Scalable and Modular Verification Environment for the Next Generation IP-Subsystems

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

Masters of Technology
in
Electronics Communication Engineering
(VLSI Design)
By

Rupal Jain
(15MECV21)

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382481
May 2017

Scalable and Modular Verification Environment for the Next Generation IP-Subsystems

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

Masters of Technology
in
Electronics Communication Engineering
(VLSI Design)
By

Rupal Jain
(15MECV21)

Under the Guidance of

Internal Guide
Dr. Usha Mehta
Professor (EC Dept.)
ITNU - Nirma University

External Guide
Panchanathan, Srilatha
Engineering Manager
Intel Technology Pvt. Ltd.

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382481
May 2017

Declaration

This is to certify that

1. 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.
2. Due acknowledgement has been made in the text to all other material used.

Rupal Jain

Certificate

This is to certify that the Major Project entitled **"Scalable & Modular Verification Environment for the Next Generation IP Subsystem "** submitted by **Rupal Jain (15MECV21)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design , NIRMA University, Ahmedabad is the record of work carried out by her 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,have not been submitted to any other university or institution for award of any degree or diploma.

Dr. Usha Mehta
Internal Guide

Dr. N. M. Devashrayee
PG Coordinator (VLSI Design)

Dr. Dilip Kothari
Head, EC Dept.

Dr. Alka Mahajan
Director, IT-NU

Date:

Place: Ahmedabad

Acknowledgement

It gives me immense pleasure to express my gratitude towards people who have been a part of my internship journey. I consider my internship opportunity at Intel as a great milestone in my career. I am highly thankful to Mr. Srinivas P Raghothaman (Project Manager) and Srilatha Panchanathan (Project Guide) for their constant support and encouragement.I am also thankful to all my team members for their needed help. I would like to thank Intel Ltd for providing me this internship opportunity. I am highly indebted to Dr. Niranjan M. Devashrayee and Dr. Usha Mehta for their timely guidance and support. I am thankful to Dr. Dilip Kothari, Head of Electrical Engineering Department for allowing me to undertake this thesis work.
Last, but not the least, I am thankful to my Parents for being a great source of motivation in my life.

Rupal Jain
(15MECV21)

Abstract

Design reuse and verification reuse are important to satisfy time to market requirements. Reuse of verification environment across different designs of the domain improves the verification efficiency. This work will provide the comprehensive approach to create a scalable and modular framework which addresses the key challenges faced, with very little effort across multiple aspects of verification process. Re-usability in this work is been reflected in reuse of verification components. Tools like Perl Template Toolkit were used to because of its fast performance capability. It saved our coding time, effort and coding related errors. The tool named as random test measurement was developed to increase the debugging capabilities for the real time low power requirement problems. It generated the coverage related statistics to calculate average duration of simulation, number of its occurrence(sequences) and the duration of those sequences which can predict the amount of power consumption. Complex design of IP blocks involves multiple complex scenarios such as connectivity break and multiple registers are present in the design and hence, the connectivity check has been taken care by creating an interrupt connectivity checker, and different methods to test each and every register has been performed thoroughly. Hence the verification environment in this project is made scalable and modular so to make testbench components, register sequences and tests, coverage model reusable.

Contents

List of Figures

Chapter 1

Introduction

According to the Moore's law, transistors in an integrated circuit will double after every 18 months. With the increase in design complexity, verification complexity also increases exponentially. As we know, 70% of the total time is devoted in verifying the design, reducing this is essential to catch up time to market. Hence to reduce verification time and complexity, different verification methodologies are used to verify a design like System Verilog, OVM, VMM, UVM. In this project OVM methodology is used. The main intent of this project is to increase re-usabilty. Various techniques and concepts like scalability, modularity etc. are used to increase the re-usability which reduces the tremendous verification effort. The major verification time is spent on functional debugs as well. Hence some automated tools or features are added in the testbench environment so that the debugging becomes faster and easier. During functional verification of IP blocks, connectivity break or data lost issues can occur, so in such case the connectivity between the blocks in the IP should be verified properly. Since registers also plays an important role in design, with the increase in the complexity of design, the number of registers to be verified also increases. Hence verification of registers becomes tedious task as there are thousands of registers in a single IP which needs to verified. Hence various methods are used in this project to to make this work simpler and easier.

1.1 Problem Statement

As Conventional verification methodologies cannot meet time to market demands of verifying the current multimillion gate SoCs. Recent industry research has already raised flags for the lack of breakthroughs which are currently used verification methodologies. Hence the inadequate verification capabilities is the main reason behind restricting the design complexity, rather than the technology. One of the biggest challenge is building a flexible, scalable and modular testbench environment which is essential and hence reduces

the verification effort and time without compromising the quality of the verification block. Automated techniques for reusable verification must be used for more effective results.

1.2 Motivation

To overcome such difficulties, verification environment should focus on the improving the abstraction of verification, improving the reusability and introduction the automation. On the basis of the above three method mentioned, the verification environment of IP is made scalable and modular so as to increase re-usability and reduce verification time, keeping low power in mind. Various debugging techniques, verification method and tools were developed which helped in achieving the goals of this project.

1.3 Thesis Outline

This thesis is divided into eight chapters,
Chapter1 provides the problem which were faced earlier and the main objective of this thesis to overcome them.
Chapter2 discusses the literature survey on the building blocks of an IP and other topics which helped in this project.
Chapter3 provides the description of pre-silicon verification, verification testbench and the methodology used(OVM).
Chapter4 provides why re-usability is essential and what all methods helps in making the environment re-usable.
Chapter5 in this chapter the detailed analysis of project work and their corresponding results are explained, how to make the verification environment effective and techniques to increase the efficiency of debugging time.
Chapter6 concludes the project followed by references

Chapter 2

Literature Survey

To accomplish this work efficiently, various methods like verification of registers by using RAL(Register Abstraction Language) methodology, knowledge of perl template toolkit and system verilog coverage is required. Hence this chapter briefly discuss these methodologies. Various terminologies like scalability, modularity, verification environment, RAL etc. are discussed in this chapter. These techniques were used in this project so as to make the verification environment efficient and reusable.

2.1 Perl Template Toolkit

Perl template toolkit is the collection of perl modules which are flexible, fast, extensible and powerful processing system. It is generally used for generating dynamic and static contents of the web. It provides an easy way to process template files, filling in embedded variable references with their equivalent values.
[1]This module defines an object class whose instances are the compiled template documents. The constructor method ”new()” expects a reference to be a hash array. The ”process()” method is then called on the object and by passing reference to that object as the 1st parameter. This will install any locally defined blocks in the BLOCKS cache. The main BLOCK subroutine is then executed by passing the reference on as a parameter. The text returned from the template subroutine is then returned by the process() method. Hence the template toolkit is a very powerful tool, it not only saved our coding time and effort but also saved from making error. [1]It is fast, flexible and uses a fast parser which compiles templates into Perl code for maximum run-time efficiency. The modules that have toolkit, are highly configurable and the architecture around which they are built is designed to be extensible. Hence if there are many files which needs to be generated or if the same code, this is the best tool as it takes the input and the template, process it and generate outputs.

2.1.1 Methods of Templating a Template

There are two ways to template a file:

1. Default style of template.
2. Perl style of template.

2.1.2 Default Style of Template

By default, template directives are embedded within the character sequences [%... %].
Example:

1. **Perl file:**
   my $vars = {
   name =>'i2c0',
   address =>'56FF4',
   };
   $tt->process('$file,%i2cbase,$output_file) or die $template->error();', $vars);
2. **Template file:**
   [%-FOREACH i2c = i2c_base-%]
   //————————————————————————
   //[%i2c.name%] registers
   //————————————————————————
   #define i2c_I2C 2
   #define [%i2c.name%]_A (0x[%i2c.i2c_address%])
   #define [%i2c.name%]_B ([%i2c.name%]_BASE + 0x000)
   #define [%i2c.name%]_C ([%i2c.name%]_BASE + 0x004)
   #define [%i2c.name%]_D ([%i2c.name%]_BASE + 0x008)
   //————————————————————————
   [%-END-%]
3. **Output file:**
   //————————————————————————-
   //i2c0 registers
   //————————————————————————-
   #define i2c_BASE 2
   #define i2c0_A (0x00084AC)

```
#define          i2c0_B          (i2c0_BASE + 0x000)
#define          i2c0_C          (i2c0_BASE + 0x004)
#define          i2c0_D          (2c0_BASE + 0x008)
//————————————————————————————————————-
```

2.1.3 Perl Style of Template

To use the perl style, set the INTERPOLATE option.

Example:

1. **Perl file:**

   ```
   my $vars = {
   name =>'i2c0',
   address =>'56FF4',
   };
   $tt->process('$file,%i2cbase,$output_file) or die $template->error();', $vars);
   ```

2. **Template file:**

   ```
   [%-FOREACH i2c = i2c_base-%]
   //————————————————————————————————————
   //[%i2c.$name%] registers
   //————————————————————————————————————
   #define          i2c_I2C          2
   #define          [%i2c.$name%]_A          (0x[%i2c.$i2c_address%])
   #define          [%i2c.$name%]_B          ([%i2c.$name%]_BASE + 0x000)
   #define          [%i2c.$name%]_C          ([%i2c.$name%]_BASE + 0x004)
   #define          [%i2c.$name%]_D          ([%i2c.$name%]_BASE + 0x008)
   //————————————————————————————————————-
   [%-END-%]
   ```

3. **Set Interpolate:**

   ```
   use Template;
   my $Template = template->new({
   Path =>'/user/bin/template',
   set_interpolate =>1,
   } or die "$template::error_received \n";
   ```

4. **Output file:**

```
//------------------------------------------------
//i2c0 registers
//------------------------------------------------
#define          i2c_BASE          2
#define          i2c0_A            (0x00084AC)
#define          i2c0_B            (i2c0_BASE + 0x000)
#define          i2c0_C            (i2c0_BASE + 0x004)
#define          i2c0_D            (2c0_BASE + 0x008)
//------------------------------------------------
```

2.2 Functional Coverage

As design complexity increases, to increase the quality and completeness to testing, by using constrained random testing (CRT). This approach reduces the tedious task of writing directed test-cases which were earlier, one for every feature in design. Now if the testbench uses random method to hit corner cases and the test stimulus is randomized then to measure the progress of the design, coverage is used. Coverage is used for measuring the progress to ensure the completeness of the design verification. The coverage tool that is URG(Unified Coverage Report Generator) gathers the information during the simulation or regression and then it post processes to produce the coverage report. With the help of this report, coverage holes can be modified for the existing test cases or we can create new tests to fill these holes. This iterative process continues till the desired coverage level is achieved.

