Verification & Debugging Challenges In The Scalable Low-Power IP Subsystems

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering

(VLSI Design)

By

Naishal Shah

(15MECV25)

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

MAY 2017

Verification & Debugging Challenges in the Scalable Low-Power IP Subsystems

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering

(VLSI Design)

By

Naishal Shah
(15MECV25)

Under The Guidance of

Prof. Vaishali Dhare

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
MAY 2017

Declaration

This is to certify that

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

II. Due acknowledgement has been made in the text to all other material used.

Naishal Shah
15MECV25

Certificate

This is to certify that the Major Project report entitled **"Verification & Debugging Challenges In The Scalable Low-Power IP Subsystem" s**ubmitted by **Naishal Shah** (15MECV25), towards the partial fulfillment of the requirements for the degree of **Master of Technology** in **VLSI Design** (Electronics and Communication Engineering Department) of Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Vaishali Dhare
Internal Guide
VLSI Design

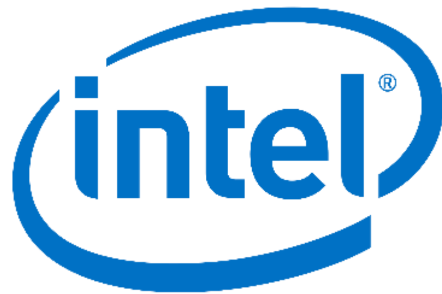
Dr. N M Devashrayee
P.G Coordinator
VLSI Design

Dr. Dilip Kothari
Head, EC Department

Dr. Alka Mahajan
Director, IT-NU

Date :

Place: Ahmedabad



Certificate

This is to certify that the Project report entitled "**Verification & Debugging Challenges in the Scalable Low-Power IP Subsystem**" submitted by **Naishal Shah** (15MECV25), towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design of Nirma University, Ahmedabad is a record of the work carried out by his under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this report, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Project Manager
Mr. Srinivas Raghothaman

Project Guide
Mrs. Srilatha Panchanathan

Abstract

Due to advancement in VLSI technology, transistors have been scaled down a lot incorporating more complex design in single System on Chip (SoC).
As the complexity of designs increases, verification emerges as a dominant step concerned with time and cost in the development of a system-on-chip. Increased design complexity mandates the need for functional verification.
The bug that is found at early level of abstraction will reduce the total cost incurred on a single chip so 70 % of the time is devoted in verifying the design.

Aim of the project is to build a scalable verification infrastructure and verification component to meet the verification challenges faced during verification process and to increase the timing efficiency of the Verification Engineer in debugging.
While verifying a complex design, many debugging challenges will be faced by a verification engineer. This report tries to discuss some of the verification and debugging challenges, faced during verification of a complex design and strategy to overcome this challenges.

Acknowledgement

I wish to express my heartfelt appreciation to all those who have contributed to this Major Project, both explicitly and implicitly, without the cooperation of whom, it would not have been possible to complete this Major Project.

Needless to mention that Prof. Vaishali Dhare (Internal Project Guide), Mr. Vishalkumar Dewan & Mrs. Srilatha Panchanathan (External Project Guides) who had been a source of inspiration and for his timely guidance in the conduct of my work for all his valuable assistance.

Finally, yet importantly, I would like to express my heartfelt thanks to my colleagues, mates for their help and wishes for the successful completion of this Major Project Part-1.

Naishal Shah
15MECV25

Table of Contents

Declaration ..... **i**

Certificate ..... **i**

Abstract ..... **i**

Acknowledgement ..... **v**

List of Figures ..... **ix**

Chapter 1: INTRODUCTION **1**

1.1 Organization Profile 1

1.2 Motivation 2

1.3 Verification Process 2

1.4 Thesis Outline 3

Chapter 2: Pre-Silicon Verification Flow **4**

2.1 VLSI Flow Process 4

2.1.1 Design Specification 4

2.1.2 Structural and Functional Description 5

2.1.3 Logic Design/ Register Transfer Level 5

2.1.4 Gate-Level Netlist 5

2.1.5 Physical Implementation 5

2.2 The Verification Process 6

2.3 Basic Test-bench Functionality 7

2.4 Directed Testing & Constrained Random testing 7

2.4.1 Directed Testing 7

2.4.2 Constrained-Random Testing 8

2.5 Testbench Components 9

2.6 Summary 9

Chapter 3: Verification Methodology **10**

3.1 Introduction to Open Verification Methodology 10

3.1.1 Features of OVM Methodology 10

3.1.2 OVM and Coverage-Driven Verification 11

3.1.3 OVM Test bench and Environments ... 11
3.2 OVM Overview ... 13
3.2.1 Data Item (transaction) ... 13
3.2.2 Driver (BFM) ... 13
3.2.3 Sequencer ... 13
3.2.4 Monitor ... 14
3.2.5 Agent ... 14
3.2.6 Environment ... 15
3.3 System Verilog Class Library ... 16
3.3.1 Other OVM Facility ... 17
3.3.2 OVM Factory ... 17
3.3.3 Transaction Level Modelling (TLM) ... 18
3.4 Summary ... 18
Chapter 4: IP Description ... 19
4.1 IP Overview & Features: ... 19
4.2 Project Overview ... 20
Chapter 5: Literature Review ... 22
5.1 GPIO ... 22
5.2 Random Constraints ... 23
5.2.1 Random Variables ... 23
5.2.2 Constraint Block ... 23
5.2.3 inside Operator ... 24
5.2.4 Distribution – dist. Operator ... 24
5.2.5 Randomization ... 24
5.3 Functional Coverage ... 25
5.4 Unified Coverage Report ... 26
5.5 Register Abstraction Layer (RAL) ... 27
5.6 On Chip Bus Interconnect Protocol Overview ... 28
5.6.1 OCP Specifications ... 28
5.6.2 AMBA AXI 4.0 Specifications ... 29
5.7 Power Aware Verification Terminology ... 31

5.7.1 Isolation Cell ... 31
5.7.2 Retention Registers ... 31
Chapter 6: Verification & Debugging Strategy ... 32
6.1 Complex Constraints solver ... 32
6.2 Coverage Report Parser ... 33
6.3 Scalability of the GPIO test cases ... 34
6.4 RAL Based Register Verification ... 35
6.5 AXI to OCP Converter ... 37
6.6 Retention & Isolation Checking Strategy:- ... 38
Chapter 7: Simulation Results & Coverage Results ... 40
7.1 Constraint Solver Results ... 40
7.2 Coverage results ... 41
7.3 Converter Upstream and Downstream Transaction Waveforms ... 42
7.4 Retention Verification Results:- ... 43
Conclusion ... 44
Future Work ... 44
References ... 45

List of Figures

Figure 1. VLSI Design Flow 4
Figure 2. Directed Test Progress over Time 8
Figure 3. Test Bench Component – A design Environment 9
Figure 4. Typical Verification Environment 12
Figure 5. OVC Environment 15
Figure 6. OVM Class Hierarchy 16
Figure 7. Block Diagram of an IP 19
Figure 8. GPIO Controller Block Diagram 22
Figure 9. A High Level Picture of the VCS Coverage Flow 27
Figure 10. Structure of the RAL based Environment 28
Figure 11. OCP on Chip Bus Architecture 29
Figure 12. Read Channel Architecture 30
Figure 13. Write Channel Architecture. 30
Figure 14. Flow Chart of Constraint solver 32
Figure 15. Coverage Parser Tool Flow 33
Figure 16. Block diagram of Register Verification 35
Figure 17. General Arch. Diagram of the IP blocks Interfacing with Fabric 37
Figure 18. Functional Diagram of the AXI to OCP converter 37
Figure 19. Flow for Isolation and Retention Checker Generation 39
Figure 20. Simulation Result generated by VCS-Mx 40
Figure 21. Filtered Output generated by Script 40
Figure 22. Variables which All values & Probability 41
Figure 23. URG Coverage Report Format 41
Figure 24. Tkdiff Result (Coverage and Cover group Difference) 42
Figure 25. AXI Slave Write Interface Signals 42
Figure 26. AXI Slave Read Interface Signals 43
Figure 27. Retention/Isolation bug Distribution for IP Blocks 43

Chapter 1

INTRODUCTION

1.1 Organization Profile

Intel Corporation is the world's largest semiconductor company and the inventor of the x86 series of microprocessors, the processors found in most personal computers. For more than three decades, Intel Corporation has developed technology enabling the computer and Internet revolution that has changed the world. Founded in 1968 to build semiconductor memory products, Intel introduced the world's first microprocessor in 1971. Today, Intel supplies the computing and telecommunications industries with chips, boards, systems, and software building blocks that are the ingredients of computers, servers and networking and communications products.

Intel is a leader in semiconductor manufacturing and technology and has established a competitive advantage through its scale of operations, agility of its factory network, and consistent execution worldwide. Intel has 26 FABs (Fabrication Units) and 13 ATM (Assembly Test and Manufacturing) centers worldwide. Intel produces the silicon for its high-performance microprocessors, chipset and flash memory components in its fabrication facilities (FABs). After the silicon-based products are created, they are sent to Intel's assembly and test facilities (ATMs) where each wafer is cut into individual microprocessors, placed within external packages, and tested for functionality.

Intel started its operations in India in 1988. Originally it was a sales and marketing office, Intel India soon expanded due to the country's I.T. and engineering talent pool. Now the majority of work done at Intel India is software and hardware engineering. Intel's Bangalore operations include the most Intel divisions in any country outside the United States.

1.2 Motivation

In 1965, Gordon Moore observed that the number of transistors in a single integrated circuit was doubling every two years and he predicted that this exponential growth would continue in perpetuity. As advances in methodologies, tools and process have enabled this growth, the task of verifying the functionality of these designs has become increasingly complex. The verification challenges are described below –

- As the number of transistors increases, the functionality is becoming more complex as more features are added.
- The cost of errors is increasing rapidly as the cost of masks increases.

The complexity of SoC devices are increasing day by day, along with them the time spent in verification is also increasing exponentially. According to recent statistics the mean time spent in verification of the total project time is more than 50%. As shown in the figure 1 the mean time spent in verification in the year 2007 was around 46 % which got increased to 51% in 2010 and to 53% in 2012. Owing to this there is a very high need to speed up the verification process.

1.3 Verification Process

The entire verification process can be sub-divided as shown in figure 2: As depicted in the figure 2 the major time spent in verification is in functional debugs, the next portion of time is taken by the test bench development the next is consumed by creating and running test cases and the remaining is consumed from test planning. A similar analogy can also be applied to developing a verification environment of a scaled up version. If some of these tasks can be automated the device can be verified faster. This is the main goal of project to automate as many tasks as possible and thereby speeding up the verification process.

1.4 Thesis Outline

This thesis is divided into seven chapters.

In this chapter **(Chapter 1)**, we have presented the motivation and need for the project.

Chapter 2 provides the brief description of the Pre-Silicon Verification Guidelines which include the basic verification flow and effective way of test bench development.

Chapter 3 provides complete OVM Methodology, one of the popular Verification methodologies used in industries for the Verification and discusses its complete structure.

Chapter 4 provides the complete overview of the project with the IP description

Chapter 5 discusses the literature survey on the building blocks of an IP and other topics which helped in this project.

Chapter 6 is the most important chapter in which the project is explained how to make the effective debugging and various verification challenges faced.

Chapter 7 summaries the results on the debugging infrastructure built in the verification environment and followed by conclusion and references.

Chapter 2

Pre-Silicon Verification Flow

2.1 VLSI Flow Process

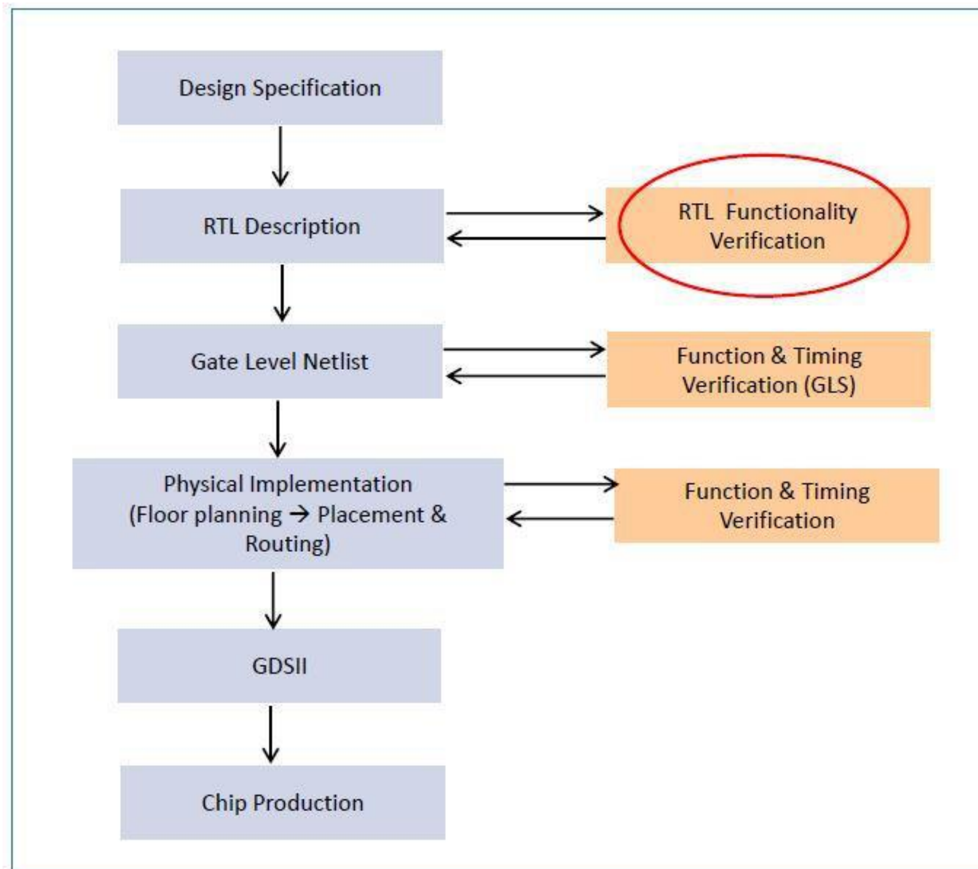


Figure 1. VLSI Design Flow

Below are the steps in the VLSI Design flow: -

2.1.1 Design Specification

- Goals and constraints of the design
- Functionality (what will the chip do)
- Performance figures like speed and power
- Technology constraints like size and space (physical dimensions)

2.1.2 Structural and Functional Description

Kind of architecture (structure) to be used for the design, example RISC/CISC, ALU, pipelining etc. To make it easier to design a complex system, it is normally broken down into several sub systems. The functionality of these subsystems should match the specifications.

