Functional Verification of

Local Move Link

Sub System at Module Level

Major Project Report

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Technology

In

ELECTRONICS & COMMUNICATION ENGG.

(VLSI DESIGN)

Prepared By

Gaurang K. Parmar
(07MEC009)

Under the Guidance of

Mr. Pranav Joshi
Project Leader - ASIC
E-Infochips,
Ahmedabad.

Prof. Usha Mehta
EC Department,
Institute of Technology,
Nirma University.

Department of Electronics & Communication Engineering,
Institute of Technology,
Nirma University of Science & Technology,
Ahmedabad - 382481.

CERTIFICATE

This is to certify that the M.Tech dissertation Major Project Report entitled **"Functional Verification of LMvL Subsystem at Module Level**" submitted by **Gaurang K. Parmar (07 MEC 009)** towards the partial fulfillment of the requirement for the Degree of **Master of Technology (Electronics & Communication Engineering)** in the field of **VLSI Design** of Institute of Technology, Nirma University of Science and Technology, Ahmedabad is the record of the work carried our under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this dissertation-project work to the best of my knowledge have not been submitted to any other University or Institute for the award of any degree or diploma.

Date:

Place: Ahmedabad

Project Guide
Prof. Usha Mehta
Institute of Technology
Nirma University,
Ahmedabad

Dr.N.M.Devashrayee
P.G. Co-ordinator
VLSI Design
Institute of Technology
Nirma University,
Ahmedabad

HOD
Prof. A. S. Ranade
Dept. of EE Engineering
Institute of Technology
Nirma University,
Ahmedabad

Director
Dr. Ketan Kotecha
Institute of Technology
Nirma University,
Ahmedabad

ACKNOWLEDGEMENT

It is an immense pleasure to express my gratitude towards all those people whose contribution was of paramount importance in achieving the goals of this project.

I am indebted to **Mr. Nilesh Ranpura**, Project Manager, E infochips Ltd. for giving me an opportunity to work in such a reputed Organization and providing the most compatible environment for working as a trainee.

I particularly record my profound gratitude to **Mr. Pranav Joshi,** Team Leader, Asic Group, for the valuable guidance, support, and encouragement played a very important role in the successful completion of this project. His generous attitude and expertise in the field of VLSI fundamentally important towards achieving the objectives of the project.

I pay my special regard to **Alay Shah**, **Ketul Parikh, Vilas Jadav**, and **Manish Ladani**, **Malkesh Adesra, Vaibhav Tekale**, **Pratik Adesra** and others in the team for their able guidance, timely help and training that made my way towards the goal easy.

I extend my thanks to **Prof. A. S. Ranade,** Head of Electrical Engineering Department of Institute of Technology, NIRMA University.

I express my special thanks to **Prof. N.M. Devashrayee,** PG Coordinator, EC Department, NIRMA University for his unconditional support and encouragement. I am thankful to **Prof. Usha Mehta**, EC Department, NIRMA University, for her continuous support and guidance. I am also thankful to all the faculty members of the department for their direct or indirect support. At last my deepest gratitude goes to those persons whose names are not mentioned here but they directly/indirectly helped us in achieving success in this project.

Gaurang Parmar

LIST OF FIGURES

Contents

4. Local Move Link

ABSTRACT

The gate counts and system complexity growing exponentially, with that engineer confront the most perplexing challenge in chip design cycle: Verification. Verification of the design RTL is done at various phases of the chip design flow at different abstraction levels. The Major Project traverses through the chip design flow and functionally verifies the module of the chip. This is done at low abstraction level concentrating on the core functionality of the module. The inputs to module are forced through the testbench and its interfaces are not looked upon. This is the functional verification of chip at module level and is done at the RTL design phase. The designer updates the RTL as per the feedback. After the RTL is been finalized after fixing all the bugs, it is send to the fabrication unit.

The chip under functional verification is a tester instrument chip, which had various blocks like Timing Generator block, Memory Pattern Generator block. To verify the Timing Generator block and Memory Pattern Generator block verification environment had to be developed around these blocks. So the Environment should me very generic and flexible in order to verify all different kind of blocks in the chip. Various test cases were developed to cover all the necessary features of the blocks. The test cases were fired and the waveforms were analyzed to debug the RTL as well as test bench issues. Once all the test cases are passing, code coverage is done using a code coverage tool. The code coverage results are analyzed to uncover any dead code as well as logic which were never exercised.

The main objectives are to learn the fundamentals of chip verification as well as ensure functional correctness of blocks under verification from Environment to final coverage against their specifications. The block to be verified is Local Move Link. To exercise the modules for their core functionality test cases are to be written and simulated based on the features. The failing tests have to be debugged for any RTL issues and once the design is stable daily regression are to be carried out with random seeds to make sure that the designs are bug free. At the end Code Coverage is to be carried out to reach to each and every hidden corners of RTL and based on it new test scenarios are added.

About eInfochips, India

eInfochips is an IP driven Product Development Services Company . The company offers a full range of services & solutions in Chip/ASIC, Embedded Systems and Software Product Development for various industries. The domain knowledge enables us to partner with companies in diverse industries worldwide including Video, Security/Surveillance, Semiconductor, Consumer Electronics, Industrial Automation, Medical/Healthcare, Automotive, Networking, Avionics & Defense, Machine Vision/Image processing.

eInfochips has ability to address the entire spectrum of solutions differentiates us from the others. eInfochips' team takes the complete ownership of execution and its success. We ensure that the technical skill of our team is constantly upgraded keeping in view the ever changing demands of the technology industry.

Over the past decade, eInfochips has executed successfully over 300 projects in varied areas for different customers ranging over different industries and different technologies. Many of our projects required cross-platform, cross-technology skills. We have grown as a design house with 730 strong team of professionals spread across the globe.

Vision, Core Values and Brand Promise

Vision:

Innovative technology company that transforms society by creating leaders and generating stakeholder value.

Core Values:

- Customer First
- Disciplined Execution
- Embrace Impossible Challenges
- Self improvement through continuous learning
- Serving Society through Technology

Brand Promise: On Time, Every Time

Key Corporate Milestones at eInfochips:

- eInfochips wins GESIA Award for the Best Innovation by ICT (Information & Communication Technology) Company.
- eInfochips breaks into NASSCOM's 100 IT Innovators - 2007 list.
- Partners with PolyVision to Design Innovative Visual Communication Tool.
- eInfochips wins CII Award for "Emerging ICT Enterprise of Gujarat".
- Selected to operate QLogic™ Design Center.
- eInfochips' CEO receives AMA Outstanding IT Entrepreneur of the Year Award 2004.

Embedded System Design Services

Einfochips provides concept-to-market Product Design Services to established technology companies as well as technology startups. In this extremely fast-paced competitive market, realizing an innovative cost-effective product is becoming exceedingly difficult. eInfochips provides services to enable its clients maintain quality standards to meet the stringent demands for product reliability, high performance & low cost.

eInfochips‘s Product Engineering Services help our clients to stand out in the market amongst all competitors by assisting them to bring new products quickly to the market with highest reliability. eInfochips’ high-quality, cost-effective & business-focused approach and extensive experience in Product Design Services helps our customers reap significant year-on-year benefits. With our company’s product development expertise, we focus on turning customer's concepts into world-class products through our unique set of value-added product realization services & solutions.

Chip Design

eInfochips is a leading provider of IP driven silicon design and verification services and solutions. Our capabilities extend from spec-silicon-system, with expertise spanning front- end design, ASIC/SoC verification through verification methodologies and Hardware Verification Languages (HVLs), physical design and verification, ASIC prototyping (pre-silicon validation), post-silicon validation and industry-standard design and verification IP development. eInfochips' offshore design centers have delivered

multitude silicon and product design solutions to technology companies worldwide, helping customers reduce their time to market and build market strength.

eInfochips offers semiconductor IP development in OVM, URM, RVM, VMM and AVM methodologies employing SystemVerilog, e, SystemC and Vera HDL. Our Design Quality Engineers and Process Analysts are trained on DO-254 (Avionics standards) by approved trainers enabling us to provide flight critical applications ASIC/SoC/FPGA Design & Verification services. eInfochips' ASIC physical design expertise covers RTL synthesis, floor planning, place & route, clock tree synthesis, physical verification and STA on 45nm, 65nm, 90nm, 130nm, 160nm, 180nm technologies. Our comprehensive domain expertise in Networking, Automotive, Avionics, Wireless, Consumer, Semiconductor, Video and Storage qualifies us as your ideal consultant in ASIC Design and Verification Services.

Semiconductors

eInfochips is a prominent design services vendor to the semiconductor industry, providing "Spec-to-Silicon-to-System" engineering services to semiconductor chip companies, EDA tool companies and product development companies. Our group has developed the requisite skill sets and industry specific solutions and provides complete Product Design Services

- ASIC/SoC Design & FPGA Prototyping
- ASIC/SoC/FPGA Verification & Validation
- ASIC/SoC/FPGA Turnkey Services from Design to Tape-out
- IP Product Development, Integration & Support
- Chip Bring-up & Hardware/Board Design Services
- Complete Reference design development

eInfochips executed different projects of the entire silicon development spectrum, including chip tester FPGA Module, EDA Tool verification, 15 million gates chip physical layout, RF Device Tester System, Digital video processing board high definition, and navigation system just to name a few.

Chapter 1

Introduction

1.1 General Description

Today, as the system complexity increases, the traditional capture and simulate methodology has changed to design simulate & synthesize. The logic, functionality and gate counts in a chip are increasing tremendously. With gate count and system complexity growing exponentially, engineers confronts the most perplexed challenge in the product design: functional verification. A bulk of time consumed in design of new ICs and systems is now spent on verification. Engineers are compelled to use the best verification and design tools available to shorten design cycle time. The true path to rapid and accurate system verification includes both tool and methodology innovation.

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System On Chip (SOC) designs, verification consumes about 70% of the design effort. The number of verification engineers is usually twice the number of design engineers. When design projects are completed, the code that implements the test benches makes up to 80% of the total code volume. Verification of design is done at various levels of design phase. Different type of verification is done on the design. The foremost of them is the functional verification that verifies the functionality implemented by the design with respect to the specifications.

1.2 Goal of the Project

The main goal of the project is to verify the submodule of a DUT fully by developing generic environment, and test cases with the help of EDA tools, verification languages and other related important softwares.

1.3 Scope of Work

- The above design is divided in following phases:
- Reading Functional Interface Specifications.
- Identifying Features and writing Testplans.
- Writing Transactor and Testcases.
- Debugging and regression.
- Code coverage and Functional coverage.
- Results and Analysis.

1.4 Organization of Report

This project report is organized into seven chapters.
Chapter 2, Introduction to Verification, mainly describes the types of Verification, Importance of Verification, Verification Methodology and Verification Languages.

Chapter 3, The Verification Methodology and Flow, presents Verification Methodology Flow and description and importance of each step in the Verification process flow.

Chapter 4, Local Move Link, presents the functional Interface Specification of the Local Move Link. It also contains the detailed functionality and implementation of each and every sub block of Local Move Link.

Chapter 5, Local Move Link Sub System Verification, presents the Environment for the Verification of LMvL. It also contains the procedure for the LMvL Verification.

Chapter 6, Functional Coverage Analysis, describes the functional verification and the coverage generated in the ICCR tool. It also contain the integration of the vManager and ICC for the analyzing the coverage.

Finally Chapter 7 EDA tools, contains various tools and verification languages used in this project.

Chapter 2

Introduction to Verification

2.1 What is Verification?

I will start with the IEEE definition of Verification. IEEE defines Verification as "***Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."*** Let's first understand the definition. The design which is to be verified implements some specific functionality as per the requirements. This is to be done by examination and not mere observation. It has to be supported by some objective evidences [8].

Verification can be done at various granularity levels. Depending on the features of the design to be verified, the abstraction level is decided. It follows methodology to accomplish it accurately and quickly. It is a parallel process with the design. Verification engineers and design engineers have to interact a lot to get a verified design at the end of the design cycle. Various EDA tools and languages are available for verification.

Verification Process is conceptually represented using a reconvergence model. It also illustrates what exactly is being verified. The purpose of verification is to ensure that the result of some transformation is as intended or as expected. This is analogous to what we do daily in our life. e.g. confirming bank transactions with the available balance.

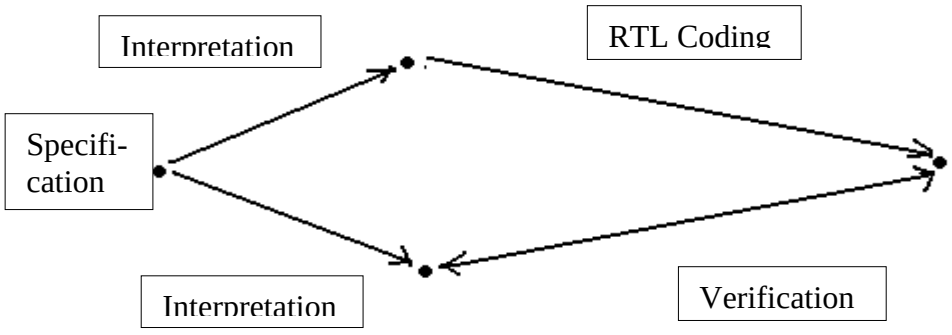


Figure 2.1.1: Functional Verification Reconvergent Model [8]

A design team interprets a written specification document and produces what they believe to be functionally correct synthesizable HDL code. If the same individual performs the verification of the RTL coding that initially required interpretation of a specification, then the common origin is that interpretation and not the specification. In this case, verification verifies designer's interpretation and not the specification. If that interpretation is wrong, then verification will not be able to highlight it. Hence, the process of verification also starts from the specification and verifies the RTL coding against the specifications. Choosing the common origin and reconvergence points determines what is being verified. For functional verification, it is the RTL coding verification against specifications.

Currently, verification is on critical path. It is on target of new tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher level of abstraction and automation. If effort can be parallelized, additional resources can be applied effectively to reduce the total verification time. For this, it is necessary to be able to write and debug testcases in parallel with each other as well as in parallel with the implementation of the design. Higher level of abstraction enables to work fast but this reduces control and hence should be used wisely. The verification process can be at higher level of abstraction by working at the transactions or bus cycle levels instead of dealing with lower level zeros and ones.

2.2 Types of Verification [8]

As explained earlier, in a reconvergence model, the points and path decides the types of verification. Different tools are used for different types of verification. Broadly, verification can be functional verification, formal verification, model checking and testbench generators.

2.2.1 Functional Verification

Functional verification is to verify the functionality implemented by the design against the specification. This can be done at various granularity levels. This depends on how much depth in verification is required based on the deadline. It can show that

the design meets the intended specifications but cannot prove that design is free from any discrepancy. More explanation on this is in the later chapters.

2.2.2 Formal Verification

Establishing properties of hardware or software designs using logic, rather than just testing or informal arguments is formal verification. This involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove, say, that the implementation satisfies the specification.

Equivalence Checking is a type of formal verification. This process mathematically proves that the origin and output are logically equivalent and that the transformation preserves its functionality. It compares two netlist to ensure that some netlist post processing such as scan chain insertion, clock tree synthesis or any manual modification, did not changed the functionality of the circuit. It is also used to verify that the netlist correctly implements the original RTL code. It can be used to see that the synthesizer tool is honest. It can also be used to verify that two RTL descriptions are logically equivalent. Equivalence checking is interested in comparing Boolean and sequential logic functions and not mapping these functions to a specific technology.

Model Checking is a recent application of formal verification. It is a method to algorithmically verify formal systems. This is achieved by verifying if the model satisfies a formal specification. The specification is often written as temporal logic formulas. A model-checking tool accepts system requirements or design (called models) and a property (called specification) that the final system is expected to satisfy. The tool then outputs yes if the given model satisfies given specifications and generates a counterexample otherwise. The decision process often uses some form of binary decision diagram (BDD). Here, assertions or characteristics of a design are formally proven or not proven. For example, all state machines in a design could be checked for unreachable or isolated states. Even deadlocks can be detected. The greatest obstacle to model checking is identifying, through interpretation of the design specification, which assertions to prove. Only a subset of identified assertions is feasible to prove.

2.2.3 Testbench Generation

Here, there is no reconvergence point. The RTL code is the common origin. With the help of code coverage metrics and the source code under analysis, testbench generators generate testbenches to either increase code coverage or to exercise the design to violate a property.

2.3 Functional Verification

I will be concentrating more on functional verification as my project is functional verification of an Tg(Timing Generator) Drv module Stay Functionality -design. The main purpose of functional verification is to ensure that a design implements intended functionality. The starting point for functional verification is specifications. The verification engineer interprets the specification and verifies whether the design coincides with the specifications or not. Functional verification, as a process can show that a design meets the intent of its specifications, but cannot prove it. One can easily prove that a design does not implement a desired functionality by identifying just one discrepancy. But, the converse is not. No one can prove that there are no discrepancies. Functional verification can be accomplished using three complementary but different approaches: black box, white box and grey box.

Black Box Verification

As the name suggests, the design to be verified is looked upon as a black box. i.e. only the interface is known for the design. The internal information of the design is not known. The intention is to verify that the design generates required output for a specific input applied. With a black box approach, the functional verification must be performed without any knowledge of the actual implementation of the design. All verification must be accomplished through the available interfaces, without direct access to the internal state of the design, with knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. It is difficult to set up interesting state combination or to isolate some functionality. It is equally difficult to observe the response from the input and locate the source of the problem.

The advantage of black box approach is that it does not depend on any specific implementation. Whether the design is implemented in a single ASIC, multiple FPGAs or board, is irrelevant. It forms a true conformance verification that can be used to show that a particular design implements the intent of a specification regardless of its implementation. In very large or complex design, black box approach requires some non functional modifications to provide additional visibility and controllability. Additional software accessible registers to control some internal states can be provided. In complex design, some module is taken from third party. This IP (Intellectual Property) is fully verified, but to verify its working within our design, it is verified using black box approach.

White Box Verification

As the name suggests, white box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This approach has the advantage of being able to quickly setup an interesting combination of states and inputs or isolate any functionality. Results can be observed as verification progresses and the source of any problem can be located. This approach is tightly integrated with a particular implementation and cannot be used on alternative implementations or future redesigns. It also requires detailed

knowledge of the design implementation to know which significant conditions to create and which results to observe. In my project, for module level functional verification, all the modules were verified using white box approach. This approach ensures that design behave properly with respect to any functionality. All FIFOs, counters or datapaths are appropriately steered and sequenced.

This approach is the main verification used to verify any design. It is the foremost important approach to verify any design that is done for the first time. It verifies the design from all respects. The designer modifies and updates the design as per the feedback from the functional verification team. The final outcome is the completely verified and almost correct design which can be synthesized and fabricated further.

Grey Box Functional Verification

Grey box approach is a compromise between the aloofness of a black box approach and the dependence on the implementation of white box approach. As in black box approach, a grey box approach controls and observes a design through its top level interfaces, but it is aware of the internal controls and can use them. This approach is used based on the priority of the features to be verified. If functionality is to be verified is not prime one, then to attain the deadline, grey box approach can be used.

2.4 Verification Methodology

Methodology is the step by step procedure to be followed for successful accomplishment of any project. As digital logic designs grow larger and more complex, functional verification has become the number one bottleneck in the design process. Reducing verification time is crucial to project success. The only way to address this problem is to adopt a reuse-oriented, coverage-driven verification methodology built on the rich semantic support of a standard language. For Verification, the Methodology used contributes great to the final conclusion for the process. Design Methodology covers from plan to closure and it includes the Verification Methodology midway. Some Approaches are stated under:

Constrained Random Stimulus Generation

Traditional verification relies on directed tests, in which the testbench contains code to explicitly create scenarios, provide stimulus to the design, and check (manually or with self-checks) results at the end of simulation. Directed testbenches may also use a limited amount of randomization, often by creating random data values rather than simply filling in each data element with a predetermined value. By building randomization into the types of scenarios that are created, not just in the data values that get generated, additional tests are much more likely to hit corner cases and thereby find more design bugs.

Coverage-Driven Verification

Coverage metrics serve two critical purposes throughout the verification process. The first is to identify holes in the process by pointing to areas of the design that have not

yet been sufficiently verified. This helps to direct the verification effort by answering the key question of what to do next — for example, which directed test to write or how to vary the parameters for constrained-random testing.