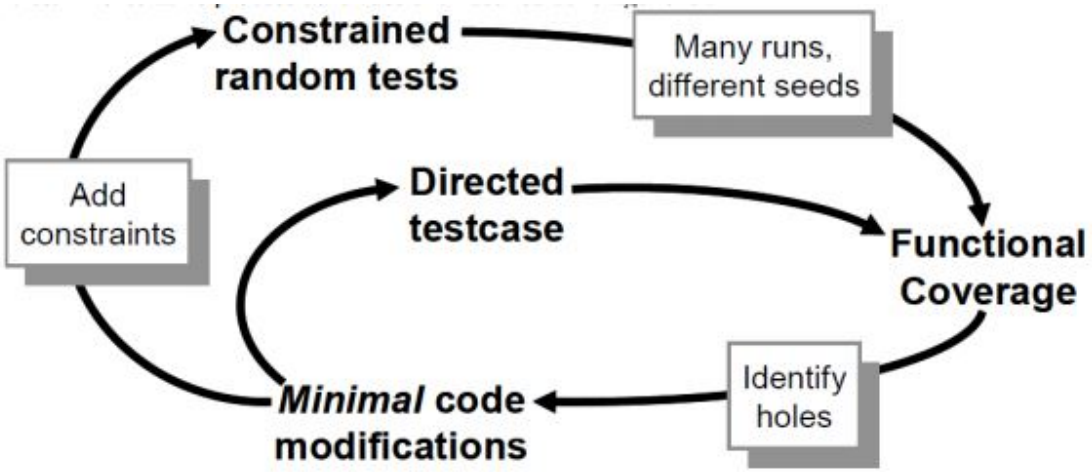


Figure 2.1: Coverage Convergence

Functional coverage is used to measure the functionality achieved for the design. It is also known as specification coverage. To increase the coverage percentage we need to increase the random tests with the random seeds to generate new stimulus. Hence these random tests should run again and again to generate new stimulus. Each simulation has coverage information and this information is merged together by urg. The coverage information can be only calculated if the test passes. If the simulation fails the coverage cannot be calculated as the information is discarded.
Various coverage terminologies are:

1. **Cover Group**
   Covergroup is like a class. Multiple covevergroup can be included in a class. The covergroup like class can only use after new() method. Multiple coverpoints are sampled by the triggering event at the same time.

2. **Cover Points**
   The coverpoint takes the snapshot of the observed values for an expression or single variable.

3. **Bins**
   Bins are used for measuring or counting the functional coverage . The bins hit only when the coverpoint expression matches. Each time the bins are hit, the functional coverage increases.

2.3 Register Verification

With the increase in complexity of design, the number of registers per device increases. These registers are very helpful in communicating information and to perform chip operation. Hence the verification of these registers are very important and should be done properly. As a single IP has thousands of registers to verify such large amount of resister is very tedious. To do this the verification engineer should have knowledge of the specification of IP and the test-bench should be hooked so that the access of these registers becomes easy. As the number of registers are not fixed from project to project, the testbench should be flexible enough. Also the design specifications keeps on changing during the design development. A script is used to generates testbench component for these registers. This register specification given by RTL design engineers is input to this script. This script generates a register abstraction layer file(RAL). This file changes with he change in design. Hence each time the design changes the script is used to generate the updated RAL file.

Registers are mainly classified into categories:

1. Interrupt Registers(maskable and non-maskable).
2. Status Registers.
3. Configuration Registers.
4. Mask Registers.

2.3.1 Features of Register Verification

Features of Register Verification are as follows:

1. RAL has the entire description of the registers. It supports the data structure which store the values of the configuration register. Sequence writes into the RAL registers and the DUT registers at the same time. Hence these RAL registers are known as shadow registers. The register name and the address of the shadow registers is same as that of DUT registers, so it becomes easy to debug.
2. Backdoor Register Access: There are two type of register access. i) Frontdoor access, and ii) Backdoor access. [2]Front door access uses physical bus . To write a value in to DUT registers, it takes some clock cycles in front door access. And writing for thousands of registers is resource consuming. Remember, only one register can be assigned at a time. One cannot make sure that only one method is called at one time. To make sure that only one method is assessing the bus, semaphore is used. In back door access, registers are access directly. In zero time. Accessing to these locations using back door will save simulation time. There should be a switch to control these feature. So after verifying the actual access path of these registers, we can start using back door access. In verilog, using Hierarchy reference to DUT register, we can bypass this path.
3. Shadow registers should always contain default value of the register. This default value of the register should be specified in RAL and should match RTL specification. These default values are read back from shadow register and DUT registers and then compared. The values should match. This check should happen after reset takes place.
4. Tasks for read and write operations to the RTL registers should be made. The read and write to the register happens by the name and the address of the register. For reading and writing in bulk locations looping is used.

5. Every register in the test-bench should contain these information. RAL should have address, offset, width, reset value or default value, access permissions or the attribute of registers, register value, register name as string, description of register.

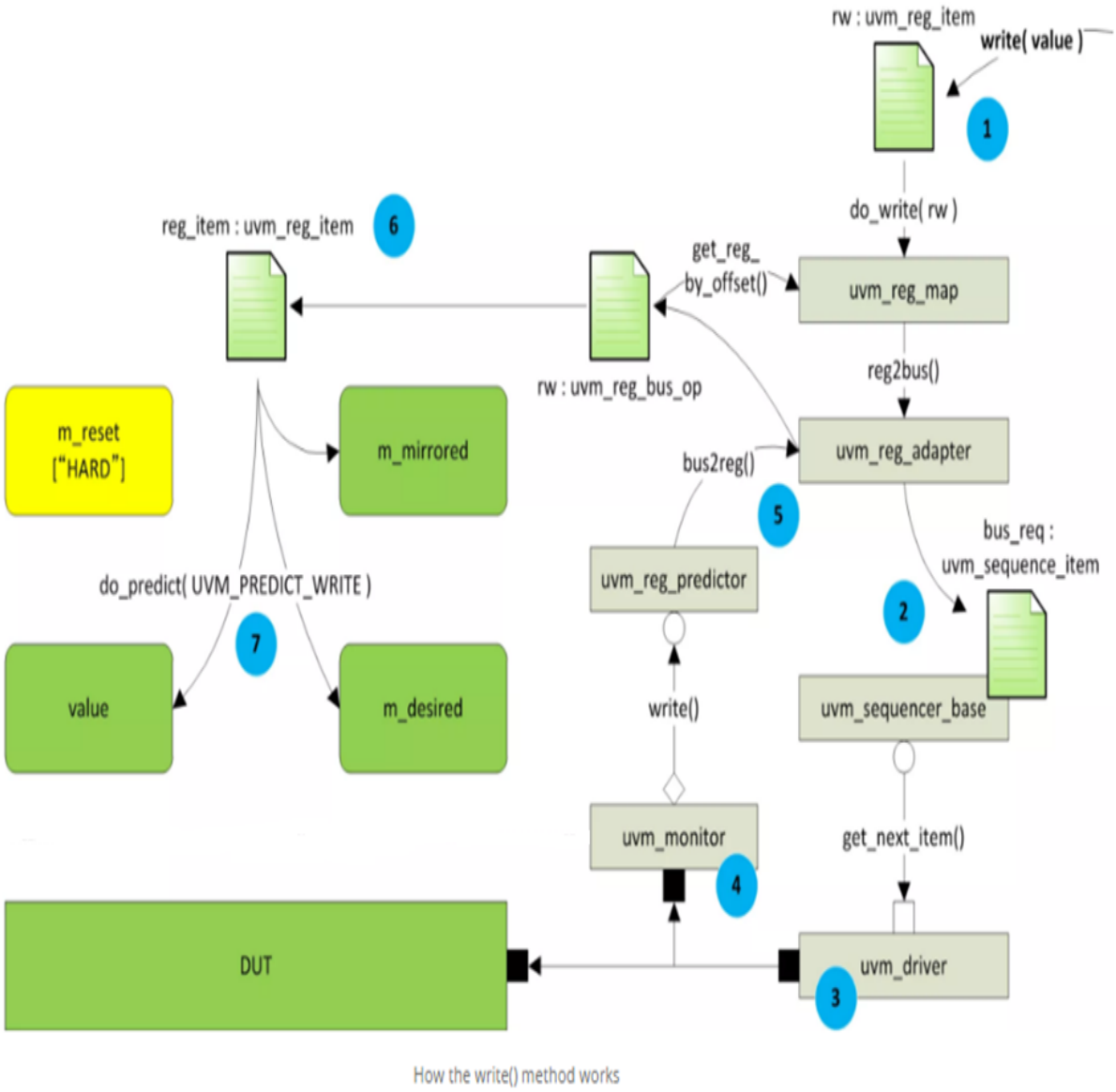


Figure 2.2: Procedure of Register Verification

2.3.2 Method used in Register Verification

Tasks should be developed to accomplish the task of Register Verification.

1. Read task
2. Write task
3. Update and predict task
4. Print task
5. Check and compare function
6. randomize task

Write random task:
Write values in the registers are randomized. But there are some registers which can have some restrictions of the data or the values while writing to the register. Hence these registers are constrained to some values. Some registers values can be any constrained random value. Randomization should be done while writing to the register.

Update task:
Control registers like enable register, interrupt register and its status register can affect the functionality of the RTL, hence update and predict value of the register should be used. When writing to such register, RTL values are compared with the RAL's expected values of the register.Update only happens when predict flag is set to 1.

Check and compare task:
This task compares the value of RTL registers with the RAL registers. These values are checked if they are matching or not. For configuration register predict flag is made zero but for interrupt and its status register predict flag is 1 and hence it is checked with expected value.

Access permission or the attributes:
Each register should have attribute mentioned in RAL as well as RTL. These access type or the permission will decide when only read is allowed and when to write.

Types of attributes:

1. Read/Write(RW).
2. Read Only(RO).
3. Write Only(WO).
4. Read Only, Write is Done by Hardware(RO/V).
5. Clear on Read(RO/C).
6. Writing 1 clears the register(RW/1C).
7. set_config_val sets the value to 1 or 0.
8. Not applicable (NA)

By default the access type of register RW if not defined, but if entire field is not defined the access type is NA. The access type NA is used only in case like when the register is accessible by IP and not by SoC. The procedure for register verification is explained in figure2.2. The RAL is the feature of UVM. Hence a cross module reference module language is used to make RAL file compatible with OVM.