2.1.3 Logic Design/ Register Transfer Level

The sub systems, top level systems once defined, need to be implemented. It is implemented using logic representation (Boolean Expressions), finite state machines, Combinational, sequential Logic, Schematics etc. Basically the RTL describes the several sub systems. It should match the functional description. RTL is expressed usually in Hardware Description Languages which is used to describe a digital system such as Verilog or VHDL. Functional/Logical Verification is performed at this stage to ensure the RTL designed matches the idea.

2.1.4 Gate-Level Netlist

Once Functional Verification is completed, the RTL is converted into an optimized Gate Level Netlist. This step is called **Logic/RTL synthesis**. This is done by Synthesis Tools such as Design Compiler (Synopsys), Blast Create (Magma), RTL Compiler (Cadence) etc. A synthesis tool takes an RTL hardware description and a standard cell library as input and produces a gate-level netlist as output. Standard cell library is the basic building block for today's IC design. Constraints such as timing, area, testability, and power are considered. Synthesis tools try to meet constraints, by calculating the cost of various implementations. It then tries to generate the best gate level implementation for a given set of constraints, target process. The resulting gate-level netlist is a completely structural description with only standard cells at the leaves of the design. At this stage, it is also verified whether the Gate Level Conversion has been correctly performed by doing simulation (GLS Gate Level Simulation).

2.1.5 Physical Implementation

The next step in the ASIC flow is the Physical Implementation of the Gate Level Netlist. The Gate level Netlist is converted into geometric representation that is the layout of the design.

The layout is designed according to the design rules specified in the library. The design rules are nothing but guidelines based on the limitations of the fabrication process.

The Physical Implementation step consists of three sub steps:

- Floor planning.
- Placement.
- Routing

The file produced at the output of the Physical Implementation is the GDSII file. It is the file used by the foundry to fabricate the ASIC. This step is performed by tools such as Blast Fusion (Magma), IC Compiler (Synopsys), and Encounter (Cadence) etc. Physical Verification is performed to verify whether the layout is designed according the rules.

2.2 The Verification Process

What is the goal of verification? Is it, "Finding bugs," then it is only partly correct. The goal of hardware design is to create a device that performs a particular task, such as a network router, or radar signal processor based on a design specification. The purpose as a verification engineer is to make sure the device can accomplish that task successfully i.e., the design is an accurate representation of the specification. Bugs are what we get when there is a discrepancy. The behavior of the device when used outside of its original purpose is not the verifier's responsibility, although he/she wants to know where those boundaries lie.

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code. A verification engineer must also read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features.

There are different ways to test the design. The easiest ones to detect are at the block level, in modules created by a single person. It is almost trivial to write directed tests to find these bugs, as they are contained entirely within one block of the design. After the block level, the next place to look for discrepancies is at boundaries between blocks. Interesting problems arise when two or more designers read the same description yet have different interpretations. The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. The

Verification Engineer's job is to find the disputed areas of logic and maybe even help reconcile these two different views. When the DUT is complex then the directed test will become a tedious job. So it demands a specific pre-planned verification plan for the testing all the scenarios in the DUT.

2.3 Basic Test-bench Functionality

The purpose of a test bench is to determine the correctness of the design under test (DUT). This is accomplished by the following steps.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness

2.4 Directed Testing & Constrained Random testing

2.4.1 Directed Testing

Traditionally, the task of verifying the correctness of a design, probably used directed tests. Using this approach, verification engineer look at the hardware specification and write a verification plan with a list of tests, each of which concentrated on a set of related features. Armed with this plan, he/she write stimulus vectors that exercise these features in the DUT. Then simulate the DUT with these vectors and manually review the resulting log files and waveforms to make sure the design does what is expected. Once the test works correctly, check it off in the verification plan and move to the next one. Figure 4 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. If there is enough time, it is able to write all the tests needed for 100% coverage of the entire verification plan.

What if we do not have the necessary time or resources to carry out the directed testing approach? When the design complexity doubles, it takes twice as long to complete or requires twice as many people to implement it. Neither of these situations is desirable. It demands a methodology that finds bugs faster in order to reach the goal of 100% coverage.

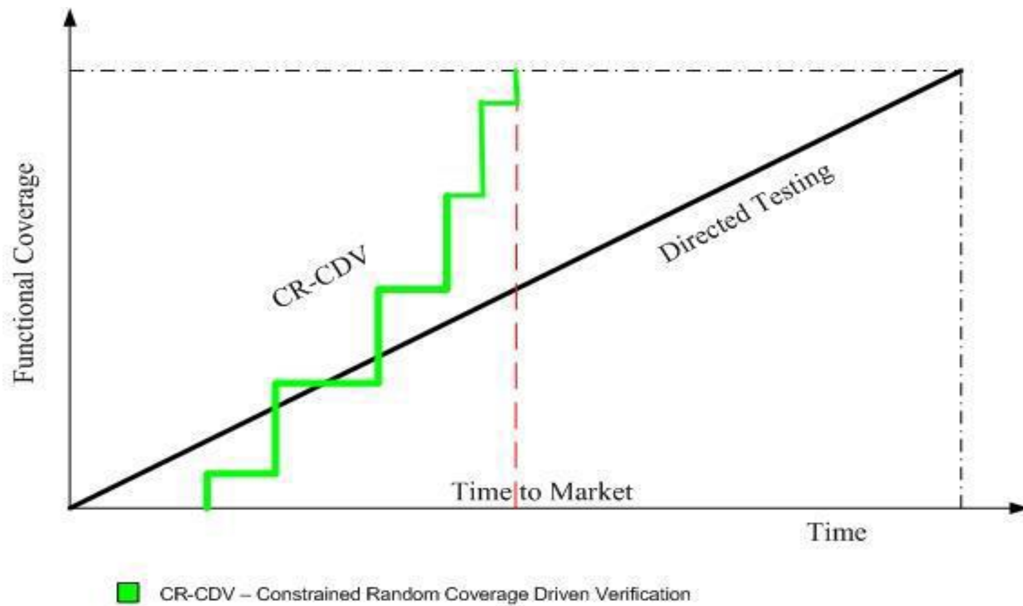


Figure 2. Directed Test Progress over Time

2.4.2 Constrained-Random Testing

Although it requires the simulator to generate the stimulus, it doesn't want totally random values. The Use of System Verilog language to describe the format of the stimulus (Eg: "address is 32-bits; Op-code is ADD, SUB or STORE; length < 32 bytes"), and then simulator picks values that meet the constraints. Constraining the random values to become relevant stimuli is one the key feature of System Verilog. These values are sent into the design, and are also sent into a high-level model that predicts what the result should be. The design's actual output is compared with the predicted output.

Figure 5 shows the paths to achieve complete coverage. Start at the upper left with basic constrained-random tests. Run them with many different seeds. From the functional coverage reports, find the holes where there are gaps in the coverage. Now make minimal code changes, perhaps by using new constraints, or by injecting errors or delays into the DUT. Spend most of the time in this outer loop by writing directed tests for only the few features that are very unlikely to be reached by random tests.

2.5 Testbench Components

In simulation, the test bench wraps around the DUT, just as a hardware tester connects to a physical chip, as shown in Figure 6. Both the test bench and tester provide stimulus and capture responses. The difference between them is that test bench needs to work over a wide range of levels of abstraction, creating transactions and sequences, which are eventually transformed into bit vectors. A tester just works at the bit level.

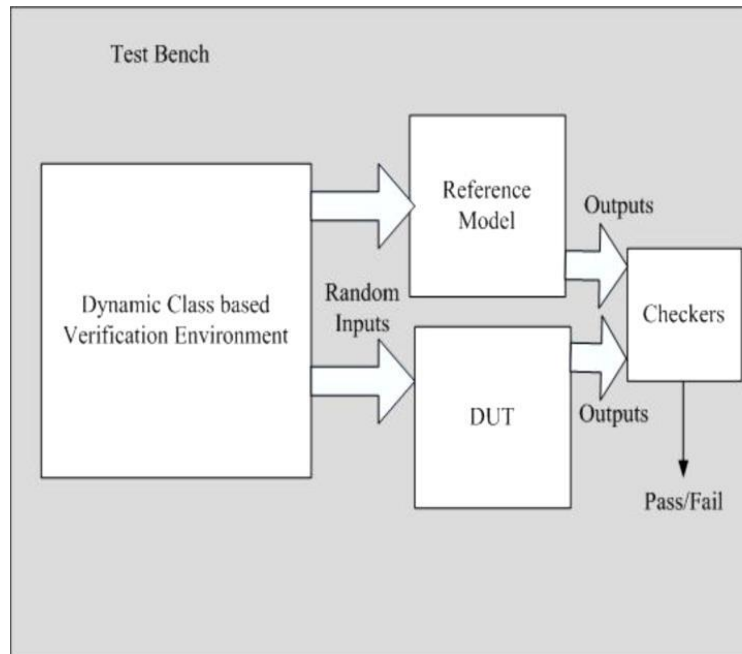


Figure 3. Test Bench Component – A design Environment

2.6 Summary

- The continuous growth in complexity of electronic designs requires a modern, systematic, and automated approach to create test benches.
- The cost of fixing a bug grows by tenfold as a project moves from each step of specification to RTL coding, gate synthesis, fabrication, and finally into the users hands.
- Directed tests only test one feature at a time and cannot create the complex stimulus and configurations that the device would be subjected in the real world.
- To produce robust designs, it is mandatory to use constrained-random stimulus combined with functional coverage to create the widest possible range of stimulus.

Chapter 3
Verification Methodology
3.1 Introduction to Open Verification Methodology
Traditionally RTL simulations are performed in Verilog or VHDL, now System Verilog and OVM as replacing whatever language and coding style that used for the test benches. OVM test benches are more than traditional HDL test benches, which might wiggle pins on the design-under-test (DUT) and rely on the designer to inspect a waveform diagram to verify correct operation. OVM test benches are complete verification environments composed of reusable verification components, and used as part of an overarching methodology of constrained random, coverage-driven, verification.

The key objectives of OVM are to enable productivity and verification component reuse within the verification environment. This is achieved through the separation of tests from the test bench, through having standardized conventions for assembling verification components, by allowing verification components to be highly configurable, and through the addition of automation features not provided natively by System Verilog.

OVM is supported by a library of System Verilog classes. OVM was created by Mentor Graphics and Cadence based on existing verification methodologies originating within those two companies, including Mentor's AVM, and consists of System Verilog code and documentation supplied under the Apache open-source license.

3.1.1 Features of OVM Methodology
OVM Methodology is a widely used Verification Methodology in all the semi-conductor industry. The features or advantages of OVM Methodology are as follow:

- Reusable across abstraction layers and design (parameterization, generalization, minimize dependencies, well defined semantics)
- Modular (localization of functionality, localization of data, communication through well-defined interfaces)
- Use standard interfaces (provide external view of object, hide implementation details and define interface semantics)
- Support stepwise refinement and abstraction (keep things at the highest level possible)

- Implemented in System Verilog and currently supported by VCSMX

3.1.2 OVM and Coverage-Driven Verification

OVM provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking test benches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests
- Ensure thorough verification using up-front goal setting
- Receive early error notifications and deploy run-time checking and error analysis to simplify debugging

Using CDV, it is possible to thoroughly verify the design by changing test bench parameters or changing the randomization seed. CDV environments support both directed and constrained-random testing. However, the preferred approach is to let constrained-random testing to do most of the work before devoting effort to writing time-consuming, deterministic tests to reach specific scenarios that are too difficult to reach randomly

3.1.3 OVM Test bench and Environments

An OVM test bench is composed of reusable verification environments called OVM verification components (OVCs). An OVC is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design sub module, or a full system. Each OVC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The OVC is applied to the device under test (DUT) to verify the implementation of the protocol or design architecture. OVCs expedite creation of efficient test benches for the DUT and are structured to work with any hardware description language (HDL) and high-level verification language (HVL) including Verilog, VHDL, ***e***, System Verilog, and System C. Figure 7 shows an example of a verification environment with three interface OVCs. These OVCs might be stored in a company repository and reused for multiple verification environments. The interface OVC is instantiated and configured for a desired operational mode.

