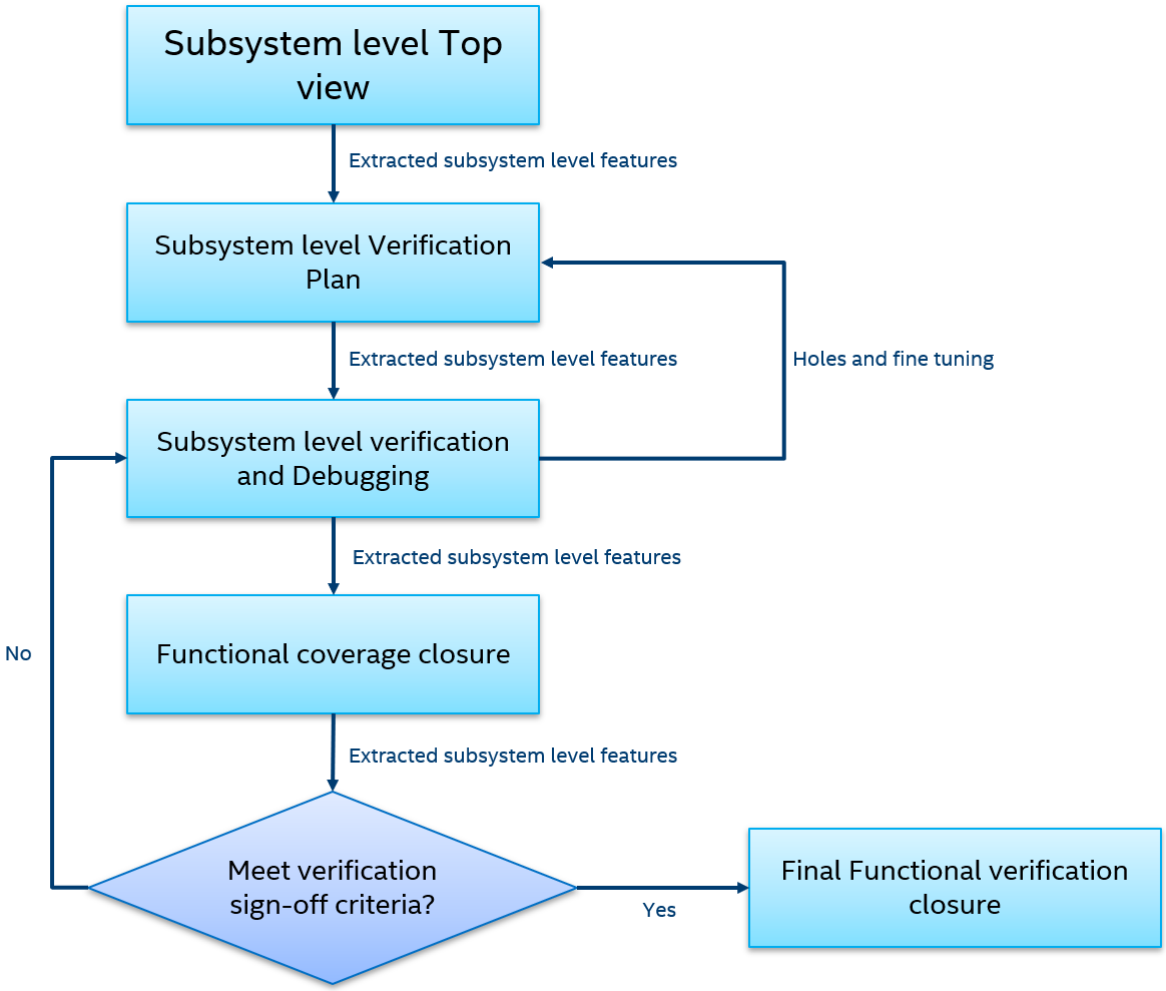


Figure 4.1.1: Subsystem Verification Flow

stating verification process. Scoreboards are developed to ensure correct connectivity between the components. This is very important process, as wrong connections can cause wrong I/O functions. At subsystem level some basic components are developed for end-to-end functionality check. Hence, subsystem is having its own uvm_driver, uvm_monitor, uvm_sequences and checkers along with some reusable sub-IP components.