Chapter 3

Pre-Silicon Verification

Due to the advancement in the technologies functional requirements in designs are increasing day by day which leads to more complex designs. More and more logic gates are getting embedded into a chip to achieve performance and functionality of the device.

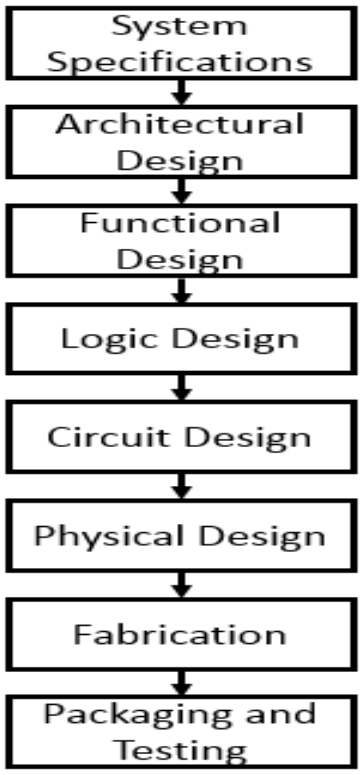


Figure 3.1: VLSI Design Flow

Specifications describe the architecture, functionality and the interface of the circuit to be designed. Then behavioral description of the design with the help of specifications is

created in architectural designs. Then comes functional design in which the RTL description is written using Hardware Description Language. This description is then simulated to check the functionality. Now in logic design, with the help of EDA tools RTL description will be converted into gate level netlist. A gate level netlist is the description of circuits in terms of the gates and the connections between them. These connections should be made by keeping power, timing and area specifications in mind. After logic design the design is converted into transistor level. The physical layout from the transistor level is made, which will be sent to the fabrication. Finally the testing and packaging of the chip is done.

3.1 Basic Testbench Functionality

SystemVerilog is specially developed for the verification purpose. Complexity of testbench increases rapidly to reduce the difficulty during the development, testbench is created from the scratch and the method is adopted in OVM. OVM uses system Verilog as HVL.
Coverage plays very important role in deciding the functional verification. SystemVerilog provides cover group concept in the language. Registers are used to take the samples of the cover groups because the designs is configured using these registers only. The SystemVerilog OVM Class Library provides all building blocks to quickly develop the reusable, test environments and the well constructed verification components. The library consists of macros, base classes, and utilities.
Design is verified by following these steps:

- Generate the stimulus vectors.
- Send the Stimulus to the DUT.
- Monitor the response generated by the DUT.
- Verify the response generated.
- Generate report about the DUT performance.
- Some kind of feedback to show the quality of testbench.

The testbench is built to verify the functionality of the design by generating corner cases and test scenarios. The stimulus is provided by the stimulus generator in the testbench to exercise the DUT through driver. The stimulus from the driver is given to the DUT and the scoreboard. The data from the DUT is collected by the monitor through mailbox. Then the data is compared in the scoreboard.

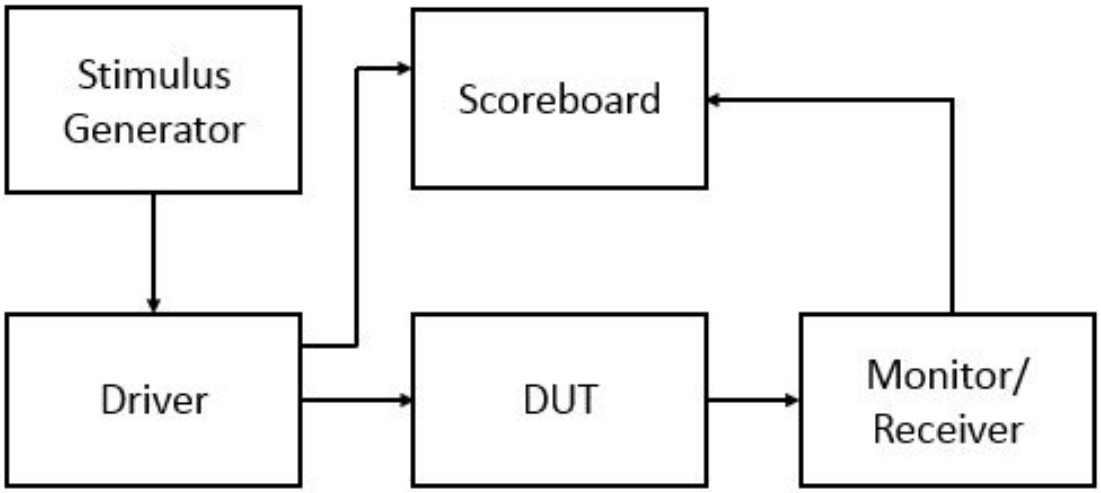


Figure 3.2: Basic Testbench

3.2 Verification Methodology

OVM is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. The letters OVM stand for the Open Verification Methodology. OVM was created by Cadence and Mentor based on existing verification methodologies originating within those two companies. The methodology used in this project is OVM(Open Verification Methodology). OVM is a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level. This could be behavioral, register transfer level, or gate level.

3.2.1 OVM Features

Features of OVM are:

- Verification quality is dramatically improved by constrained random verification and the verification environment becomes more flexible and transparent.
- Accurate feedback(result) is provided by the automated coverage collection report which helps in improving the verification by emphasising the verification plan.
- OVM facilitates verification reuse by using modular verification environment. Reusabilty helps in saving time and effort.

Three C's that is, Checker, coverage and constraints plays important role in verification.

1. **Checker:** Checkers check the functional correctness of the design. Implementation of checker can be done by using system verilog assertions or by using some regular procedural code.

2. **Coverage:** Coverage determines the functional completeness of the verification. It tells us how much goals has been achieved and how much more is required.

3. **Constraints:** Constraints help to shape the stimulus which is to be provided to DUT. It helps in generating interesting corner cases.

3.2.2 OVM Phases

OVM uses predefined phases for testbench. Each phase is determined by its virtual method whose derived component can overridden to incorporate the specific behavior of the component.

1. **build() phase :** This phase is used for constructing and configuring the child's component.

2. **connect() phase :**This phase is used for connecting ports and exports of the testbench components.

3. **end_of_elaboration() phase :** This phase is used for the final modification in the configuration.

4. **start_of_simulation() phase :** This phase is used for printing hierarchical topologies and table.

5. **run() phase :** This is the main phase of the testbench where actual verification by passing the stimulus takes place.

6. **extract() phase :** This phase gathers all the required information.

7. **check() phase :** This phase checks the compared data is matching or not and it also checks for the errors, and fatal.

8. **report() phase :** It is the last phase, it reports the pass/fail of the test.

Only build() method is the only phase which follows top down approach. All other phases follows bottom up approach. The run() phase consumes time as all the verification related functionality are done in this phase. If forking is used, then all he forked off threads are completed first then this phase ends. Except run phase all other phases are of 0 time consuming.

3.2.3 OVM Testbench

The OVM verification components (OVCs) are written in SystemVerilog which is structured as follows:

1. Design under test or DUT.
2. Interface to the DUT.
3. Top level module
   (a) Process to run the test.
   (b) Instantiation of interface.
   (c) Tests which instantiates the verification environment.
   (d) Instantiation of DUT.
4. Verification environment (or testbench)
   (a) Instantiation of driver.
   (b) Transaction of the data item.
   (c) Driver
   (d) Instantiation of sequencer.
   (e) Top-level of verification environment.
   (f) Sequencer or the stimulus generator.
5. Response checking
   (a) Scoreboard
   (b) Monitor

The entire hierarchy of the OVM testbench is defined as follows:

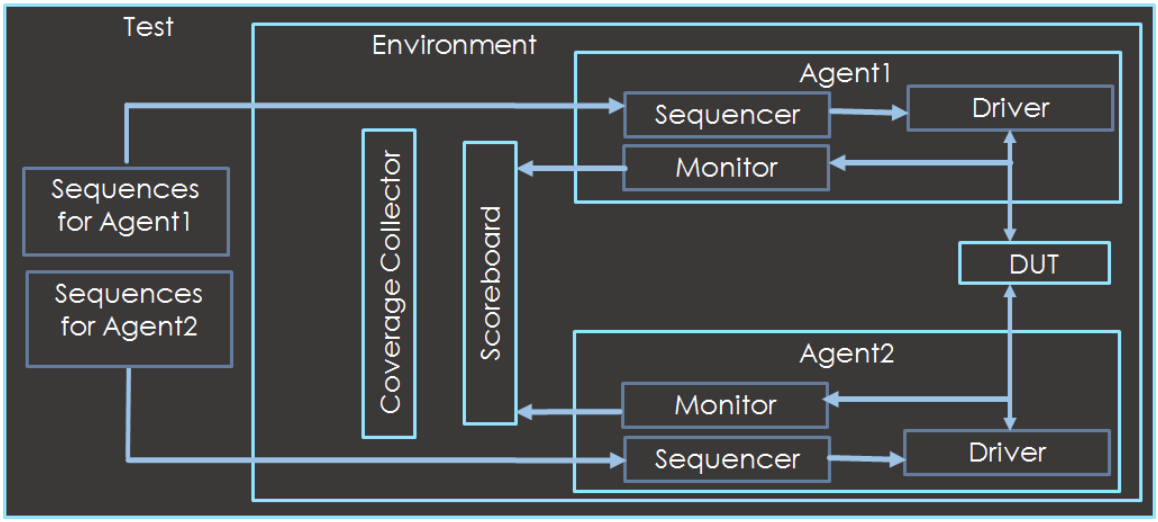


Figure 3.3: OVM Testbench

1. **Sequence item** The sequence library will have one or more sequence items which are used to either define what pin level activity will be generated by the agent or to report on what pin level activity has been observed by the agent.

2. **Sequencer** The role of the sequencer is to route sequence items from a sequence where they are generated to/from a driver.

3. **Driver** The driver is responsible for converting the data inside a series of sequence items into pin level transactions.

4. **Monitor** The monitor observes pin level activity and converts its observations into sequence items or packets which are sent to components such as scoreboards which use them to analyze what is happening in the test-bench.

5. **Agent :** It instantiates Sequencer, Driver and Monitor. There are two types of agents one is active which has sequencer, driver and monitor, the other one is passive which consists of monitor.

6. **Scoreboard :** A scoreboard is an analysis component that checks that the DUT is behaving correctly. A scoreboard will usually compare transactions taken from master and a slave agent.