Assertions based Verification

The capabilities of any verification environment can be enhanced by the addition of assertions, which are statements of design intent. Ideally, as the designer writes the RTL, he or she documents with assertions the requirements on how the design is expected to behave and the assumptions on interfaces with adjoining blocks. Assertions can range from low-level statements about how specific design elements should behave to high-level, end-to-end rules about how information should flow through a design. Assertions can be specified in many ways, including with general RTL expressions, special statements within hardware verification languages, and the built-in assertion constructs

2.5 High Level Verification Languages

Verilog and VHDL were widely used for verification languages. Due to increase in complexity of functionality within designs, the need for developing language that would aid verification grew. As a part of it, many new verification languages developed. These languages are very powerful in creating conditions which

Verification Engineers require to verify the design from all aspects. Many features of powerful languages are blended together in these languages to support all new approaches. I will be discussing some high level verification languages. These languages are now evolving fr both design and verification. Today's system-on-a-chip designs require multi-discipline engineering teams with a range of skills covering embedded software, system architecture, RTL design and verification. Traditionally these teams use a variety of C modeling styles for architecture design and a variety of hardware description languages (HDLs) and hardware verification languages (HVLs) for RTL design and verification. These traditional methods have led to very complex design flows, prohibited reuse, and have increased the total time to market and development costs for today’s chip designs.

Two industry standards have emerged to allow convergence of the different C-based and HDL and HVL-based approaches. These are SystemC, for C-based system-level modeling and SystemVerilog, providing a unified language for RTL design and verification. Both SystemC and SystemVerilog span multiple levels of abstraction. These languages can support verification at transaction level of abstraction. They enable ease call to functions of other languages and hence provide good interface. Assertion based approach is well supported. Object oriented approach and reusability is taken care of in these languages.

System Verilog

IEEE 1800TM SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard. SystemVerilog is a major extension of the established IEEE 1364TM Verilog language. It was developed originally by Accellera to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips. SystemVerilog is targeted primarily at the chip implementation and verification flow, with powerful links to the system-level design flow. SystemVerilog has been adopted by 100's of semiconductor design companies and supported by more than 75 EDA, IP and training solutions worldwide.

System C

SystemC provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software. SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools. The Open SystemC Initiative (OSCI) is an independent not-for-profit organization composed of a broad range of companies, universities and individuals dedicated to supporting and advancing SystemC as an open source standard for system-level design.

Vera

Vera is an industry-leading testbench automation product that increases design quality by finding simple as well as corner-case bugs, quickly. Vera allows engineers to create coverage-driven tests using advanced testbench concepts like constrained-random stimulus generation, real-time data and temporal checking and extensive analysis of functional coverage. Vera combines next-generation constraint solving and coverage analysis engines with a proven reference verification methodology and interfaces to leading Verilog and VHDL simulators. Vera supports the OpenVera® hardware verification language, including OpenVera Assertions, and is an integral part of the Synopsys Discovery™ Verification Platform.

Specman – e Language

It is the most powerful HVL. Specman is the compiler/debugger/simulator is for e language. Specman Elite offers a comprehensive verification environment that is based on the e hardware verification language (HVL). The Verisity's Specman Elite is acquired now by Cadence. It is playing an important part in developing reusable verification components.

2.6 The Cost of Verification

Verification is a necessary evil. It always takes too long and costs too much.Verification does not generate a profit or make money: after all it is the design being verified that will be sold and ultimately make money. Yet verification is indispensable. To be marketable and create revenues, a design must be functionally correct and provide the benefits that the customer requires. Today, in the era of IP and reusable components, verification is foremost to gain trust in the design.

Chapter 3

The Verification Methodology & Flow

3.1 Introduction

Methodology is defined as

(1) "*a body of methods, rules, and postulates employed by a discipline*"

(2) "*a particular procedure or set of procedures*"

(3) "*the analysis of the principles or procedures of inquiry in a particular field*".

The common idea here is the collection, the comparative study, and the critique of the individual methods that are used in a given discipline or field of inquiry. Methodology refers to more than a simple set of methods; rather it refers to the rationale and the philosophical assumptions that underlie a particular task.

Verification is a critical part in the specification to silicon path. Hence, it should be done with proper planning and proper methods to make the process effective and quicker. The ultimate aim is to attain the most critical challenges while maximizing overall speed and efficiency. Following an appropriate path is very important for this. Hence, for any verification process, first the approaches, methods, algorithms, sequences, etc is decided upon from all aspects.

3.2 Verification Methodology

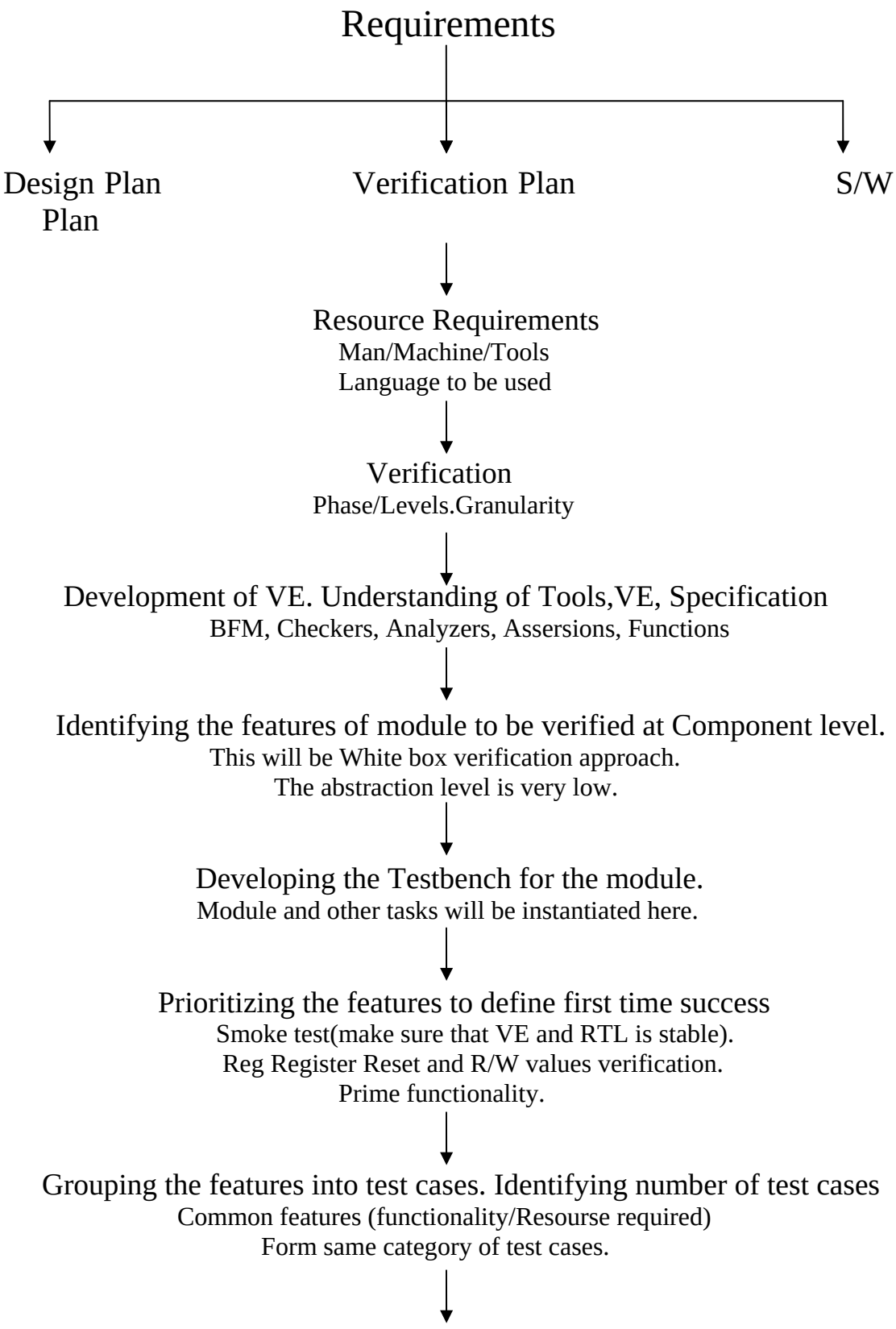
As already mentioned earlier, the methodology adopted plays a vital role in the progress and accomplishment of the process. Verification remains the single biggest challenge in the design of system-on-chip (SoC) devices and reusable IP blocks. As designs continue to grow in size and complexity, new techniques emerge that must be linked by an effective methodology for significant adoption and deployment. The SoC industry needs a reuse-oriented, coverage-driven verification methodology built on the rich semantic support of a standard language. Different approaches are possible for targeting verification of a design.

3.2.1 The Bottom Top Methodology

In the project, Bottom Top Methodology is followed. The designers have written the RTL Code for the design. But the design is not complete yet. It will be modified and new features will be added based on the feedback from verification process. In Bottom Top Methodology, the verification starts with from the base of the design. In the project, the complete design is divided into functional modules which can be individually verified. These modules are to be verified using white box approach with full visibility. The designer and verification engineer interact to find bugs with the design and modify the RTL to get a functionally correct design. The level of abstraction at module level is very low. Not many assumptions are made. Interest is to verify the design with transaction in each signal. The modules are to be verified for all possible inputs and even the invalid inputs. Corner cases are to be identified to verify the design from all aspects. After the completion of this phase, the level of abstraction rises. Now all the modules are to be integrated to and verification is done for correctness of the interactions between the modules. At this time, the modules are assumed to be functionally correct and only there mutual interactions are verified. After this phase system level verification is done. The design is verified for its functionality with the all peripherals and system components. The level of abstraction is highest here. The approach will be clearer as I start with the Verification flow.

3.3 Verification Flow [4]

I will be discussing the complete flow followed by the company to achieve the verification. I will start with common approach for any designing and will proceed with concentration on the verification. Any project is the outcome of some requirement. As without any requirement, there is no profit in going for any project. Hence, depending on the market requirement and market availability for the project product, a project is finalized. Figure 3.3.1 below shows the Verification Flow followed for the project. It shows the complete flow from Plan to Closure. Different Tools are used at each step. The Flow goes in loop as the project progresses. Depending on the feedback from tool or updation of RTL due to addition of new feature or fixing of some bug, the flow undergoes in a loop.


Requirements
Design Plan
Plan
Verification Plan
S/W
Resource Requirements
Man/Machine/Tools
Language to be used
Verification
Phase/Levels.Granularity
Development of VE. Understanding of Tools,VE, Specification
BFM, Checkers, Analyzers, Assersions, Functions
Identifying the features of module to be verified at Component level.
This will be White box verification approach.
The abstraction level is very low.
Developing the Testbench for the module.
Module and other tasks will be instantiated here.
Prioritizing the features to define first time success
Smoke test(make sure that VE and RTL is stable).
Reg Register Reset and R/W values verification.
Prime functionality.
Grouping the features into test cases. Identifying number of test cases
Common features (functionality/Resourse required)
Form same category of test cases.

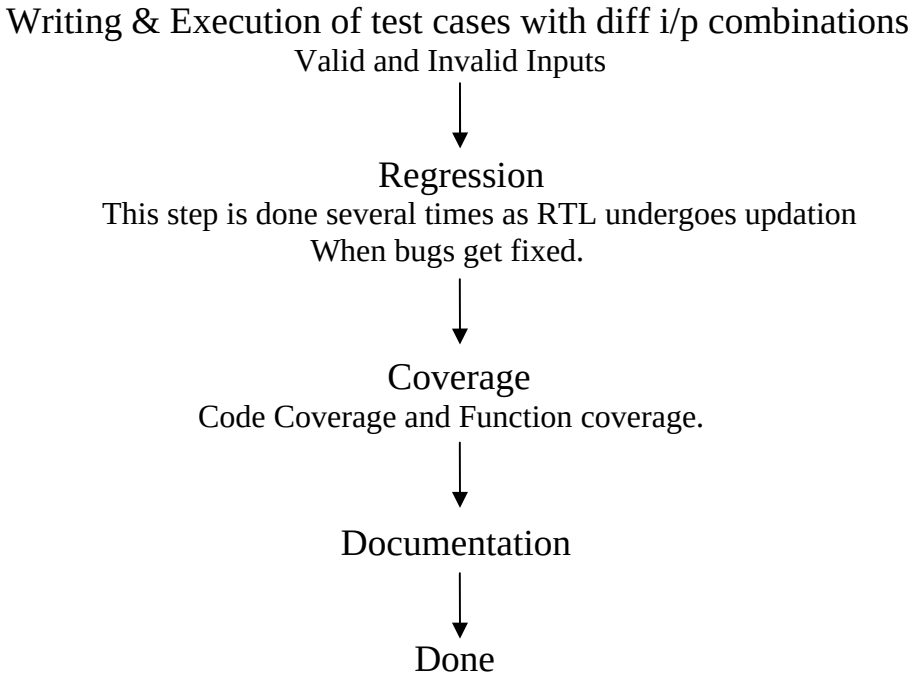


Figure 3.3.1: Verification Flow

3.4.1 Design Plan, Software Plan & Verification Plan

The Project Plan decides other plans for the project. These are the Design Plan, Software Plan and the Verification Plan. These three teams play part in the project. These are the most important teams responsible for successful completion of the project. Design Plan and Software Plan states all the requirements from there point of view.

3.4.2 Verification Plan

The Verification plan acts as a specification for the verification effort. It is used to define what first time success is, how the design is verified and which testbenches are written. The verification plan states everything related to Man/Machine & Tool requirements. It gives the approach to verify every module. The verification plan includes:

1. Requirements (Man/Machine) Verification.
2. The languages to be used for verification.
3. The tools to be used for all tasks in verification.
4. The engineers to be worked on the project and the duties to be assigned to them.
5. The requirements in terms of System OS, S/W Drivers, Application S/W, etc.

6. The CHIP configuration.
7. Level of granularity/abstraction for all modules to be verified.
8. Division of the complete verification process into different phases with deadlines defined for each of them.
9. All the verification specific things.

The verification process is divided into phases.

First Phase: This is the Block level verification. Each block is assigned to individual and is verified with interactions with the designer.

Second Phase: This is Cluster level verification. Few blocks with high mutual interactions are combined and are verified.

Third Phase: This is Full Chip level verification. All blocks are combined and the functionality of the complete design is verified.

Fourth Phase: This is System level verification. The chip is verified with the system in which it will be used. Third Party Models are included here.

3.4.3 Development of VE

As the complete plan for all three major team is ready, now they start working on the project. Design team starts the designing of individual blocks depending on the architecture finalized. Software team starts with the implementation of algorithm to be used to make sure that it will work in the design. If the algorithm works well, then RTL coding for it is done. The software team then starts with the necessary software required by the project.

The verification team's task starts with complete understanding of the specification. In our project, the chip works on a protocol. So, we need to completely understand that the protocol first and then the specifications stated for each block. While verifying, it is required to also check that designer stick to the protocol. Then all the tools to be use used is studied. I have described all the tools used in our project in detail in later chapters. Then starts the development of Verification Environment. All the requirements required from VE is mentioned in verification plan. The VE is developed accordingly. The VE in our project is in C++ and Verilog. The BFMs

(Bus Functional Models), Checkers, Analyzers, Assertions, etc are written as a part of VE.

When the design will be verified, it is required that the design should be given the same environment which it will find when it will be actually used in the system. The testbench is like a universe for the design. Testbench is a closed system. It generates all the inputs and compares the outputs generated by the design. Testbench will configure the design as per the requirements by the test case which verifies design functionality. It is important to be able to develop generic testbench which is applicable for verification of all functionality of the design. The testbench is updated as per the feedback from verification process later.

3.4.4 Identifying Features

The development of VE is completed at this point. It would be modified or updated later depending on the requirements as verification proceeds. Now, each verification engineer is given individual module and it is his/her responsibility to verify it. Now, starts the identifications of the features to be verified. This is a white box approach and hence all the features have to be verified. There would be some features which are to be verified at full chip level and not at module level. e.g. interrupt propagation structure.

3.4.5 Prioritizing Features

For any design, few features are of prime importance. Bugs in such functionality leads to more bugs in the functionalities depending on them. Hence, such functionality must be considered first. In any design, you will find many configurable registers. The configuration, operation and output from the design depend on these registers. So, it is of prime importance to see that all registers are functionally correct.

3.4.6 Grouping into Testcases

The ultimate aim is to verify all the features of the design within the deadline. So, time taken is very important. Hence, the prime objective is to keep the number of test

cases less and still be able to verify all the features. Hence, the features are grouped into same category of test cases depending on the functionality they implement. Some

features require similar configuration, granularity or verification strategies. Hence, to increase the productivity, these features should be grouped into common testcase. Again here the priority among the features is always taken care of. Now, all the test cases required are known. Number of testcases required for each functionality verification is known.

3.4.7 Writing & Execution of Testcases

The test cases are written and are executed with valid and invalid inputs. All possible corner cases are applied on the design. For the failing test cases, debugging is done. The source of problem is found. The issue could be related to VE, Test Case or RTL. Once it is confirmed that the problem is related to the RTL, bug is filled for that. Designer debugs the design and locates the problem. The RTL is modified accordingly to fix the bug. The test case is executed again. This process repeats till the test cases passes.

3.4.8 Regression

As and when bugs are filed, the designer debugs the RTL and modifies it. Hence, the RTL goes on updating as the verification progresses. Hence, there can be a chance that a test case passing previously might fail with this new updated RTL. Hence, after every updating of RTL, regression of all test cases is done. Again, at the end of the verification, all test cases are executed for the same.

3.4.9 Coverage [9]

This is very important task to which defines the effort placed by verification engineers and the completion of the verification task. From the feedback from the coverage tool, new test cases required are written and executed. It is desirable to get 100% coverage. But this is a difficult task. Depending on the deadline of the project, the coverage required can be loosen.

3.4.10 Documentation

This is the final step in the process and declares the closure of the verification process. It includes documenting all the effort done and the results generated. This documentation can be used by some other person in case the design is updated later.

3.5 Designer – Verification Engineer Interaction

At all the steps discussed earlier, interaction between designers and verification engineers is required. Any open bug is discussed by both to come to a conclusion. The partition of the design into modules is done by the both teams together. Many Design for Verification features are added to the RTL to aid the verification process. Some more register or some multiplexer to bypass some functionality can be added.

3.5.1 Design for Verification

There are two major reasons for the presence of design errors (bugs) in a design. First, the sheer complexity of a module, often including multiple-state machines, makes it virtually impossible to anticipate all possible conditions to which the module can be subjected in the context of an application. Typically the state space is very large and bugs can be buried very deep into the logic. Hence, some corner cases may simply not have been anticipated in the implementation. Second, designing a module often requires the designer to assume a particular behavior on the interface, that is, make assumptions on the behavior of modules physically connected to the module under design. These assumptions are needed to assure minimum area/maximum speed micro architectures to be designed. To improve the quality of the design process we clearly have to address specification, design, and verification in a concerted manner. Similar to other seemingly independent tasks in the past such as manufacturing test, design quality needs to become the whole team's concern, and the methodology employed must support this notion. That is exactly what the DFV methodology offers coherent methodology to find design errors through constrained-random stimulus generation and advanced tools for formal state-space analysis, powerful means to eliminate ambiguity in specifications, and improved conformance checking capabilities to ensure that the imported design IP complies with required standards.

Chapter 4

Local Move Link

Now a day there is a revolutionary growth in the field of chip development, it further revolutionized growth of instruments that verify chips. So to ensure correct functioning of the chips, market of tester instruments is increasing. Depending on the customer's requirements and need, the company came up with a chip tester instrument having a SoC DART.

4.1 Local Move Link [1]

Figure 4.1.1 shows the block diagram of LMvL. From the block diagram, it is seen that there are many independent modules which performs specific functionality. Each verification engineer owns individual/multiple module block/s verification. Local Move Link (LMvL) is one of the block of chip, which interfaces the DigCap module and Processor outside the chip to analyze the capture data captured by DigCap.

The L3 Packet Assembler receives and acknowledges move requests from/to the DigCap, and starts the move of data onto the LMvL by building an L3 packet. It requests and receives data from the DigCap memory controller and forms that data into packets, which are stored into the packet buffers. The packet buffers are a staging area for packet data between the DigCap memory and the LMvL, which run at different data rates. The packet buffers also bring the packet data path across frequency domains, from clk_ddr to clk_db. The L2 Packet Burst Controller detects and reads packets from the packet buffer and bursts them following an L2 header onto the LMvL interface, which handles the physical interface to the LMvL data and control lines. The LMvL Fifo & Bypass block serves as an interface to the external LMvL data and control lines, with the ability to bypass the chip if no data moves are required. My Project work for LMvL is the Functional Verification of LMvL SubSystem – L3 Packet Assembler.

