GMIT
GALWAY-MAYO INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO

The Research & Development of an Advanced Verification Infrastructure for ASIC Devices using SystemVerilog & the Verification Methodology Manual (VMM)

In One Volume

Michael Anthony McMahon B.Eng

May 2010

Submitted for the Degree of Master of Engineering

Submitted to: Galway Mayo Institute of Technology, Galway, Ireland

Research carried out at: Chipright LTD, Galway, Ireland

Research Supervisor: Niall O'Keeffe

GALWAY-MAYO INSTITUTE OF TECHNOLOGY
09 MAR 2011
LIBRARY

Declaration

I hereby declare that the work presented in this thesis is my own and that it has not been previously used to obtain a degree in this institution or elsewhere.

Michael Anthony McMahon

Statement of Confidentiality

The material contained in this thesis should not be used, sold, assigned, or disclosed to other person(s), organisation(s) or corporation(s) without the written consent of both Chipright Ltd, and Niall O'Keeffe.

Galway Mayo Institute of Technology: Contact: Niall O'Keeffe

Tel: +35391 742057

Email: niall.okeeffe@gmit.ie

Chipright: Contact Kevin Keane

Tel: +91 444168

Email kevin.keane@chipright.com

Prologue

The research described in this thesis has been conducted over a 24-month period as part of a college/industry partnership project. This project was funded under the Innovation Partnership research grants scheme administered by Enterprise Ireland. The aim of the project is to research and develop an Advanced Verification Infrastructure using SystemVerilog and the Verification Methodology Manual (VMM).

Acknowledgements

I would like to thank the following people for their support and encouragement during the course of this project.

I would like to thank:

First and foremost, my supervisor, Niall O'Keeffe of the Department of Electronic Engineering in GMIT, for his supervision, guidance, inspiration, and his constructive suggestions throughout this research study. Without his support and encouragement this research would not have been done.

Emer O'Mahony for all her help and support throughout the course of my studies.

The senior engineers from Chipright Ltd. for their help and support while doing this project. They were of great help in teaching me to become familiar with SystemVerilog, helping with developing the testbench and giving useful insights into the practical and potential implementations of SystemVerilog.

I would also like to give a special mention to my parents, who have helped and guided me through my time at college in GMIT.

Abstract

In digital design much of the focus and attention in the past has been towards developing languages and tools primarily for use in designing an ASIC device. Today, the single biggest problem in digital design is the time that is spent in the Verification process. With this in mind the key EDA companies have focused a large proportion of their research and development budgets towards supporting new verification languages and methodologies, namely, SystemVerilog and OpenVera. SystemVerilog is the industry's first unified hardware description and verification language (HDVL). Along with developing SystemVerilog, the EDA companies have developed methodologies to support the language. The two main methodologies are the Synopsys's Verification Methodology Manual and Cadence/Mentor Open Verification Methodology. These methodologies are geared towards the implementation of functional coverage-points; use of assertion based coverage and constrained random test techniques.

This thesis outlines a VMM style test bench architecture that is structured to gain maximum efficiency from both constrained random and directed test case development. This thesis describes how directed and constrained random tests can be implemented inside a reusable directory structure that takes full advantage of the coverage and assertion techniques.

This thesis uses an IEEE-754 compliant floating-point adder model as part of a case study that illustrates a complete set of results extrapolated from using this test bench structure. An Integrated Inter Circuit (I2C) verification component has also been implemented and used to test the reusability of the test bench structure.

This thesis reviews the use of formal verification within the digital design community. Formal methods such as model checking, equivalence checking and deductive reasoning have become increasingly popular verification techniques. These methods are investigated to see if they could be used as alternative verification techniques within the verification environment.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1 Introduction

1.1Thesis Motivation

The continuous growth and complexity of digital design requires modern, systematic and automated approaches for creating test benches [1]. Given that up to 70% of the design time is spent in the verification process [5] it has become even more critical that verification engineers design test benches that are at the cutting edge of the verification industry.

The EDA vendors have recognised that a standardised approach for verification is required and this approach needs to support the structures needed to build an advanced verification environment. However, they don't actually specify how the architecture of the test bench environment should be built. It is still the verification engineer's responsibility to do this with the added pressure of making the environment re-usable for future chip sets.