7. **Coverage Collector :**A functional coverage monitor analysis component contains one or more cover groups which are used to gather functional coverage information relating to what has happened in a test-bench during a test case. A functional coverage monitor is usually specific to a DUT

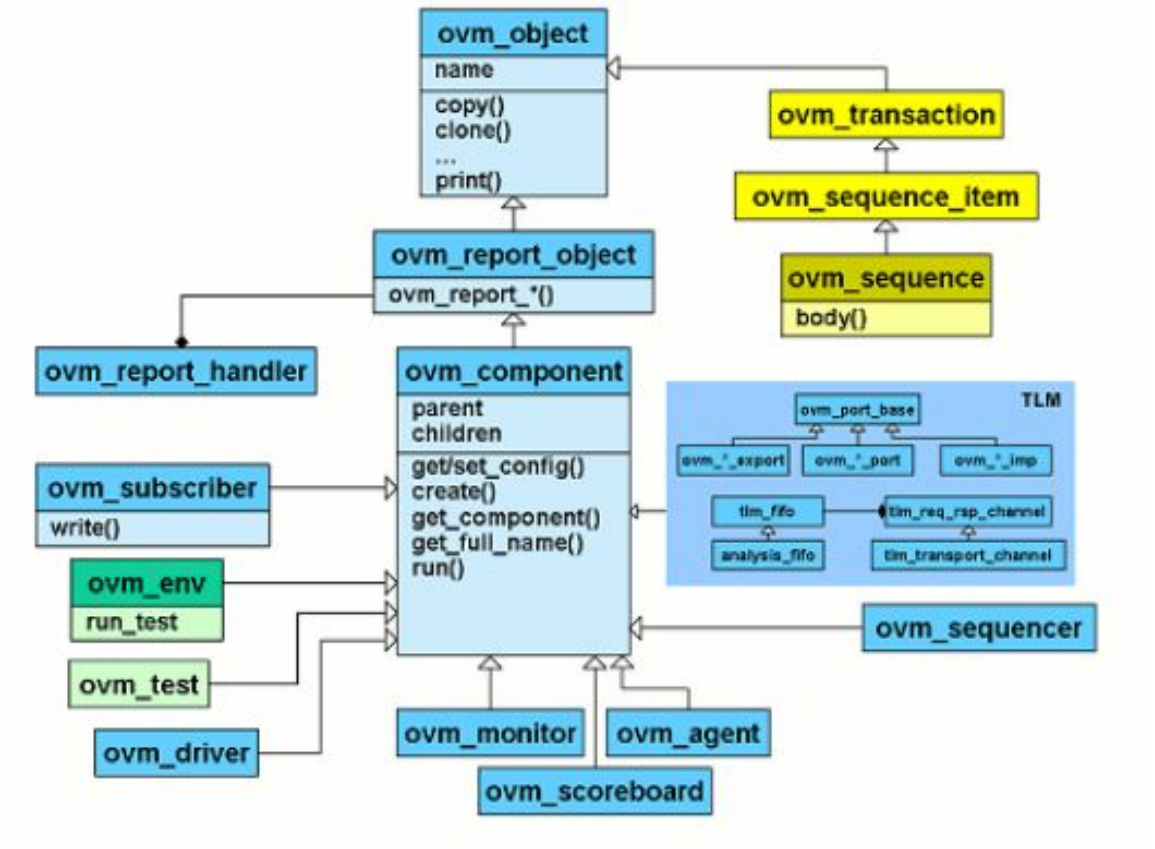


Figure 3.4: OVM Class Hierarchy

8. **Environment :** It instantiates the agent, scoreboard and the coverage collector. It consist of one or more agent. It configures the the type of agent required that is active or passive. All the internal component connection takes place at this hierarchy level.

9. **Test :** Test configures the environment. It instantiates the environment and sequence item.

Chapter 4

Reusable Verification Environment

With the increase in design complexity, verification complexity increases exponentially. Hence to catch up with time to market, the verification environment of the IP should be made re-usable. [3]Verification reuse deals with reusing the existing verification environments, components of verification environments which are developed for the other blocks or designs. It consists of verification code reuse for modules like scoreboard, monitor, data items, bus functional model [BFM], assertions, test case reuse, coverage and simulation script reuse.
[3] Functional verification for IP can be classified into three types:

1. IP block verification
2. IP integration or inter-blocks verification
3. System verification

4.1 Need for Reusable Verification

Verification reuse can drastically reduce the verification time, verification environment build effort, reduce verification risk and improves the product quality. Considering that verification consumes 50% to 80% of the total development effort hence verification reuse brings tremendous benefits to the verification team.
For the reuse of verification components, number of requirements must be met from the perspective of verification component users.
They include:

- The ability to integrate with design which implements the specific interface.
- The ability to integrate with other verification environments.

- Allows multiple instantiations.
- A user friendly interface for writing tests.
- A clear interface for extensions.

The verification process generally depends on the following factors:

- Quality of the testbench and the test plan.
- Robustness of reusable IP blocks.
- Robustness of the verification flow.

4.2 Techniques for Reusability

Different techniques for reusability are used for verification environment. This means that any particular reuse technique or concept may have different importance depending on how we anticipate the code. Hence it is recommended to incorporate as many of the subsequently listed reuse techniques as possible.
Verification code reuse migration can take many paths. Some examples are:

- From basic module level to chip level within the same project.
- From first generation project to the second generation project with modified functionality.
- From project X to project Y and then to project Z, each with possible individual variations as required.

Within these migration paths, there are be different reuse strategies. Some examples of these strategies are:

- Use original code as a templates, and then create new independent copy of files.
- Modify existing code to handle new cases and functionality. Modifications are not backward compatible as methods may have new parameters or fields/structures removed, added or changed.

4.3 Scalable and Modular Verification Environment

Scalable and modular verification environment leads to verification reuse. Setting up constrained random test environment, however, seems like a difficult task, especially when we consider that environment needs to be flexible and scalable.

4.3.1 Verification Environment

The verification environment consists of an automatic verification control system driven by a series of test cases which are usually a set of constraints or a set of scripts containing function calls to class member in object oriented environment. For reusability, the verification environment should be modeled with modular, configurable and completeness.

4.3.2 Modular Verification Environment

[4]Modularity requires specification of the module interfaces. In order to achieve modularity in verification the module interfaces have to provide the right amount of information. If the interfaces provide too much information then they are not helpful in achieving modularity in verification. On the other hand, if they provide too less information then they are not helpful in verifying interesting properties. Modular verification task is necessary for its scalability.

Figure 4.1: Benefits of Modularity

This approach enables the re-usability for the verification components which are associated with the DUT in the SOC and across various SOC where the same IP is being re-used. [5]The main advantage of re-usability is that the IP level test-cases can be easily used as is(or ported) in SoC environment. The scripts are written to convert these tests from IP level to system level. The main use of these tests are to run sanity test list on the module. Sanity tests run on a small area to check the functionality of the module.

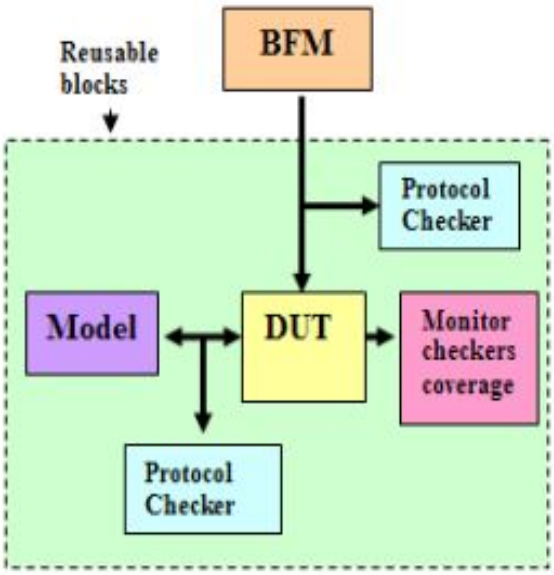


Figure 4.2: The structure of module level verification

Ample amount of time is saved by using these tests. These tests are already present in the SoC environment and hence the verification engineer can focus on other checks.

4.4 Reuse of Verification Components

All the verification components are inherited from ovm_component class. This is the standard class. In figure 4.3, the relation of specific inheritance is clearly seen. [6]In the OVM methodology the environment is instantiated in the test class. The environment instantiate the agent, the scoreboard and the coverage collector. The agent instantiate the sequencer, the driver and the monitor. There are two types of agent, passive and active. In passive agent, only monitor is used while in active driver, monitor and sequencer are used. As OVM provides factory mechanism and hence all the components are registered to the factory. This factory mechanism helps in creating reusable platform. Hence agent can be reused as active and passive where ever required and it is configured in the environment. The following code shows the factory method:

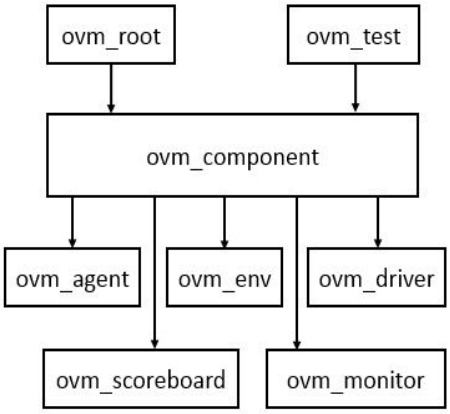


Figure 4.3: OVM Component Class

class class_env extends ovm_env;
‘ovm_component_utils(class_env)
class_master_agent master_a[];
class_slave_agent slave_a[];
function build_phase();
master_a[i] = class_master_agent::type_id::create(inst_name, this);
slave_a[i] = class_slave_agent::type_id::create(ints_name, this);
endfunction
endclass

From the above code we can see that the class of the environment is registered into the factory by using the predefined macro ‘ovm_component_utils. The ovm_master_agent and ovm_slave_agent is the same agent class with different configuration. Then the master and slave agent is created and the memory is allocated for each agent. The is_active enum is used for the configuration of the agent which is defined in environment. This agent is component is reused as slave.

4.4.1 Essentials of Re-usable Verification Components

[7]The obvious benefit of reuse on productivity has to do with reducing or eliminating some of the traditional steps of developing the verification environment. The danger is with

making the assumption that the components which are reused are assumed to be tested well enough. Hence while reusing the verification components various things should be kept in mind. The two major concern of reusing the verification components is adaptability and portability. Adaptability is when the components are reused into the different environment, they should work for which it was intended to work. By portability it means that the language should be supportive. For example, the RAL is an UVM feature but it can be used by OVM test with the help of cross module language.