4.1.3.1 Subsystem level scenarios and debugging

At subsystem level, it is required to apply various stimuli that can generate different scenarios for the purpose of functional verification. On the other side, it is not feasible to generate all kind of scenarios at this level, because we also need to meet time to market. Hence, few test cases are reused from sub-blocks. Also, some of the cases or code are reused form its older version, to reduce the verification efforts. After developing sufficient amount of test scenarios for subsystem, we start running those, and we start analysis of the failures. For this we run regressions. To debug the regression failures is tedious task at this level. Thorough study of

the subsystem behaviour under any condition should be known to avoid unwanted debug time. Proper messaging and coding standards are used while implementation, to make debug process easy up to some extent. We also provide uvm_verbosity to all the messages to enable and disable those messages while running regression.
UVM provides below verbosity levels for messages:

- UVM_LOW
- UVM_MEDIUM
- UVM_HIGH
- UVM_NONE
- UVM_FULL
- UVM_DEBUG

In some case, we debug RTL bug with interactive session in the verdi. That helps us to debug whole simulation with the break points.

4.2 Reuse of IP verification components

Integration of sub-IP's verification components is done in subsystem environment of verification. In this process few components are disabled by configuring knobs in particular block. Figure 4.2.1 and figure 4.2.2 shows IP verification components and Subsystem verification architecture.
General structure is having uvm_driver, uvm_monitor, uvm_sequences and uvm_sequencer. A uvm_agent is a container, in which all these classes are instantiated. uvm_agent can work actively or passively. In the passive mode only uvm_monitor is instantiated in the uvm_agent. In the active mode all class are instantiated in the uvm_agent. UVM provides TLM ports for connectivity between the classes. Packets are sampled in uvm_monitors, then they are set to uvm_scoreboards. The uvm_scoreboard is coded to implement checker between input and output data. At subsystem level uvm_agents of sub-IPs are configured in such a way that they work passively. Because, uvm_driver would not require at this level, subsystem is having its own uvm_driver to send the data to sub-blocks.

4.3 Coverage reuse

Different functional coverage places, discussed earlier. Few of those are required at this level also. Hence, IP functional coverage are also reused at this level. However, we do not require all coverage setup from sub block.
To overcome this situation, we can have two ways:

- HVP - Hierarchical Verification Plan
- Unique implementation method

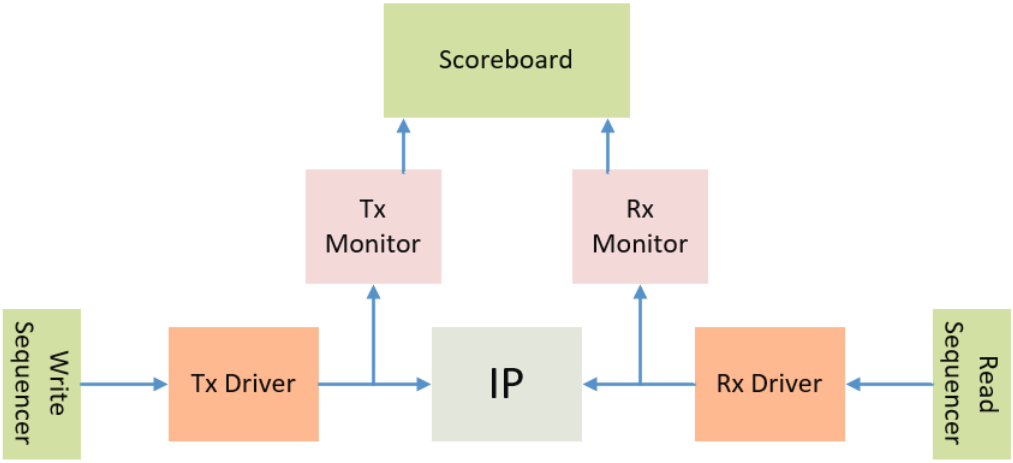


Figure 4.2.1: IP verification components

4.3.1 Hierarchical Verification Plan

HVP is maintained for the functional coverage reports after every regression run. From HVP reports system holes are debugged and fixed, to get higher coverage number. Figure shows basic flow for HVP generation.
Pros and cons of this method:

- Pros:
  - Only required coverage adds number into total coverage percentage
  - Required coverpoints or bins can be excluded
  - HVP can be manipulated as per functional testplan
- Cons:
  - It will not reduce wall clock time for simulation, rather it can be increased in some cases

4.3.2 Unique coverage coding method

Here, we develop proper method to enable and disable particular coverage from particular sub-block. Each sub-lock must follow that implementation method to make coverage reusable. Pros of this method are:

- Rechecking of reused scenario is not needed
- wall clock time for simulation can be reduced
- Coverage database size is reduced

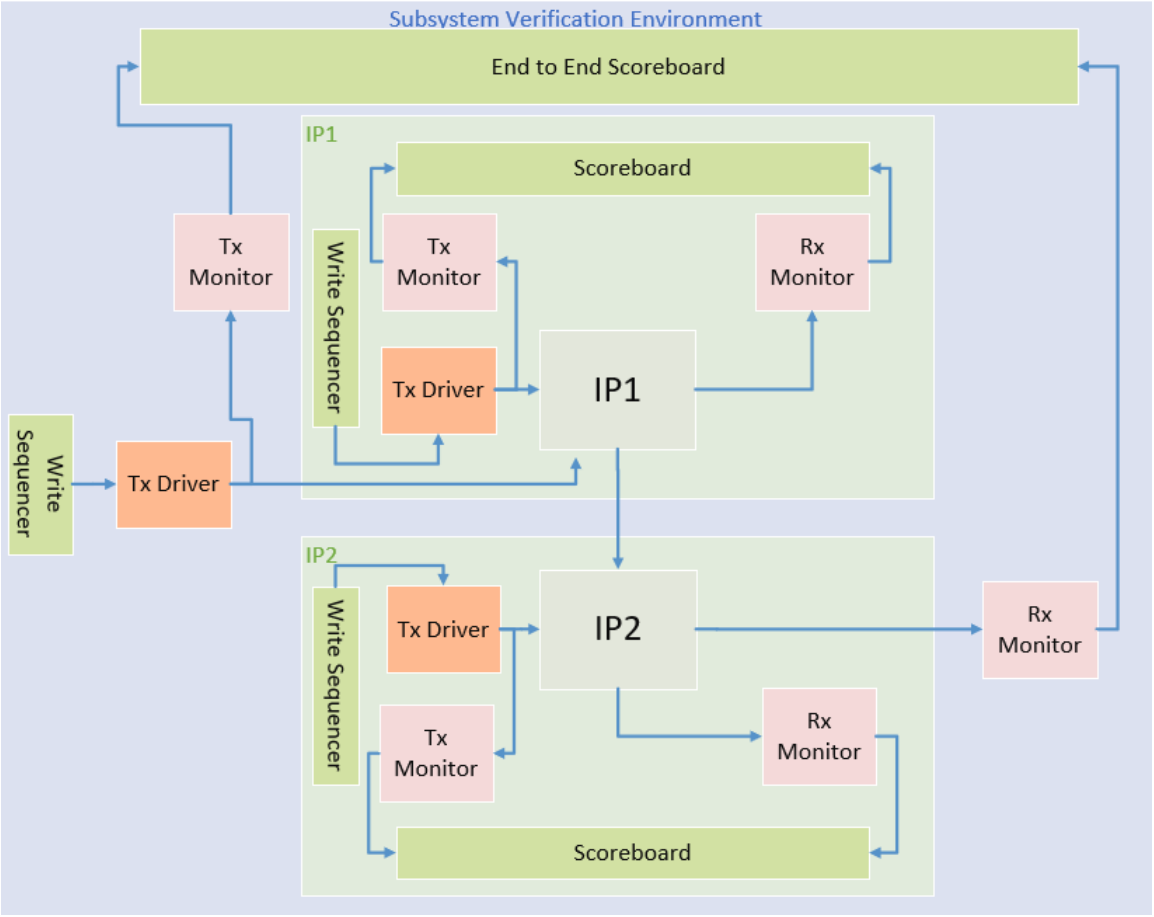


Figure 4.2.2: Integrated Verification environment

4.3.2.1 User defined Macros

We have defined user defined macros ‘CREATE_COV and ‘COV_SAMPLE for instantiating and sampling.