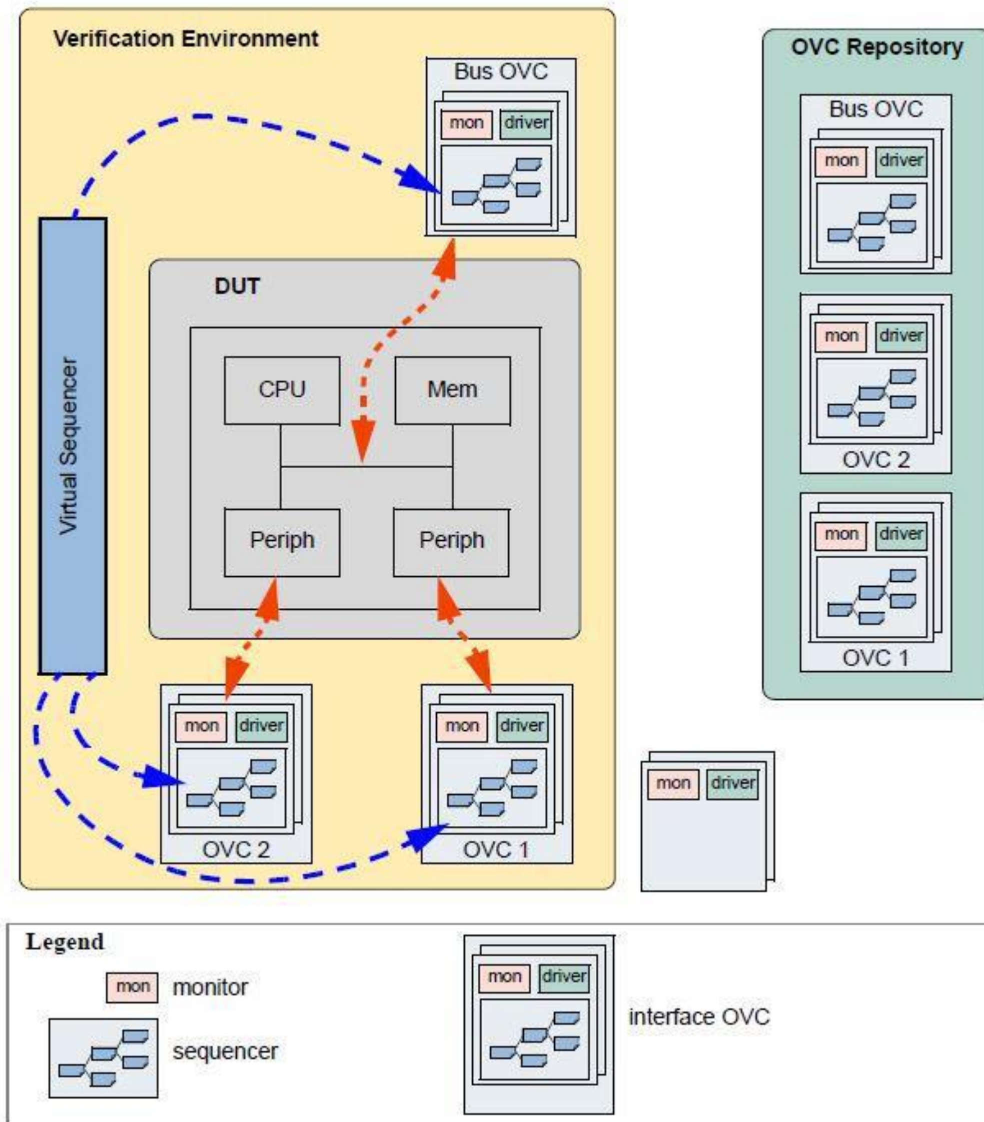


Figure 4. Typical Verification Environment

The verification environment also contains a multi-channel sequence mechanism (that is, virtual sequencer) which synchronizes the timing and the data between the different interfaces and allows fine control of the test environment for a particular test. In the figure 5 the agent is an OVC and the components inside the Agent are also OVC's

3.2 OVM Overview

The main components of an OVC are as follows:

- Data Item(Transaction)
- Driver(BFM)
- Sequencer
- Monitor
- Agent
- Environment

3.2.1 Data Item (transaction)

Data items represent the input to the DUT. Examples include networking packets, bus transactions, and instructions. The fields and attributes of a data item are derived from the data item's specification. For example, the Ethernet protocol specification defines valid values and attributes for an Ethernet data packet. In a typical test, many data items are generated and sent to the DUT.

3.2.2 Driver (BFM)

A driver is an active entity that emulates logic that drives the DUT. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals. For example, a driver controls the read/write signal, address bus, and data bus for a number of clocks cycles to perform a write transfer.

3.2.3 Sequencer

A sequencer is an advanced stimulus generator that controls the items that are provided to the driver for execution. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior allows adding constraints to the data item class in order to control the distribution of randomized values.

A partial list of the sequencer's built-in capabilities includes:

- Ability to react to the current state of the DUT for every data item generated.
- Captures the order between data items in user-defined sequences, which forms a more structured and meaningful stimulus pattern.
- Enables time modeling in reusable scenarios.
- Supports declarative and procedural constraints for the same scenario.
- Allows system-level synchronization and control of multiple interfaces.

3.2.4 Monitor

A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information and perform checking. A monitor:

- Collects transactions (data items). A monitor extracts signal information from a bus and translates the information into a transaction that can be made available to other components and to the test writer
- Extracts events. The monitor detects the availability of information (such as a transaction), structures the data, and emits an event to notify other components of the availability of the transaction. A monitor also captures status information so it is available to other components and to the test writer.
- Performs checking and coverage
    i. Checking typically consists of protocol and data checkers to verify that the DUT output meets the protocol specification
    ii. Coverage also is collected in the monitor

3.2.5 Agent

Sequencers, drivers, and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, OVM recommends that environment developers create a more abstract container called an agent. Agents can emulate and verify DUT devices. They encapsulate a driver, sequencer, and monitor. OVCs can contain more than one agent. Some agents (for example, master or transmit agents) initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or

passive. Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity

3.2.6 Environment

The environment (env) is the top-level component of the OVC. It contains one or more agents, as well as other components such as a bus monitor. The env contains configuration properties that enable to customize the topology and behavior and make it reusable. For example, active agents can be changed into passive agents when the verification environment is reused in system verification. Figure 8 illustrates the structure of a reusable verification environment. Notice that an OVC may contain an environment-level monitor. This bus-level monitor performs checking and coverage for activities that are not necessarily related to a single agent. An agent's monitors can leverage data and events collected by the global monitor. The environment class (ovm_env) is architected to provide a flexible, reusable, and extendable verification component. The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.

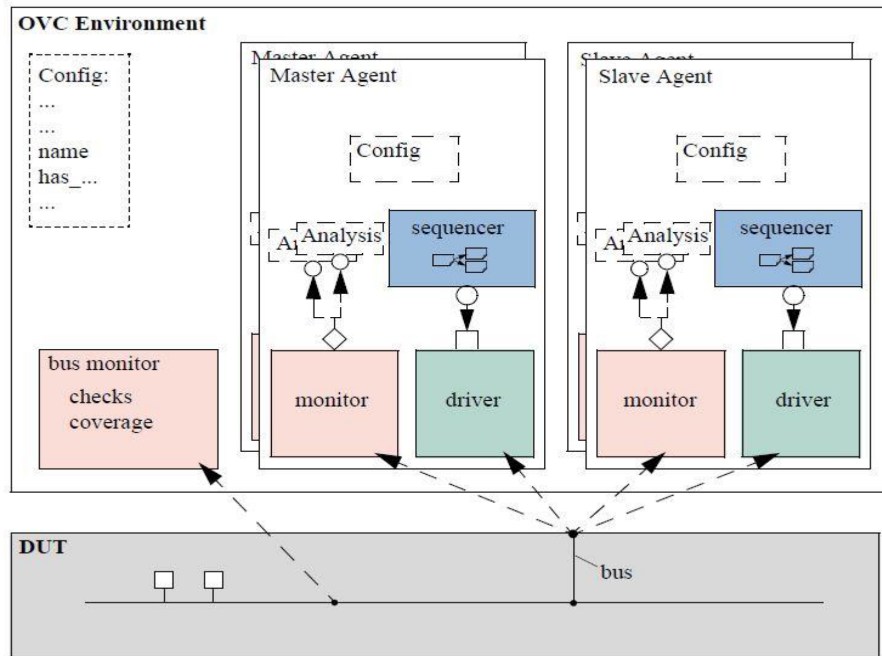


Figure 5. OVC Environment

3.3 System Verilog Class Library

The System Verilog OVM Class Library provides all the building blocks that are needed to quickly develop well-constructed, reusable, verification components and test environments. The library consists of base classes, utilities, and macros.

Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. These phases are defined in the base class library but can be extended to meet specific project needs.

Figure 6. OVM Class Hierarchy

The advantages of using the System Verilog OVM Class Library include:

- A robust set of built-in features The System Verilogma OVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory methods, and more.
- Correctly implemented OVM concepts each component in the block diagram in Figure 8 is derived from a corresponding System Verilog OVM Class Library component. Figure 9

shows the same diagram using the derived System Verilog OVM Class Library base classes. Using these base-class elements increases the readability of the code since each component's role is predetermined by its parent class.

3.3.1 Other OVM Facility

The System Verilog OVM Class Library also provides various utilities to simplify the development and use of verification environments. These utilities support debugging by providing a user-controllable messaging utility. They support development by providing a standard communication infrastructure between verification components (TLM) and flexible verification environment construction (OVM factory).

The System Verilog OVM Class Library provides global messaging facilities that can be used for failure reporting and general reporting purposes. Both messages and reporting are important aspects of ease of use.

It Includes:

- OVM Factory
- TLM(Transaction Level Modelling)

3.3.2 OVM Factory

The factory method is a classic software design pattern that is used to create generic code, deferring to run time the exact specification of the object that will be created. In functional verification, introducing class variations is frequently needed. For example, in many tests it might want to derive from the generic data item definition and add more constraints or fields to it; or it might want to use the new derived class in the entire environment or only in single interface; or perhaps it must modify the way data is sent to the DUT by deriving a new driver. The factory allows substituting the verification component without having to provide a derived version of the parent component as well.

The System Verilog OVM Class Library provides a built-in central factory that allows:

- Controlling object allocation in the entire environment or for specific objects.
- Modifying stimulus data items as well as infrastructure components (for example, a driver).

3.3.3 Transaction Level Modelling (TLM)

OVM components communicate via standard a TLM interface, which improves reuse. Using a System Verilog implementation of TLM in OVM, a component may communicate via its interface to any other component that implements that interface. Each TLM interface consists of one or more methods used to transport data. TLM specifies the required behavior (semantic) of each method but does not define their implementation. Classes inheriting a TLM interface must provide an implementation that meets the specified semantic.

Thus, one component may be connected at the transaction level to others that are implemented at multiple levels of abstraction. The common semantics of TLM communication permit components to be swapped in and out without affecting the rest of the environment.

3.4 Summary

- This chapter summarized various OVM based components and explained each and every component.
- The advantages of using a System Verilog class based structure and OVM gives enough flexibility and randomness in the present verification environment.
- Explained how to develop each and every component in the OVM Environment for the development of the Verification.
- OVM gives the power of re-usability and randomness which helps to find the hidden bugs in the RTL Design

Chapter 4

IP Description

4.1 IP Overview & Features:

It is a Soft IP, which is designed with the goal of "Always On, Always Sensing" and it provides the following functions to support this goal:

1. Acquisition / sampling of sensor data.
2. Low power operation through clock and power gating of the different blocks
3. The ability to operate independently when the main processor is in a low power state
4. Compatibility with various operating systems, such as Win 7, Windows 8, and Android
5. Ability to provide sensor-related data to other subsystems.

It consists of the following key components:

A processor with a combined cache for instructions and data.

- ROM space intended for the boot loader.
- SRAM space for code and data.
- Interfaces to Sensor peripherals. (I2C, SPI, UART, GPIO)

<u>Block Diagram</u>:-

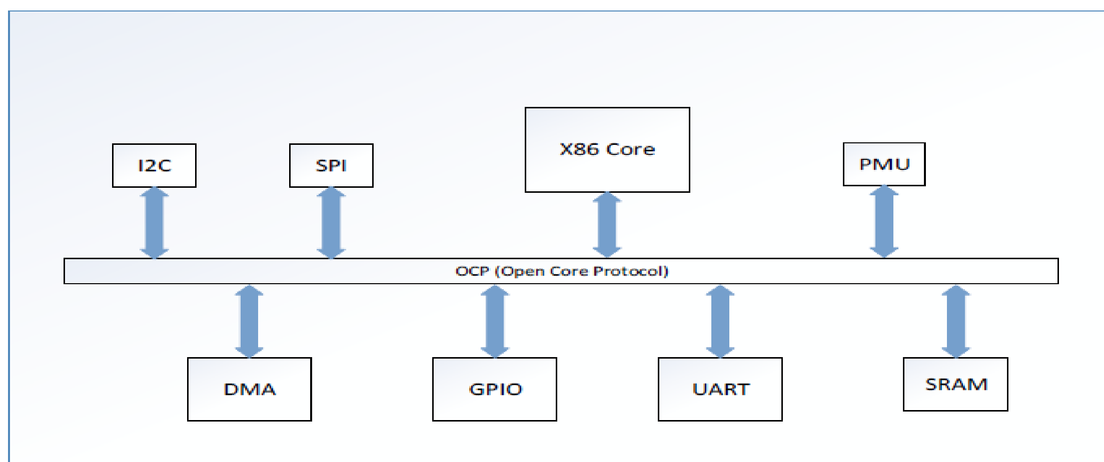


Figure 7. Block Diagram of an IP

4.2 Project Overview

As the complexity of designs increases, verification emerges as a dominant step concerned with time and cost in the development of a system-on-chip. So different verification strategies need to be implemented to verify a complex design. While verifying a complex design, many debugging challenges will be faced by a verification engineer. This report tries to discuss some of the debugging challenges faced during verification of a complex design.

In the present IP subsystems as shown in the block diagram, subIPs such as DMA, SPI, I2C, GPIO, UART, Corex86, SRAM etc. will be present for different purposes.

DMA helps in transferring data between the memories, either between internal memories or from external memory to internal memory. SPI bus is a synchronous serial communication interface used for short distance communication. I2C is typically used for attaching lower-speed peripheral ICs to processors. SRAM is used to store the data.