4.5 Conclusion

Reusing verification code from module level to the SoC level is now the highly expected.[7]The five classes of reasons for reusing verification components and environment are productivity, reliability, consistency, manageability, and standardization. This level of reuse is the most simplest way to reduce effort and spend time on improving the verification plan. If modules are completely reused in different projects or is expected to be reused in new generation subsystems or products then both verification components and the corresponding design modules needs some amount of functional improvisation.

Chapter 5

Detailed Analysis of the Work

This chapter will discuss the work and their corresponding results. The mail goal of this work is to achieve reusable verification environment in terms of verification components reuse and to save verification time. Verification components like interrupt connectivity checker, coverage model, sequences and its test are written which can be further used in different projects. Tools like perl template toolkit is used to obtain the assertions for 128 interrupts coming from different IP's can just be using a single generalized template. The random test measurement tool is created to increase the efficiency of debugging time.

5.1 Perl Template Toolkit

As there are many interrupts which were routed from different IP's to IP processor, the assertions for each IP interrupt was written. Assertions are used to validate the behaviour of a design, whether the design is working correctly or not. In System Verilog, the assertions are of two type:
1). immediate (assert)
2). concurrent (assert property).
These assertions will check the connectivity between the ip and the programmable interrupt controller.

As there were 128 IP's and the assertions were required to be written for the interrupts coming from these IP's and all these assertions are same except for the interrupt name, hierarchical path, clock and RTE number. Hence, with the help of template toolkit assertions for all the interrupts were written. In template file, template of concurrent assertions were used with dynamic variable used are IP name, clock, RTE(redirection table entry) number and the hierarchical path was given in the input file. Then the input file and the template file

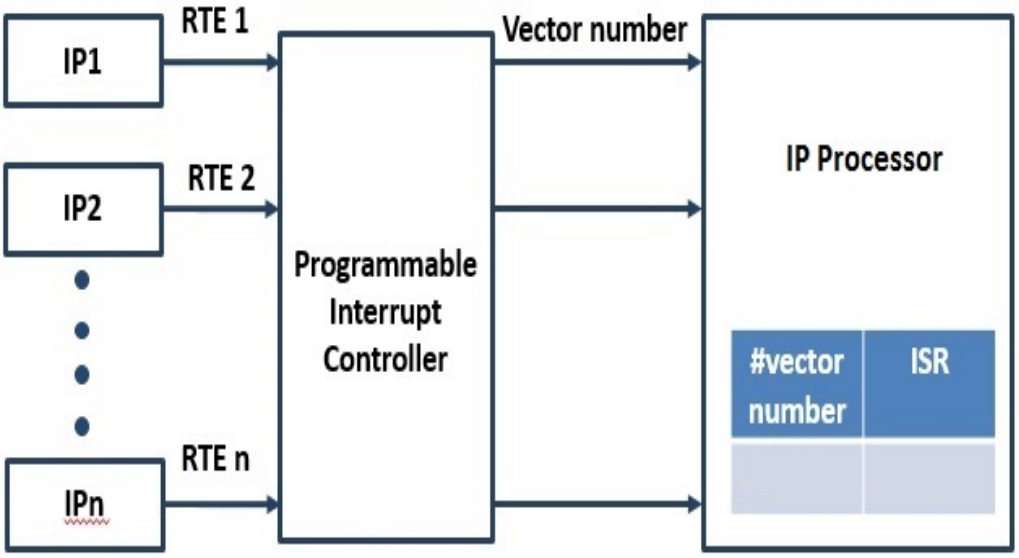


Figure 5.1: Interrupt monitor

are processed by the perl file to generate output .sv(extension) file which has the complete assertions for all the interrupt connectivity.

```
#!usr/bin/perl
use strict;
use warnings;
use Template;
use intr_connect_assert;

my %interrupt_data = intr_connect_assert::show_intr_connect_assert();
my $template = Template->new;
my $input_template = $ARGV[0];
my $output_file = $ARGV[1];

$template->process($input_template,\%interrupt_data,$output_file) or die $te
mplate->error;
```

Figure 5.2: Perl File for Assertions

```
[%-FOREACH interrupt = interrupt_data-%]
//-----------------------------------------------------------------------
//[%interrupt.interrupt_name%] INTR RTE [%interrupt.RTE_number%]
//-----------------------------------------------------------------------
property [%interrupt.interrupt_name%]_int_connect;
@(posedge [%clock%]) disable iff(!rst)

 ($rose ([%interrupt.hirarical_path%])&& (rte_mask_[[%interrupt.RTE_number
%]] == 1'b0)) |-> ##[1:$] $rose(vector_num[[%interrupt.RTE_number%]]) ;

 endproperty

[%interrupt.interrupt_name%]_int_connect_cov: assert property ([%interrupt
.interrupt_name%]_int_connect)
//$display("Property [%interrupt.interrupt_name%]_int_connect succeeded\n"
);
else
$display("Property [%interrupt.interrupt_name%]_int_connect failed\n");

[%interrupt.interrupt_name%]_int_cov:  cover property([%interrupt.interrup
t_name%]_int_connect);
[%END%]
```

Figure 5.3: Template File for Assertions

```
//-----------------------------------------------------------------------
//ip0 INTR RTE 0
//-----------------------------------------------------------------------
property ip0_int_connect;
@(posedge clk) disable iff(!rst)

 ($rose (ip_path)&& (rte_number[0] == 1'b0)) |-> ##[1:$] $rose(vector_number[0]) ;

 endproperty

ip0_int_connect_cov: assert property (ip0_int_connect)
//$display("Property ip0_int_connect succeeded\n");
else
$display("Property ip0_int_connect failed\n");

ip0_int_cov:  cover property(ip0_int_connect);
```

Figure 5.4: Output Assertions File

5.2 Random Test Measurement

[8]Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

– Brian W. Kernighan.

Hence timing efficiency of debugging has been increased by creating a random test measurement tool.

5.2.1 Power Management of Random Sequences

The IP on which the work has been carried out enable a ”always-on-always-sensing” feature. Since the IP is always on and always sensing the battery or power used will be high, hence some techniques are used to reduce this power. Now while verifying such IP random traffics are generated to check whether IP is going under power gated state or sleep state etc. Various IP's sends traffic to verify the entire device hence it takes large simulation time. Simulation time depends on the amount of time each IP is kept awake by the traffic sent. Longer the thread(sequence) is the simulation time is more. And hence it was very difficult to figure out which IP is taking was taking longer time.

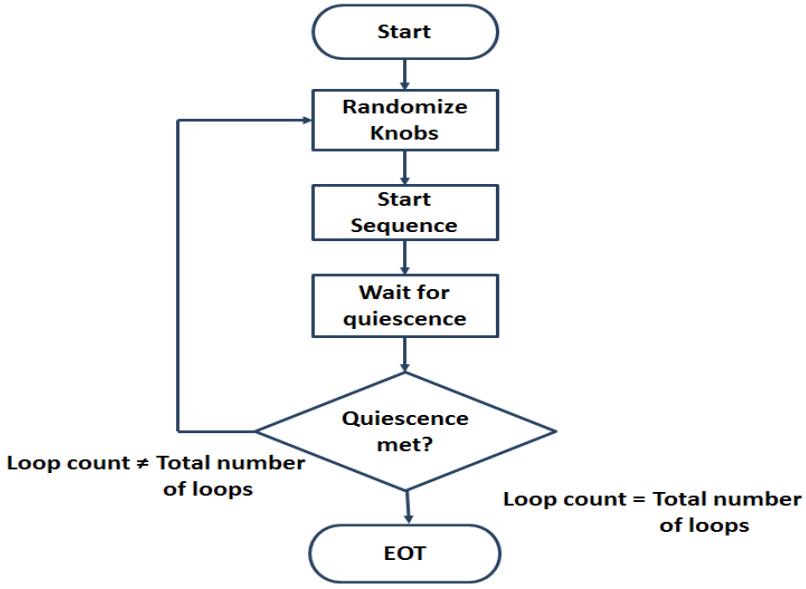


Figure 5.5: Flow Chart of Random Sequences

Figure 4.1 shows the working of the tool, it takes the sequences(random traffic) by runningrandom seeds of tests. Then this sequence is randomized and the triggering of sequencestakes place and then it waits for the sequence to end. After that it checks for the

presence ofany other sequence. If yes then it repeats the loop otherwise the end of test state is called.
To increase the timing efficiency of debugging by creating a tool known as random test measurement. It performs the following tasks:

1. It gives the number of IP's which are active.
2. It checks the concurrency of the occurrence of threads, i.e, how many threads started parallel.
3. Time taken by each thread

5.2.1.1 Simulation Result

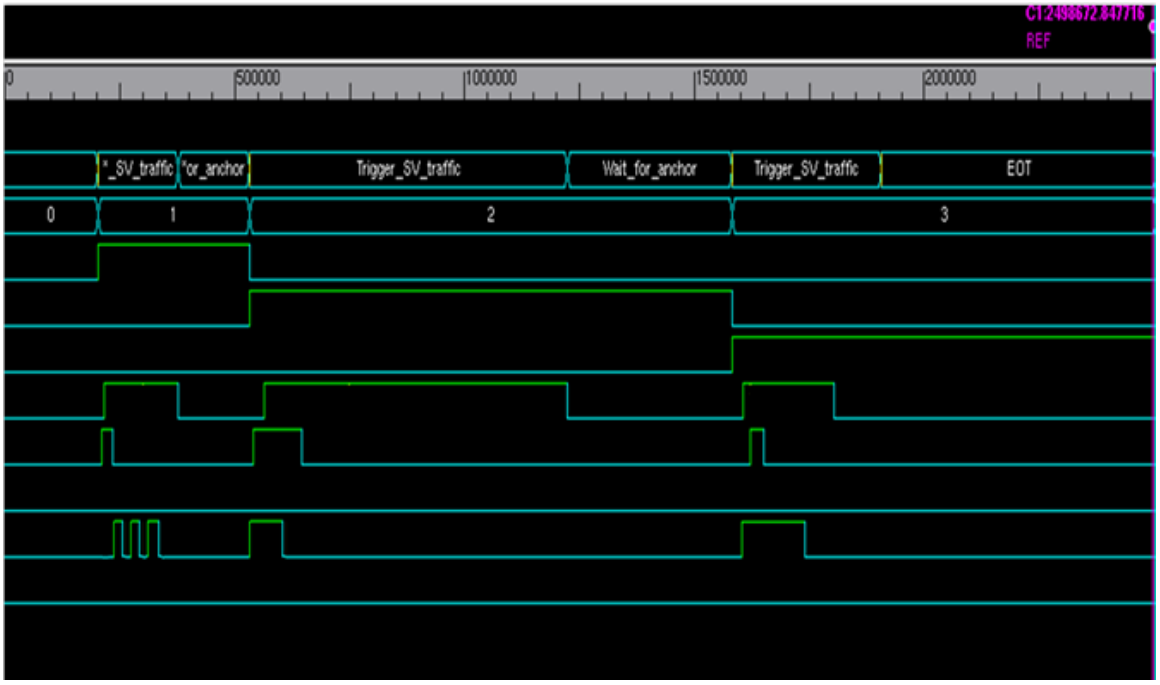


Figure 5.6: Results for Duration of Sequences

Here the fsm have three loops which defines, when SoC is in power off state then the time taken by the IP to go under power off. And each sub loop will contain multiple threads from each IP.