1. COV_CREATE(cg_name, instance_name, arguments, verbos)
   - instance_name : name of the cg instance
   - cg_name : covergroup name
   - arguments : arguments used for covergroup
   - verbos : verbosity level
2. COV_SAMPLE(instance, arguments)
   - instance : name of cg instance
   - arguments : arguments used while sampling

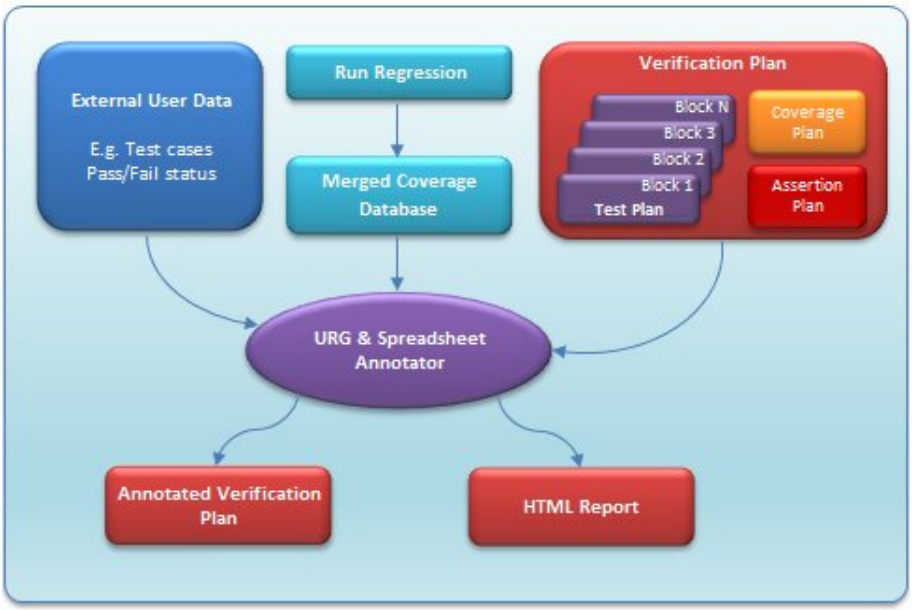


Figure 4.3.1: Basic flow for HVP [12]

Verbosity levels:

- COV_VERB_TOP
- COV_VERB_SUBSYSTEM
- COV_VERB_SUBBLOCK
- COV_VERB_BLOCK
- COV_VERB_MODULE
- COV_VERB_NULL

Verbosity is passed at two places, first is command line and second is covergroup instance. If simulation verbosity is higher than or same as CG instance verbosity, then that covergroup will be enabled, otherwise it will be disabled while simulation. Figure 4.3.2 shows sample code for this method, comparing with normal system verilog method. Flow for calculating coverage is set for subsystem team. Steps I have followed for this are listed below:

- Prepare coverpoint definitions
- Filter reusable coverpoints and get those from sub blocks
- Implement coverage for remaining coverpoints
- Add features and meaures in the HVP as per coverage plan
- Coverage debug

- Without Macros

```
covergroup cg_abc_transaction ;
   coverpoint a;
   coverpoint b;
endgroup : cg_abc_transaction
---------------

function new (string name);
      super.new(name);
     cg_abc_transaction.new();
endfunction : new
task ...
 cg_abc_transaction.sample ();
```

- With Macros

```
covergroup cg_abc_transaction ;
   coverpoint a;
   coverpoint b;
endgroup : cg_abc_transaction
---------------

function new (string name);
      super.new(name);
     `COVERAGE_CREATE(cg_abc_transaction,
"", (),  COVERAGE_VERB_MOD)
endfunction : new
task ...
`COVERAGE_SAMPLE(cg_abc_transaction, ())
```