The communication between each of these subIPs will take place consistently for data transactions either for transmitting or receiving the data. Data transactions maybe from external memory to the internal memory through DMA, any data which is coming from outside of the IP through I2C, any data transactions within the IP through SPI or any GPIO transactions.

In verification environment, for random test cases a random stimulus will be created for such transactions. These stimulus are constrained randomized to meet the particular functionality and hit the scenario. As the complexity of the design increases, constraints of the random stimulus become more complex. After the regression run for a large amount of the time if scenario doesn't hit it costs verification engineer lot of wastage of time. There is no proper tool exists that can solve these complex constraints. So, we developed such tool which can solve these constraints. And this in turn help us to know if any of these specified constraints fails. This will help a verification engineer to check whether the intent of the stimulus is met or not and also shows the quality of the stimulus created to verify the design.

As soon as we start running a test case, the corex86 starts executing the important functions which maybe either an API of each subIP or an Interrupt handler to service the interrupts. For verification engineer it is very difficult to know when these important functions are being called. Instruction

Debugger tool helps in finding the regime of these important functions in the simulation window which are executed by the corex86 during the simulation.

Sometimes just by knowing the regime of important functions is not enough to debug the issues. Hence the monitor at the corex86 interface helps to know whether the function is enabled and scheduled for execution by the core. This helps the verifier to debug very fast and efficiently.

One of the verification challenge is when moving from old project to the new project with different set of specifications. It is very difficult to port the test cases to the new project as the project specification has changed. If we are having a project specific parameters, which can be used to stamping out these test cases, then porting from old project to new project becomes very easy.

Chapter 5

Literature Review

5.1 GPIO

A general-purpose I/O (GPIO) controller for use in generating and capturing application specific input and output signals. The GPIO controller is connected to the OCP Fabric. When the pin mode is chosen as GPIO it can be programmed as an output or input.

When programmed as an input, a GPIO can serve as an interrupt source. For example, GPIO when configured as inputs & connected to the interrupt outputs from sensors. When programmed as an output, GPIO pins can be connected to the power control pins on sensors or to external FETs that can gate the power to the external sensors.

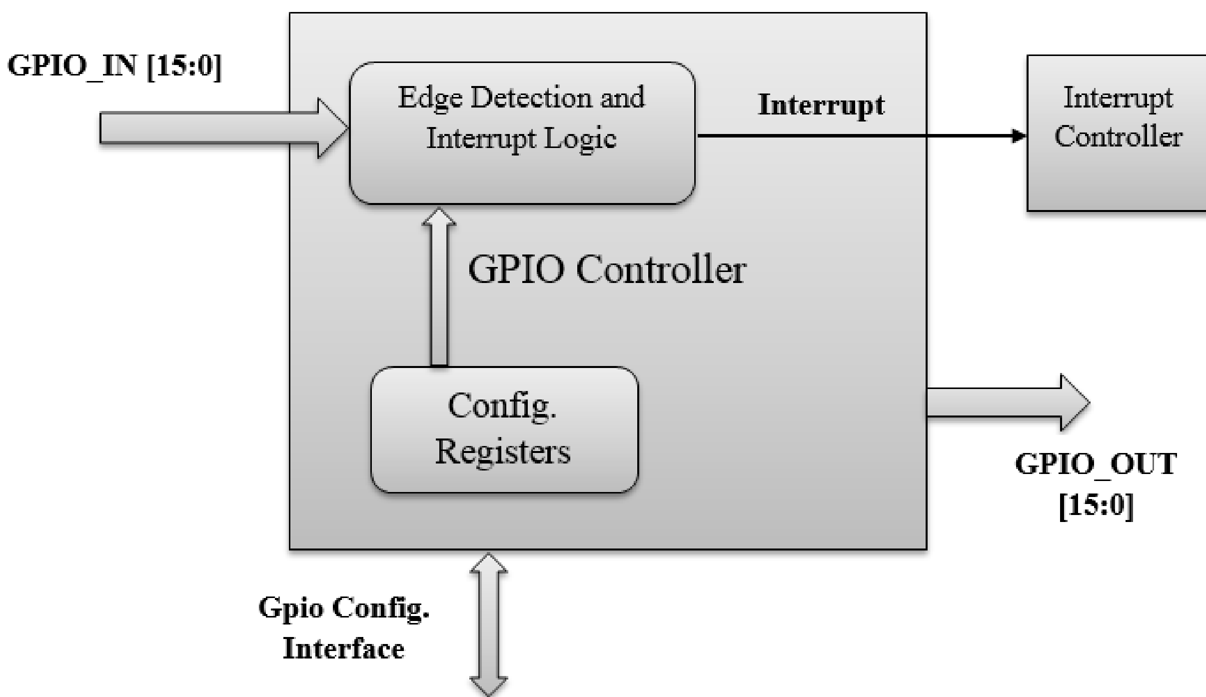


Figure 8. GPIO Controller Block Diagram

5.2 Random Constraints

5.2.1 Random Variables

- Random variables within class can be declared using *rand* or *randc.*
- Variables declared with *rand* are uniformly distributed over their range.
    - Example :-

      rand byte header;

      Header will be 8 bit unsigned integer with the range from 0-255.
- Variables declared with *randc* are random cyclic variables that cycle through all the values in a random permutation of their declared range and never repeat in a iteration. They can only be of type *bit* or *enumerated* types and can be limited to a maximum size.
    - Example :-

      randc bit [1:0] length;

      Length can take values 0,1,2,3.

5.2.2 Constraint Block

- Constraint expressions defined in Constraint blocks determine the values of random variables.
- Constraint blocks are class members, like tasks, functions, and variables.
- Multiple constraint can be defined within a class
- Constraint block names must be unique within a class
    - Example : -

      rand integer x,y,z;

      constraint c1{ x > 0 ; x< 500;

      y < x; z > x; }

      Value of x **1-499**, Value of y **less than** x & Value of z **greater than** x.

5.2.3 inside Operator

- All values specified by the inside operator have an equal probability of being Chosen.
  - Example :-

```
rand bit[7:0] z;
constraint c1 { z inside {5,10,[12:15],20}; }
```

    Z will be assigned the values specified with equal probability.

5.2.4 Distribution – dist. Operator

- Distribution is a form of constraints that support sets of weighted values.
- The := operator assigns the specified weight to the item, or if the item is a range, to every value in the range.
- The :/ operator assigns the specified weight, or if the item is a range, to the range as a whole. So the weight of each value in the range is range_weight/n.
  - Example : -

```
rand bit[7:0] y;
constraint c1{ y dist {5:=1, 7:=2, 9:=5} };
```

    Value of y will be 5, 7, and 9 with ratios of 1, 2, and 5.

5.2.5 Randomization

- Variables in an object are randomized using *randomize()* class method
- Every class contains pre_randomize() and post_randomize() methods, which are automatically called by randomize() before and after computing new random values
  - Example : -

```
Class packet;  //Class definition
rand bit [7:0] data;
endclass
task data_generator; //Task to Generate Random data packets
packet  p;                // Object of the class packet
p.randomize(data); //Randomize the data variable.
endtask
```

5.3 Functional Coverage

Functional verification comprises a large portion of the resources required to design and validate a complex system. Often, the validation must be comprehensive without redundant effort. To minimize wasted effort, coverage is used as a guide for directing verification resources by identifying tested and untested portions of the design.

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project, in order to reduce the number of simulation cycles spent in verifying a design.

Broadly speaking, there are two types of coverage metrics. Those that can be automatically extracted from the design code, such as code coverage, and those that are user-specified in order to tie the verification environment to the design intent or functionality. This latter form is referred to as Functional Coverage, and is the topic of this section.

Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions—captured as features of the test plan—have been observed, validated and tested.

The key aspects of functional coverage are:

- It is user-specified, and is not automatically inferred from the design.
- It is based on the design specification (i.e., its intent) and is thus independent of the actual design code or its structure.

Since it is fully specified by the user, functional coverage requires more up front effort (someone has to write the coverage model). Functional coverage also requires a more structured approach to verification. Although functional coverage can shorten the overall verification effort and yield higher quality designs, these shortcomings can impede its adoption.

The System Verilog functional coverage extensions address these shortcomings by providing language constructs for easy specification of functional coverage models. This specification can be efficiently executed by the System Verilog simulation engine, thus, enabling coverage data

manipulation and analysis tools that speed up the development of high quality tests. The improved set of tests can exercise more corner cases and required scenarios, without redundant work.

The System Verilog functional coverage constructs enable:

- Coverage of variables and expressions, as well as cross coverage between them.
- Automatic as well as user-defined coverage bins
- Associate bins with sets of values, transitions, or cross products.
- Filtering conditions at multiple levels.
- Events and sequences to automatically trigger coverage sampling.
- Procedural activation and query of coverage.
- Optional directives to control and regulate coverage.

5.4 Unified Coverage Report

VCS can monitor and evaluate the coverage metrics of Verilog, VHDL, and mixed HDL designs during simulation to determine which portions of the design have not been tested. The results of the analysis are reported in a number of ways that allow you to see the shortcomings in your testbench and improve tests as needed to obtain the complete coverage. Functional coverage is the determination of how much functionality of the design has been exercised by the verification environment.

The Unified Report Generator (URG) generates combined reports for all types of coverage – (Line, Toggle, Condition, FSM, Branch, Assertion) information. The reports may be viewed through the design hierarchy, module lists, coverage groups, or through an overall summary "dashboard" for the entire design/test bench.

The reports consist of a set of HTML or text files. The HTML version of the reports take the form of multiple interlinked HTML files. For example, a "hierarchy.html" page shows the design’s hierarchy and contains links to individual pages for each module and its instances. The HTML file that the URG writes can be read by any web browser.

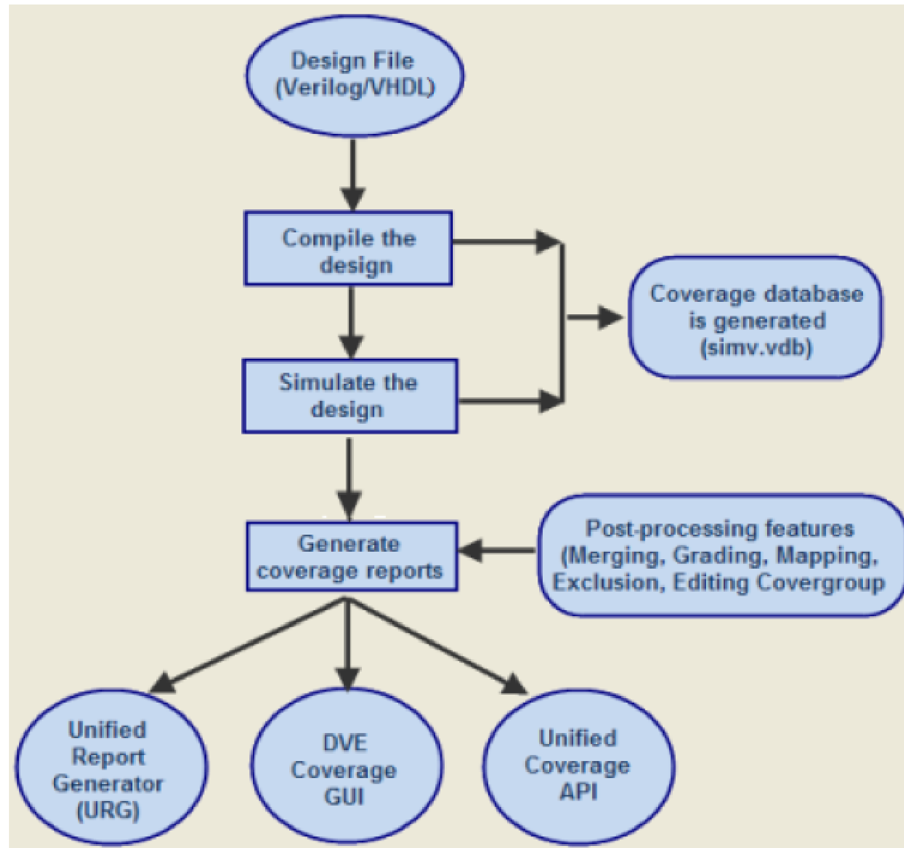


Figure 9. A High Level Picture of the VCS Coverage Flow

5.5 Register Abstraction Layer (RAL)

The Register Abstraction Layer (RAL) is a System Verilog or application package used to automate the creation of a high-level, object-oriented abstraction layer for memory-mapped registers and memories in a design under verification (DUV). The abstraction mechanism allows verification environments and tests to be migrated from block to system levels without any modifications. It also allows fields to be moved between physical registers without requiring modifications in the verification environment or tests.