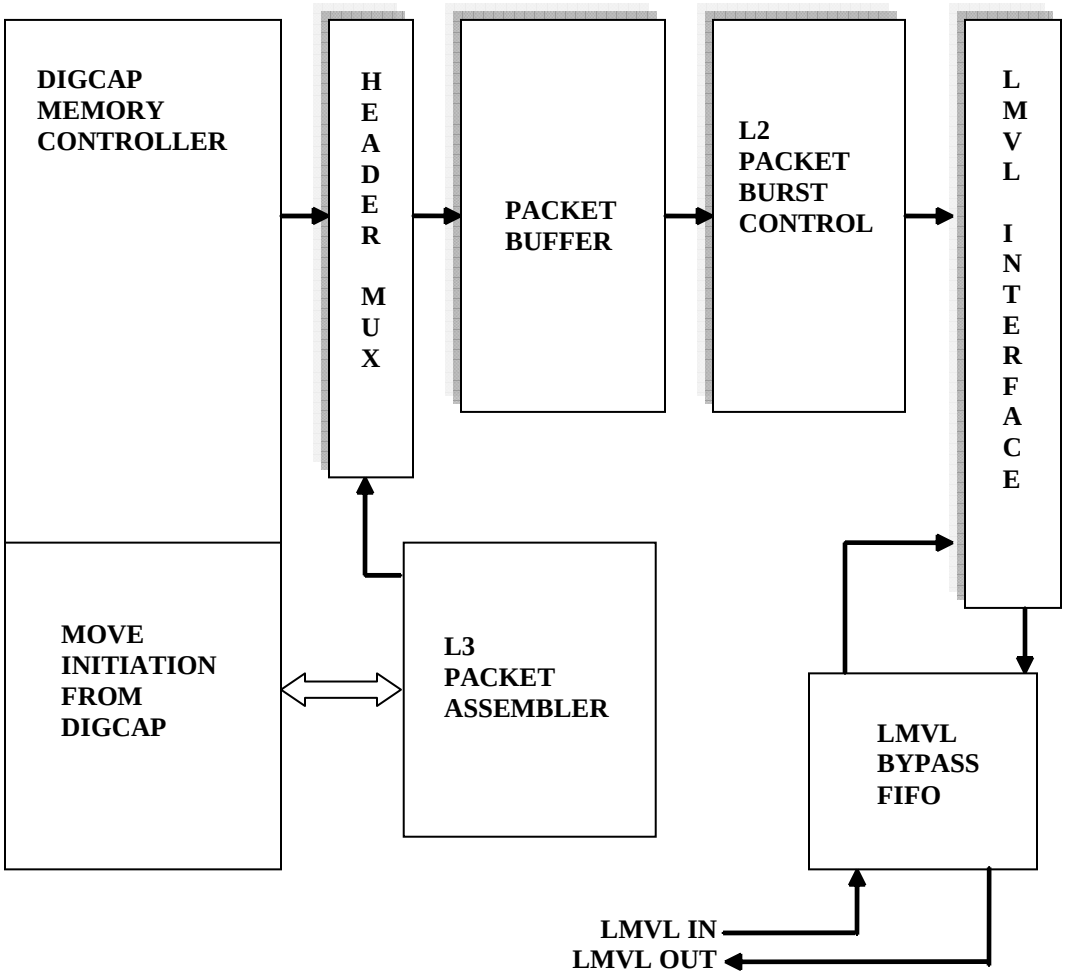


Figure 4.1.1: Block Diagram of LMvL[1]

4.2 L3 Packet Assembler

There is one L3 Packet Assembler per LMvL. It builds packets from capture data in the capture DDR-SDRAM and loads them into the packet buffers of each of the 8 channels. It Accepts move requests from the DigCap and sends requests for data to the DigCap memory controller. It Builds the L3 packet header and trailer, and assembles the packets by combining the L3 header and trailer information with the capture data from the DigCap memory. The packets are assembled as they are stuffed into the packet buffers. Packet Assembler Signals the L2 Packet Burst Control block when a packet is ready by

way of the packet buffer status bits. It also handshakes with the per-chip DigCap to inform it that the requested move has completed.

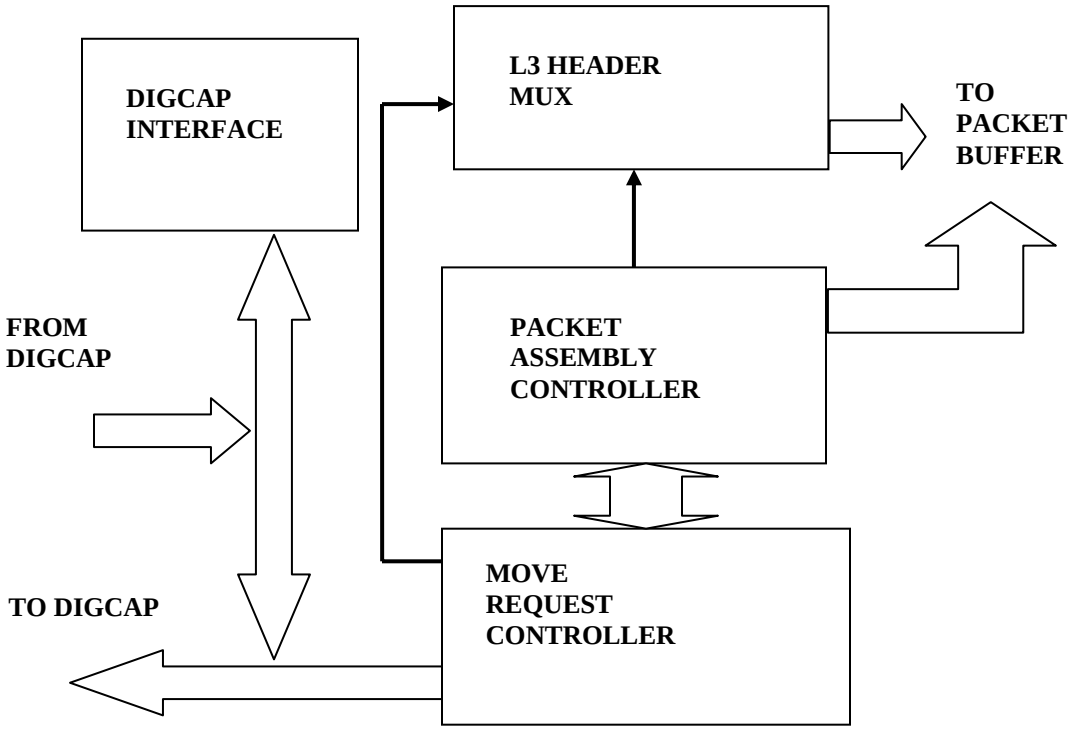


Figure 4.2.1: Block Diagram of LMvL Packet Assembler [1]

4.3 DigCap interface and requirements

The L3 Packet Assembler needs to interface with the DigCap functional block.

- DigCap initiates a move by asserting the i_mv_rqst_ddr pin. This pin is all the LMvL knows about move initiation.

- DigCap identifies when all data for the current move has been sent to the LMvL by asserting the i_rqst_done_ddr pin on the same cycle that the last data word is transferred from DigCap to LMvL.

- The DigCap cannot send another move request until the LMvL has acknowledged the previous move by asserting the o_mv_done_ddr line high.

- The 16 bit Segment Identification Number from the DigCap to be inserted into word 3 of the L3 header is sent to the LMvL on the i_ch#_pkt_payld lines during move initiation. ID is defined on when i_mv_rqst_ddr is asserted.

4.4 Packet Assembly Controller

As shown in Figure 4.4.1 It interfaces to the Packet Buffers, the Header Mux, and the Move Request Controller. The Packet Assembly Controller controls the flow of L3 header and L3 trailer data into the packet buffers, and monitors the flow of payload data into the packet buffers from the capture DDR-SDRAM controller.

It writes packet size and the "last" bit into the Packet Buffer status buffers. It also sets and monitors the Packet Buffer "Packet Present" bits to indicate a packet is ready to be sent and to determine if the current packet buffer is empty, ready for a new packet. The Packet Assembly Controller is made up of three counters and two state machines.

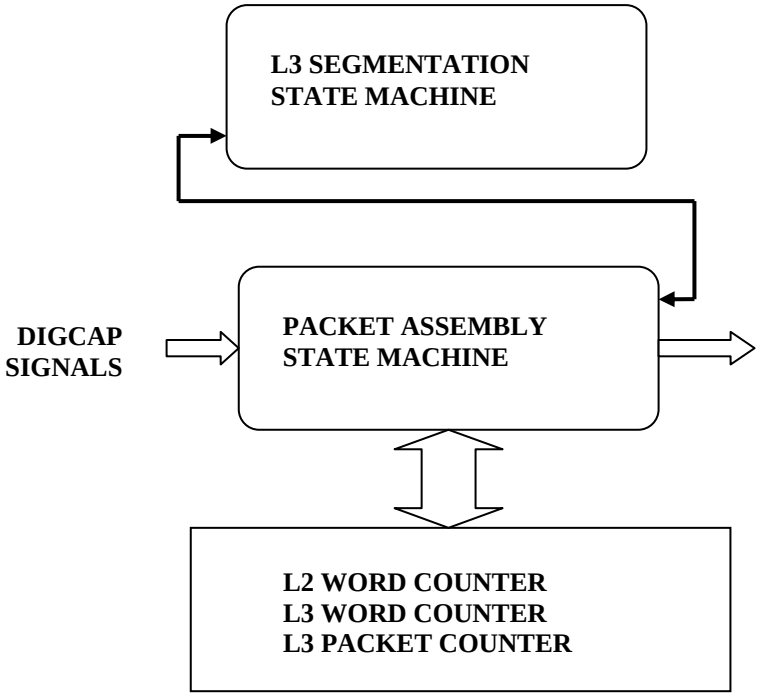


Figure 4.4.1 Block Diagram of LMvL Packet Assembly Controller

4.4.1 Packet Assembler State Machine

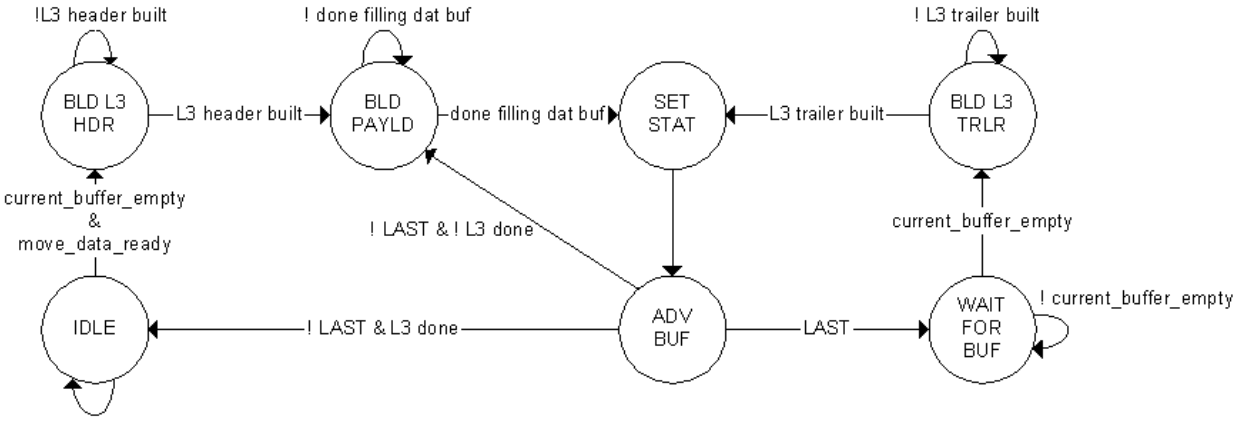


Figure 4.4.1.1: State Machine of LMvL Packet Assembler

The Packet Assembler State Machine is implemented with the inputs coming from DigCap, Move Request Controller, Packet Buffer and DRAM Controller. The output state is given to the Move Request Controller State Machine, Header Mux and Packet Buffer.

4.4.2 Packet Assembler Counters

L2 Packet Word Counter [4:0]: It counts running count of the number of 16 bit words loaded in the current packet buffer. Reset to zero when advancing to the next packet buffer. Increments during L3 header build, or when packet payload write enable = 1. This value is used in various next state calculations as well as to define the packet status buffer "size" field.

L3 Packet Word Counter [27:0]: It counts running count of the number of 16 bit words already loaded into the packet buffer for the current L3 packet. Valid values are all within address range. Reset to zero before starting a new L3 packet. Increments during L3 header build, L3 trailer build, and on every cycle when packet payload write enable = 1. This value is compared to the value in the payload size register to determine when the current L3 packet is complete. This value is converted to a 32-bit word count value to become the "word count" field in the L3 trailer.

L3 Packet Counter [7:0]: Count of the number of L3 packets sent. Incremented each time the packet assembly state machine leaves the assembly state build L3 trailer. Databus can reset and program this value. Software will reset this value at the beginning of a pattern burst. This count becomes the "L3 Count" field in the L3 header.

4.4.3 L3 Segmentation State Machine

The L3 Segmentation State Machine determines what type of L3 packet is currently being burst. There are 4 types of L3 packets:

1. L3_FIRST = 2'b01
2. L3_MID = 2'b00
3. L3_LAST = 2'b10
4. L3_ONLY = 2'b11

The type is dependent on:

1. The size of the transfer from the DigCap.
2. The size of L3 packets as defined by L3 Pay Size Register.
3. The previous L3 packet type.

The L3 packet type is used to define the L3 trailer Start/End Indicator field, so the output of this L3 Segmentation State Machine module feeds directly into this trailer field in the Header Mux.

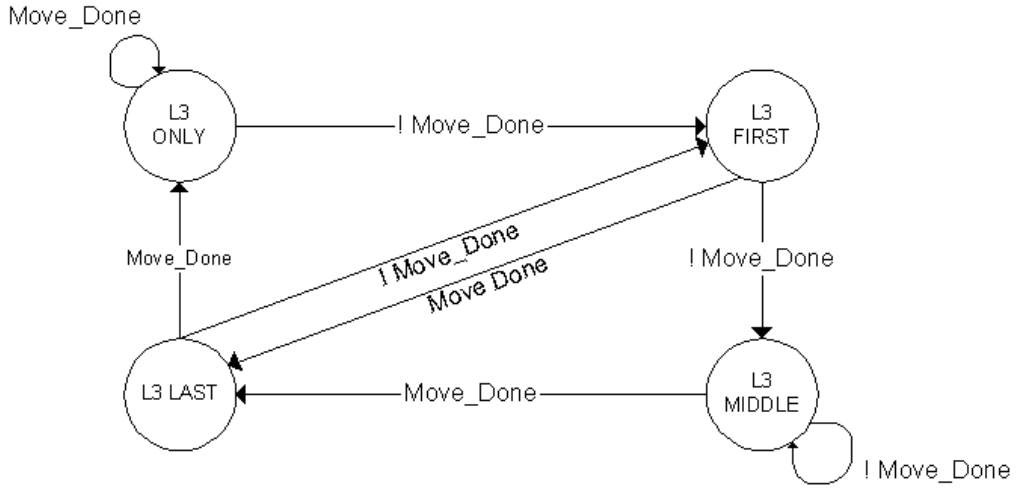


Figure 4.4.3.1: L3 Segmentation State Machine

4.4.4 L3 Header Mux

The L3 Header Mux gives the Packet Assembly Controller the ability to multiplex in header and trailer information onto the data path between the capture DDR-SDRAM controller and the Packet Buffers.

It also contains logic that generates a channel mask forwarded to and used by the packet buffers to determine which channels are participating in the current move. Some values are assigned on the fly by DigCap or by the state machines inside the L3 Packet Assembler. A channel mask is generated and passed along with the Packet Data and write enable outputs of the Header Mux to the Packet Buffers. The per channel channel packet payload write enable register is used to communicate which channels will participate in the current move request. The value on that register is latched when the move request register pin is asserted by DigCap.

4.5 Move Request Controller[1]

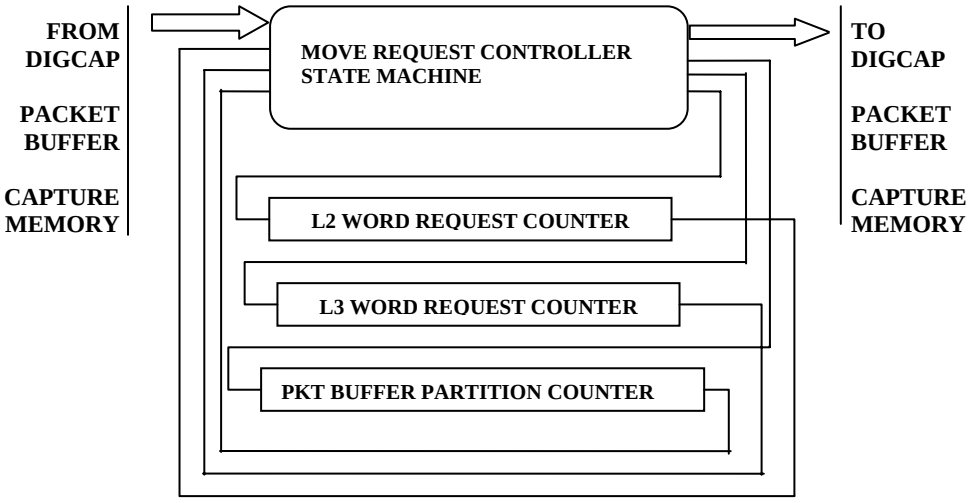


Figure 4.5.1: Move Request Controller

The Move Request Controller interfaces with the DigCap, Packet Buffers, and the other blocks of the L3 Packet Assembler. It receives requests to build and send an L3 packet

and interfaces with the external memory controller to access the payload data for the L3 packet.

Move requests are received from the DigCap by way of the move request register signal. This signal indicates when the data is available to be accessed. Only one move request is submitted at a time. No new requests are accepted until the current request is completed and acknowledged.

The LMvL does not know what size the total data transfer will be until all the data has been received from the DigCap. When all data for the current move request has been retrieved from capture memory and transferred to the LMvL RLM, the DigCap signals that the transfer is complete on the request done register signal. The DigCap asserts the request register signal on the same cycle that the last data word of the move is transferred from the DigCap to the LMvL.

When the LMvL completes the pending move request (final complete L3 packet loaded into the Packet Buffer), output move done register is set high, indicating to the DigCap that the move is complete. Only then is the DigCap allowed to submit another move request.

The Move Request Controller will request only enough data to fill an L3 Packet the size of that specified by L3 packet size register, and will then wait for the Packet Assembly Controller to finish building the current L3 packet and the header for the next L3 Packet before submitting more data requests.

Data requests are sent to the DigCap memory controller by way of the output 32 bit request register is set high during a clock cycle to submit a request for 32 bits of data per channel. (Inside the DigCap, this signal increments a counter, which represents pending 4 byte word requests). Requests for data sent to the DigCap are based on available packet buffer space.

The Move Request Controller determines if the currently selected packet buffer is empty by oring together the “packet present” bits from each channels packet buffer. It knows how much space is available by keeping track of which packet buffer is the first in an L3, indicating it has space for 256 bits. All other packet buffers have 384 bits of available space to fill.

4.5.1 Counters

Move Request Pending Buffer Count [1:0] - Count of packet buffer partitions with pending data requests submitted to the DigCap. Incremented by the Move Request State Machine in the advanced buffer pointer state.

L2 Word Request Counter [4:0] – Counts running count of the number of 16 bit words already requested from the DigCap DDR-SDRAM controller.

L3 word request counter [25:0] – Counts running count of the number of 32 bit words already requested from the DigCap DDR-SDRAM controller.

4.6 PACKET BUFFER

The Packet Buffer is per Channel Move Block. It is positioned between the L3 Packet Assembler and the L2 Packet Burst Controller. L3 packets staged in the packet buffer in preparation for transmission onto the LMvL.

Each Packet Buffer contains a Data Buffer holds the actual packet payload from the DigCap DDR-SDRAM, as well as L3 header information.

The Data Buffer is made from a 72 X 16 register array. It is partitioned down to three 24 X 16, or 384 bit data buffers.

Status Buffer holds data describing the contents of the data buffer. The Status Buffer is a 3 X 6 buffer made from discrete registers. Each of the three locations hold 6 bits that

represent status of the corresponding data buffer. The first 5 bits are the SIZE field, which represent the number of 16 bit words in the corresponding Data Buffer. The 6th bit is the LAST field, which indicates if the data in the corresponding Data Buffer is the last L2 Packet of an L3 Packet.