Figure 4.3.2: With/without user defined method for coverage

4.4 Performance Testing

Consider figure 4.4.1
Only verification of features and behaviour under all possible stimuli is done by functional verification. However, features and functionality are not only concerns, also performance measures like bandwidth, latency and reliability do matter. Performance testing will determine whether the DUT meets performance measures under expected workloads. Chip sent to market with poor performance metrics due to nonexistent or poor performance testing are likely to gain a bad reputation and fail to meet expected sales goals. That's why performance testing of the DUT along with functional verification is carried out. Performance testing is not to find bugs but to eliminate performance bottlenecks. We have written the performance test cases for Ethernet subsystem to generate different workloads situations. As shown in figure Ethernet subsystem has below components: PCIE Interface with 3 different configurations - 1x16, 2x8, 4x4; Address Translation Unit; Packet Processor; 4 Mac Ports. Variants of performance testing: System Topology – In this testing we configured DUT with different topology (such as -1 Mac Port and 1 PCIE Home, Multiple Mac port and Multi PCIE Home etc.) and ensured performance under different situations. Cache miss rate – In this testing we configured the local cache of the sub IPs to achieve different cache miss rate. (such as 10%, 50%, 100% etc.) Packet Size – In this testing I built different packet size and ensured the performance of the system with those. To generate different packet size, we used IPv4, IPv6 and VLAN headers. AT enable/disable – AT is address translation unit, used for mapping between virtual and physical functions.

4.5 Sim Profiling

Sim profiling is a way to analyse program execution time. Complex code in system core logic can increase time for simulation run as well as real time. To analyse execution time manually for each and every block is very tedious. Manually, we need to analyse first, which block is

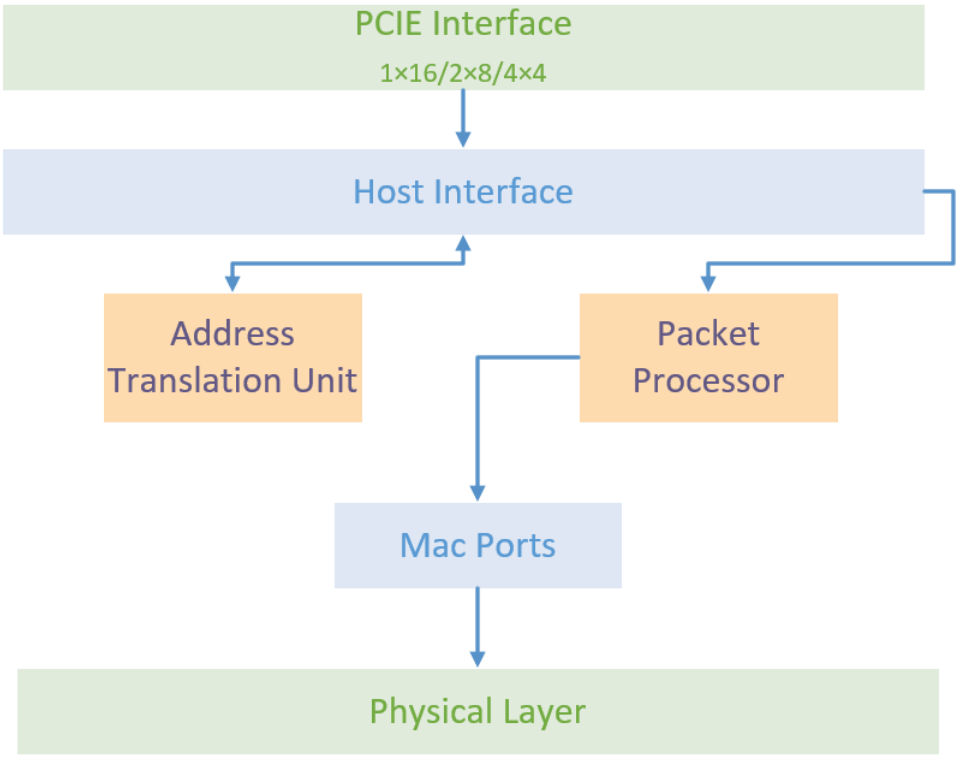
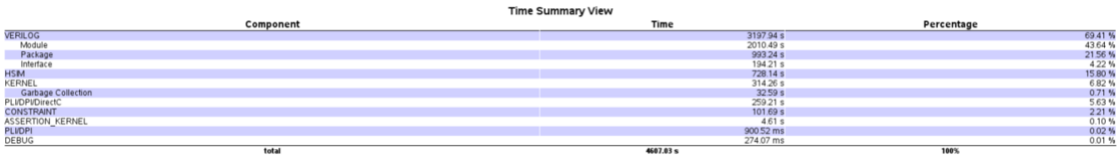