5.2.2 Coverage for Random Test Measurement

The random test measurement tool can not provide enough information with signal simulation as it is used to calculate information from random tests. Hence for multiple seeds the

result will be different. Hence coverage collector is used to calculate average information for all the seeds.
The coverage collector for random tests measurement gives three information. It calculates:

1. duration of each loop
2. total number of threads(count) in each loop.
3. duration of each thread.

5.2.2.1 Coverage Reports

Summary for Variable duration_of_loop1

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	9	2	7	77.78

User Defined Bins for duration_of_loop1

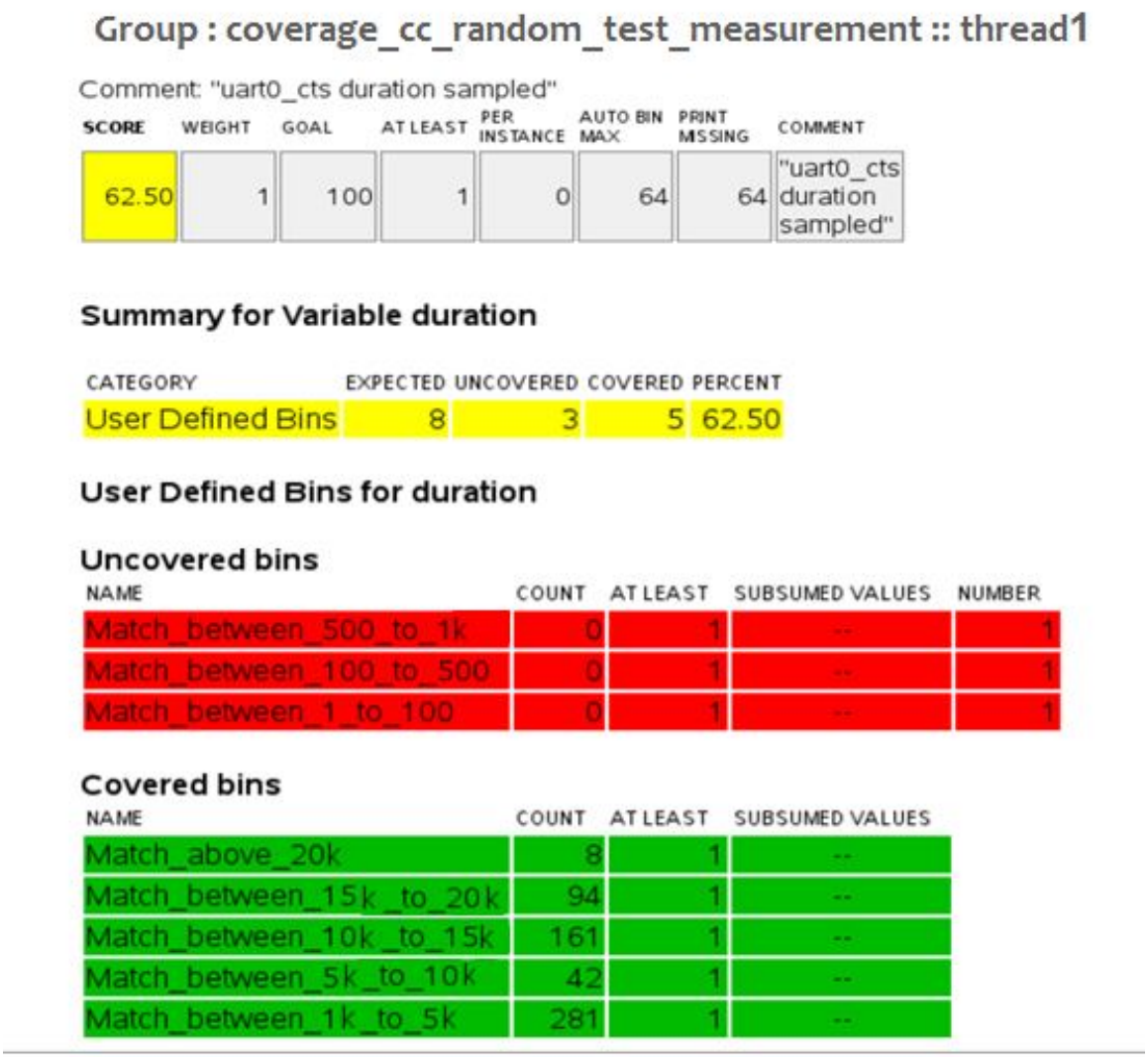
Uncovered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES	NUMBER
Match_above_300k	0	1	--	1
Match_in_5k	0	1	--	1

Covered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES
Match_in_300k	2	1	--
Match_in_240k	4	1	--
Match_in_180k	14	1	--
Match_in_120k	69	1	--
Match_in_60k	82	1	--
Match_in_20k	18	1	--
Match_in_10k	10	1	--

Figure 5.7: Coverage Report for Loop Duration



Group : coverage_cc_random_test_measurement :: thread1

Comment: "uart0_cts duration sampled"

SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
62.50	1	100	1	0	64	64	"uart0_cts duration sampled"

Summary for Variable duration

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	8	3	5	62.50

User Defined Bins for duration

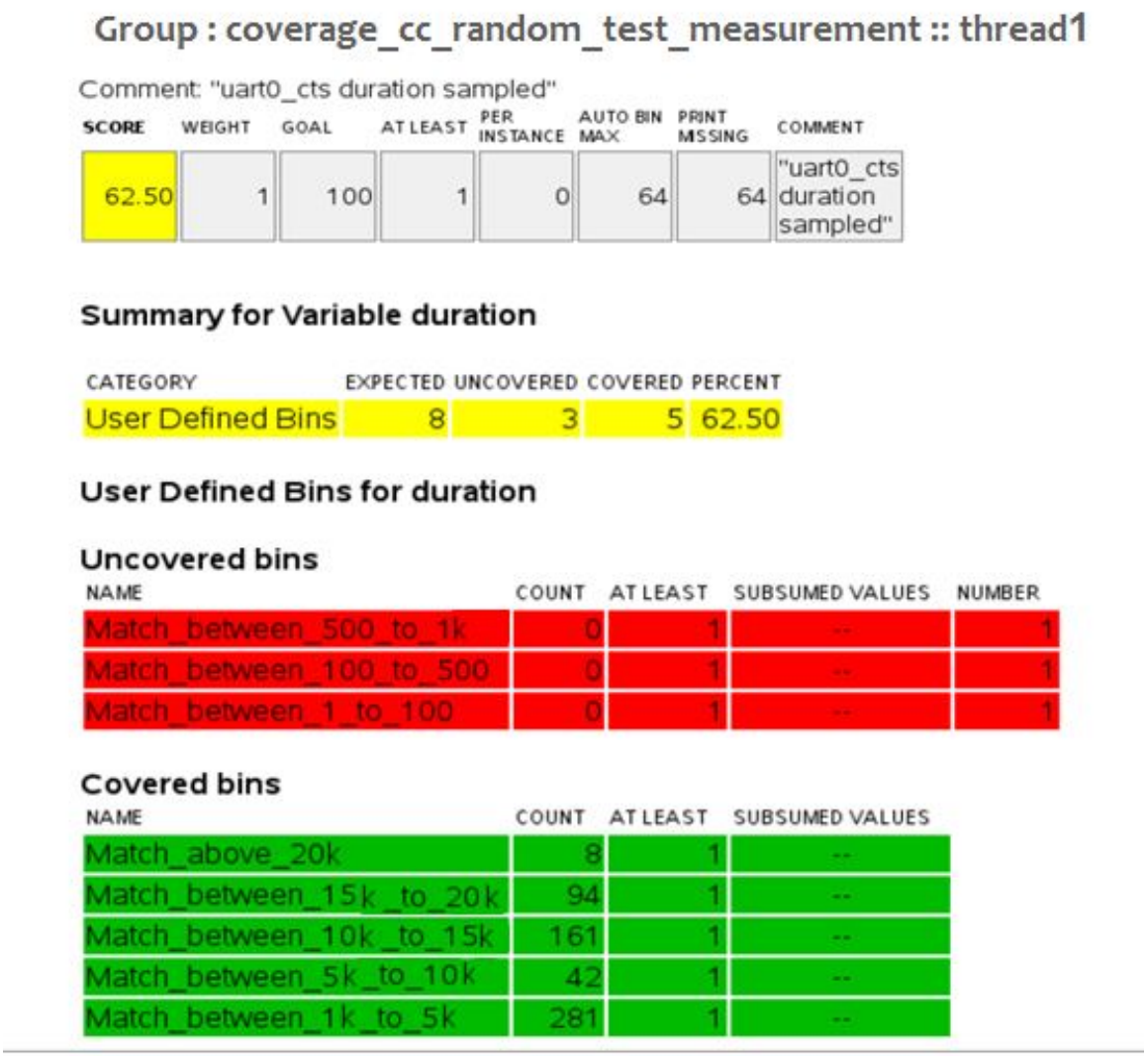
Uncovered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES	NUMBER
Match_between_500_to_1k	0	1	--	1
Match_between_100_to_500	0	1	--	1
Match_between_1_to_100	0	1	--	1

Covered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES
Match_above_20k	8	1	--
Match_between_15k_to_20k	94	1	--
Match_between_10k_to_15k	161	1	--
Match_between_5k_to_10k	42	1	--
Match_between_1k_to_5k	281	1	--

Figure 5.8: Coverage Report for the Thread Count of a Sequence



Group : coverage_cc_random_test_measurement :: thread1

Comment: "uart0_cts duration sampled"

SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
62.50	1	100	1	0	64	64	"uart0_cts duration sampled"

Summary for Variable duration

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	8	3	5	62.50

User Defined Bins for duration

Uncovered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES	NUMBER
Match_between_500_to_1k	0	1	--	1
Match_between_100_to_500	0	1	--	1
Match_between_1_to_100	0	1	--	1

Covered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES
Match_above_20k	8	1	--
Match_between_15k_to_20k	94	1	--
Match_between_10k_to_15k	161	1	--
Match_between_5k_to_10k	42	1	--
Match_between_1k_to_5k	281	1	--

Figure 5.9: Coverage Report for the Duration of Uart Sequence

5.3 Interrupt Connectivity Checker

Connectivity checker, as the name suggest is used to check the connectivity between the source and the destination. The source will be the periphery of different IP blocks. There are four destination path so, the destination for the single IP will be passed through the de-multiplexer and hence to the desired path. The de-multiplexer will have 4-bit selection logic to select the desired path. Based on this selection logic the desired path is selected. This checker is scalable, as it can check the interrupt connectivity for any number of IP's. There are three methods to check the connectivity: assertions, objections and with the help system verilog procedural code. In this project, objection based and the procedural method were used to check the connectivity of the of the IP block.