Status Flags called “Packet Present Bits”: are used by the controllers on each side to status if the Packet Buffers are empty. The three “Packet Present” bits correspond to each of the three partitions in the data and Status Buffers. with the L3 Packet Assembler able to set the “Packet Present” bit when finished filling the corresponding data and Status Buffers, and the L2 Burst Controller able to clear the “Packet Present” bit when finished reading the corresponding data and status buffers. The reset value of the Packet Present bits is low. The packet buffer is sized and partitioned such that three maximum sized LMvL L2 Packet payloads may be stored at one time.

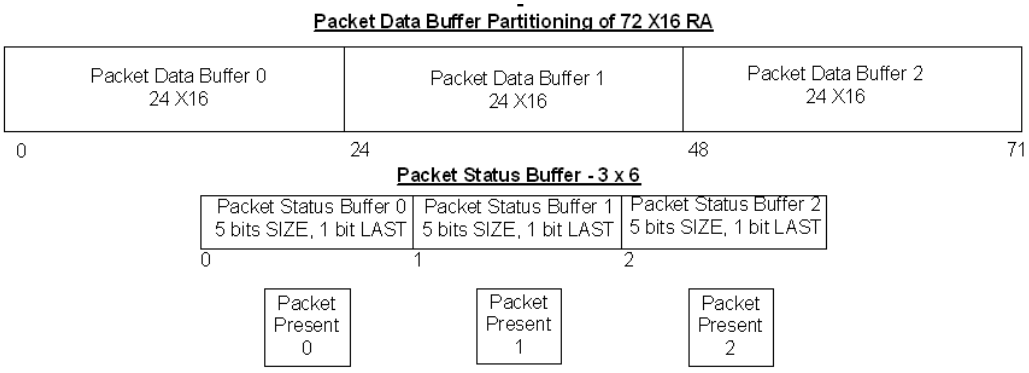


Figure 4.6.1: Packet Buffer Partitioning

4.7 L2 Packet Burst Control [1]

It detects if a packet in the Packet Buffer is ready to be sent. If a packet is ready, a request is sent to the LMvL Interface to catch the Bus Available Token. The LMvL Interface responds to indicate when the Bus Available Token is latched, signaling that the packet burst may begin. The L2 Packet Burst Control module then sends an L2 header onto the LMvL Interface module, followed by L2 packet data read from the Packet Buffers. The L2 Packet Counter keeps track of the number of L2 packets that have been sent. It is set to zero when the packet burst state machine is in the IDLE state. It is incremented each

time a packet burst is completed, when the Burst Control State Machine is in the BURST DONE state. It is compared to the maximum L2 packets per bus available token possession Databus register to determine if another packet should be sent. If the counter is greater then the limit, the burst control state machine goes into the IDLE state and the Bus Available Token is released.

4.8 LMvL Interface Block

In the following Figure 4.8.1 after reset, the LMvL Interface is in a pass through mode. Data is received from the previous device, registered, and then sent on to the output ports to the next device. When the bus available token is received, the LMvL Controller state machine sends data and/or forwards the bus available token. The controller then returns to pass through mode. The following state diagrams, state table, and waveform diagrams describe this modules operation in more detail.

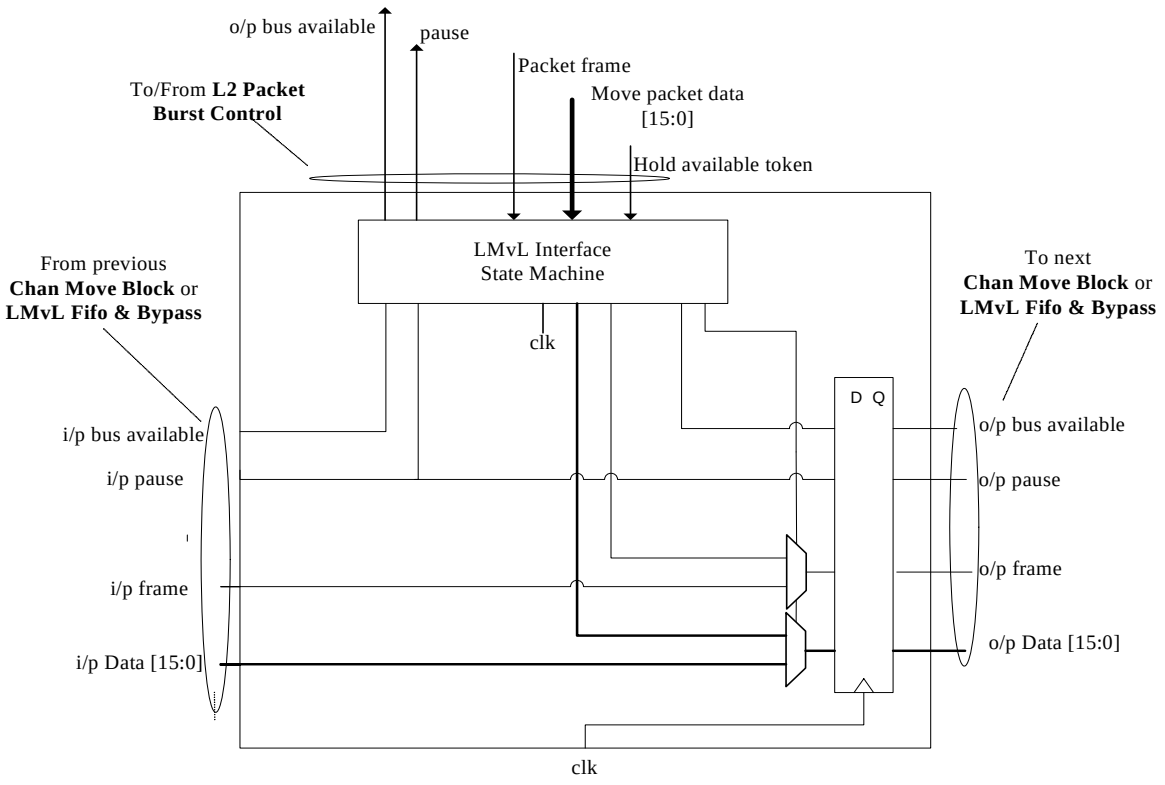


Figure 4.8.1: LMvL Interface Block Diagram

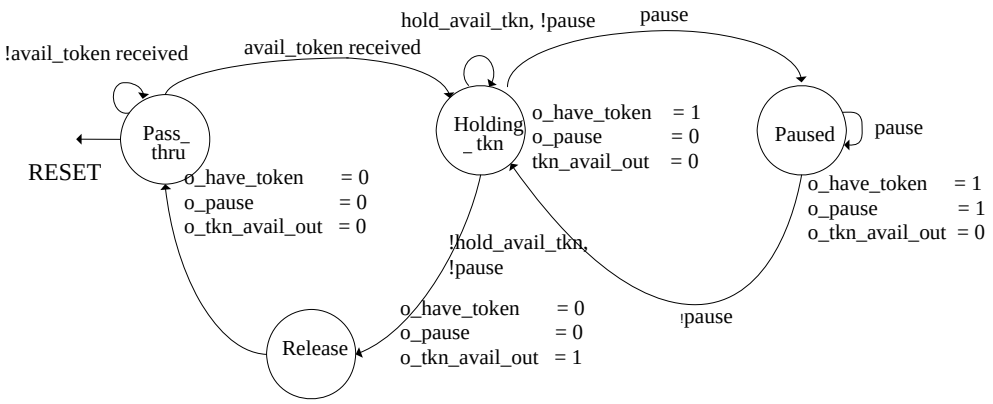


Figure 4.8.2: LMvL Interface State Machine

The LMvL interface state machine connects to three sets of signals. The first are the signals coming from the previous channel (or LMvL Fifo & Bypass module for channel 0) which include all the LMvL token, framing, and data bits. The second are the LMvL signals going to the next channel (or LMvL Fifo & Bypass module for channel 7). The third are the signals going to and from the L2 Packet Burst Controller module, which include framing and data bits, as well as handshaking signals to communicate orders to hold the bus available token and identify when it is ok to burst a packet.

PASS_THRU

At reset, the LMvL interface state machine is in the PASS_THRU state. In this state, o_have_token_db, which is tied to the select on the output mux, is low. This creates a data path from the LMvL inputs from the previous channel, to the LMvL outputs to the next channel. The state machine does not leave PASS_THRU state until the bus available token is received on the I_lmvl_bus_avail_in_db line.

HOLDING_TKN

In the HOLDING_TKN state, the o_have_token_db signal goes high, disconnecting the bypass path, and connecting the inner L2 packet burst controller to the LMvL outputs. o_have_token_db is also sent to the L2 packet burst controller which goes high to indicate that this channel has possession of the bus available token and it is now ok to

send a packet onto the LMvL. The state machine will stay in this state (in other words, “hold the bus available token”), until the L2 packet burst controller allows it to release the token, sending the state matching into the RELEASE_TKN state, or it is paused by the I_lmvl_pause_in_db signal, sending the state machine into the PAUSED state.

When the L2 packet burst controller has no data to send, the I_hold_avail_tkn_db signal will be low. When a channel goes into HOLDING_TKN state that does not have any packets ready to be sent, the state machine will advance to the RELEASE_TKN state in only one cycle. This is important to understand when calculating the per channel latency of the bus available token through channels with no packet data. If the L2 packet burst controller has data to send, the I_hold_avail_tkn_db signal will be high. The state machine will release the bus available token until the I_hold_avail_tkn_db signal returns low.

PAUSED

When the SLC needs to flow control the LMvL, it does so by holding the “pause” bit high. Then the I_lmvl_pause_in_db signal goes high with the state machine in the HOLDING_TKN state, the state machine jumps into the PAUSED state. In this state, the o_pause_db signal to the L2 packet burst controller goes high, signaling that a new packet burst should not be started. The output mux select signal, which is connected to o_have_token_db, stays high to allow the current packet burst to complete without interruption. The state machine stays in the PAUSED state until the I_hold_avail_tkn_db signal low.

RELEASE_TKN

When not paused, and the L2 packet burst controller is done (or doesn’t want to start) sending packets, the state machine jumps from the HOLDING_TKN to the RELEASE_TKN state. In this state, the o_lmvl_bus_avail_out_db signal is set high. Since o_lmvl_bus_avail_out_db is low in all other states, and RELEASE_TKN is a one cycle state, a one clock cycle wide pulse is sent onto the o_lmvl_bus_avail_out_db line.

Note that the bus available token will NOT flow through the channel path inside the Tempe device from the board level LMvL unless at least one of the L2 packet burst controllers in the TEMPE has set the O_hold_avail_tkn_db signal. Instead, the bus available token will flow through the bypass path inside the Tempe. The O_hold_avail_tkn_db signal is sent to the LMvL Interface module for that channel, as well as the per Tempe LMvL Bypass & Fifo module, to identify when to bring the bus available token onto the internal (non-bypassed) path. If no channels within a Tempe have packets ready to be burst, all the LMvL Interface state machines in that Tempe will remain in the PASS-THRU state.

Data is received from the previous device, registered, and then sent on to the output ports to the next device. When the Bus Available Token is received, the LMvL Controller State Machine sends data and/or forwards the Bus Available Token.

The first are the signals coming from the previous channel (or LMvL FIFO & Bypass module for channel 0) The second are the LMvL signals going to the next channel (or LMvL FIFO & Bypass module for channel 7). The third are the signals going to and from the L2 Packet Burst Controller module, which include framing and data bits, as well as handshaking signals to communicate orders to hold the Bus Available Token and identify when it is ok to burst a packet.

4.9 LMvL FIFO & Bypass

LMvL FIFO and Bypass circuit is a per chip block in the LMvL. Receives LMvL clock, data, and control signals from the previous device in the ring into a FIFO, which makes them synchronous with the local clock domain. Forwards and receives LMvL data and control to and from the per-channel LMvL Interface Blocks. Monitors if the device is ready to move a packet onto the LMvL. If ready to move a packet, the Bus Available Token is captured and sent to the internal LMvL path. If the device is not ready or has no data to send, it provides a Bypass path for the LMvL, and immediately forwards the Bus Available Token to the next device. When bypassed, the device adds less latency to the

LMvL ring. The block is only allowed to switch out of Bypass mode when it takes possession of the Bus Available Token.

4.10 VERIFICATION ENVIRONMENT FOR LMvL

To simplify the task of Verification at functional level, the verification environment is made as generic as possible. Most of the portion is based on MACROS and Registers. The VE comprises of C++ (C Side) and Verilog components (V Side).

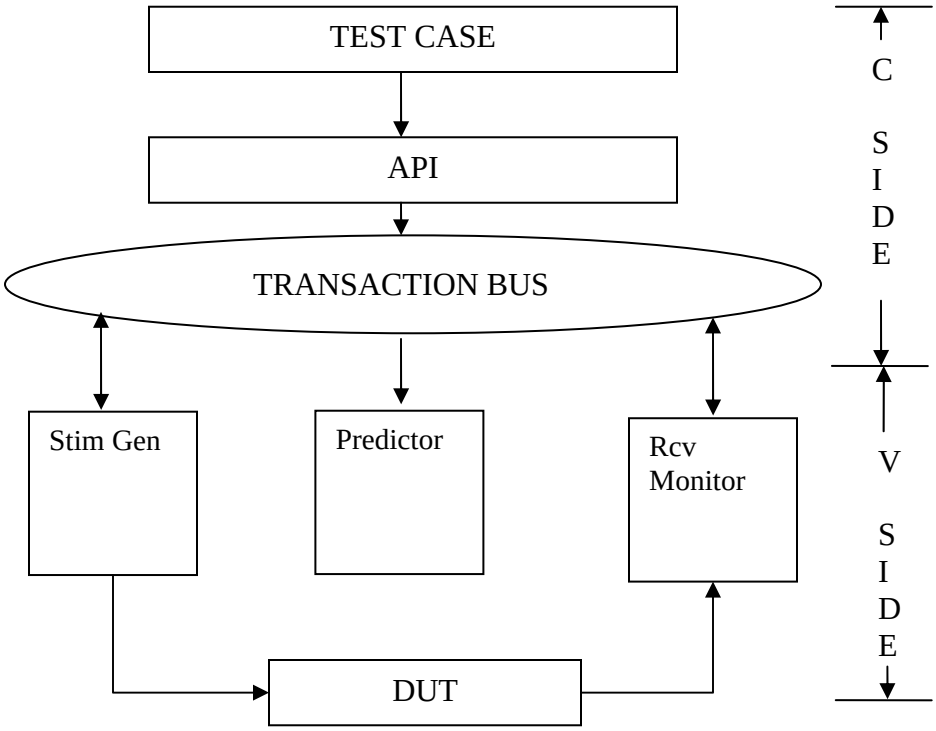


Figure 4.10.1: Drv TG Verification Environment Components

They interact with each other through SystemC. The C++ Components forms the basic platform for the VE. The Verilog components deal with the test bench portion. Generation of clocks, source and sink drivers are part of Verilog components. It has a common test bench file which is used by module blocks, It is complied selectively based on the working module block.. All top level headers are defined in a separate file. It includes many tasks to govern the complete verification process.

4.11 C++ ENVIRONMENT [4]

- Test Case
- API
- X Bus
- Stimulus Generator(C Side only)
- Predictor/Drv Monitor
- Monitor/Rcv Monitor(C Side only).

The C++ components deal with initial generation of Testcases and API that act as interface between Testcases and VE. Testcases uses API routines to pass their configurations to VE.

TestCase

Testcases configure DUT and VE basically. To do that it uses API's *apiTestFlowCtrl.h* to configure DUT and VE parameters to directed values. According to Feature requirement testcase generate specific scenario using this file of API and pass it to other VE components StimGen, Predictor, Monitor. Using this file Status, Control, Configuration related information are also provided. Testcase can also configure VE parameters and DUT parameters to random value within their scope using *tgDriveScenarioGen.h* file. This file define default random constrain of all configuration parameters. Testcase uses *TestCommon.h* file to have other platform file ex. *msg.h, types.h* etc.

API

It acts as interface medium between testcase and rest of the VE. Various API routines are used for VE configuration, control, handshaking, synchronism between VE elements. Under API directory there are files have all VE and DUT configuration parameters declaration and initialization.ex. *tgDrvTestCase.h,* define default constrains for VE and DUT parameters using SystemC ex. *tgDriveScenarioGen.h,* files that governs flow of data through VE ex. *apiTestFlowCtrl.h*. *PatternDrvRcvStart(),* routines to abstract hardware configuration from testcase for DUT and write down bitcharts(registers) ex.

testFlowUpdHwCfg(testcase) **,**routine to abstract VE configuration from testcase and then send information to StimGen, Predictor, Monitor through X Bus ex. *testFlowUpdVerCfg(testcase),*routines to read status of StimGen, DrvMonitor, RcvMonitor using X Bus ex. *PatternDrvRcvBusy().*

API sends/receives information (configuration, control, status) to StimGen, Predictor, Monitor in form of packets. There are three kind of packets with which API deals mainly.

1) Config Packet: Includes configuration parameters for VE components (StimGen, Predictor)
2) Ctrl Packet: Includes parameters that are used to start VE components (StimGen, Predictor, Monitor), to stop VE components (StimGen, Predictor) or to reset VE components (StimGen, Predictor, Monitor).
3) Staus Packet: Includes parameters that are used to check current state of VE components (StimGen, Predictor, Monitor) .

Transaction(X) Bus

Any VE components communicate with testcase using X Bus only. Any client can put a packet onto the X Bus but only those clients can take packet from the X Bus which is registered for that type of packet. VE elements do not need to register to transmit on Transaction Bus. There are total three types of clients on Transaction Bus:

1) Send Client: Using routine *TRR_Send(Datatype*)* routine; where Datatype* is pointer to data structure client transmit on Bus.
2) ReadImmediate Client: Using routine *TRR_ReadImmediate(DataType*);* where Datatype* is pointer to data structure, it is considered that zero time expires when this call is made and that the data structure is returned.
3) Request Client: Using routine *TRR_Request(Datatype*);* where

Datatype* is pointer to data structure, request for particular data structure is made.

For listening particular type of data structure VE element has to declare it self for particular data type using routine *TRR_DECLARE_CLIENT(<routine>, Datatype*);* where Datatype* shows pointer to data structure and routine is receipt routine, and it has to register itself with receipt routine using

TRR_REGISTER_SendClient(<routine>) *or*

TRR_REGISTER_ReadImmediateClient(<routine>) *or*

TRR_REGISTER_RequestClient(<routine>). For Request Client server can give its delayed response using routine *TRR_Response(Datatype*)* ; where Datatype* shows pointer to data structure. File corresponds to X Bus related operations is *trRouterIntf.h* file included into API, StimGen, Predictor, Monitor files.

StimGen

StimGen is used to generate data stream according to the configuration parameters obtained from testcase and drive this data stream Vside of StimGen to DUT and to Predictor as well. For stimulus generation, synchronization, handshaking.. StimGen has files that shows configuration, control and status parameters. Ex. *tgDrvStimCfg.h, tgDrvStimCtrl.h, tgDrvStimStatus.h* files

As StimGen can receives configuration, control, status packets from X Bus so as mentioned earlier it has to register itself for which kind of packet it is interested and also for which of its routine. So StimGen declares itself as

TRR_DECLARE_Client (txCfg, trTgDrvStimCfgT)*

TRR_DECLARE_Client (txCtrl, trTgDrvStimCtrlT)*

TRR_DECLARE_Client(rxStatus, trTgDrvStimStatusT)* in *tgDrvStimTran.h* file ;where txCfg, txCtrl, rxStatus are receipt routines and *trTgDrvStimCfgT, trTgDrvStimCtrlT, trTgDrvStimStatusT* are data types.

StimGen registers itself as

TRR_REGISTER_SendClient (txCfg)

TRR_REGISTER_SendClient (txCtrl)

TRR_REGISTER_ReadImmediateClient(rxStatus)

;where *txCfg, txCtrl, rxStatus* are receipt routines so whichever packets come on to the X Bus, having any of the data types described earlier, are been fed to corresponding StimGen routines only. Based on the configuration StimGen generates data stream, forward it to V Side. It also forward this data stream to predictor by putting packet on to the X Bus using *TRR_Send(&trTgDrvDatT).*

After generation of stimulus for DUT StimGen form a bundle and using library routines ex. *WriteBFM(*bundle, size).* It forward the stimulus data stream from C side to V side. To have definition of *WriteBFM(*bundle, size)* we have to include library file *platform.h* file In later section of V Side of Environment we will discuss how this is mapped with verilog side.