With this in mind the main goal is to develop a new and more effective intuitive way of designing test benches. This thesis describes the implementation of constrained random test stimuli, functional coverage, and assertions and also describes an approach for creating test cases that allow the use of both constrained random tests and directed tests within a single environment. The environment built should also have the capability to be easily modified where a Device Under Test (DUT) of similar structure can be verified.

The verification environment is built using SystemVerilog and VMM. SystemVerilog is the industry's first unified hardware description and verification language (HDVL). VMM provides a set of rules and recommendations for constructing test benches using SystemVerilog. This is done through a set of base classes which are used to describe important elements of the test bench

1.2 Thesis Contributions

The contributions of this research are as follows:

- Summary of existing verification methodologies used in the Application Specific Integrated Circuit (ASIC) verification industry.
- Development of an advanced verification environment for a floating-point adder. The DUT in this case has been provided by Chipright Ltd [29]. The environment has been developed using the SystemVerilog language and Synopsys's Verification Methodology Manual (VMM).
    - Development of a Bus Functional Model (BFM) using SystemVerilog constructs to create an IEEE-754 compliant floating-point model for testing purposes.
    - Use of VMM Object Oriented (OO) base classes in the environment.
    - Creation of a reference model to verify that the DUT has successfully implemented floating-point addition.
    - Identification of functional coverage points inside the DUT and creation of a functional coverage plan
    - Creation of functional coverage groups to collect data during simulations and analysis and comparison of the results with the functional coverage plan.
    - Creation of assertions for the floating-point adder within the interface file.
    - Verification of the adder using constrained random test stimuli in conjunction with the reference model.
    - Collection of all functional coverage reports, assertion reports, sequencing reports and display in Hypertext Mark-up Language (HTML) format.
    - Execution of simulations to obtain full verification of the adder model as determined by the functional coverage reports.
    - Development of a test bench architecture that is structured to gain maximum efficiency from both constrained random and directed test case development.

- Extension of the verification environment to support a more complex BFM, thus proving that the testbench infrastructure can support both "intensive data verification" and more finely tuned "control of sequence and scenarios" verification.
- Investigation of the use of formal verification practices in the verification industry.

1.3 Thesis Structure

Figure 1.1 illustrates the steps taken throughout the project and the structure of the thesis.