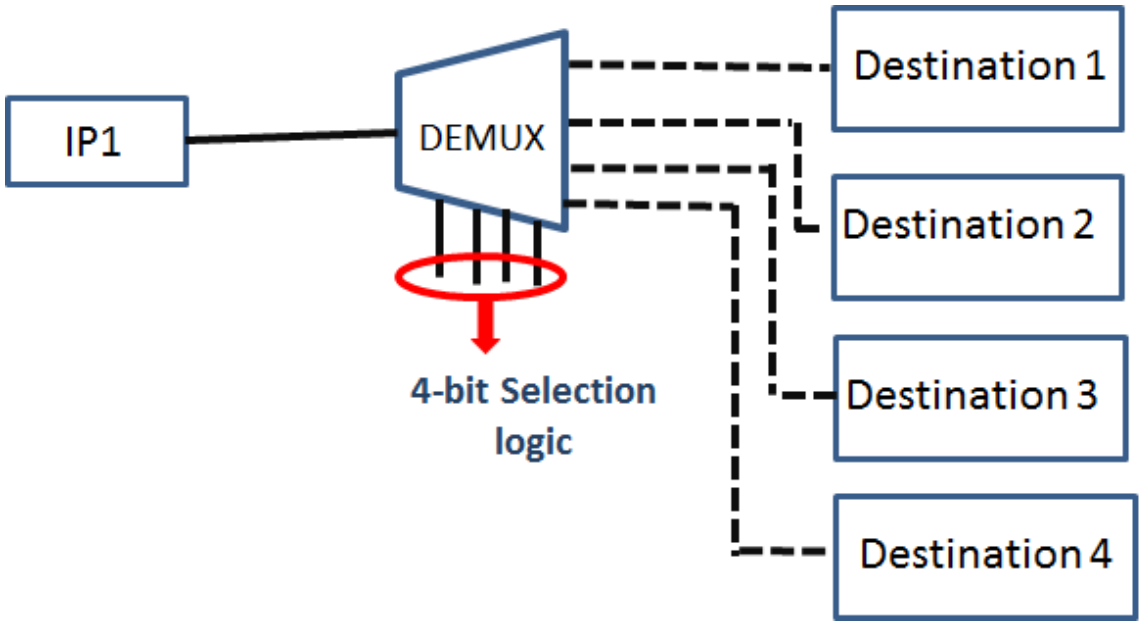


Figure 5.10: Functional Block of Interrupt Connectivity Checker

Interrupt connectivity checker should perform four checks:

1. **Check1:** The interrupt signal at the peripheral should match with the signal at the destination.
2. **Check2:** The interrupt signal is serviced by the ISR or not.
3. **Check3:** The unconnected interrupt pins should not toggle.

4. **Check4:** There are many interrupts which are ored and are given to the mux logic. Hence assertion based check is performed to check the interrupt connectivity from IP peripheral to the or gate logic.

5.3.0.1 Objection Based Check

1. **Check1:** In objection based method, the checker raises the objection at the periphery of IP, on the positive edge(and negative edge) of interrupt, waits for a clock and then drops the objection at the positive level triggered(and negative level triggered) at the destination.

2. **Check2:** To check whether the interrupt is serviced or not, the objection is raised on the posed of interrupt and the objection is dropped at the negative edge of interrupt.

3. **Check3:** It checks for the unconnected pin status.

This method of checking was not correct as it skipped many bugs. So the other way of implementation was procedural check.

5.3.0.2 Procedure Check

By using this method the checker performed all four checks. It used if..else and for condition. All checks are same except for the check1. The check1 checks for all interrupts, that source interrupt(IP level interrupt) should be equal to destination interrupt. The destination is chosen by mux logic.

5.4 Interrupt Coverage

Coverage provides the measure of the functional completeness of the verification and it determines when the set goals of the verification plan is been meet. Interrupt coverage measures the quality of the test which generates an interrupt that are routed to various destinations, like fabric, SoC, IP processor.

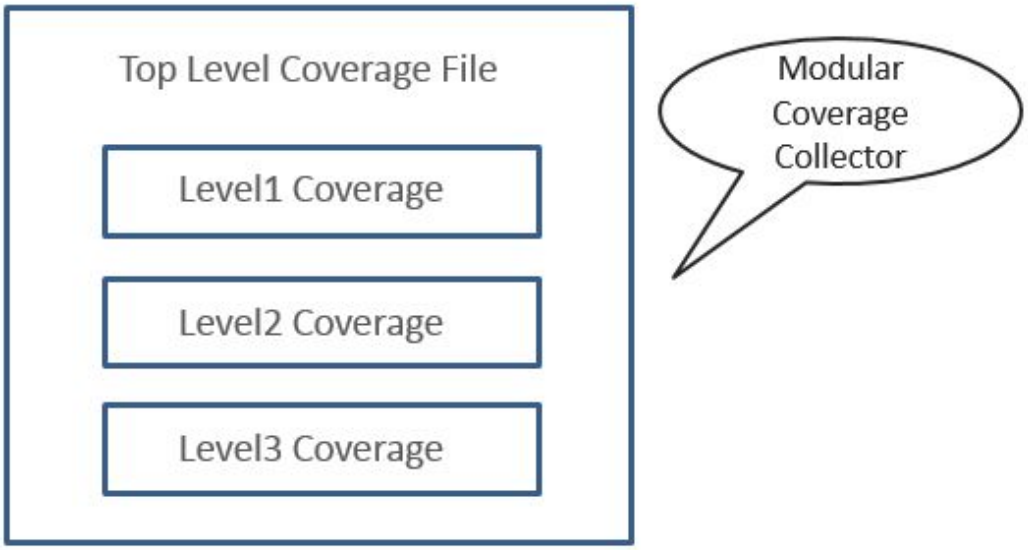
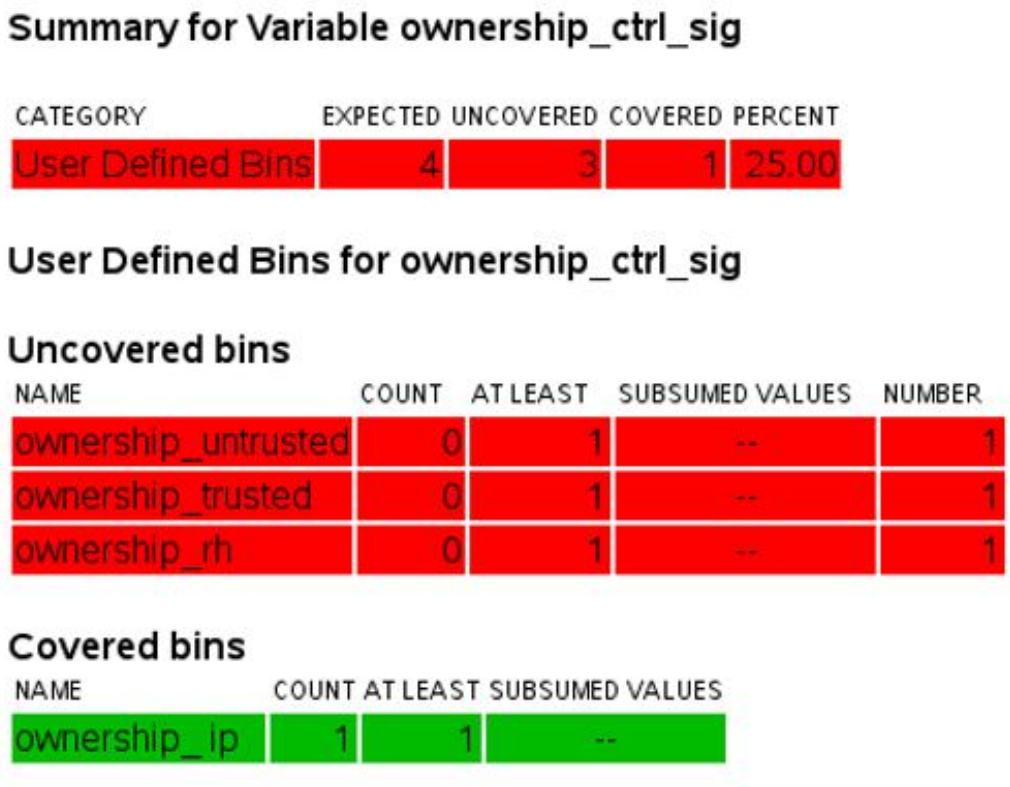


Figure 5.11: Functional Block of Interrupt Coverage

As shown in the Figure 5.11, coverage collector is made modular such that it can be re-used as and when required by different project. It consist of top file which instantiate three sub level coverage collector modules: Levl1, level2 and level3 coverage modules.

Level1 Coverage determines every IP interrupt crossed with the interrupt routing (selection logic of the demux). It stress the demux logic. It checks the stimulus generated by the tests are able to route the interrupt to the proper destination. If all the interrupts are covered by the tests, 100% coverage can be achieved The covergroup is sampled at the posedge of each interrupt which makes it scalable. And the destination is kept as bins, hence when posedge of interrupt coincides with the routing logic bin count is increased. Level2 coverage determines the coverage for the concurrency of occurrence of interrupt at same time. It stress the bridge to measure the efficiency of fabric by sending one interrupt and multiple interrupt.