Predictor (Drv Monitor)

Predictor collects stimulus from StimGen through Transaction Bus and predict that what should be the output of DUT. For synchronization, handshaking, configuration Predictor source code is divided in various files that have configuration, control, status parameters ex. *tgDrvPredCfg.h, tgDrvMonCtrl.h, tgDrvMonStatus.h* files.

tgDrvPredCfg.h

```
Class tgDrvPredCfg
{
  // Configuration variables
  int MajorCycCnt,PreCycCnt;
  bool InfRepEn;
  .
   }
```

Predictor declares itself as

TRR_DECLARE_Client (txDrvDat, trTgDrvDatT)*

TRR_DECLARE_Client (txRcvDat, trTgMonDatT)*

TRR_DECLARE_Client (txTgDrvPredCfg, trTgDrvPredCfgT)*

TRR_DECLARE_Client (txTgDrvMonCtrl, trTgDrvMonCtrlT)*

TRR_DECLARE_Client (rxTgDrvMonStatus, trTgDrvMonStatusT)*

;where *txDrvDat, txTgRcvDat, txTgDrvPredCfg, txTgDrvMonCtrl, rxTgDrvMonStatus* are receipt routines and *trTgDrvPredCfgT, trTgDrvMonCtrlT, trTgDrvMonStatusT, trTgDrvDatT , trTgMonDatT* are data types.

Predictor registers itself as

TRR_REGISTER_SendClient (txDrvDat)

TRR_REGISTER_SendClent (txRcvDat)

TRR_REGISTER_SendClient (txTgDrvPredCfg)

TRR_REGISTER_SendClient (txTgDrvMonCtrl)

TRR_REGISTER_ReadImmediateClient(rxTgDRvMonStatus)

;where *txTgDrvPredCfg, txTgDrvMonCtrl, rxTgDrvMonStatus, txDrvDat, txRcvDat* are receipt routines so whichever packets come on to the X Bus, having any of the data types described earlier, are been fed to corresponding Predictor routines only. Based on the data stream, forwarded by StimGen, Predictor generates the expected data. Predictor has only C Side implementation. Note Drv Monitor is also registered for the packet that comes from Monitor as a result of DUT response and compared with expected value of Predictor.

Monitor (Rcv Monitor)

Monitor C side receives data from its V side and then forward the outcomes of DUT on to the X Bus which are received by Drv Monitor where they are compared with expected output of Predictor. Monitor is registered to receive control packet.

As Monitor just act as buffer medium to transfer data from DUT to C side, it is interested to know only no of cycles that it has to fetch from DUT that is exactly same as transmitted by StimGen C side to its V side to DUT. So Monitor declare itself as

TRR_DECLARE_Client (txTgRcvMonCtrl, trTgRcvMonCtrlT)*

TRR_DECLARE_Client (txTgRcvMonStatus, trTgRcvMonStatusT)*

;where *txTgRcvMonCtrl, txTgRcvMonStatus* is receipt routines and *trTgRcvMonCtrlT, rTgRcvMonStatusT* is data type of packet

Monitor registers itself as

TRR_REGISTER_SendClient (txTgRcvMonCtrl)

TRR_REGISTER_SendClient (txTgRcvMonStatus)

;where txTgRcvMonCtrl, txTgRcvMonStatus is receipt routine. So whichever packets come on to the X Bus, having data types described earlier, are been fed to corresponding Monitor C side receives data from its V side and then put onto X Bus that is compared at Drv Monitor. To take DUT response from SC Interface to C side library functions of library file platform.h is used ex. *ReadBFM(*bundle,size)* In later section of Verilog Environment we discuss this platform matching of C side and V side using SC Interface.

4.12 VERILOG ENVIRONMENT

The Verilog components deal with the test bench portion. It includes V side of StimGen, Monitor and DUT section. Along with StimGen, Monitor - Generation of clocks, source and sink drivers are part of Verilog components. All top level headers and V side environment are instantiated in a separate file. There is also a separate file to instantiated C side of the environment. It includes many tasks to govern the complete verification process. The following sections explain in detail, the overall
contribution of all these files. The module specific test bench is in its folder of module. This file is having the instantiation of the module and the other required supported verilog modules.

As discussed earlier that in StimGen using various library routines ex. *WriteBFM(*bundle, size)* stimulus packets are sent to V side to DUT & to have definition of *WriteBFM(*bundle, size)* we have to include library file *platform.h* and *zipSC.h* file that maps C side function *WriteBFM(*bundle, size)* to SC Interface function *SIM_Write()* and *cside_Interface.h* library file maps SC Interface function *SIM_Write()* function to

verilog task *Write_BFM. Write_BFM* task includes library function *TBP_getdata(status, data)* take data from SC Interface which are forwarded to DUT.

Similar to StimGen Monitor receives response of DUT through binding of ports into testbench and forward the response to SC Interface through library functions ex. *Read_BFM* task at Verilog side of Monitor which has library routine *TBP_Putdata()* to send data to SC Interface. Which is further forwarded to C side using SC Interface library routines ex. *SIM_Read()* maps with C side function *ReadBFM(*bundle, size)* to form packet for comparison with expected values. Again *platform.h, zipSC.h, cside_Interface.vh* files are included as for platform mapping.

The testbench is constructed using a modular approach. There are three main components to the testbench as indicated below. These sections are merely broken up for clarity, maintainability and portability across the various aspects of VE.

The V Side Testbench Section

This section includes .cmd file ex. *config_tg.cmd* file that defines transactors of RTL, other top level modules wrappers ex. _wrapper.v, V sides of all VE component (StimGen, Drv Monitor, Rcv Monitor, Clock) by including corresponding .cmd files, RTL Netlist.

The C Side TestBench Section

This section includes .txt file ex. *tg_config_table.txt* that defines C side of all VE components (StimGen, Predictor, Monitor) and also defines their C to V side SC interface.

Chapter 5

Local Move Link Subsystem Verification

This Chapter discusses the complete verification process for the Local Move Link LMvL Interface Block. As already discussed earlier, this is the first phase where module level verification is done. The complete LMvL Interface block is divided into functional blocks capable of being verified individually and also completely. The design is to be verified against all the functionality specified in its hardware specification and which states the features of the design. The chapter discusses the complete verification process as the flow followed.

5.1 Identification of Features

This is the first step to the verification. Before deciding how to verify, it is necessary what is to be verified. For that module/block Functional Interface Specification is studied very preciously without any assumptions. This Functional Interface Specification provided by Designer is able to explain all aspects of functionality to be implemented by each module. The addresses and configuration of all registers is explained here. Hence, all behaviors specified into Functional Interface Specifications can be considered as Expect Model for the verification Engineers.

As said earlier as Designers and Verification Engineers interpret the specifications independently and hence the verification will be for the specification and not for the interpretation. Now, as the functionality desired by the module/block is understood, the next step is to identify the features of the module/block. Identified features are reviewed by Designer as some tine it may possible that some of the functionality doesn’t need any verification or they can not be verified. Identification of features will help in identifying the number of test cases required to be written. The features are grouped into categories and test cases are identified for each group. LMvL Interface verification features can be well summarized by dividing LMvL Interface into Four branches.

5.2 Developing the Testbench

The Testbench connects C Side and V side into one for verification. The Testbench is applicable to all test cases of LMvL Interface. The LMvL Drv module is instantiated in the Testbench. Other required sub modules like Paragon are also instantiated in the Testbench. This Test Bench Works in conjunction with the generic Testbench that is applicable to all the Functionalities of module.

The generic Testbench is based on Macros and complier directives. Hence, during compilation only the portion of the Testbench that is applicable to individual module functionality is complied. The Testbench is updated as and when required based on test case requirement and updation in the RTL. One important thing is that it is possible that for verifying particular functionality, register is not present in the RTL. In such cases, software accessible registers are made in the reserved space of the addresses. These are then written in the Testbench as per the change in the signal and can then be accessed for verification.

5.3 Prioritization of Features

This is an important task as the aim of completing the verification at proper deadline is also equally important. Below are few points which are taken care of while verification.

- The first important task is to verify that the RTL and VE are stable and ensure that both of them works fine in co-ordination with each other. This will ensure that there is no major issue in the VE or Testbench or RTL, because of which the test case might fail. Normally, this takes some time. This test is called harness test. It can be thought of a top level test.

- Now, some harness test cases are written to verify that the RTL is stable for the major functionality. In this test cases, interest is to verify only the that RTL gets configure according to the top level configuration. No internal generation of any signal is looked upon.

- Now, starts the verification of the inner functionality of the module. The level of granularity is quite low. But here the functionality checked is sorted out as per the importance. The first aim is that all functionality is to be verified. Then, depending on time and requirement, the granularity is reduced.

- Other important point affecting the test cases is the fixing of bugs. Due to delay in fixing of some bugs, it is possible to start with test cases which are of some lesser priority.

5.4 Grouping of the Test Cases

As per the complexity of the feature and its implementation in the module, the number of test cases can vary. Hence, as already shown earlier, few test cases group together to and fall into some common category which shares common tasks and require same type of debugging. The grouping of test cases is done with many considerations. Functionality falling into same category requires similar type of verification approach. e.g. Any particular block related functionality is responsible for generating many different combinations of configuration based on occurrence of events. Verification of such interrupts generation requires generation of that type of event. All of them require similar type of top level configuration and hence should be grouped into common category.

5.5 Environment Settings

While starting to use the environment for the first time, it is required to make a working folder where all the environment components, various modules' Test Bench and Test Cases will reside as per specific path. It is required to Check Out the environment from the Clear Tool.

The second requirement is to set the module onto which we are working. This will indicate the module to be verified. For LMvL Interface, we just have to enter into directory of module. This will use all the LMvL specific code from the Verification

Environment. It is possible to have multiple such directories as the project progresses. All test cases in specific directory will execute in the environment it sees within its project directory. The only requirement for this is to set the variable project accordingly. It is just required to enter into project directory name at the command line and the directory required to be worked with. Code Build and Code Compile will be done within the directory only.

5.6 Transactor

The test cases use the API calls to command the Transactor to do a transaction. The Transactors transforms a command from the test cases into stimuli of signal values and these stimuli are sent to the DUT. For a given stimulus the response from the DUT is processed in the Transactor and the status or the data is send back to the test case.

5.7 Developing and Debugging Test cases

As verification progresses test cases are developed for the features lying into common category. For LMvL Interface block test case strategy for features mentioned into previous sections are as under.

Procedural Algorithm to make sure LMvL Interface block functional correctness when enabling channel in bypass (not holding token) and enabling other channels in Holding Token. The token holding is depending on the pack modes. [Appendix A]

- To make sure that correct functionality of LMvL Interface block by enabling and disabling the holding token.
- Set the pack mode for enable and disable the selected channels for data transfer.
- Set the channel to a holding token state with the help of LMvL Interface State Machine. So the bypass path will disconnect and connecting the L2 packet burst controller to the LMvL outputs.

- The State Machine will stay in this mode until the L2 packet burst allows to release the token. When the L2 packet burst controller has no data to send the hold signal will be low. And token is released.

- Now the token is passed to the next channel and if that channel is not enable than it should pass the token by releasing it to the next channel.

- The same procedure will continue for all pack mode configuration.

PASS/FAIL CRITERIA

If the disable channel grab the token than the data will not passed and it will release the token as soon as it gets (i.e. bypass).

For bug analysis debug the test case with the help of Simvision or try to read the simlog file. The bug from the Simvision can be seen from below two Figure 5.7.1 and Figure 5.7.2.

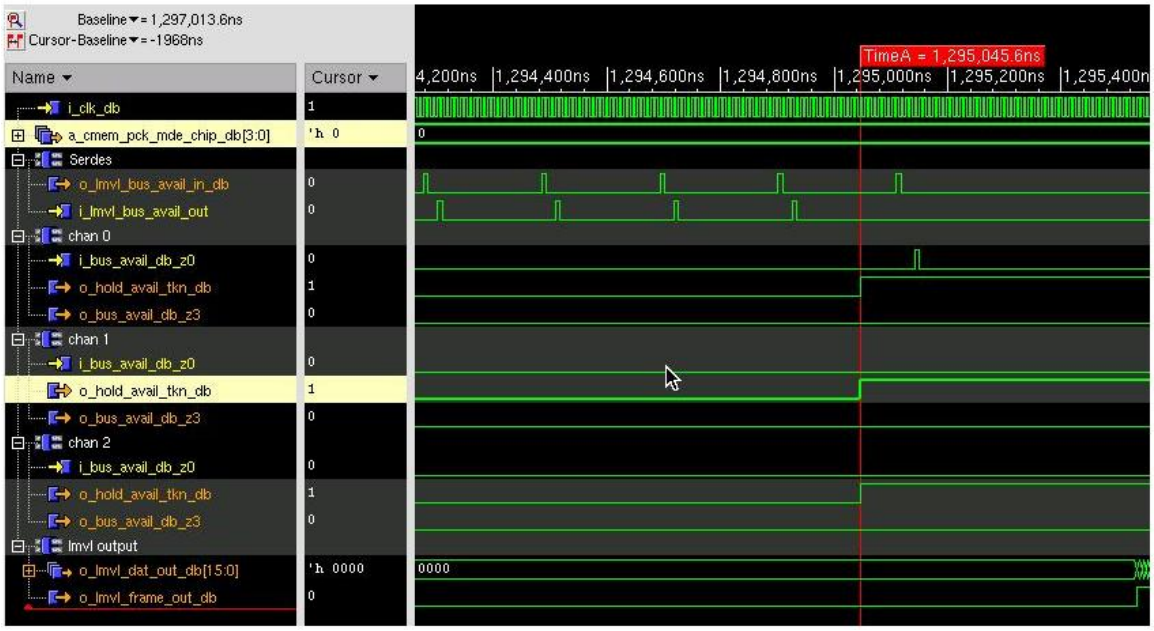


Figure 5.7.1: Waveform for Bug Identification

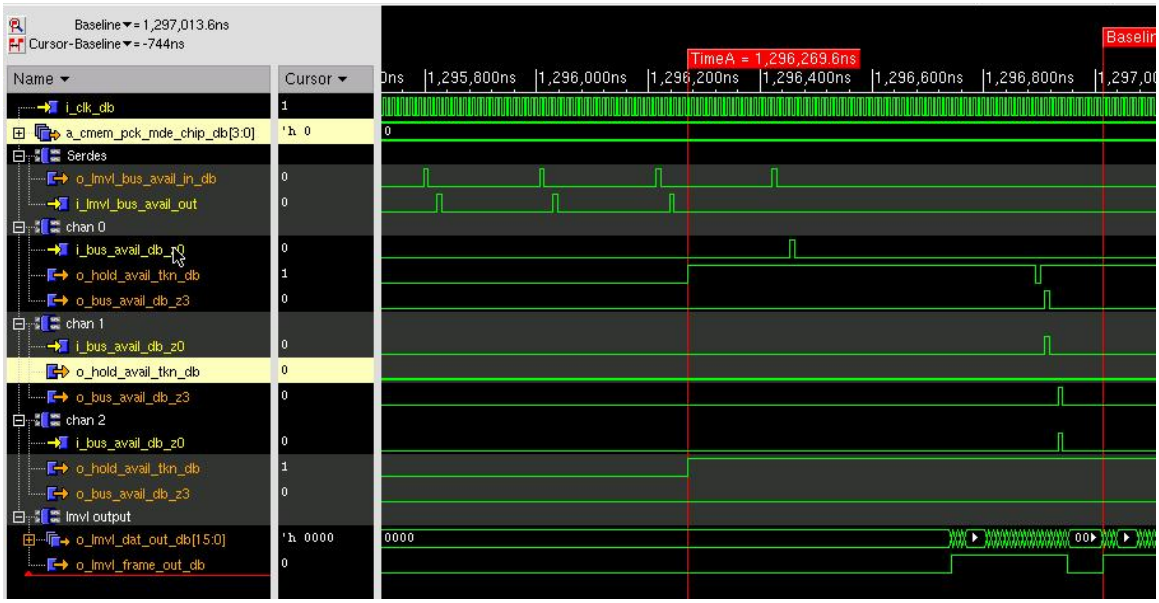


Figure 5.7.2: Waveform after Bug Resolved

Branch: pause
Feature:

The i/p LMvL pause Serdes input signal is always routed on the o/p LMvL pause Serdes output signal.

NOTE: The dagger i/o latency of the pause signal is no more than 15 LMvL clk cycles

Measure the Pause latency time by counting no. of LMVL clk cycles between negative edge of i/p pause signal and negative edge of o/p pause signal. Measure the period of BUS Available signal. It should be asserted high for 1 DB clk cycle.

Procedural Algorithm:

- Configure the LMVL SERDES x'actor to send the token to LMVL RTL. Measure the Pause latency time by counting no. of LMVL clk cycles between negative edge of i/p pause signal and negative edge of o/p pause signal.

- Send a burst to the LMVL SERDES x'actor where it drives the i/p LMvL pause signal for few i/p LMvL clk cycles. At the same time the i/p LMvL frame & i/p LMvL bus available signals will be driven low.

- The LMVL SERDES x'actor returns the control back to testcase ASA the i/p LMvL pause signal will be driven low.

- Now at each edge of o/p LMvL clk, read back the o/p LMvL pause signal value at o/p of LMVL SERDES RTL and count the o/p LMvL clk edges till o/p LMvL pause signal value goes low.

- Derive the no. of lmvl cycles from the count and compare it with expected lmvl cycle values.
  => The derived latency count should be 15 lmvl clk cycles.

Measure the period of BUS Available signal. It should be asserted high for 1 DB clk cycle.

- As soon as LMVL SERDES x'actor finished the transmition of above burst, it comes to default state where it asserts the i/p LMvL bus available to high for one DB clk cycle.

- Since Dagger is not ready to send the data(it will be in pass-through mode), i/p LMvL bus available serdes signal is routed to o/p LMvL bus available serdes o/p signal.

- Now at each edge of o/p LMvL clk, read back the value of o/p LMvL bus available signal. As soon as o/p LMvL bus available goes high, start counting the o/p LMvL clk edges for which i/p LMvL bus available signal remains high.

- Compare the derived period, for which o/p LMvL bus available signal remains high, with expected BUS Avail signal period.
  => The o/p LMvL bus available signal should be high for 1 DB clk cycle.

PASS/FAIL CRITERIA:

Test case will report error based on:

1. Comparison of derived pause latency cycles with expected lmvl Cycles (15 LMvL cycles).
2. Comparison of derived o/p LMvL bus available period with expected DB Cycles (1 DB cycle.)

As the verification progresses, issues arises. When any issue arises, first thing is to find the source of issue. This is done by debugging using output generated by simulation. In my project, as the simulation progresses, log files are created. They specify the complete flow of simulation.

The debugging is done as under.

1. Check the log file to check that whether simulation is completed or is left midway. Log file shows different messages ex. Error, Panic, Milestone, Debug etc. This will show the point of problem in terms of at processing of which stream, does the problem and from where does it arises, whether from testcase or from VE. The RTL can be viewed for this to some extend.

2. If the log file is not having any issue, then the next step is Waveform viewer, Simvision is used to see the transactions in the signals. Unwanted signals can be filtered out and important signals can be zoomed out to identify the problem.

The final conclusion to be made is that whether the problem is because of VE, TestBench, Testcase or RTL. Once it is confirmed that the problem is because of RTL, bug is reported.

5.8 Bugs Reporting

As the verification progresses, issues arises. This is a good sign of progress. If no issue arises, that means that something is wrong in the Testbench or VE. So, when any issue arises, first thing is to find the source of issue. This is done by debugging using output generated by simulation. Once it is confirmed that the source of issue is somewhere in the RTL and not in the VE, Testbench or Test case, a bug is filed for it. Brief description is given so that any one can understand the issue. The test case and the logs generated are sent to the designer who is assigned the bug.

5.9 Looping Structure

As the bug get resolved, designer checks in the updated RTL. This RTL is to be checked out and the test case is run on it. The results are again analyzed if the test case fails. The bug id remains same but its status changes. Designer again locates problem in the RTL and this process continues in a loop till finally the test cases passes. It is many times

required to update the VE or Testbench to make the test fail pass. Again, designer can add new registers or some extra hardware as a part of DFV for the purpose to aid the verification. Once Testcase behavior comes out as per expectations and all open issues are resolved test case is reviewed by designer or other verification experts to make sure test case able to reach to targeted corner of RTL, only after completion of review test case status is considered as closed.

5.11 Regression

As and when bugs get fixed and the RTL gets updated, it is required to do regression for all previously passing test cases with the new updated RTL. It is to verify that in the task of fixing one bug, some other bug has not aroused. Hence, a list file is prepared and all test cases undergo regression. At the end of the verification process, once again all test cases are regressed to get the final picture and mark that the process is completed.