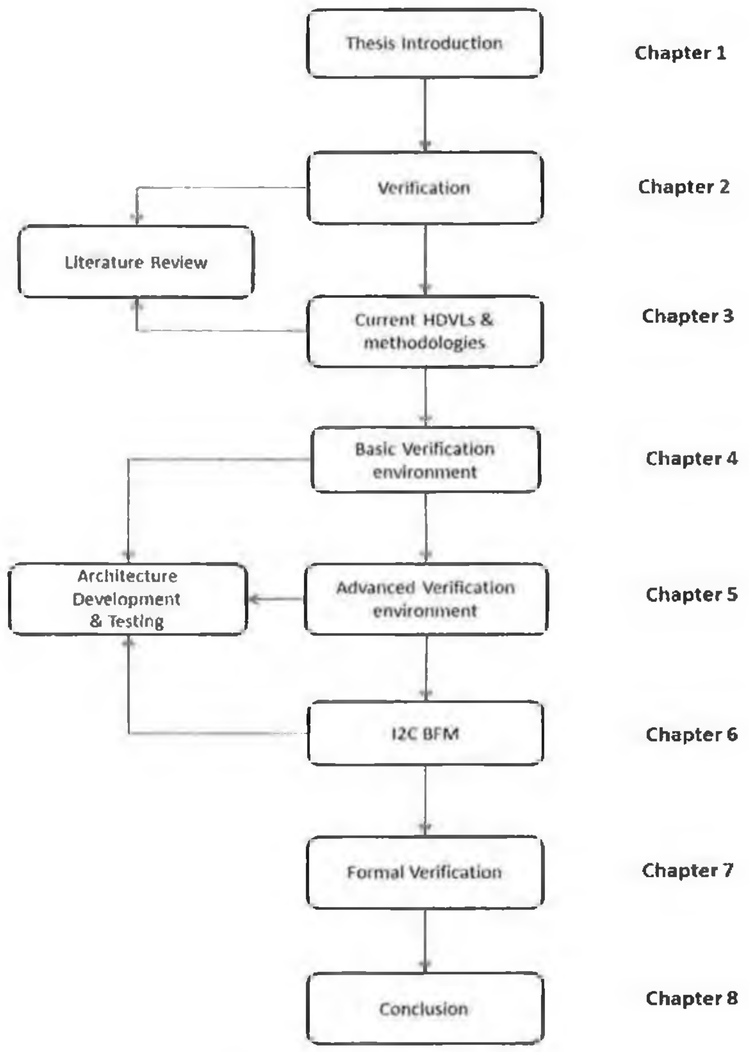


Figure 1.1 Thesis Structure

The literature review carried out for this project is presented in chapter 2 and 3 of this thesis. The following areas were researched as part of a literature review.

Chapter 2 analyses the different types of verification techniques within ASIC verification. This chapter also describes the challenges associated with ASIC verification.

Chapter 3 reviews the different hardware and verification languages within the micro-electronics industry. Also reviewed are Synopsys's VMM and Cadence/Mentor OVM and the different tools that support these methodologies.

Chapter 4 describes the implementation of a basic VMM based verification environment to test a floating-point adder. An IEEE-754 Floating-Point adder has been used as a case study in the implementation of a verification environment using the VMM methodology. The test bench has been written in the SystemVerilog language and includes the following elements:

- Constrainted random test stimuli to represent floating-point numbers
- A Bus Functional Model for the adder under test
- Assertions to check the design intent.

Chapter 5 outlines the architecture of the test environment. The test bench infrastructure is analysed and some key points are identified to yield a flexible solution that can meet industry level verification requirements. The chapter contains the test plan document that is used to help determine when the DUT has been completely verified. As part of the results analysis phase, the VMM Planner application from Synopsys is identified, described and used within the verification environment to ultimately help interpret the test results.

Chapter 6 describes how the verification infrastructure created and described within Chapter 5 can be re-used to facilitate the development of a more complex BFM. As part of a case study, the industry standardI2C protocol has been chosen. The BFM is built using the same template as outlined in Chapter4. This BFM is integrated into the verification environment and used to communicate with a DUT containing an I2C interface.

Chapter 7 describes the formal verification process and identifies how it is different to the constrained random approach undertaken in this project. As an added part of the research work, formal verification techniques have been reviewed with emphasis on determining if parts of this approach could be utilised within the advanced verification environment.

Chapter 8 outlines the results and conclusions of the research work. The key points of the research project are identified and discussed with recommendations for future research work.

Chapter 2 Review of ASIC Verification Techniques

2.1 Introduction

This chapter reviews ASIC Verification Techniques. Verification is the activity that determines the correctness of the design that is being developed. It ensures that the design meets the specification required of the product and the intent of which the product should operate [2].

Verification has become a major element within the digital design development process. Various industry surveys highlight that verification is the single largest component in a project, taking up more than half of the total project's staffing, schedule and cost [3]. It is often the limiting factor to project completion, and is becoming the single largest bottleneck in the ASIC industry.

Verification is complex, time-consuming, and sometimes poorly understood. As a result, the verification effort represents one of the bigger risks to the successful completion of a project.

This chapter gives an overview of what ASIC verification is and the importance of it within the microelectronics industry. Also described within this chapter are the challenges that verification has to overcome. A key point in explaining verification is that it is so vast and covers many areas within the electronics industry. Verification may be classified into three types: Applications, Software and Circuit Development. An example of each type is given below

Applications:

- File verification: Checks the formal correctness or integrity of a file

Software Development:

- Formal verification: Uses mathematical proofs to check the correctness of an algorithm
- Intelligent verification: Updates the testbench to changes in Register Transfer Level (RTL)

- Runtime verification: Verification technique that combines formal verification and program execution
- Software verification: Assures that software fully satisfies all of the requirements

Circuit Development:

- Functional verification: Verifies the design of digital hardware
- Analogue verification: Verifies analogue or mixed-signal hardware
- Physical verification: Verifies an IC layout design

The main verification techniques used within ASIC verification are functional verification and formal verification. Functional verification is the main verification type implemented throughout this project. Formal Verification is discussed from a research point of view in chapter 7.

2.2The Goal of Verification

The goal of ASIC verification can be simply stated: to prove that a design will work as intended. There are four components to achieving this goal [3]:

1. Determine what the intent is
2. Determine what the design does
3. Compare the two to ensure that they match
4. Estimation of the level of confidence of the verification effort.