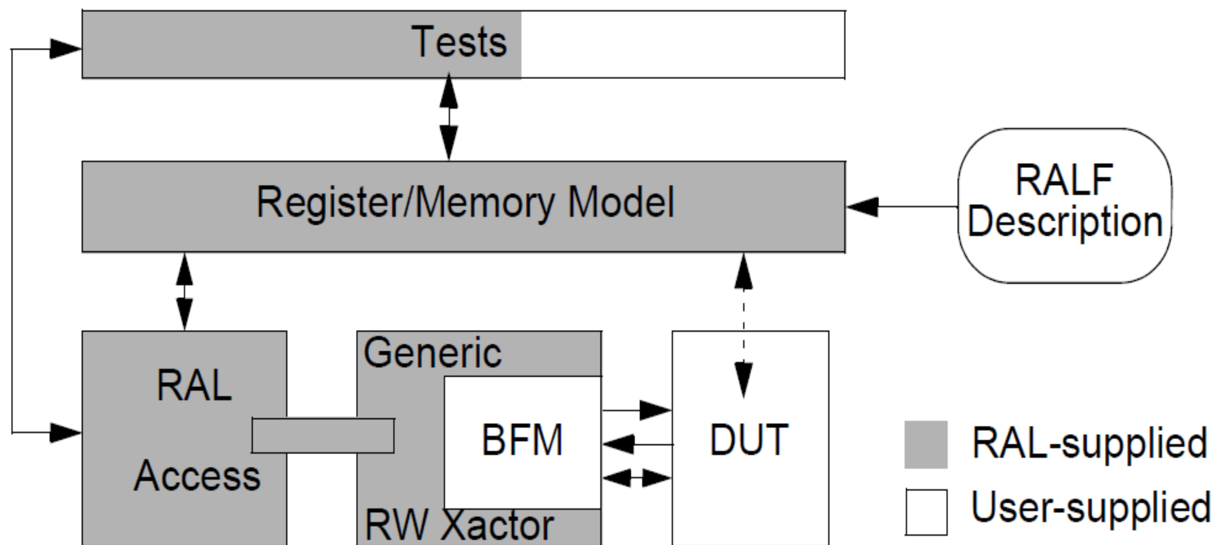


Figure 10. Structure of the RAL based Environment

The RAL includes predefined testcases that you can use to verify the correct operation of registers and memories in a design under verification. It includes usage assertions to detect incorrect register and memory accesses. A functional coverage model is included to accurately measure how thoroughly the registers and memories have been exercised. The RAL supports front-door and back-door access to provide redundant paths to the register and memory implementation, and verify the correctness of the decoding and access paths. The RAL also supports designs with multiple physical interfaces, as well as registers, register files and memories shared across multiple interfaces.

5.6 On Chip Bus Interconnect Protocol Overview

5.6.1 OCP Specifications

The Open Core Protocol (OCP) defines a high-performance, bus independent interface between IP cores and OCP is composed of uni-directional signals driven, and sampled by the rising edge of the OCP clock. The OCP is fully synchronous (with the exception of reset) and contains no multi-cycle timing paths with respect to the OCP clock. All signals other than the clock signal are strictly point-to-point.

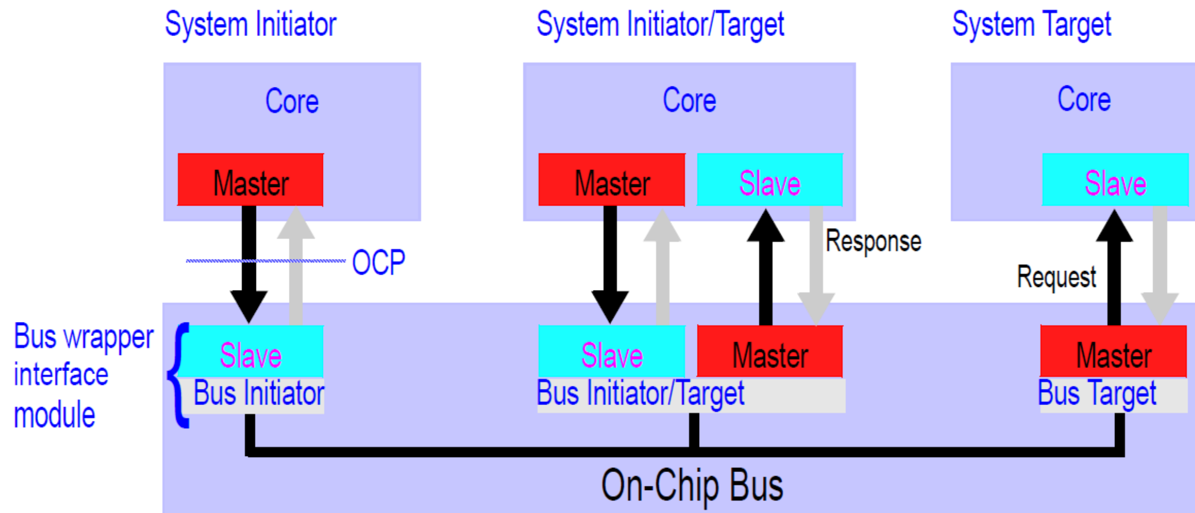


Figure 11. OCP on Chip Bus Architecture

The OCP defines a point-to-point interface between two communicating entities such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave.

5.6.2 AMBA AXI 4.0 Specifications

The AMBA AXI protocol supports high-performance, high-frequency system designs.

The AXI protocol:

• is suitable for high-bandwidth and low-latency designs

• provides high-frequency operation without using complex bridges

• meets the interface requirements of a wide range of components

The key features of the AXI protocol are: -

• separate address/control and data phases

• support for unaligned data transfers, using byte strobes

• uses burst-based transactions with only the start address issued

• separate read and write data channels that can provide low-cost *Direct Memory Access* (DMA)

• support for issuing multiple outstanding addresses

• support for out-of-order transaction completion

• permits easy addition of register stages to provide timing closure.

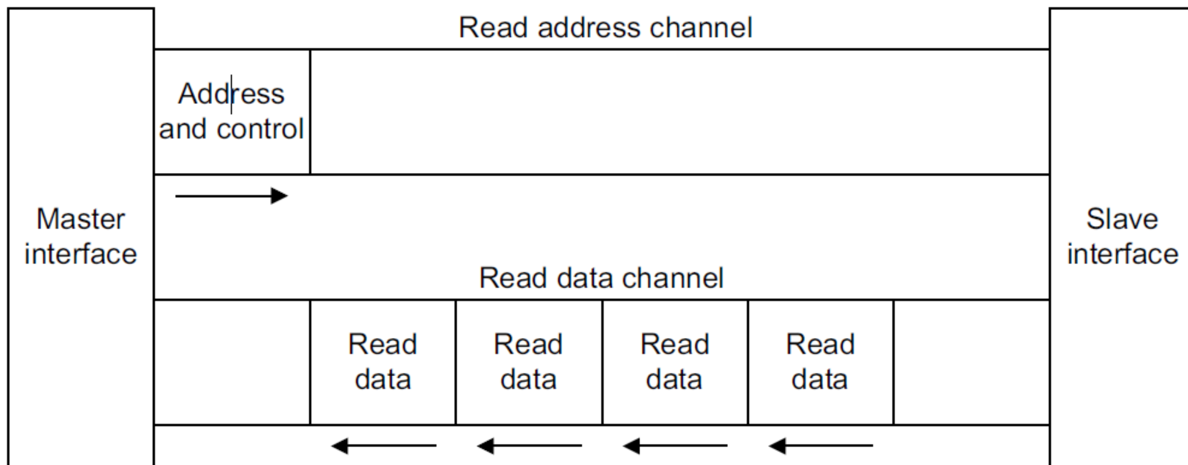


Figure 12. Read Channel Architecture

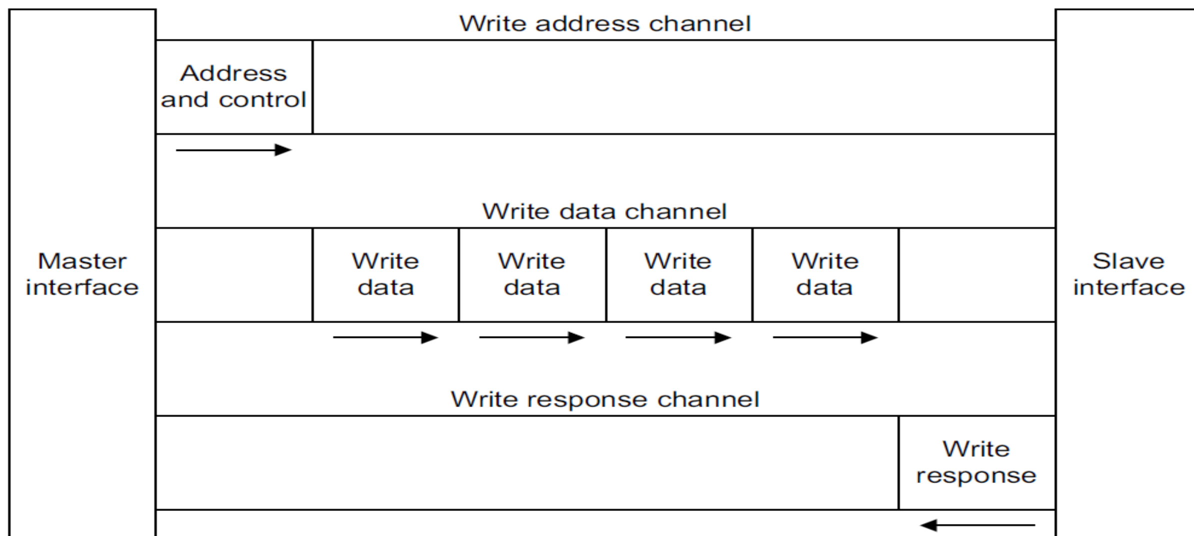


Figure 13. Write Channel Architecture.

5.7 Power Aware Verification Terminology

5.7.1 Isolation Cell

Isolation cells are used to prevent short circuit current. As the name indicates these cells isolate power gated block from the normally powered on block. Isolation cells are specially designed for low short circuit current when input is at threshold voltage level. Isolation control signals are provided by power gating controller. Isolation of the signals of a switchable module is essential to preserve design integrity. Usually a simple OR or AND logic can function as an output isolation device.

5.7.2 Retention Registers

Multiple state retention schemes are available in practice to preserve the state before a module shuts down. The simplest technique is to scan out the register values into a memory before shutting down a module. When the module wakes up, the values are scanned back from the memory. That memory cells which holds the values of the power gated cells during PG state are inferred as a retention registers.

When power gating is used, the system needs some form of state retention, such as scanning out data to a RAM, then scanning it back in when the system is reawakened. For critical applications, the memory states must be maintained within the cell, a condition that requires a retention flop to store bits in a table. That makes it possible to restore the bits very quickly during wakeup. Retention registers are special low leakage flip-flops used to hold the data of main register of the power gated block. Thus internal state of the block during power down mode can be retained and loaded back to it when the block is reactivated. Retention registers are always powered up. The retention strategy is design dependent. During the power gating data can be retained and transferred back to block when power gating is withdrawn. Power gating controller controls the retention mechanism such as when to save the current contents of the power gating block and when to restore it back.

Chapter 6

Verification & Debugging Strategy

6.1 Complex Constraints solver

Constraint solver is a tool which is used for solving very complex constraints statically before running the long regression run. It reduces the debugging time if the scenario doesn't hit.

- **Solver Tool Flow : -**

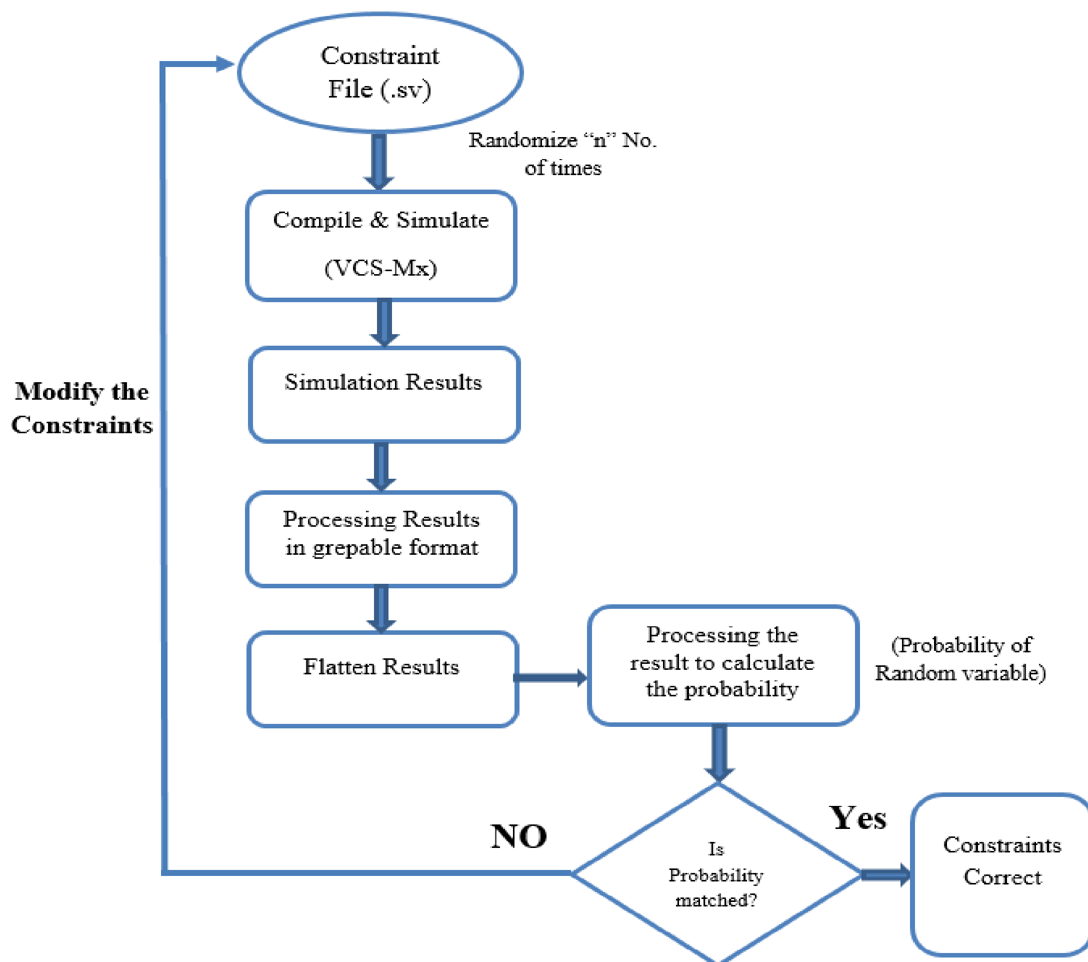


Figure 14. Flow Chart of Constraint solver

Pcregrep is the command used in the Linux command line interface which searches files for character patterns, in the same way as other grep commands do, but it uses the PCRE regular expression library to support patterns that are compatible with the regular expressions of Perl.