Regression uses scripts for executing all the test cases. The script contains all the required commands for the compilation and simulation of the test cases. Hence, at the end of regression, it provides results for all the test cases. Regression lasts for many hours depending on number of test cases.

5.11 Coverage

Once all the issues are fixed and all identified test cases passes, coverage tool is used to get the coverage of the code in terms of blocks, expressions and other metrics for all the test cases. Finally, accumulated result for all the test cases gives the picture of the code which is not covered by existing test cases. Hence, this output serves as an input to the identification of new test cases.

Incisive from cadence is used as coverage tool. Coverage for the RTL will be the accumulation of coverage due to all these test cases. RTL consists of many modules. Hence, the coverage tool will show the coverage in graphical manner. For Block

Coverage, tool extracts modules from RTL and shows the result for them. For expression Coverage, tool extracts expressions from RTL modules and shows the result for them.

5.12 Documentation

The project ends with documenting all the work done. This will serve as reference for future and also it is to be submitted to the client to show the final status of the verification process in terms of various metrics of coverage. Major documents prepared are as under:

1. Documenting the Verification Environment: In this document, complete Verification Environment developed is explained with respect to each block. The VE is generic and can be reused again with updated design.

2. Test Plan: The Test Plan is prepared at the start of the project which serves as specification for the verification task. Test Plan is updated as and when new features are added to the design and the new test cases are identified because of this and feedback from coverage.

3. Bug Report: This document specifies all the bugs filed during the complete verification process with brief description of the issue, date of filing, test cases affected and its final status.

Chapter 6

Functional Coverage Analysis

6.1 Functional Coverage[12]

Functional coverage in HDL designs is a relatively new concept. Functional coverage focuses on functional aspects of a design and provides a functional view of verification completeness. Functional Coverage helps you in identifying uncovered functions of the design. Functional coverage offers a very good insight on how the verification goals set by a test plan are being met. Functional coverage is performed on user-specified functional coverage points. These coverage points can be specified either in PSL or SystemVerilog. The user-defined functional coverage points specify scenarios, error cases, corner cases, and protocols to be covered as well as specify analysis to be done on values of a variable. Functional coverage ensures that the functionality of the design is tested thoroughly.

Functional coverage can be further classified as:

Control-oriented functional coverage: Control-oriented functional coverage is userdefined and easier to interpret. It is an extension of assertion-based verification and this type of coverage requires more user input/work but identifies interesting functions directly. In Incisive Comprehensive Coverage, control-oriented functional coverage points are specified using PSL (Property Specification Language) assert and cover statements or SVA (SystemVerilog Assertions) assert and cover directive. The coverage to be measured is either directly specified using the PSL/SVA statements or is interpreted from them.

Data-oriented functional coverage: Data-oriented functional coverage offers simple coverage metrics, which are relatively easy to measure and interpret. They include coverage of variable values, binning, specification of sampling, and cross products. It can also be used for user-specified state and transition coverage, although FSM coverage is better addressed via automatic extraction of FSMs and automatic state and transition coverage on extracted FSMs. Data-oriented functional coverage helps

in identifying untested data values/subranges. In Incisive Comprehensive Coverage, data-oriented functional coverage is specified using SystemVerilog constructs.

6.2 Incisive Comprehensive Coverage Report Tool (ICCR)

The Incisive Comprehensive Coverage reporting component ICCR is used to generate reports and analyze the coverage data. You can generate and print reports in batch mode with the ICCR while you can generate and analyze the reports in the GUI mode through the ICCR GUI. The reporting tool uses the Functional Coverage data accumulated during simulation to generate textual and graphical reports. This reported data can be used to evaluate the quality of the test stimuli and to detect areas that require more testing.

Generating reports is the final step in the Functional Coverage analysis process. You can generate reports after loading the functional coverage data generated during one or more simulation runs. In report generation and analysis, the count denotes the number of times coverage point is covered during a simulation run. The report generation tool ICCR can generate reports in the various formats.

ICCR is mainly used to analyze the functional coverage analysis. The LMvL packet is taking the data of failed vector array coming from DigCap which is Stored at DDR. The data array or packet is analyzed here with the help of ICCR. The Data Oriented Functional coverage summary report of that packet is shown on GUI below.

As shown in the Figure 6.2.1 the Data Oriented functional coverage result for the packet is about 98 %.

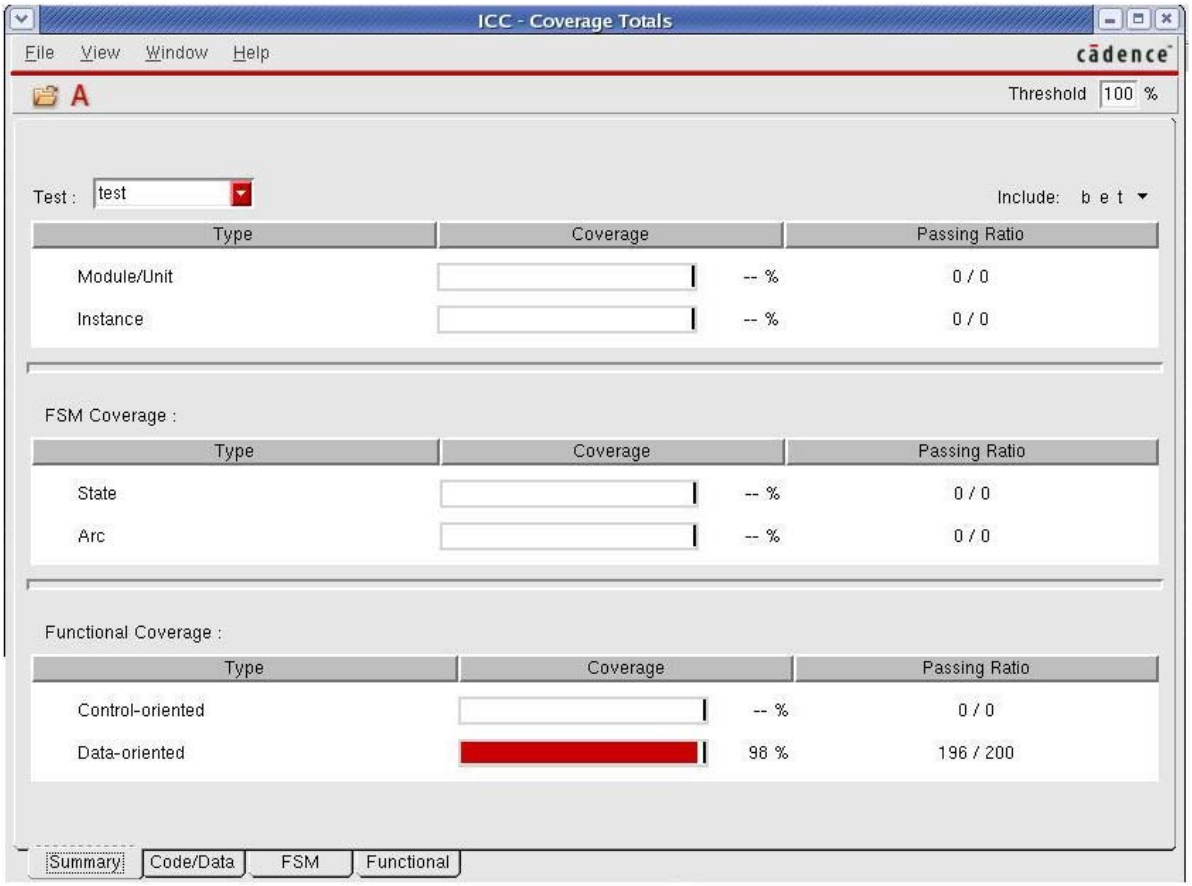


Figure 6.2.1: ICC coverage Totals[12].

The detailed covered bins and covergroup report on GUI is shown in the Figure 6.2.2. where the red dot indicates the missed bin and the source cover code is given at the bottom of the results. The red color text corresponds to the yellow highlighted text in the GUI.

The covermodule of the packet, detailed report of the coverage is shown in the APPENDIX B.

The TCL commands for generating the coverage results in ICCR are shown below.

load_test test_name

report_summary -module module_name

report_detail -module -both module_name

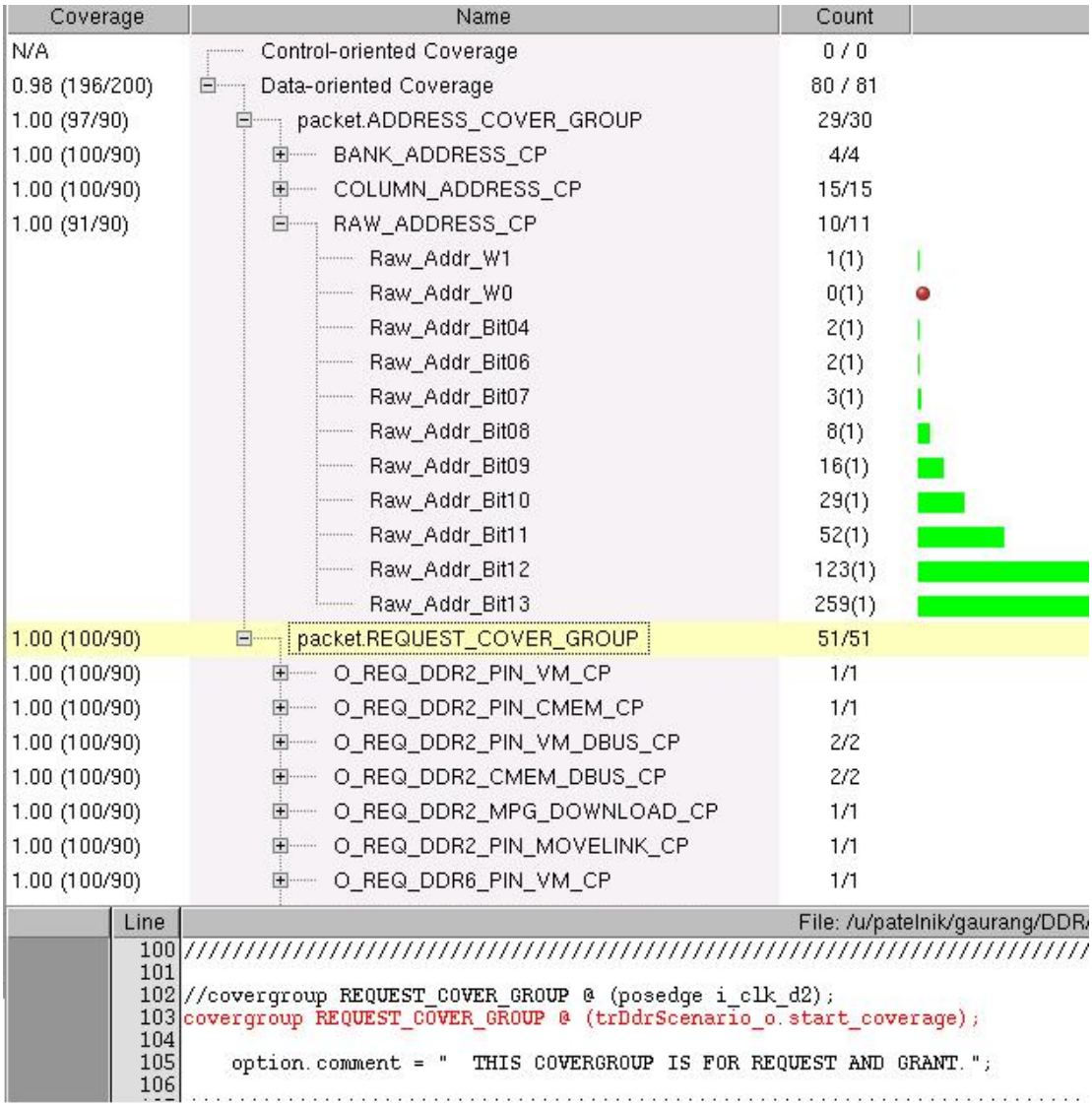


Figure 6.2.2: ICC Functional Coverage Results on GUI.

6.3 vManager – ICC Integration[13]

A verification environment generates huge amounts of data on a daily basis, in the form of logs, and waveform and coverage databases. The typical user of the verification environment creates additional data in the form of source files, makefiles, and scripts. Organizing and abstracting all of this data into useful, relevant information for each member of the project team is one of the significant contributions Enterprise Manager makes to the verification process.

Incisive Manager is a verification process automation tool that facilitates everyday tasks of verification. Incisive Manager makes it easier for the verification manager to track project status by:

- Enabling metric-based tracking of project status from verification plan to closure
- Generating charts of coverage and failure data
- Generating HTML-based summary reports

Incisive Manager also facilitates the tasks of the verification engineer by enabling the verification project team to:

- Find and fix bugs at a faster rate
- Identify and fill holes in verification at a faster rate
- Manage multiple sessions with large numbers of parallel test runs

Incisive Comprehensive Coverage (ICC) is the coverage tool delivered as part of the Incisive platform and offers coverage features to perform code coverage, control coverage (PSL/SVA coverage), data coverage (SystemVerilog covergroups), FSM coverage, and toggle coverage. The ICC - Incisive Manager integrated solution helps you perform verification process automation using Incisive Manager to manage the verification runs. You can perform ICC runs and failure analysis through Incisive Manager. You can automatically generate coverage model and vPlan from an ICC run. The integrated solution also allows analysis and reporting of ICC coverage data from Incisive Manager.

Below are the necessary steps for the VM-ICC Integration.

Step 1

set path = (<INSTALL_DIR>/tools/bin $path)

Where the <INSTALL_DIR> is the directory of the source of the incisive compiler.

Step 2

setenv LD_LIBRARY_PATH
<INSTALL_DIR>/tools/lib:${LD_LIBRARY_PATH}

where the LD_LIBRARY_PATH is the load library path. Ensure that vManager & Specman are installed.

Step 3

setenv SPECMAN_PATH <INSTALL_DIR>/tools/vmgr

where the SPECMAN_PATH is the path of the installes vManager.

Step 4

setenv WORKAREA `pwd`

Set the WORKAREA to your current directory.

Step 5

“load <Dir>/cdn_icc_top.e”

From vManager Console load the "cdn_icc_top.e" file to enable interface of ICC and VM.

Step 6

“load setup”

From vManager Console write the below mentioned command

OR

After enabling the integration from Incisive Enterprise Manager click "setup".

Step 7

From Incisive Enterprise Manager go to file menu and click on "Start Session"

to load your file vsif file "my_session.vsif".

Step 8

From Incisive Enterprise Manager click on "refresh" to get get the pass or fail status.

By performing above steps the vManager shows the status of the processes about run. The window shows the pass, fail, run and wait status in the GUI as shown in the Figure 6.3.1 below. There is another window of the vManager for the prompt command execution as shown in the Figure 6.3.2.

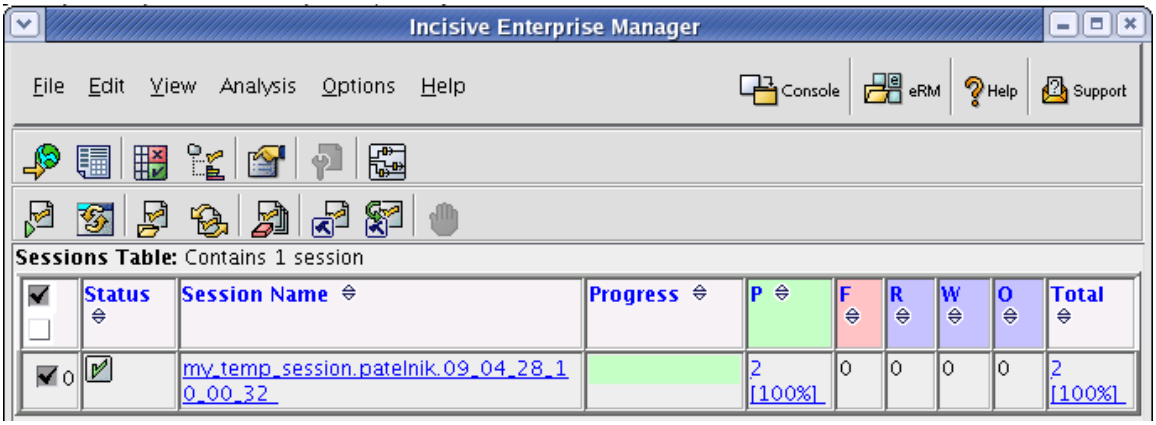


Figure 6.3.1: vManager Output File Status[13]

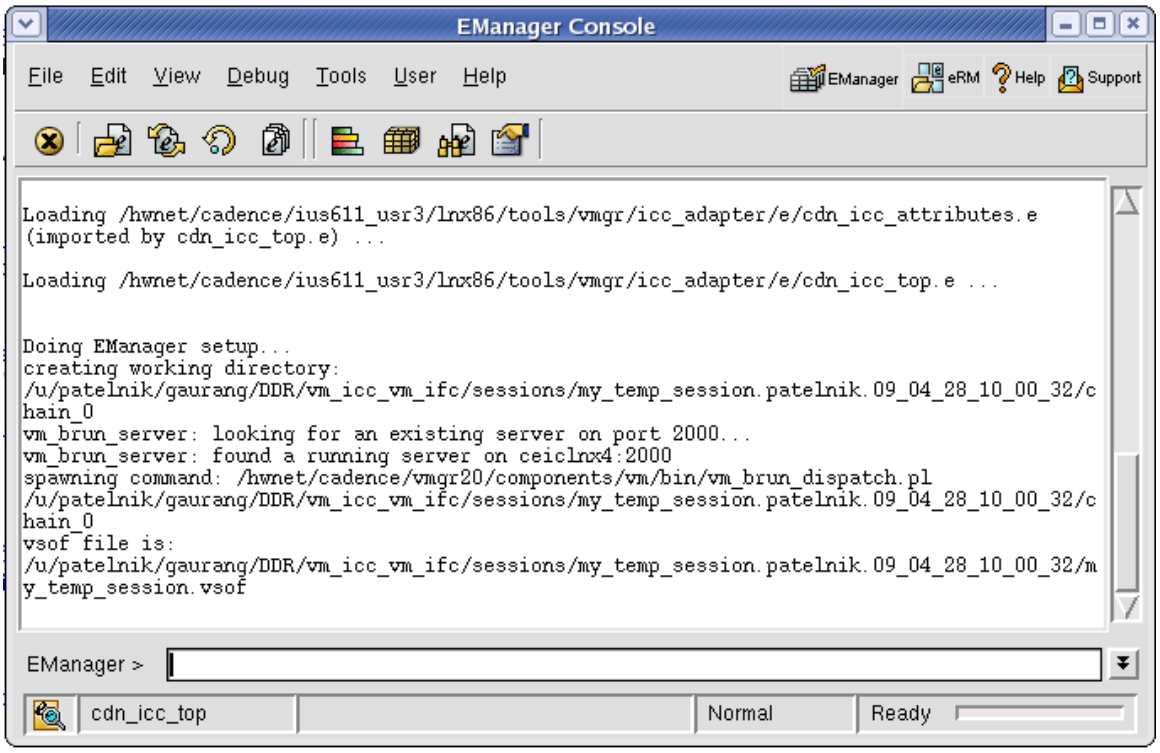


Figure 6.3.2: vManager Console[13]

6.4 Analyzing the Coverage

Step 1

Write the "reload" command in vManager Console to reload the icc_cdn_top.e file for enabling the integration.

OR

From Incisive Enterprise Manager click on "reload"

Step 2

Load the coverage model file generated for the DUT during simulation run by executing the following command at the console in the EManager Console window.
“load ./vm_auto/mic_cov_model.e;”

Where vm_auto is the automatic generated coverage directory which contains the cover model file, vsof file and verification plan file.

OR

Load the file automatic generated Cover Model file from vm_auto dir in your top WORKAREA.

Step 3

Do the "setup" in Incisive Enterprise Manager window.

Step 4

Load the automatic generated “vsof” file by selecting "Read Session" from the File menu in the Incisive Enterprise Manager window.

Step 5

Click on "vplan" on Incisive Enterprise Manager window.

Step 6

To load the automatic generated “vPlan”, select Read vPlan from the File menu in the Verification Plan Tree window.