2.2.1 Determining Intent

Determining the intent of the system is essential for verification to succeed [3]. The intent could be defined as what the system is supposed to do, which may be different from what it actually does.

In some cases, the intent may be obvious, when it is possible to clearly specify the functions which the device must perform. However, for most reasonably complex digital systems even the intent may not always be clear. How should a device operate when an error is introduced, or when two competing actions are received at the same time, or when a resource is oversubscribed? It is often the architectural specification that describes the intent of the system and the implementation specification that documents how the design is intended to implement the intent as seen in figure 2.1.

The Architectural specification is usually created by examining many use-cases that are specific scenarios that describe how the device will work, and determining the system intent. The Architectural specification is defined to satisfy all of the use-cases.

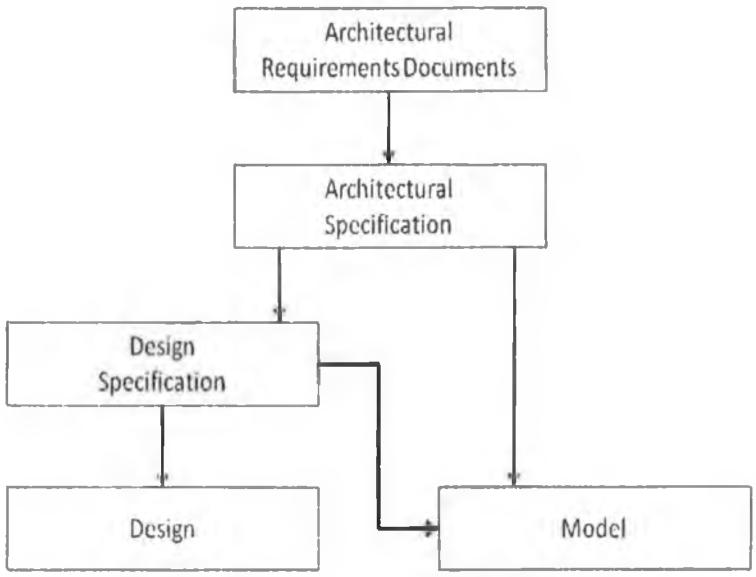

Architectural Requirements Documents
Architectural Specification
Design Specification
Design
Model

Figure 2.1 Specification Hierarchy

2.2.2 Determining What a Design Does

The next part of ASIC verification is to determine what a design does. This is the first step in comparing the design with the intent. Since the design has not yet been implemented in a prototype, a model of the design is needed. The model is described in a software language that allows simulations to be run. The method used to test the model is quite traditional: stimulate it and see what happens. The poking is frequently referred to as stimulus injection, while the observation involves collecting and checking the outputs of the simulation.

2.2.3 Comparing the Intent with the Design

Ideally, one would like to take the intent and the design, stimulate them in the same way, and check that the results match. This would provide a direct way to validate that the two are identical. The issue with this is that intent is not something that can usually be modelled. In some cases, it may not even be fully understood. Instead, something else must be used to represent the intent of the system. In some cases, people have built executable design specifications. That can be useful, but of course, the specification may also have errors. In other cases, the intent is captured as a reference model that is supposed to behave as the system is intended to work. These are some of the uses of the model block shown in figure 2.1. Still another is to provide a series of tests and expected results to run against a design. These approaches, and most others, rely on comparing two different models, and examining the discrepancies. It is hoped that two models will not have exactly the same errors.

2.2.4 Determining Verification Completeness

There are a number of other complexities that arise during the process of functional verification. One of the most frequently discussed is determining the completeness criteria of the verification.

Just as the system intent is often not fully defined, the design model may never be fully tested. There are several reasons for this. First, it is difficult to ensure that the complete intent of the system is known.

Anything that has not been included in the architectural specification or functional specification is unlikely to show up in the design or to be tested, even though the function may be required for the system to work. Secondly, it is difficult to know if the design has been tested sufficiently. Any issue that is found provides proof that the questions have not all been previously asked. It is rare that someone can prove that no other questions need asking. It is difficult to show that no more errors can be found. Another limitation of verification is the impracticality of running a complete test. Any reasonably sized design is too big to examine completely. Ideally an engineer wishes to see that all possible stimuli have been applied in all possible states of the design

As a result, functional verification is rarely certain and rarely complete. Given the importance of success to many verification projects, estimation methods are used to provide an approximation of the quality level of the functional verification.