6.2 Coverage Report Parser

Unified Report Generator (URG) generates Coverage Report form the coverage database for all types of the coverage. It generate report in the **two format 1. HTML file format 2.txt file format.** URG can generate the difference of the two coverage database of the same executable which is simulated for two different seeds. But it cannot generate the difference of the two different coverage database of the two different executables.

Coverage Report parser is the tool which helps verifier in knowing if there is any other cover groups or variable added or deleted and difference in the coverage statistics as well.

This tool is based on the HTML Parser, HTML TableExtract – CPAN Perl based Modules used to parse out the useful data from the coverage report (HTML format). And that Extracted data can be diff. by **tkdiff** tool.

- **Tool Flow :-**

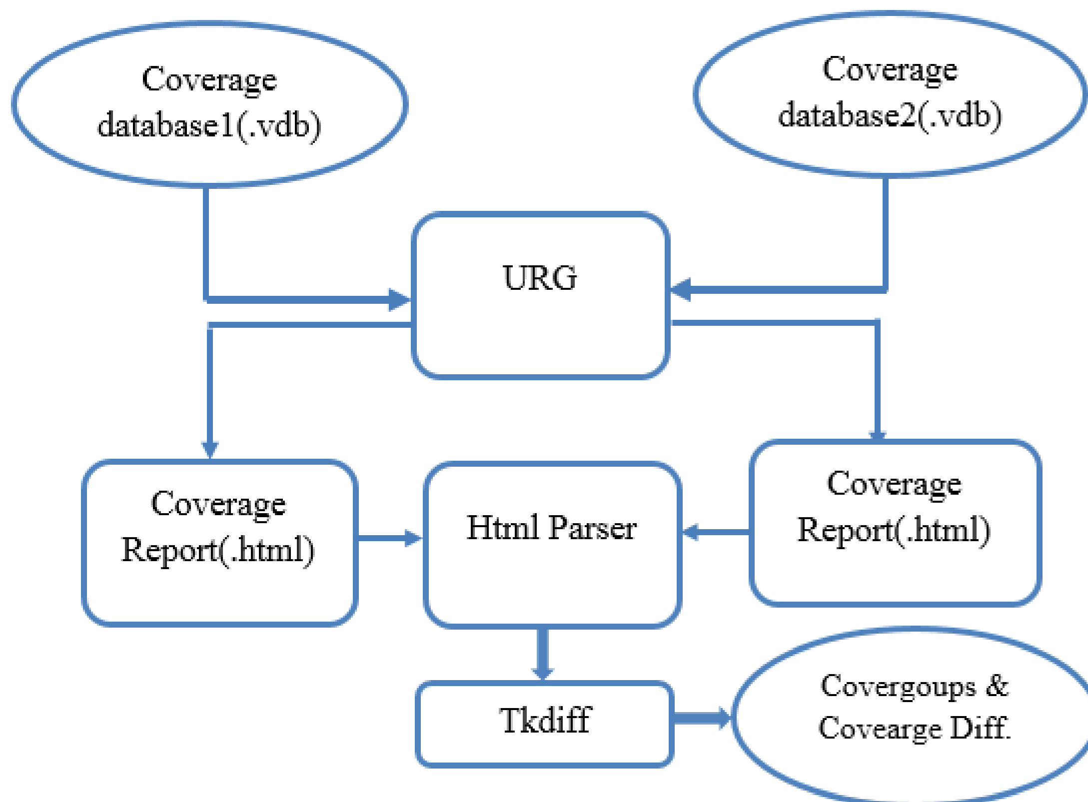


Figure 15. Coverage Parser Tool Flow

6.3 Scalability of the GPIO test cases

One of the verification challenge is when moving from old project to the new project with different set of specifications. It is very difficult to port the test cases to the new project as the project specification has changed. If we are having a project specific parameters, which can be used to stamping out these test cases, then porting from old project to new project becomes very easy and make the verification environment more scalable.

- GPIO Test cases

  In the Previous Project only 32 pins of the GPIO were supported by the IP. As new project came in, specifications revised and it requires the 128 pins of the GPIO to be supported by the IP. In order to meet the specification, test cases of the GPIO need to be scaled to 128 pins.

  Test cases contains two part 1.SV part 2. C part

  C part is used to configure the control registers, and it is compiled by the GCC compiler which load the data & instructions into memory. Processor executes them and to read and write to control registers and message registers.

  SV part is used to generate the sequences and these sequences (transaction) are driven by sequencer and driver to the DUT. There are some of the assertions and checkers are introduced for checking the condition and intent of the test is met or not.

  Test cases ate of two types 1. Focused tests 2. Random Tests

  1. Focused tests: - These tests are used to verify the particular functionality/feature of the GPIO in which at a time single pin is asserted and if the pin is configure to generate the interrupt, interrupt monitor check for it.

  2. Random Tests: - Random tests are used to verify the design’s most of all features exhaustively. Here Random stimulus are generated using System Verilog’s Constraint Randomization feature. GPIO pins are selected randomly to assert the pulses.

6.4 RAL Based Register Verification

Register description file contains register name, register description, fields of register, offset value of register. Each register is of 32 bits and the register is divided into fields. Each fields contains field name, field description, field size, bit position inside register, its access type or attribute, its default value. Default value of the register will be combined from the fields default value. The register description cannot be directly used as it is converted into RAL (Register Abstraction Layer) by using a script, which can be directly used by the test bench.

The OVM based test is cleared, which contains own phases.

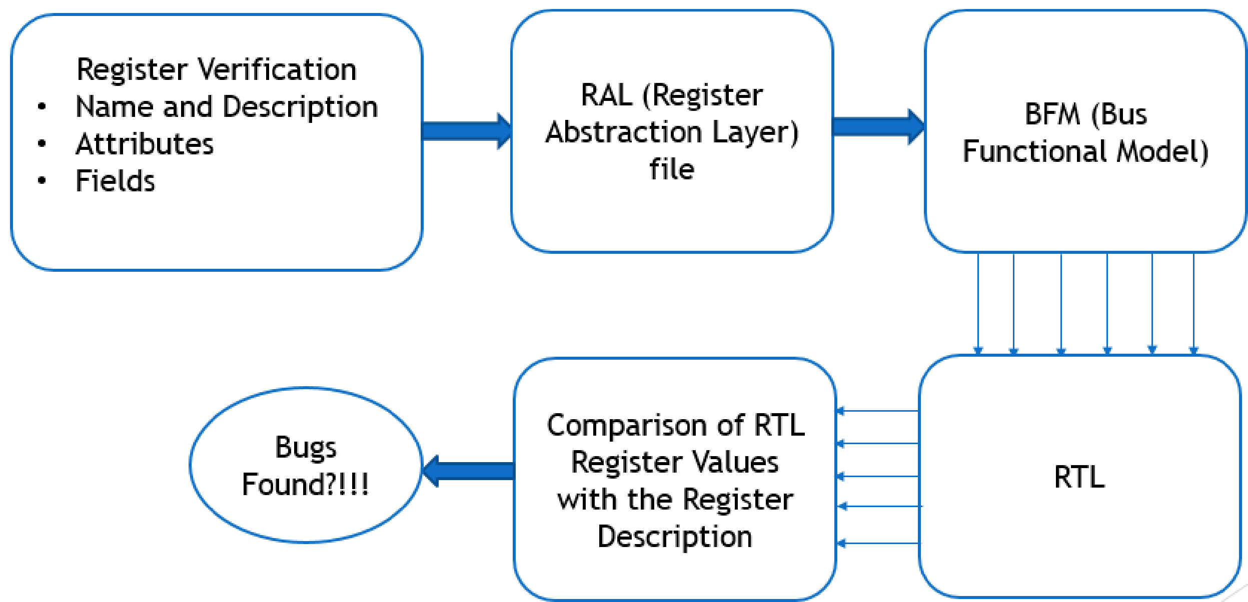


Figure 16. Block diagram of Register Verification

1. **Build Phase**: - In this phase the environment of the test is created and configured according to the project requirement.
2. **Connect Phase:** This phase calls the sequence, determines the amount of time the test should run that is the drain-out time which is generally kept as 100ms.
   This drain out time should be selected carefully because this determines the amount of the time the test should take. Hence the intent of the test should be achieved.

The Sequence for the test is created. This sequence is assigned to the proper sequencer by ovm factory method that is ‘ovm_sequence_utils (sequence_name , sequencer_used). This is a self – checking test. It means that it does not use the third component of the testbench like scoreboard, monitor. These tests frontdoor access method. That is each register will be called and if any activity in the bus transaction is captured. This sequence creates the pointer to the RAL file and by using this pointer all the registers in the RAL file is populated in an array with all information like, name, base address, offset address, width of the registers, field of register, and its attributes. This

populated register is then randomized. Some registers and the bits of the registers are masked or sometimes skipped for the checking. This is done because we do not want to enable the functionality of the RTL. For this case the dummy model or the IP is required and hence registers or the field like enable bit, reset bits, interrupt bits are masked. Now the task which contains read write function will be called. The three checks are performed.

1. **Reset Value Check**: It checks the default value of register. This happens as soon as the IP powers up and the reset happens. In this checks only the register address is required. Then the read and compare happens, If the value compare does not matched the 'ovm_error' is called.

2. **Attribute Check**: In this check the write to the register is done and we read back the entire register. The write value will be random value and the register locations will be random. The read only bits will not be modified and hence the bit will be treated as 0. Here when we write the RTL register, the shadow copy of the register that is RAL register also gets updated. Then we compare the value of the shadow register and the RTL register. So in this check we write to one register and we read back the value of register and then we compare the value of the register.

3. Write All Read All Check: - Some errors were not captured while performing attribute check. The bugs like, if two register locations are shorted, writing one and reading one register will not be able to catch such bugs. Hence we write to all registers and read all registers back.

6.5 AXI to OCP Converter

Need for Protocol Conversion:-

Generally Embedded IP and SoC both have different on chip bus Connects. When SoC's one of the IPs try to communicate with another IP's any device it can be done through the Bus Interconnect. But due to different Bus protocols it's mandatory to use the some bridge or converter between them which translates the one protocol to another.

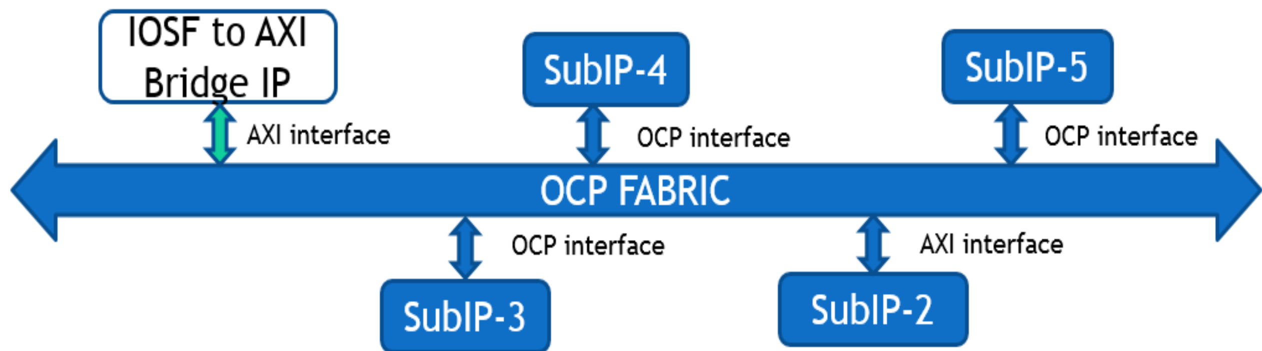


Figure 17. General Arch. Diagram of the IP blocks Interfacing with Fabric

Upstream & Downstream Flow of the AXI to OCP Monitor:-

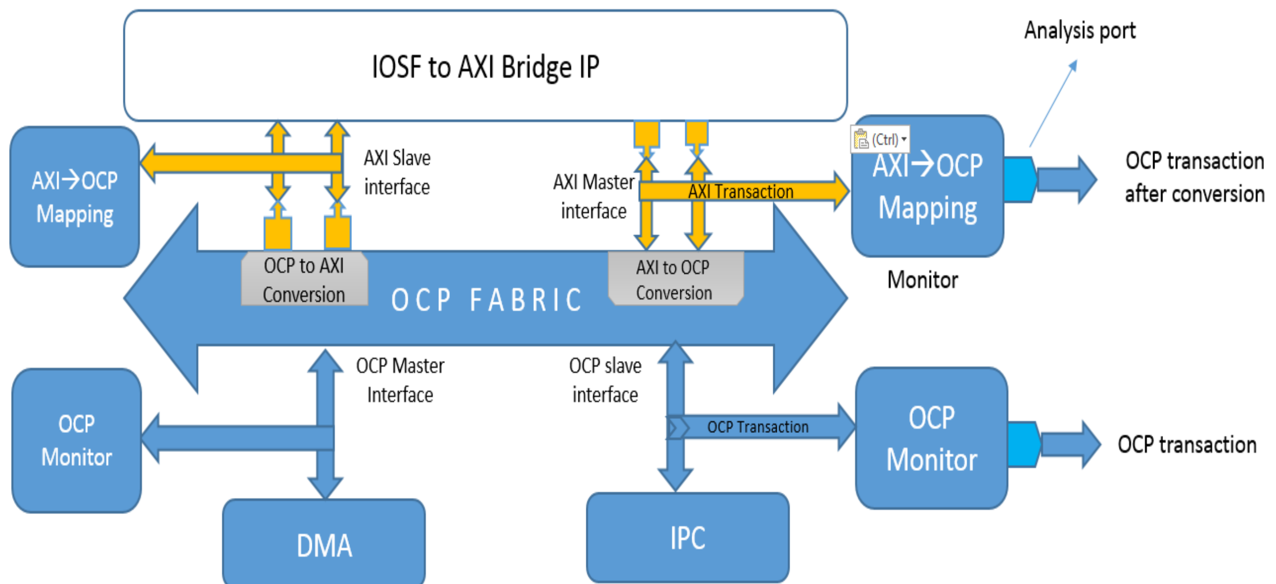


Figure 18. Functional Diagram of the AXI to OCP converter

As shown in the Block diagram Upstream Flow Includes the Upstream Write and Upstream Read transfers. For Upstream transactions IP blocks serves as master quantity whereas bridge acts as a slave, here DMA(master) can send the command to the bridge IP (slave) for Read from Memory location or Write to the Memory location. Same way Downstream Flow includes the downstream read and write transfers. For downstream, Bridge is master and IP blocks of the whole IP is serve as a slave. Here Host would like to access the some of the Sensor data acquired by the IP via Bridge depends on the Use-case. These communication with the IP can be done through the Inter process communication (IPC). IPC controls the flow of the inbound and outbound transfers based on request.