In level3 coverage counters at the destinations are used to count number of interrupts are sent to the particular destination. It takes the snapshot whenever a counter changes by sampling it for 3 “destination”. It basically measures the requirement of interrupt to be sent at the destination.



Summary for Variable ownership_ctrl_sig

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	4	3	1	25.00

User Defined Bins for ownership_ctrl_sig

Uncovered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES	NUMBER
ownership_untrusted	0	1	--	1
ownership_trusted	0	1	--	1
ownership_rh	0	1	--	1

Covered bins

NAME	COUNT	AT LEAST	SUBSUMED VALUES
ownership_ip	1	1	--

Figure 5.12: Interrupt Coverage for Level1, Interrupt routed to IP Processor

5.5 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. This register description cannot be directly used as it is not compatible with the testbench environment. Hence it is converted into RAL(Register Abstraction Layer) by using a script, which can be directly used by the testbench.

The ovm based test is created, which contains ovm phases.

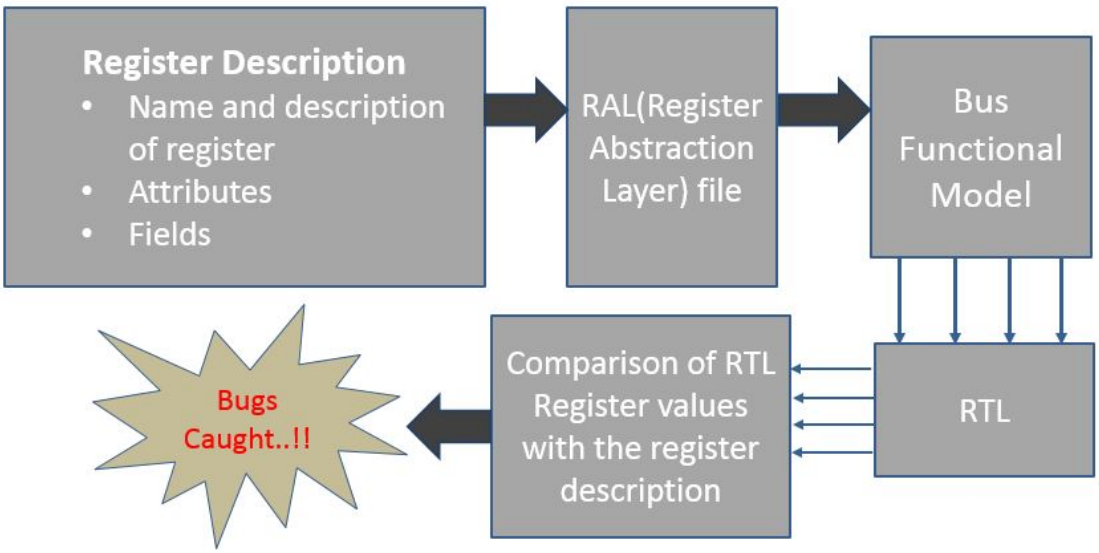


Figure 5.13: Block Diagram of Register Verification

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 100microsecond. This drain out time should be selected carefully because this determines the amount of 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, filed 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 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 filed like enable bit, reset bits, interrupt registers or interrupt bits are masked. Now the task which contains read write function will be called. The three checks are performed:

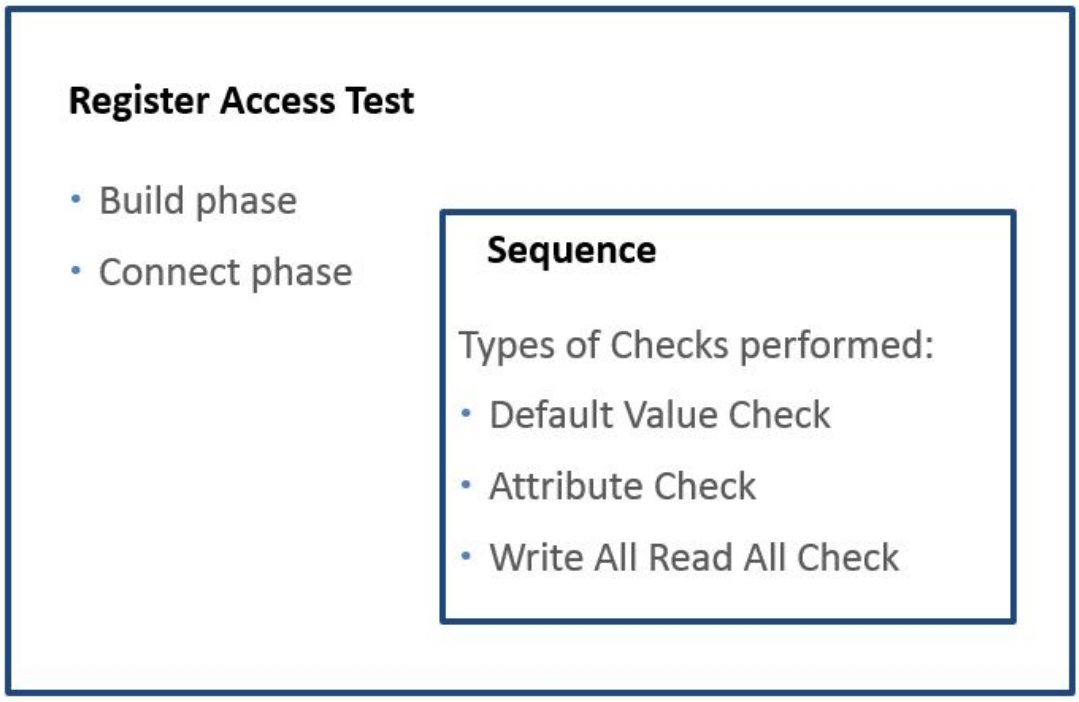


Figure 5.14: Block Diagram of Register Verification

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 check only the register address is required. Then the read and compare happens. If the value compare does not matches 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 will be treated as 0. Here when we write to 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.

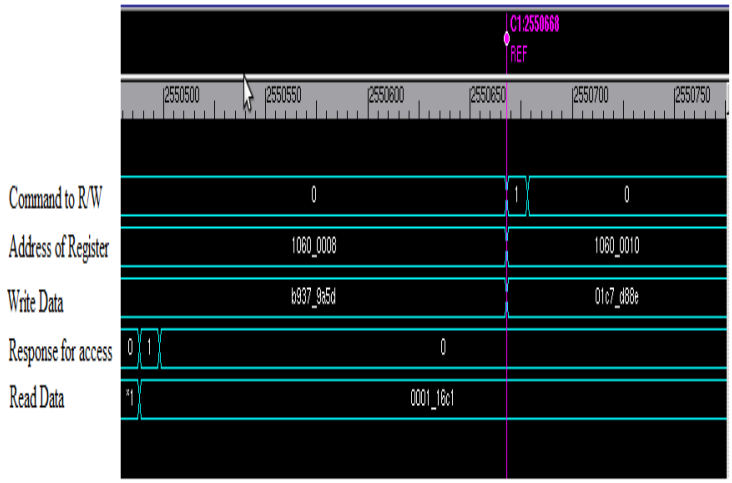


Figure 5.15: Result of Register Verification

5.5.1 Results and Waveforms

From the above figure it can be seen that there is error while testing register with address 1060000010.
The error from the log is:
OVM_ERROR (path_of_flie)(1096) @ 2551428: reporter [PMU_INTR] RAL¿ [Attribute] - [Attr=RW] Read Data Mismatch: File =map_pmu_config_i, Register_name=PMU_GENERAL_INTR[31:1 Field_name=PMU_INTR, Access =sequencer_type, RTL VALUE=0000000000000007, DESIRED=00000000000001c7

OVM_INFO pmu_sequence_file.sv(97) @ 2551428: reporter [pmu_seq] :: :: reg: PMU_GENERAL_IN :: :: op: READ_AND_CHECK :: :: rd_val= 0000000000070002, wr_val = 0000000001c7d88e.

It is an attribute check with write one read one register. Here in register PMU_GENERAL_INTR from bit 16 to bit 31, bit 16 to bit 19 is RW, which is read correctly but from bit 20 to bit 27 is RW in specification that is RAL but RO in RTL. Hence from bit 20 to bit 27 is modified to 0.

Chapter 6

Conclusion

With the advancement of tools and technologies in design, the necessity for reusable verification has araised. The main focus of this project was towards the reuse of components created in verification environment and to save verification time. Hence the components discussed like interrupt connectivity checker, interrupt coverage, register tests and their sequences can be used in different projects. The register tests and their sequences are also used at the SoC level. Various techniques are discussed which can be very useful to increase the reuasbility. The significant amount of time spent in verification can also be saved in debugs. Hence a random test measurement tool is developed to reduce the effort for debugging hence it saved time for verification. Tools like perl template toolkit is used in which assertions for 128 interrupt were processed by using single interrupt template and more number of interrupts can be generated as per the project requirement. Some reusable tests and sequences were written to verify each and every registers in the IP.

References

[1] Andy Wardley. ”Template-Toolkit Tutorial”. 2.27, 2013.

[2] Gopikrishna and Naresh Maddipati. ”Register Verification Tutorial”. *http://www.testbench.in/TB_32_REGISTER_VERIFICATION.html*.

[3] Jia Wei Han Qi, Zheng Jiang. ”IP Reusable Design Methodology”. *IEEE*.

[4] C. Heitmeyer Tevfik Bultan and J. O’Leary. ”Panel on design for verification”. *IEEE*, 2005.

[5] Rizal Prasetyokusuma Ranga Kadambi Teng-Peow Ng, Anjali Vishwanath. ”Reusable Verification Environment for Core based Designs”. *Development Centre, Microcontroller, Infineon Technologies Asia Pacific Pte Ltd, Singapore*.

[6] W. Ni and J. Zhang. ”Research of reusability based on UVM verification,”. *IEEE 11th International Conference on ASIC (ASICON)*, pages pp. 1–4, 2015.

[7] W. M. McCracken D. S. Guindi, W. B. Ligong and S. Rugaber. ”The Impact of Verification and Validation of Reusable Components on Software Productivity”. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. : Software Track, Kailua-Kona*, Volume II:pp. 1016–1024, 1989.

[8] Harry Foster Chief Scientist Verification Design Verification Technology. ”Verification is a Problem, but is Debug the Root Cause?”. *Mentor Graphics*, 2010.