2.3The ASIC Verification Challenge

Functional verification is a simple problem to state but a challenging one to address [4]. The increasing size and complexity of designs and shortening time to market means that verification engineers must verify larger and more complex design in a shorter time than in previous projects. An effective solution to meeting this increased demand for achieving verification closure must address the flowing verification challenges:

- Reusability
- Efficiency
- Productivity
- Code Performance
- Completeness

The challenge in **verification reusability** is to increase portions of the verification environment infrastructure that can be reused in another part of the project or in a completely different project [4]. This is done by sharing features that are similar with those in the current project. By using standardised interfaces or functions, a high degree of reuse can be achieved. Blocks can be reused in any project that makes use of the same standardised interface. As well as using standardised interfaces, identifying common functionality in the verification environment planning stage can lead to further reuse of verification infrastructure.

The challenge in **verification efficiency** is to minimise the amount of manual effort required for completing a verification project. The reason for reducing manual effort in a verification project is that it is error prone, omission prone and time consuming. In contrast, automated systems can complete a significant amount of work in a short time. However automated systems must be built manually. To improve verification efficiency, careful analysis is required of the trade-off between the extra effort required for building an automated system and the gains it affords. An example of such a system is the coverage driven random verification methodology where the effort to build an automated system for stimulus generation and automated checking leads to significant improvements in verification efficiency, and hence productivity. Before building an automated system an important consideration to be identified is that the automated system requires a consistent infrastructure on which it can be developed, and also imposes a user model on how an engineer interacts with it. Deployment of an automated system requires consistency both in infrastructure and engineering approach, both of which take time and targeted effort to achieve.

The challenge in **verification productivity** is to maximise work produced manually by verification engineers in a given amount of time. Achieving higher productivity has become a major challenge in functional verification. Significant improvements in the design flow have afforded design engineers with much higher productivity. However improvements in verification productivity have lagged those on the design side, making functional verification the bottle neck in completing the design. The goal of functional verification is to reduce the gap between productivity and verification. To achieve this verification is moving to a higher level of abstraction and leveraging reuse concepts.

The challenge in **verification code performance** is to maximise the efficiency of verification programs. This consideration is in contrast with verification productivity, which deals with how efficiently verification engineers build the verification environment and verify the verification plan. The time spent on a verification project is usually dominated by the manual work performed by verification engineers. In most projects verification performance has usually been a secondary consideration to designing and building verification environments. An important area in which verification performance becomes a primary consideration is in running regression test suites where the turnaround times are dominated by how efficient verification programs operate. Expert knowledge of tools and languages used for implementing the environment is a mandatory requirement for improving verification performance.

The challenge in **verification completeness** is to maximise the part of the design behaviour that is verified. The major challenge in improving verification completeness is in capturing all of the scenarios that must be verified. This however is a manual, error prone, and omission prone process. There have been significant improvements in this area by moving to coverage driven methodologies. Coverage driven methodologies require a measure of completeness whose calculation requires strict planning, tracking and organisation of the verification plan. This strict requirement on verification plans naturally leads to exposing the relevant scenarios that may be missing. Fine-tuned verification planning and management methods have been developed to help with the planning and tracking of verification plans.

2.4 Functional Verification

The main purpose of functional verification is to ensure that a design implements the intended functionality [5]. As shown by Figure 2.2, functional verification tries to establish a relationship between a design and its specification. Without functional verification, one must trust that the transformation of a specification into RTL code has been performed correctly, without misinterpretation of the specification's intent.