6.6 Retention & Isolation Checking Strategy:-

Nowadays Power targets of a product have equal importance with feature set of a product. Battery life is playing key role in product success.

Some of the common low power techniques are:-

- Clock gating
- Supply Voltage Reduction
- Power switching
- Logic Power gating
  - SW controlled power gating
  - HW Autonomous Power gating etc.

Hardware Autonomous Power gating Technique is one of the fine techniques in which device will be in the power gated state but transparent to the Software. In the Technique portion of the Gated domain logic is retained, rest of the logic will be in the power gated state.

Challenges in Verifying Retention list:-

- Issues with Retention flop list:
  - Missing retention flops [Results in Functional failures]
  - Over retention [No-functional issue but impacts Die size. Retention flop area is 2 x normal flop]
- Propagation of incorrect state after power down exit to a visible checkpoint is not guaranteed even with exhaustive stimulus. Hence all missing retention flops may not yield into test/check failures.
- Debugging functional failure due to missing retention flop is very costly in-terms of time and resources at Gate level verification or on Silicon.

Isolation & Retention Checker Generation Flow: -

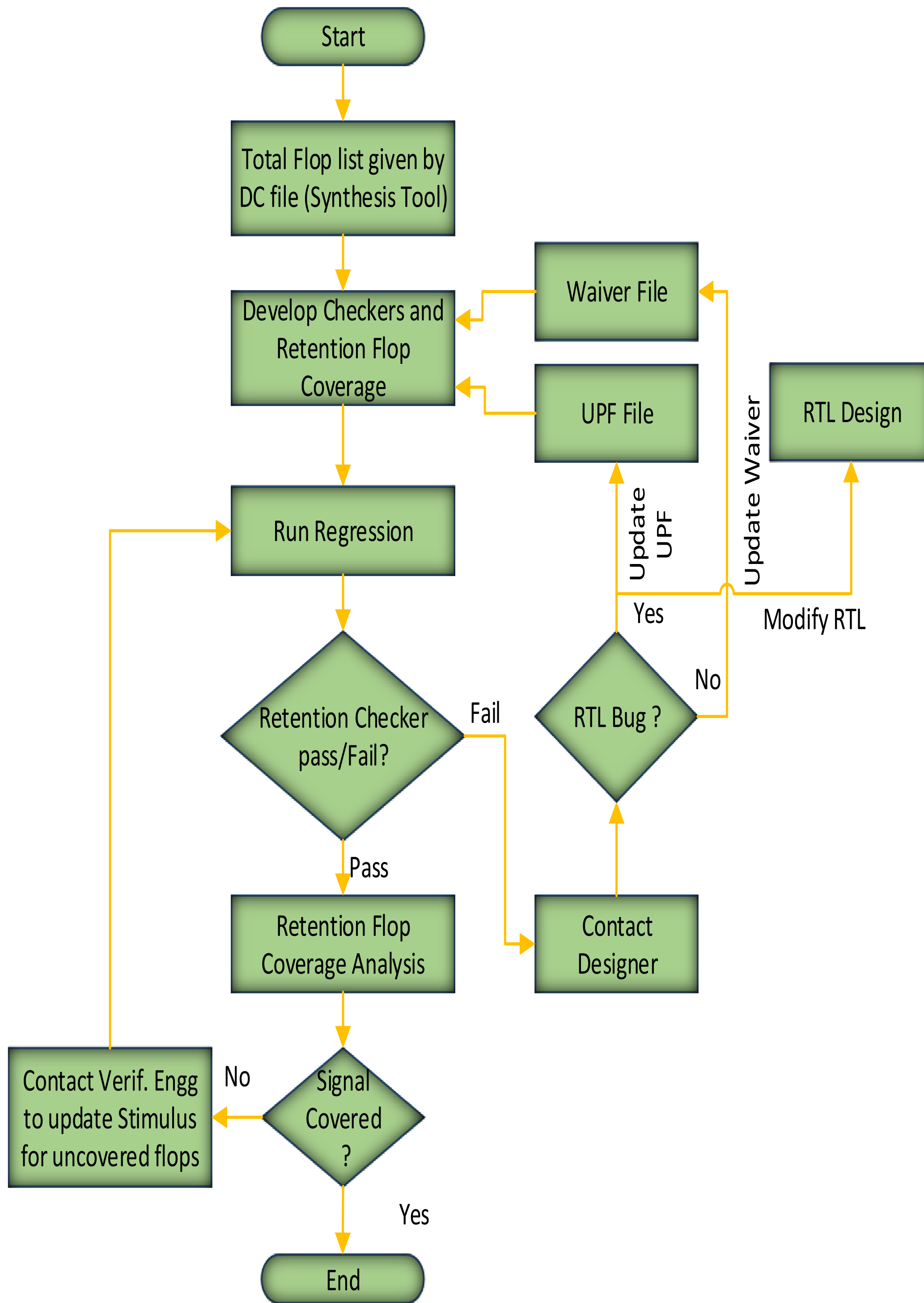


Figure 19. Flow for Isolation and Retention Checker Generation

Chapter 7

Simulation Results & Coverage Results

7.1 Constraint Solver Results

- Compile & simulation Result

```
8 ---------------------------------------------------------------------
9 iteration_start
0 -----------------------------------------------------------------------------
1 Name                         Type                   Size               Value
2 -----------------------------------------------------------------------------
3 <unnamed>                    func_rand_PG_seeds     -                     @4
4   var1                       integral               1                    'b1
5   var2                       integral               2                   'b10
6   var3                       integral               1                    'b1
7   var3                       integral               1                    'b1
8   var4                       integral               1                    'b1
9   var5                       integral               6                   'd19
0   var6                       integral               6                   'd16
1 -----------------------------------------------------------------------------
2 -----------------------------------------------------------------------------
```

Figure 20. Simulation Result generated by VCS-Mx

- Processed Output

```
1 VAR_A: 'b1; VAR_B: 'b01; VAR_C: 'b0; VAR_D[0]: 'b11;VAR_D[1]: 'b0;VAR_D[2]:
   'b1;VAR_D[3]: 'b11;VAR_D[4]: 'b1;VAR_D[5]: 'b01;VAR_D[6]: 'b1;VAR_D[7]: 'b
  0;VAR_D[8]: 'b11;VAR_D[9]: 'b10;VAR_D[10]: 'b01;VAR_D[11]: 'b10;VAR_D[12]:
  'b10;VAR_D[14]: 'b1;VAR_D[15]: 'b11;VAR_D[16]: 'b01;VAR_D[17]: 'b11;VAR_E:
  'b01;VAR_F[0]: 'b0;VAR_F[1]: 'b0;VAR_F[2]: 'b0;VAR_F[3]: 'b0;VAR_F[4]: 'b0;
2 VAR_A: 'b1; VAR_B: 'b01; VAR_C: 'b0; VAR_D[0]: 'b11;VAR_D[1]: 'b0;VAR_D[2]:
   'b1;VAR_D[3]: 'b11;VAR_D[4]: 'b1;VAR_D[5]: 'b01;VAR_D[6]: 'b1;VAR_D[7]: 'b
  0;VAR_D[8]: 'b11;VAR_D[9]: 'b10;VAR_D[10]: 'b01;VAR_D[11]: 'b10;VAR_D[12]:
  'b10;VAR_D[14]: 'b1;VAR_D[15]: 'b11;VAR_D[16]: 'b01;VAR_D[17]: 'b11;VAR_E:
  'b01;VAR_F[0]: 'b0;VAR_F[1]: 'b0;VAR_F[2]: 'b0;VAR_F[3]: 'b0;VAR_F[4]: 'b0;
3 VAR_A: 'b1; VAR_B: 'b01; VAR_C: 'b0; VAR_D[0]: 'b11;VAR_D[1]: 'b0;VAR_D[2]:
   'b1;VAR_D[3]: 'b11;VAR_D[4]: 'b1;VAR_D[5]: 'b01;VAR_D[6]: 'b1;VAR_D[7]: 'b
  0;VAR_D[8]: 'b11;VAR_D[9]: 'b10;VAR_D[10]: 'b01;VAR_D[11]: 'b10;VAR_D[12]:
  'b10;VAR_D[14]: 'b1;VAR_D[15]: 'b11;VAR_D[16]: 'b01;VAR_D[17]: 'b11;VAR_E:
  'b01;VAR_F[0]: 'b0;VAR_F[1]: 'b0;VAR_F[2]: 'b0;VAR_F[3]: 'b0;VAR_F[4]: 'b0;
4 VAR_A: 'b1; VAR_B: 'b01; VAR_C: 'b0; VAR_D[0]: 'b11;VAR_D[1]: 'b0;VAR_D[2]:
   'b1;VAR_D[3]: 'b11;VAR_D[4]: 'b1;VAR_D[5]: 'b01;VAR_D[6]: 'b1;VAR_D[7]: 'b
  0;VAR_D[8]: 'b11;VAR_D[9]: 'b10;VAR_D[10]: 'b01;VAR_D[11]: 'b10;VAR_D[12]:
  'b10;VAR_D[14]: 'b1;VAR_D[15]: 'b11;VAR_D[16]: 'b01;VAR_D[17]: 'b11;VAR_E:
  'b01;VAR_F[0]: 'b0;VAR_F[1]: 'b0;VAR_F[2]: 'b0;VAR_F[3]: 'b0;VAR_F[4]: 'b0;
5 VAR_A: 'b1; VAR_B: 'b01; VAR_C: 'b0; VAR_D[0]: 'b11;VAR_D[1]: 'b0;VAR_D[2]:
   'b1;VAR_D[3]: 'b11;VAR_D[4]: 'b1;VAR_D[5]: 'b01;VAR_D[6]: 'b1;VAR_D[7]: 'b
  0;VAR_D[8]: 'b11;VAR_D[9]: 'b10;VAR_D[10]: 'b01;VAR_D[11]: 'b10;VAR_D[12]:
  'b10;VAR_D[14]: 'b1;VAR_D[15]: 'b11;VAR_D[16]: 'b01;VAR_D[17]: 'b11;VAR_E:
  'b01;VAR_F[0]: 'b0;VAR_F[1]: 'b0;VAR_F[2]: 'b0;VAR_F[3]: 'b0;VAR_F[4]: 'b0;
```

Figure 21. Filtered Output generated by Script

➢ Variable Probability Result

```
prob_calc + (/nfs/iind/disks/mg_disk1465/naishals/perl_task/perl_ex) - GVIM3
File Edit Tools Syntax Buffers Window Help

 1 var1 :-
 2 'b1:87.49%   'b0:12.51%
 3 var2 :-
 4 'b10:44.37%   'b11:44.07%   'b1:11.56%
 5 var3 :-
 6 'b1:35.35%   'b0:64.65%
 7 var4 :-
 8 'b1:80.16%   'b0:19.84%
 9 var5 :-
10 'b1:90.90%   'b0:9.10%
11 var6 :-
12 'd19:7.55%   'd14:7.54%   'd10:8.96%   'd16:7.74%   'd17:7.63%   'd20:7.48%   'd7:1.39%   'd8:1.69%   'd3:1.58%   '
   d12:7.88%   'd13:7.68%   'd15:7.40%   'd6:1.44%   'd11:7.54%   'd18:7.54%   'd4:1.44%   'd0:1.57%   'd9:1.49%   'd5
   :1.64%   'd2:1.23%   'd1:1.59%
13 var7 :-
14 'd16:7.11%   'd19:7.81%   'd17:7.58%   'd10:9.18%   'd6:1.52%   'd13:7.80%   'd18:7.63%   'd7:1.50%   'd20:7.77%
```

Figure 22. Variables which All values & Probability

7.2 Coverage results

➢ URG Coverage Report

Group : coverage_overlap_flows::cov_overlap_flows

SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
69.23	1	100	1	0	64	64	

Summary for Group coverage_overlap_flows::cov_overlap_flows

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	247	76	171	69.23

Variables for Group coverage_overlap_flows::cov_overlap_flows

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST	AUTO BIN MAX	COMMENT
all_flows_cov	247	76	171	69.23	100	1	1	0	