Figure 4.4.1: Ethernet Subsystem

taking long, then we need to modify core logic for that block. Instead this manual work we are using VCS sim profiling tool to reduce human efforts. VCS sim profiling tool can be enabled


Time Summary View

Component	Time	Percentage
VERILOG	3197.94 s	69.41 %
Module	2010.49 s	43.64 %
Package	993.24 s	21.56 %
Interface	194.21 s	4.22 %
HSM	728.14 s	15.80 %
KERNEL	314.26 s	6.82 %
Garbage Collection	32.59 s	0.71 %
PLIDPIDirectC	259.21 s	5.63 %
CONSTRAINT	101.69 s	2.21 %
ASSERTION_KERNEL	4.61 s	0.10 %
PLIDPI	900.52 ms	0.02 %
DEBUG	274.07 ms	0.01 %
total	4607.83 s	100%

Figure 4.5.1: VCS Report for Time Profile

using below switches during compilation.

1. ”elab_opts -simprofile=time”
2. ”elab_opts -simprofile=mem”

Figure 4.5.1 shows time profile results.

Chapter 5

Conclusion

Detailed knowledge of UVM and use of verdi tool for verification of deign. Detailed study of design is very important as verification prerequisite step, to avoid additional time and cost in the verification process later. Major challenge is coding IP verification components in such a way that they can be reused at different levels. Along with this, It requires to re-spin the logic of the uvm_monitor and uvm_driver few times, because of misinterpretation of DUT behaviour. Hence, detailed knowledge of behaviour of the design is very importannt before starting actaul verification process. Componenets of UVM supports phase wise simulation, to handle proper begining and ending of each phase, pre-defined uvm_objecction is used.
Automation scripts is used as post processing of simulation to collect feature and scenario wise failure reports. That reduces human efforts and process pf initial debugging is simplified. Also sim profiling helps with collecting block wise clock timing report and memory report.
Subsystem is integrated version of sub-IPs, hence we can reuse as many sub blocks as possible, at this level integration is very important process. Wrong connections can cause false verification, that can add unwanted delay in tape-in. Also, at this level rather than thorough verification of sub blocks, interfaces and connections are mainly focused for verification process. As many configurablity as possible is must for environment. Beginning and end of every phase must be handled carefully. Number of testcases developed are not important for verification closure, rather code coverage and functional coverage must be up to the mark for closure.

References

[1] P1800.2/D7, ”IEEE Draft Standard for Universal Verification Methodology Language Reference Manual”, Oct 2019.

[2] Dilip Prajapati ”SoC Functional verification flow”, Dec, 2017.

[3] IEEE Std 802.3, ”IEEE Standard for Ethernet”, published at New York, NY 10016-5997 USA, Sep, 2015.

[4] UVM cookbook, available at ”https://verificationacademy.com/cookbook/uvm”.

[5] ”Functional Coverage”, ”http://www.asic-world.com/systemverilog/coverage1.html”.

[6] ”7 Levels of verification cycle”, ”https://www.edn.com/the-7-levels-of-ip-verification/”.

[7] ”Concepts of Virtual sequence and Sequencer”, ”https://www.chipverify.com/uvm/uvm-virtual-sequence”.

[8] Debug features of Synopsis verdi tool, ”https://www.synopsys.com/verification/debug/verdi.html”.

[9] ”7 Layers of OSI MOdel”, ”https://www.webopedia.com/quick_ref/OSI_Layers.asp”.

[10] How does Ethernet work, ”https://www.youtube.com/watch?v=5u52wbqBgEY”.

[11] ”PCI Express Standards”, ”https://en.wikipedia.org/wiki/PCI_Express”.

[12] ”Basic HVP generation flow”, ”https://www.design-reuse.com/articles/42844/smart-tracking-of-soc-verification-synopsys-hierarchical-verification-plan.html”.