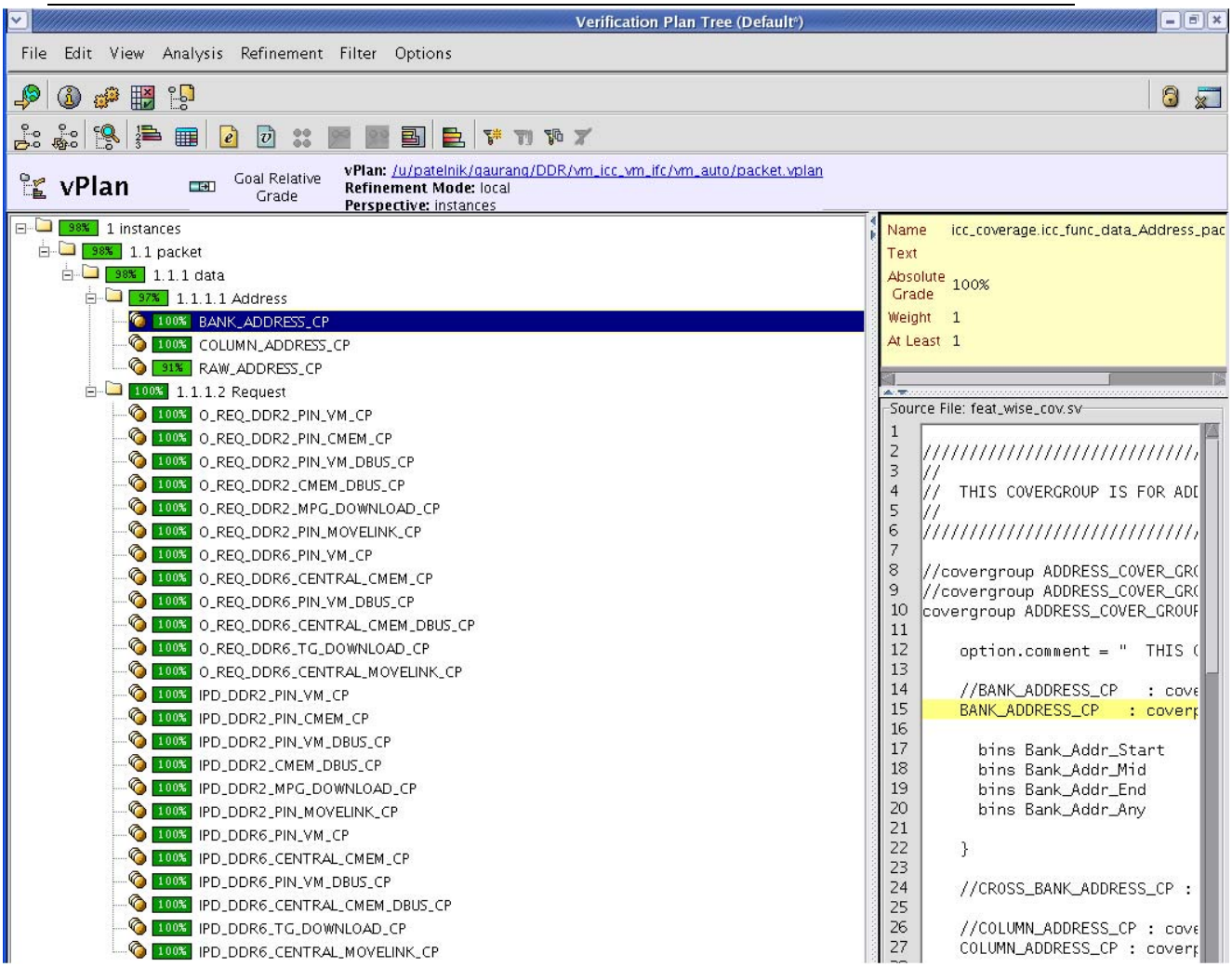


Figure 6.4.1: VM Coverage Analysis Results in VPLAN window.

Chapter 7

Languages, S/W & EDA Tools

7.1 Platform for Verification

The Platform for Verification is provided by languages. These languages are for coding designs, writing scripts to command processes, etc. The languages used are explained here in detailed.

7.1.1 C/C++ [7]

This is the most popular language used since long time. Its flexibility is responsible for its wide use. Most languages have evolved from C/C++ and inherit its features. C++ was the result of dynamic growth of software technology. It gave rise to object oriented approach. Because of this strong feature, C/C++ is always present somewhere in any project. The idea behind VE is to make it generic and applicable to all phases of verification. It is to be made such that it can be updated later to make reusable with such other chip. Hence, C++ was selected as the base for VE.

7.1.2 SystemC & Verilog

SystemC is Hardware Description Language, whose rapidly growing use is because of its compatibility with C/C++. It is also popular for randomization so. SystemC is used to support SCV Randomization for coverage requirements as well as to fill the gap between C++ and Verilog for the project. Verilog is the most popular Hardware Description Language used for Chip Design. Its growth has being like a revolution. Verilog supports designing at many levels of Abstraction importantly Behavioral, RTL and Gate level. It supports any level of hierarchy in the design. It is the most preferred language for verification. Popular language System Verilog has developed from fusion of features of Verilog and C.

Basic features of Verilog are as under:

1. Verilog HDL is a general purpose HDL that is easy to learn and easy to use. It is similar in syntax to the C language.
2 Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL or

behavioral code.

3. Verilog HDL is supported by most popular logic synthesis.
4. All fabrication vendors provide Verilog HDL libraries for postlogic synthesis Simulation. Thus, designing a chip in Verilog allows the widest choice of vendors.

7.1.3 Perl

PERL stands for Practical Extraction and Report Language. Perl is a dynamic programming language created by Larry Wall and first released in 1987. Perl borrows features from a variety of other languages including C, shell scripting (sh), AWK, sed and Lisp. The overall structure of Perl derives broadly from C. Perl is procedural in nature, with variables, expressions, assignment statements, brace-delimited code blocks, control structures, and subroutines.

Perl also takes features from shell programming. It is used as a CGI (Common Gateway Interface) language, a programming language for Unix, linux or for Windows. Perl has many and varied applications, compounded by the availability of many standard and third-party modules. Perl is often used as a glue language, tying together systems and interfaces that were not specifically designed to interoperate, and for "data managing", converting or processing large amounts of data for tasks like creating reports. In fact, these strengths are intimately linked. The combination makes perl a popular all-purpose tool for system administrators, particularly as short programs can be entered and run on a single command line.

7.1.4 Makefile

“make” is a utility for automatically building large applications. Files specifying instructions for make are called Makefiles. make is most commonly used in C/C++ projects, but in principle it can be used with almost any compiled language.

The basic tool for building an application from source code is the compiler. make is a separate, higher-level utility which tells the compiler which source code files to process. It tracks which ones have changed since the last time the project was built and invokes the compiler on only the components that depend on those files. Although in principle

one could always just write a simple shell script to recompile everything at every build, in large projects this would consume a prohibitive amount of time. Thus, a makefile can be seen as a kind of advanced shell script which tracks dependencies instead of following a fixed sequence of steps.

Today, programmers increasingly rely on Integrated Development Environments and language-specific compiler features to manage the build process for them instead of manually specifying dependencies in makefiles. However, make remains widely used, especially in Unix-based platforms. The makefile just contains a list of file dependencies and commands needed to satisfy them.

7.2 OS: RED HAT LINUX

Linux was originally created by Linus Torvalds during his graduate studies at the University of Helsinki in Finland. Linus wrote Linux as a small PC-based implementation of UNIX. During the summer of 1991 Linus made Linux public on the Internet. In September of that same year, version 0.01 was released. A month later, version 0.02 was released, with version 0.03 following several weeks later. In December, Linux was numbered at 0.10, and by the end of the month, virtual memory (disk paging) was added. Within a year, Linux had a thousand more features and was well on its way to becoming a self-compiling, usable operating system.

Linus made the source code freely available and encouraged other programmers to develop it further. They did, and Linux continues to be developed today by a worldwide team, led by Linus, over the Internet. It supports a wide range of Softwares and Hardwares. Again, it is a secure system and hence is preferred the most when networking is involved in the project. e.g offshore projects. . The Operating System installed in the work station is i386 -RedHat Linux v3.0. All the EDA Tools and the applications to be used through the project are supported by this OS. The secure networking to be used for working at the work station at US, is thoroughly supported by this OS. All the work stations are installed with this OS.

7.3 Supporting Applications

For any project, along with the main languages and software, some supporting software applications are required to manage the project. Some commonly used for verification are discussed under.

7.3.1 S/W: ClearQuest[6]

ClearQuest a database for bugs. It lets people report bugs and assigns these bugs to the appropriate developers. Developers can use ClearQuest to keep a to-do list as well as to prioritize, schedule and track dependencies. In verification, all the issues and bugs are filed in ClearQuest for the developer. Each bug can be issued a priority based on urgency for its resolve. Each bug can have interred dependencies. Each bug is assigned a unique id. Any authorized person can login into the ClearQuest and view and update or add some comments the status for any bug. Each bug is composed of many fields. Few of them are mentioned below:

Bug ID, Reported By, Date Found, Root Cause

This shows unique id of Bug, Registered Reporter Name, Date and time relevant information, Root cause shows reason of bug in terms of VE, RTL coding.

Status, Investigator, Date Last Modified

This field is used to show status of a bug(open, resolved), Investigator who will work on fixing of bug, last modification date.

Severity, Project, Bug Type, Testcase

This field targets to show severity of bug ex. urgent, Platform and Project belongs to bug, Testcase name where bug is to be noticed.

Description Log, Resolution Log, Verification Log

Log of description shows expectation of bug reporter and misbehavior noticed by bug reporter, Log of Resolution shows summary of resolution actions taken by bug investigator. After fixing bug by investigator, summary of verification done by bug reporter.

Dependency

If a bug can't be fixed until another bug is fixed, that's a dependency. For any bug, you can list the bugs it depends on and bugs that depend on it. ClearQuest can display a dependency graphs which shows which the bugs it depends on and are dependent on it.

Attachment

Adding an attachment to a bug can be very useful. Test cases, screen shots and editor log can help pinpoint the bug and help the developer reproduce it. If you fix a bug, attach the patch to the bug. This is the preferred way to keep track of patches since it makes it easier for others to find and test.

7.3.2 ClearCase[6]

The ClearCase implements SCM (Software Configuration Management). Its main features or we can say advantages are 1) Version Control: version all types of files and directories. 2) Build Management: ensure integrity of all software elements, accurately reproduce every release. 3) Workspace Management: work in parallel with other developers. 4) Process Control: record and report actions, history and milestones. Its version control system keeps track of all work and all changes in a set of files, typically the implementation of a software project, and allows several (potentially widely separated) developers to collaborate. All the source files are managed in a common platform called as a VOB (Versioned Object Base) *[Ref-5]*.

VOB plays duties like server for one or more projects. ClearCase uses client-server architecture: a server stores the current version(s) of the project and its history, and clients connect to the server in order to check-out a complete copy of the project, work on this copy and then later check-in their changes. Typically, client and server connect over a LAN or over the Internet, but client and server may both run on the same machine if ClearCase has the task of keeping track of the version history of a project with only local developers. The VOB normally runs on Windows XP, Windows 2000, Unix including Red Hat Linux.

Several developers may work on the same project concurrently, each one editing files within his own working copy of the project, and sending (or checking in) his modifications to the server. To avoid the possibility of people stepping on each other's toes, the VOB will only accept changes made to the most recent version of a file. Developers are therefore expected to keep their working copy up-to-date by incorporating other people's changes on a regular basis. If the check-in operation succeeds, then the version numbers of all files involved automatically increment, and the VOB writes a user-supplied description line, the date and the author's name to its log files. ClearCase can also run external, user-specified log processing scripts following each commit. These scripts are installed by an entry in ClearCase's log info file, which can trigger email notification, or convert the log data into a web-based format. Clients can also compare different versions of files, request a complete history of changes, or check-out a historical snapshot of the project as of a given date or as of a revision number. Clients can also use the "gvp uwa –v -merge" command in order to bring their local copies up-to-date with the newest version on the server. This eliminates the need for repeated downloading of the whole project. ClearCase can also maintain different "branches" of a project. For instance, a released version of the software project may form one branch, used for bug fixes, while a version under current development, with major changes and new features, forms a separate branch.

Few terms related to ClearCase are discussed below:

Main Branch: basically means the code and all its various versions in the repository. Main Branch consists of the main trunk which contains all the main code files, and sometimes may be branch for each separate releasable product.

Release Branch: Branch designated for holding official releases of a product. It facilitates moving and tracking changes among releases. Reduce complexity and length of version tree.

Developer Branch: Branch designated for individual development, facilitates to track individual activity.

Working copy: Your local copy of the source code. You don't work on the server code directly, instead you work on the working copy and changes are merged together.

Few useful Commands guideline related to ClearCase are discussed below:

prune

Mark your private files as removed. This means that you are updated with latest repository, your local checkin files are no more available.

merge

Keep you in cope with latest release available at repository, but checkout files are kept unchanged.

mkelem:

Insert a new file into VOB. Will be visible in the repository only after a checkin and a gather command is issued.

checkout

Checks out the source files defined by modules. Note that multiple checkouts can be made in different directories, this can be very confusing.

checkin

Checks in the source files defined by modules to your local workspace, will not be visible in the repository Note that multiple checkins can be made in different directories, this can be very confusing.

gather

Commit your changes into the the repository.

diff

Check difference with your private files against the repository.

remove

Remove the file from ClearCase revision control. This command moves the repository file permanently from VOB so no way to recover it.

7.4 EDA TOOLS

EDA Tools plays very important role in this industry. For every task to be accomplished, these EDA Tools tend to reduce the time required and performs the task accurately. Developments of EDA Tools are at the target of every vendor as its demand is increasing tremendously. They have revolutionized every process in the development of any design.
In the following topics, I will be discussing the Major EDA Tools use by me for the project. All design tools are by the company Cadence.

7.4.1 Compilation & Simulation[11]

Tool: Affirma NCsim v 06.11-s002

Company: Cadence

For the C++ compilation, freeware g++ compiler from GCC is used. This is free software. It is available with Linux Operating System. The NC-Verilog Simulator delivers high-performance, high-capacity Verilog simulation with transaction/signal viewing and integrated coverage analysis. It is fully compatible with the Incisive functional verification platform, so design teams can easily upgrade to the Incisive Unified Simulator and Incisive XLD team Verification, with native support for Verilog, VHDL, SystemC Verification Library, PSL/Sugar assertions, and Acceleration-on-Demand. Verilog IEEE 1364-1995 and a majority of IEEE 1364-2001 extensions, SystemVerilog (IEEE-1800). It compiles directly to host processor machine code for maximum performance.

The NC-Verilog Simulator provides the industry’s premier simulation performance for Verilog designs using the unique native-compiled architecture of the Incisive Unified Simulator. It produces efficient native machine code directly from Verilog for high-speed execution. Linked list scheduling of the resulting data structures pre-processes signal actions and maximizes the effectiveness of modern caching algorithms available in

today's computing platforms. The NC-Verilog performance profiler identifies bottlenecks. Designers find areas of high activity by viewing how each module contributes to overall performance. Minor changes can greatly improve simulation performance by identifying the areas that consume the most simulation time. NC-Verilog 64-bit capacity simulates designs larger than 100 million gates.

The unified NC-Verilog simulation and debug environment makes it easy to manage multiple simulation runs and analyze the design and testbench. Its transaction/waveform viewer and schematic tracer quickly trace design behavior back to the source. The NC-Verilog source viewer lets designers examine their design, set complex breakpoints to control simulation execution, and access results in both interactive and post processing debug modes. The NC-Verilog Simulator provides access to a wide variety of coverage metrics to help determine how well tests have exercised the design. These include block coverage, path coverage, expression coverage, state variable coverage, state transition coverage, state sequence coverage, and toggle coverage. Integrated coverage analysis and display tools speed the process of determining which additional tests will need to be developed.

7.4.2 Waveform Viewer[11]

Tool: Simvision v 06.11-s003

Company: Cadence

The SimVision analysis environment is a unified graphical debugging environment for Verilog-XL, NC-Verilog, NC-VHDL, and NC-Sim. You can run SimVision in either of the following modes:

Simulation mode

In simulation mode, you view "live" simulation data. That is, you analyze the data while the simulation is running. You can control the simulation by setting breakpoints and stepping through the design.

SimVision provides several tools to help you track the progress of the simulation:

- Source code window
- Navigator window

- Watch window
- Signal Flow Browser
- Cycle window
- Schematic window
- Waveform window

Many of these windows are linked, so that when you select an object in one window, it is selected in the other windows as well.

Post-Processing Environment (PPE) Mode

In PPE mode, you analyze simulation data after simulation has completed. You have access to all of the SimVision tools, except for the simulator. As in simulation mode, all of these windows are linked, so that when you select an object in one window, it is selected in the other windows as well.

7.4.3 Code Coverage[11]

Tool: Incisive v 06.11-s002

Company: Cadence

Code Coverage is an important metric for Verification Engineers to measure their effort. It gives the view of the scenarios created by the test cases to verify the RTL. The remaining can then be again generated by the Verification engineers.

Incisive by Cadence is a powerful Code Coverage Tool that gives coverage in many metrics as desired by verification engineer. It can provide the graphical view of scenarios created. Desired metrics can be selected as required. It has the capability of extracting FSM from RTL design. It supports code, path, expression, fsm (state, transition and sequence), toggle, variables and gate coverage. Such different metrics of Coverage acts as a feedback loop to improve the input stimulus.

APPENDIX A

TESTCASE

Testcase for LMvL Interface Holding Token State.

Testcase Name: cmem_lmvl_siggen_basic.cc

/*!

TEST ID: cmem_lmvl_siggen_basic

STATUS: CLOSED

REVIEWED: TESTPLAN

SCHEDULE: 07/10/07

ASSIGNED TO: parmarg

REVIEWED BY: panej

OPEN ISSUES:

TEST STRATEGY: The strategy is to release the token as soon as a channel has no data to send to the LMvL output

REFERENCE DOCS: LMvL_fis

TEST ASSUMPTIONS:

VERIFICATION AIDS: LmvlSerdesTran

RANDOMIZATIONS: BlockSize [16'h0000 – 16'hffff]

PROCEDURE:

- Set up Pattern Generator
- Enable loopback path from drv to rcv tg.
- Initialise cmem.
- Create the Data Set by Randomization .
- LMvL configuration Setting
- Select Pack Mode for LMvL
- Set the Capture Count Limit
- Call API to invoke the desired scenario and Transactor

- Start Test to Run

PASS/FAIL CRITERIA: The LmvlSerdesTran will report failures on channel which is in Holding_Token State. Even there is no data is available Channel Holding the Token instead of release.