Go to top

Summary for Variable all_flows_cov

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	247	76	171	69.23

Figure 23. URG Coverage Report Format

➢ Tkdiff Result

```
label_simple_wr_burst_wr  1    0    1    100.00  100
label_burst_rd_simple_rd  1    0    1    100.00  100
label_burst_wr_simple_rd  1    1    0    0.00    100
label_burst_rd_simple_wr  1    0    1    100.00  100
label_burst_wr_simple_wr  1    0    1    100.00  100
host_prim_rst_b           2    0    2    100.00  100

Cross Coverage Points : Aon Memory
CROSS                        EXPECTED UNCOVERED COVERED PERCENT GOAL
cross_label_burst_rd_burst_rd   2      0     2    100.00  100
cross_label_burst_rd_burst_wr   2      0     2    100.00  100
cross_label_burst_wr_burst_rd   2      2     0    0.00    100
cross_label_burst_wr_burst_wr   2      0     2    100.00  100
cross_label_simple_rd_burst_rd  2      1     1    50.00   100
cross_label_simple_rd_burst_wr  2      0     2    100.00  100
cross_label_simple_wr_burst_rd  2      0     2    100.00  100
cross_label_simple_wr_burst_wr  2      0     2    100.00  100
cross_label_burst_rd_simple_rd  2      0     2    100.00  100
cross_label_burst_rd_simple_wr  2      0     2    100.00  100
cross_label_burst_wr_simple_rd  2      2     0    0.00    100
cross_label_burst_wr_simple_wr  2      0     2    100.00  100
cross_label_simple_rd_simple_rd 2      0     2    100.00  100
cross_label_simple_rd_simple_wr 2      0     2    100.00  100
cross_label_simple_wr_simple_rd 2      0     2    100.00  100
cross_label_simple_wr_simple_wr 2      0     2    100.00  100
```

```
1883 label_simple_wr_burst_wr  1    0    1    100.00  100
1884 label_burst_rd_simple_rd  1    0    1    100.00  100
1885 label_burst_wr_simple_rd  1    0    1    100.00  100
1886 label_burst_rd_simple_wr  1    0    1    100.00  100
1887 label_burst_wr_simple_wr  1    0    1    100.00  100
1888 host_prim_rst_b           2    0    2    100.00  100
1889
1890 Cross Coverage Points : Aon Memory
1891 CROSS                        EXPECTED UNCOVERED COVERED PERCENT
1892 cross_label_burst_rd_burst_rd   2      0     2    100.00
1893 cross_label_burst_rd_burst_wr   2      0     2    100.00
1894 cross_label_burst_wr_burst_rd   2      0     2    100.00
1895 cross_label_burst_wr_burst_wr   2      0     2    100.00
1896 cross_label_simple_rd_burst_rd  2      0     2    100.00
1897 cross_label_simple_rd_burst_wr  2      0     2    100.00
1898 cross_label_simple_wr_burst_rd  2      0     2    100.00
1899 cross_label_simple_wr_burst_wr  2      0     2    100.00
1900 cross_label_burst_rd_simple_rd  2      0     2    100.00
1901 cross_label_burst_rd_simple_wr  2      0     2    100.00
1902 cross_label_burst_wr_simple_rd  2      0     2    100.00
1903 cross_label_burst_wr_simple_wr  2      0     2    100.00
1904 cross_label_simple_rd_simple_rd 2      0     2    100.00
1905 cross_label_simple_rd_simple_wr 2      0     2    100.00
1906 cross_label_simple_wr_simple_rd 2      0     2    100.00
1907 cross_label_simple_wr_simple_wr 2      0     2    100.00
1908
```

Figure 24. Tkdiff Result (Coverage and Cover group Difference)

7.3 Converter Upstream and Downstream Transaction Waveforms

➢ Signals for the Upstream Write Transactions

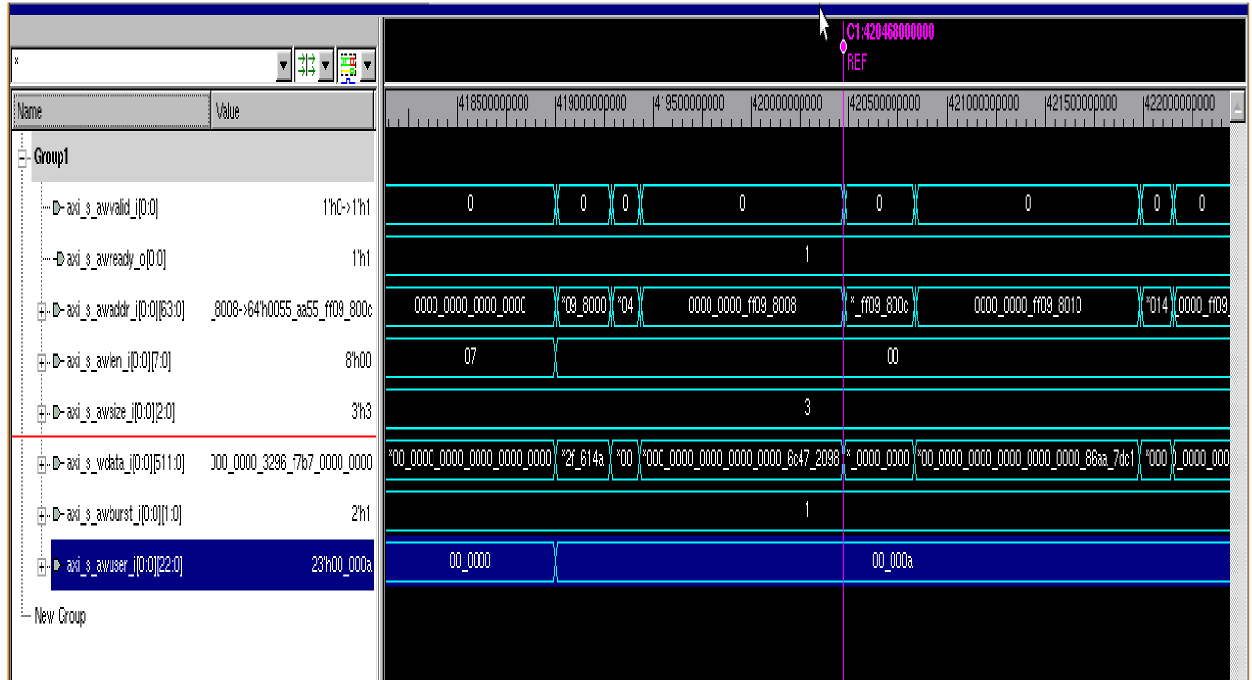


Figure 25. AXI Slave Write Interface Signals

➢ Signals for the Upstream Read Transaction

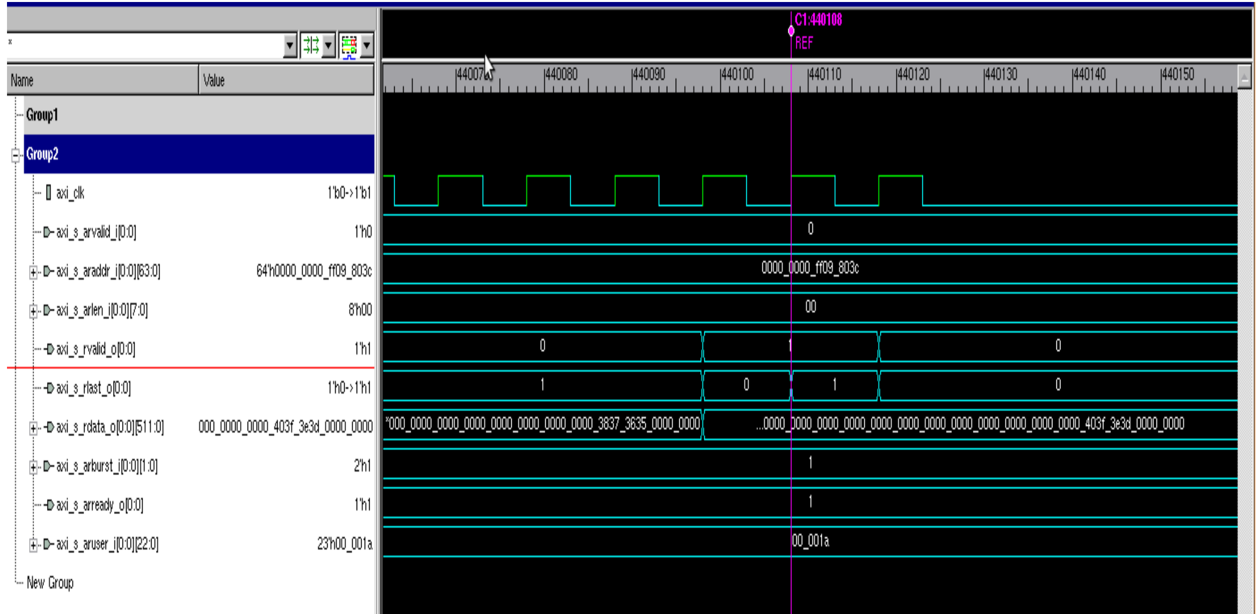


Figure 26. AXI Slave Read Interface Signals

As shown in the above figure for the Upstream read transfer at the end of the completion of the test, though ARLEN == 0, RLAST signal is getting high(protocol compliance violations). This test issue is due to the Burst Narrow transfer happening in the test which is not supported by the bridge IP.

7.4 Retention Verification Results:-

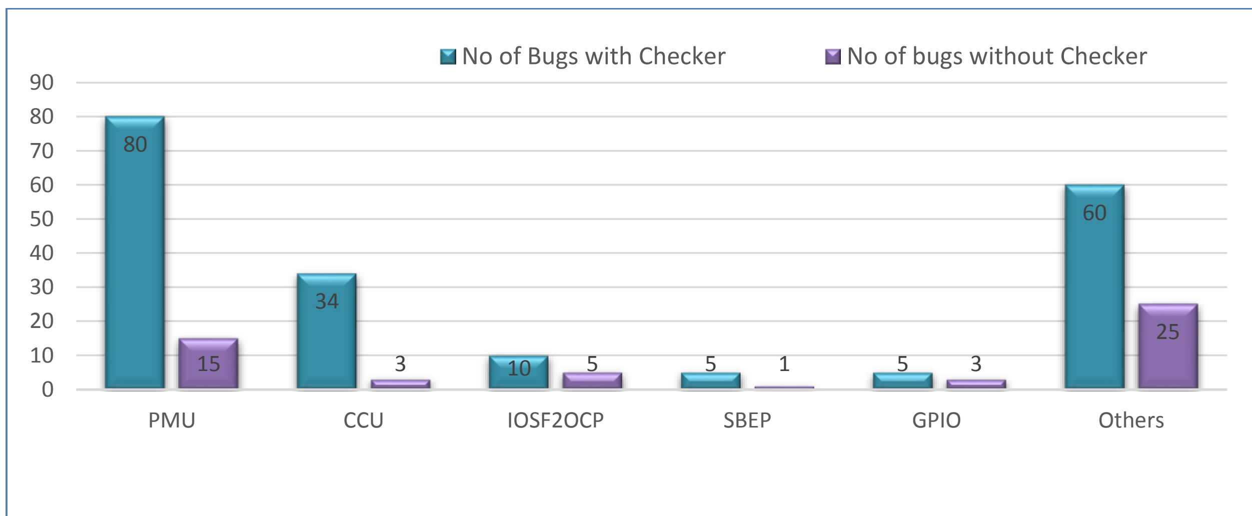


Figure 27. Retention/Isolation bug Distribution for IP Blocks

This methodology is used in at the IP Level and delivered to other SOC Level. Found ~170 missing retention flops and ~20 missing isolation cells.

Conclusion

Due to increase in design complexity and time to market, and need of the low power features by using the various verification strategies will help the verification engineer to verify the complex design in a short period of time.

Various verification and debugging approaches will help in reducing the effort put by the verification engineer in solving the debugging issues. Some of the debugging issues mentioned above can be overcome by having the mentioned debugging infrastructure and tools in the Scalable verification environment.

Also, a significant amount of time spent on verification can be saved by having these various metrics with proper architecture, flexible test benches, smoke & sanity check tests plays a vital role in delivering a quality product on time.

Future Work

As thesis is related to the verification challenges, everyday verification engineers run the regression for the every new design changes (in parallel to the designing process) and plans to verify design changes accordingly. During this process, various challenges encounter by verification engineers to verify and to debug the complex design features and flow.

Using different automation techniques and tool methodologies verification efforts put by the verification engineers can be reduced and TTM (Time to Market) can be achieved as per the planning. Due to the Scalability of the verification environment and components it will be easier to migrate from one projects to other projects which have identical specifications.

In the power aware verification techniques mentioned have the limitations to correlating of the post synthesis flop list to the UPF Flop list and RTL signals. (E.g. complex data structures such as nested arrays used and dealing with it is not a robust method). Currently A project specific lookup table is used to overcome this difficulty And exploring on ways to overcome this limitations by using other tools instead of the post- synthesis report to extract out the flop list and making ot project agnostic.

References

[1] "Challenges in System on Chip Verification" by Noah Bamford, Rekha K Bangalore, Eric Chapman, Hector Chavez, Rajeev Dasari, Yinfang Lin, Edgar Jimenez, IEEE-2006.

[2] **"**Retention and Isolation Checker Generation for Verifying Power Gating Features**"** by Srikanth Kotra, Mangesh Kondalkar , DTTC - 2015

[3] **"**Verification Approach for ASIC Generic IP Functional Verification**"** by Bhavin Patel, IJIRCCE-2013

[4] Perkins, C., "IP Encapsulation within IP", RFC 2003,DOI 10.17487/RFC2003,October 1996

[5] Designware_dw_apb_gpio_book by Synopsys, March 2015

[6] "Coverage Technology User Guide**"** H-2013.06-SP1-12 December 2014.

[7] "System Verilog for Verification : A Guide to learning the Testbench Language Features**"** by Chris Spear, Second Edition, Springer, 2012

[8] OVM User guide by Cadence & Mentor Graphics, March 2010

[9] Intel Specific Bridge IP - High Abstraction Specification(HAS) Documents