*/

```
#include "dartTestCommon.h"
#include "apiLmvl.h"
#include "apiCmnSetup.h"
#include "apiCmem.h"
#include "apiLmvlSerDes.h"

// NOTE : Run test with command line option "-define LMVL_SERDES_XACTOR"
// NOTE : To enable a_tempe_lmvl_wait_for_dat, run test with command line option "-
//SetWaitForDat"
//
//Main
extern "C" int cmem_lmvl_siggen_basic(void) {
  Tai::SetArg("wavesOn","on");
  TT_READ( A_CM_BUSY );

  apiSteering.selectAllPatgen();                    // Setup
  apiCmnSetup.setLpbk();                            // Enable loopback path from drv
                                                    // to rcv tg.
  TT_WRITE(A_TEMPE_CMEM_INIT,
M_TEMPE_CMEM_INIT_DART_DB_ONLY);        // issue init_dart_cmem to db

   // create the data set for db ddr write
  //
  // create random write data array
  u_int32 blockSize = 1024;
```

```
u_int32 wrdata[blockSize];
for(u_int32 i=0; i<blockSize; i++){
  wrdata[i] = Random.getInRange(0, 0xffff);
}

apiLmvlT::RawDdrDatT rawData;
rawData.datCnt = blockSize;
rawData.datPtr = wrdata;

//
// create lmvl/cmem config
//
apiLmvlT::ConfigT config;
config.mssnMode = apiSteeringT::CM_HSM_MSSN_MDE_DTL_SE;
config.packMode = B_TG_CMEM_DATA_PCK_SITE_0;
config.mvMode = apiLmvlT::MV_HIM;
config.startAddr = Random.getInRange(0,(0x3fff-blockSize)) & 0xfff8;
config.captSize = blockSize / 16 ; // (Blocks of 16 bit data) / 16 = segments available.
config.lastWdSize = 0;
config.reset = M_TEMPE_CMEM_INIT_ALL;
config.wrapEn = false;
config.lmvlConfig.lvl3_pay_size = 0x10;
config.lmvlConfig.move_en = 0xFF;
config.lmvlConfig.siggen_bus_en = 0x3FFFF;


if(Tai::IsPresent("SetWaitForDat")){
                        config.lmvlConfig.wait_for_dat = 0x1;
        }

//
```

```
 // start test
 //
 TT_WRITE(A_TEMPE_LMVL_SIG_CAPT_CNT_LIMIT, 0xFFFF);
 apiLmvl.testScenario( config,
                 apiLmvlT::WR_DDR,
                 rawData,
                 true,          // lmvl transactor monitoring data
                 true );        // lmvl client check siggen

 TT_READ( A_CM_BUSY );
 TT_READ( A_CM_BUSY );
 TT_READ( A_CM_BUSY );
 TT_READ( A_CM_BUSY );
 TT_READ( A_CM_BUSY );
 Tai::SetArg("wavesOn","off");

 return 0;
}
```

APPENDIX B

Functional Cover Module

```
////////////////////////////////////////////////////////////////////////////////
//
//  THIS COVER GROUP IS FOR ADDRESS FIELD OF THE PACKET.
//
////////////////////////////////////////////////////////////////////////////////

covergroup ADDRESS_COVER_GROUP @ (trDdrScenario_o.start_coverage);

   option.comment = "  THIS COVERGROUP IS FOR ROW, COLUMN AND BANK
ADDRESS.";

   BANK_ADDRESS_CP   : coverpoint trDdrScenario_o.bankAddress {

     bins Bank_Addr_Start          = { 0};
     bins Bank_Addr_Mid            = { 4};
     bins Bank_Addr_End            = { 7};
     bins Bank_Addr_Any            = { [0:7]};

   }

   CROSS_BANK_ADDRESS_CP : cross
BANK_ADDRESS_CP,trDdrScenario_o.clntType;

   COLUMN_ADDRESS_CP : coverpoint trDdrScenario_o.columnAddress {

     bins Column_Addr_Eight        = { 11'b00000001000};
     //SPECIAL PETTERN BINS
     bins Column_Addr_W1           = { 11'b00000000001, 11'b00000000010,
                                       11'b00000000100, 11'b00000001000,
                                       11'b00000010000, 11'b00000100000,
                                       11'b00001000000, 11'b00010000000,
                                       11'b00100000000, 11'b01000000000,
                                       11'b10000000000};

     bins Column_Addr_W0           = { 11'b11111111110, 11'b11111111101,
                                      11'b11111111011, 11'b11111110111,
                                       11'b11111101111, 11'b11111011111,
                                       11'b11101111111, 11'b11101111111,
                                       11'b11011111111, 11'b10111111111,
                                       11'b01111111111};
```

```
    //SPECIAL PATTERN ADDRESS BIN
    bins Column_Addr_Invert          = { 11'b10101010101,11'b01010101010};
    wildcard bins Column_Mid_bit     = { 11'b11111?00000,11'b00000?11111};

    //ADDRESS VALUE BIN
    bins Column_Addr_Bit04           = { [8:15]};
    bins Column_Addr_Bit05           = { [16:31]};
    bins Column_Addr_Bit06           = { [32:63]};
    bins Column_Addr_Bit07           = { [64:127]};
    bins Column_Addr_Bit08           = { [128:255]};
    bins Column_Addr_Bit09           = { [256:511]};
    bins Column_Addr_Bit10           = { [512:1023]};
    bins Column_Addr_Bit11           = { [1023:2047]};
    bins Column_Addr_Any             = { [0:2047]};
    bins Column_Addr_End             = { 2047};
    illegal_bins Column_Addr_Ignore  = { [0:7]}; //LAST 3 BITS ARE HARDCODED
                                       //0 SO IGNORE THE VALUES FROM 0 TO 7
  }

  CROSS_COLUMN_ADDRESS_CP : cross COLUMN_ADDRESS_CP,
trDdrScenario_o.clntType;

  RAW_ADDRESS_CP    : coverpoint trDdrScenario_o.rowAddress {

    bins Raw_Addr_Start          = { 16'b0000000000000000};
    // SPECIAL ADDRESS PETTERN BIN
    bins Raw_Addr_W1             = { 16'b0000000000000001, 16'b0000000000000010,
                                     16'b0000000000000100, 16'b0000000000001000,
                                     16'b0000000000010000, 16'b0000000000100000,
                                     16'b0000000001000000, 16'b0000000010000000,
                                     16'b0000000100000000, 16'b0000001000000000,
                                     16'b0000010000000000, 16'b0000100000000000,
                                     16'b0001000000000000, 16'b0010000000000000,
                                     16'b0100000000000000, 16'b1000000000000000};
    // SPECIAL ADDRESS PETTERN BIN
    bins Raw_Addr_W0             = { 16'b1111111111111110, 16'b1111111111111101,
                                     16'b1111111111111011, 16'b1111111111110111,
                                     16'b1111111111101111, 16'b1111111111011111,
                                     16'b1111111110111111, 16'b1111111101111111,
                                     16'b1111111011111111, 16'b1111110111111111,
                                     16'b1111101111111111, 16'b1111011111111111,
                                     16'b1110111111111111, 16'b1101111111111111,
                                     16'b1011111111111111, 16'b0111111111111111};
    bins Raw_Addr_Invert         = { 16'b1010101010101010, 16'b0101010101010101,
                                     16'b1111111100000000, 16'b0000000011111111};
```

```
    // ADDRESS VALUE BIN
    //bins Raw_Addr_Bit03          = { [0:8]};
    bins Raw_Addr_Bit04          = { [0:15]};
    bins Raw_Addr_Bit05          = { [16:31]};
    bins Raw_Addr_Bit06          = { [32:63]};
    bins Raw_Addr_Bit07          = { [64:127]};
    bins Raw_Addr_Bit08          = { [128:255]};
    bins Raw_Addr_Bit09          = { [256:511]};
    bins Raw_Addr_Bit10          = { [512:1023]};
    bins Raw_Addr_Bit11          = { [1023:2047]};
    bins Raw_Addr_Bit12          = { [2048:4095]};
    bins Raw_Addr_Bit13          = { [4096:8191]};
    bins Raw_Addr_Bit14          = { [8192:16383]};
    bins Raw_Addr_Bit15          = { [16384:32767]};
    bins Raw_Addr_Bit16          = { [32768:65535]};

  }
  CROSS_RAW_ADDRESS_CP : cross RAW_ADDRESS_CP,
trDdrScenario_o.clntType;

endgroup
ADDRESS_COVER_GROUP Address = new(); //TAKING THE INSTANCE OF
ADDRESS_COVER_GROUP

////////////////////////////////////////////////////////////////////////////////
//
//   THIS COVERGROUP IS FOR HANDSHAKING AND CONTROL SIGNALS.
//
////////////////////////////////////////////////////////////////////////////////

covergroup REQUEST_COVER_GROUP @ (trDdrScenario_o.start_coverage);

  option.comment = "  THIS COVERGROUP IS FOR REQUEST AND GRANT.";

////////////////////////////////////////////////////////////////////////////////
//
// REQUEST COVER POINTS FOR DDR2 GROUP ALL CLIENTS(read =0 ,wr = 1)
//
////////////////////////////////////////////////////////////////////////////////

  O_REQ_DDR2_PIN_VM_CP          : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_PIN_VM){
    bins O_REQ_DDR2_PIN_VM_RD                    = { 0};
    bins O_REQ_DDR2_PIN_VM_RD_TRAN               = ( 0=>0);
    ignore_bins O_REQ_DDR2_PIN_VM_WR_IGN         = { 1};
  }
```

```
   O_REQ_DDR2_PIN_CMEM_CP          : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_PIN_CMEM){
     bins O_REQ_DDR2_PIN_CMEM_WR                    = { 1};
     bins O_REQ_DDR2_PIN_CMEM_WR_TRAN               = ( 1=>1);
     ignore_bins O_REQ_DDR2_PIN_CMEM_RD_IGN         = { 0};
   }

   O_REQ_DDR2_PIN_VM_DBUS_CP       : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_PIN_VM_DBUS){
     bins O_REQ_DDR2_PIN_VM_DBUS_RD                 = { 0};
     bins O_REQ_DDR2_PIN_VM_DBUS_WR                 = { 1};
     bins O_REQ_DDR2_PIN_VM_DBUS_RD_TRAN            = ( 0=>0);
   }

   O_REQ_DDR2_CMEM_DBUS_CP         : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_CMEM_DBUS){
     bins O_REQ_DDR2_CMEM_DBUS_RD                   = { 0};
     bins O_REQ_DDR2_CMEM_DBUS_WR                   = { 1};
     bins O_REQ_DDR2_CMEM_DBUS_RD_TRAN              = ( 0=>0);
   }

   O_REQ_DDR2_MPG_DOWNLOAD_CP      : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_MPG_DOWNLOAD){
     bins O_REQ_DDR2_MPG_DOWNLOAD_RD                = { 0};
     bins O_REQ_DDR2_MPG_DOWNLOAD_RD_TRAN           = ( 0=>0);
     ignore_bins O_REQ_DDR2_MPG_DOWNLOAD_WR_IGN     = { 1};
   }

   O_REQ_DDR2_PIN_MOVELINK_CP      : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR2_PIN_MOVELINK){
     bins O_REQ_DDR2_PIN_MOVELINK_RD                = { 0};
     bins O_REQ_DDR2_PIN_MOVELINK_RD_TRAN           = ( 0=>0);
     ignore_bins O_REQ_DDR2_PIN_MOVELINK_WR_IGN     = { 1};
   }

////////////////////////////////////////////////////////////////////////////
//
//   REQUEST COVER POINTS FOR DDR6 GROUP FOR ALL CLIENTS
//
////////////////////////////////////////////////////////////////////////////

   O_REQ_DDR6_PIN_VM_CP            : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR6_PIN_VM){
     bins O_REQ_DDR6_PIN_VM_RD                      = { 0};
     bins O_REQ_DDR6_PIN_VM_RD_TRAN                 = ( 0=>0);
```

```
    ignore_bins O_REQ_DDR6_PIN_VM_WR_IGN          = { 1};
   }

   O_REQ_DDR6_CENTRAL_CMEM_CP      : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR6_CENTRAL_CMEM){
    bins O_REQ_DDR6_CENTRAL_CMEM_WR                  = { 1};
    bins O_REQ_DDR6_CENTRAL_CMEM_WR_TRAN             = ( 0=>0);
    ignore_bins O_REQ_DDR6_CENTRAL_CMEM_RD_IGN       = { 0};
   }

   O_REQ_DDR6_PIN_VM_DBUS_CP       : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR6_PIN_VM_DBUS){
    bins O_REQ_DDR6_PIN_VM_DBUS_RD                   = { 0};
    bins O_REQ_DDR6_PIN_VM_DBUS_WR                   = { 1};
    bins O_REQ_DDR6_PIN_VM_DBUS_RD_TRAN              = ( 0=>0);
   }

   O_REQ_DDR6_CENTRAL_CMEM_DBUS_CP: coverpoint
trDdrScenario_o.RdWrReq iff(trDdrScenario_o.clntType ==
DDR6_CENTRAL_CMEM_DBUS){
    bins O_REQ_DDR6_CENTRAL_CMEM_DBUS_RD             = { 0};
    bins O_REQ_DDR6_CENTRAL_CMEM_DBUS_WR             = { 1};
    bins O_REQ_DDR6_CENTRAL_CMEM_DBUS_RD_TRAN        = ( 0=>0);
   }

   O_REQ_DDR6_TG_DOWNLOAD_CP       : coverpoint trDdrScenario_o.RdWrReq
iff(trDdrScenario_o.clntType == DDR6_TG_DOWNLOAD){
    bins O_REQ_DDR6_TG_DOWNLOAD_RD                   = { 0};
    bins O_REQ_DDR6_TG_DOWNLOAD_RD_TRAN              = ( 0=>0);
    ignore_bins O_REQ_DDR6_TG_DOWNLOAD_WR_IGN        = { 1};
   }

   O_REQ_DDR6_CENTRAL_MOVELINK_CP   : coverpoint
trDdrScenario_o.RdWrReq iff(trDdrScenario_o.clntType ==
DDR6_CENTRAL_MOVELINK){
    bins O_REQ_DDR6_CENTRAL_MOVELINK_RD              = { 0};
    bins O_REQ_DDR6_CENTRAL_MOVELINK_RD_TRAN         = (0=>0);
    ignore_bins O_REQ_DDR6_CENTRAL_MOVELINK_WR_IGN  = { 1};
   }

////////////////////////////////////////////////////////////////////////////////
//
//   IPD COVER POINTS FOR DDR2 GROUP
//
////////////////////////////////////////////////////////////////////////////////
```

```
   IPD_CP : coverpoint IPD {
   IPD_DDR2_PIN_VM_CP          : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_PIN_VM){
     bins IPD_DDR2_PIN_VM_ZERO                    = { 0};
     bins IPD_DDR2_PIN_VM_ONE                     = { 1};
     bins IPD_DDR2_PIN_VM_FOUR                    = { 2,3,4};
     bins IPD_DDR2_PIN_VM_FIVE                    = { 5};
     ignore_bins IPD_DDR2_PIN_VM_IGN              = { [51:$]};
     //IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }

   IPD_DDR2_PIN_CMEM_CP        : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_PIN_CMEM){
     bins IPD_DDR2_PIN_CMEM_ZERO                  = { 0};
     bins IPD_DDR2_PIN_CMEM_ONE                   = { 4};
     bins IPD_DDR2_PIN_CMEM_FOUR                  = { 1,2,3,5};
     ignore_bins IPD_DDR2_PIN_CMEM_IGN            = { [51:$]}; //IGNORE THE
INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR2_PIN_VM_DBUS_CP     : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_PIN_VM_DBUS){
     bins IPD_DDR2_PIN_VM_DBUS_ZERO               = { 0};
     bins IPD_DDR2_PIN_VM_DBUS_ONE                = { 1};
     bins IPD_DDR2_PIN_VM_DBUS_FOUR               = { 2,3,4,5};
     ignore_bins IPD_DDR2_PIN_VM_DBUS_IGN         = { [51:$]}; //IGNORE THE
INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR2_CMEM_DBUS_CP       : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_CMEM_DBUS){
     bins IPD_DDR2_CMEM_DBUS_ZERO                 = { 0};
     bins IPD_DDR2_CMEM_DBUS_ONE                  = { 1,3};
     bins IPD_DDR2_CMEM_DBUS_FOUR                 = { 2,5,4};
     ignore_bins IPD_DDR2_CMEM_DBUS_IGN           = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR2_MPG_DOWNLOAD_CP    : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_MPG_DOWNLOAD){
     bins IPD_DDR2_MPG_DOWNLOAD_ZERO              = { 0};
     bins IPD_DDR2_MPG_DOWNLOAD_ONE               = { 1};
     bins IPD_DDR2_MPG_DOWNLOAD_FOUR              = { 2,3,4,5};
     ignore_bins IPD_DDR2_MPG_DOWNLOAD_IGN        = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR2_PIN_MOVELINK_CP    : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR2_PIN_MOVELINK){
     bins IPD_DDR2_PIN_MOVELINK_ZERO              = { 0};
```

```
     bins IPD_DDR2_PIN_MOVELINK_ONE                = { 1,4};
     bins IPD_DDR2_PIN_MOVELINK_FIVE               = { 2,3,5};
     ignore_bins IPD_DDR2_PIN_MOVELINK_IGN         = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50) gaurang
   }

////////////////////////////////////////////////////////////////////////////////
//
// IPD COVER POINTS FOR DDR6 GROUP
//
////////////////////////////////////////////////////////////////////////////////

   IPD_DDR6_PIN_VM_CP          : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_PIN_VM){
     bins IPD_DDR6_PIN_VM_ZERO                     = { 0};
     bins IPD_DDR6_PIN_VM_ONE                      = { 1};
     bins IPD_DDR6_PIN_VM_FOUR                     = { 2,3,4,5};
     ignore_bins IPD_DDR6_PIN_VM_IGN               = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }

   IPD_DDR6_CENTRAL_CMEM_CP     : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_CENTRAL_CMEM){
     bins IPD_DDR6_CENTRAL_CMEM_ZERO               = { 0};
     bins IPD_DDR6_CENTRAL_CMEM_ONE                = { 1};
     bins IPD_DDR6_CENTRAL_CMEM_FIVE               = { 2,3,4,5};
     ignore_bins IPD_DDR6_CENTRAL_CMEM_IGN         = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR6_PIN_VM_DBUS_CP      : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_PIN_VM_DBUS){
     bins IPD_DDR6_PIN_VM_DBUS_ZERO                = { 0};
     bins IPD_DDR6_PIN_VM_DBUS_TWO                 = { 1,2};
     bins IPD_DDR6_PIN_VM_DBUS_FIVE                = { 3,4,5};
     ignore_bins IPD_DDR6_PIN_VM_DBUS_IGN          = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR6_CENTRAL_CMEM_DBUS_CP : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_CENTRAL_CMEM_DBUS){
     bins IPD_DDR6_CENTRAL_CMEM_DBUS_ZERO          = { 0};
     bins IPD_DDR6_CENTRAL_CMEM_DBUS_TWO           = { 1,2};
     bins IPD_DDR6_CENTRAL_CMEM_DBUS_FIVE          = { 3,4,5};
     ignore_bins IPD_DDR6_CENTRAL_CMEM_DBUS_IGN    = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
```

```
   IPD_DDR6_TG_DOWNLOAD_CP       : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_TG_DOWNLOAD){
     bins IPD_DDR6_TG_DOWNLOAD_ZERO              = { 0};
     bins IPD_DDR6_TG_DOWNLOAD_ONE               = { 1};
     ignore_bins IPD_DDR6_TG_DOWNLOAD_IGN        = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }
   IPD_DDR6_CENTRAL_MOVELINK_CP  : coverpoint trDdrScenario_o.IPD
iff(trDdrScenario_o.clntType == DDR6_CENTRAL_MOVELINK){
     bins IPD_DDR6_CENTRAL_MOVELINK_ZERO              = { 0};
     bins IPD_DDR6_CENTRAL_MOVELINK_ONE               = { 1};
     ignore_bins IPD_DDR6_CENTRAL_MOVELINK_IGN        = { [51:$]};
//IGNORE THE INVALID VALUE OF IPD (AFTER 50)
   }

endgroup

REQUEST_COVER_GROUP Request = new();  //TAKING THE INSTANCE OF
                                      //REQUEST_COVER_GROUP
```

ACRONYMS

ABBREVIATION	FULL NAME
GVP	Gather Validate Publish
DUT	Device Under Test
ENV	Environment
BFM	Bus Functional Model
MSB	Most Significant Bit
LSB	Least Significant Bit
PISO	Parallel in Serial Out
SIPO	Serial in Parallel Out
VC	Verification Component
API	Application Program Interface
SCIF	System C Interface
TSTI	T client Standard Transactor Interface
MPG	Memory Pattern Generator
CMEM	Capture memory
DigCap	Digital Capture Instrument. .
DSSC	Digital Signal Source/Capture
HRAM	History Memory.
LFVM	Large Fail Vector Memory.
LMvL	Local Move Link.
MTC	Memory Test Capture
SRM	Sub Routine Memory (a.k.a. SVM)
SR	Synchronous Reject.
STV	Store This Vector.
SUT	Signal Under Test
VM	Vector Memory
ALU	Algorithmic Logical Unit
ISL	Instrument Synchronous Link
Patgen	Pattern Generator
FSS	Full System Specification
FIS	Functional Interface Specification
FVM	Fail Vector Memory
SR	Synchronous Reject
CR	Cumulative Reject
ADS	Alternate Data Source
ATE	Automatic Test Equipment

REFERENCES

[1] EInfochips, Local Move Link Functional Interface Specifications Manual.

[2] EInfochips, Capture Memory Functional Interface Specifications Manual.

[3] EInfochips, Capture Memory Verification Environment Reference Manual.

[4] EInfochips, SCIF methodology Functional Interface Specifications Manual.

[5] EInfochips, Memory Pattern Generator Verification Environment Reference Manual.

[6] IBM, SCM271 Essentials of ClearCase v7.0.

[7] Object Oriented Programming with C++, By E balagurusamy .second Edition.

[8] WRITING TESTBENCHES Functional Verification of HDL Models By Janick Bergeron.

[9] Andrew Pizali, P. D. (2004), Functional Verification Coverage Measurement and Analysis, Kluwer Academic Publishers (Boston).

[10] Andreas Meyer, P. D.(2003), Principles of Functional Verification, Elsevier Science(USA).

[11] Incisive v 06.11-s002 User Guide, Cadence

[12] ICCR v 06.11-s002 User Guide, Cadence

[13] VManager v 2.0 User Guide, Cadence