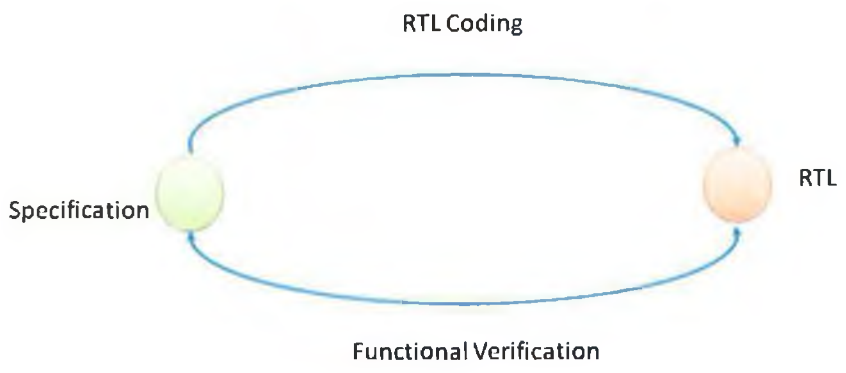


Figure 2.2 Functional Verification of the RTL with respect to the specification

It is important to note that, unless a specification is written in a formal language with precise semantics, it is difficult to prove that a design meets the intent of its specification. Specification documents are written using natural languages by individuals with varying degrees of ability in communicating their intentions. Any document is open to interpretation. One can easily prove that the design does not implement the intended function by identifying a single discrepancy. The opposite sadly, is not true: No one can prove that there are no discrepancies. Functional verification as a process can show that a design meets the intent of its specification, but it cannot prove it.

2.4.1 Verification Process

The process of verification should be primarily accomplished by placing the DUT in a testbench [3]. The testbench applies some test vectors to the design to ensure that the intent meets the specification. The testbench takes over the task of applying inputs (testcases) to the design and setting it up in a known configuration. Various input vectors are applied to the design to ensure that the response is as expected. In addition to the DUT, various other modules which check the output of the device or observe some signals of the DUT may also be instantiated in the testbench. These checkers could perform various functions in the testbench. For instance, some checkers may check for the correct signal sequence protocol on the inputs and outputs of the device. Monitors perform an additional function of watching the I/O or some specific busses in the DUT.

2.4.2 What is a Testbench?

The term "testbench" usually refers to simulation code used to create a predetermined input sequence to a design and to observe the response [5]. Test benches are implemented using VHDL, Verilog, System C, SpecmanE, and System Verilog and may include external data files or C routines.

Figure 2.3 illustrates how a testbench interacts with a DUT. The testbench provides inputs to the design and watches any outputs. The verification challenge is to determine what input patterns to supply to the design and what is the expected output of a working design when subjected to those input patterns.

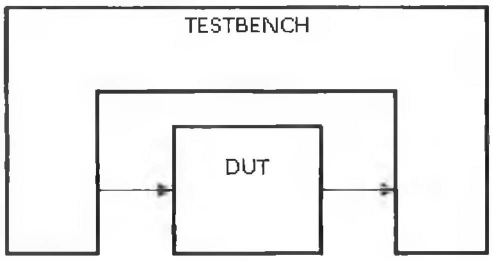


Figure 2.3 Testbench interaction with DUT

A testbench should be robust so that it provides a high level of confidence that the design being tested works correctly.

2.5 The Bottleneck in ASIC Verification

Today in the era of multi-million gate ASICs and FPGAs, reusable intellectual property (IP), and system-on-a-chip (SoC) designs, verification can consume up to 70% of the design effort [5]. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers.

2.5.1 Reasons for the Verification Bottleneck

The verification bottleneck is a result of raising the design abstraction level for the following reasons [6]:

- Designing at a higher abstraction level allows engineers to build highly complex functions with ease. However, this increase in design complexity then results in almost doubling the verification effort. Doing so has doubled the functional complexity and hence its verification scope.

- Using a higher level of abstraction for design, transformation, and eventual mapping to the end product is not without information loss and misinterpretation. For instance, synthesis takes HDL-level design and transforms it to gate level format. Verification is needed at this level to ensure that the transformation is indeed correct, and that design intent has not been lost. Raising the level of abstraction also brings about the question of interpretation of the code that is used to describe the design during simulation.

Other factors that affect the verification problem are:

- Increase in functional complexity because of the complex nature of designs today; for example, co-existence of hardware and software, analogue and digital.

- The requirement for higher system reliability forces verification tasks to ensure that a chip level function will perform satisfactorily in a system environment, especially when a chip level defect has a multiplicative effect.

Research done by Collett International Research into functional verification by collecting information from design engineers shows some of the reasons behind the bottleneck in verification [7]. The result of their finding is shown below:

- **Chip flaws because of design errors.** 82% of designs with a re-spin resulting from logic and functional flows had design errors. This means that corner cases have not been adequately covered during the verification process and that bugs remain hidden in the design all the way through tape out.

- **Chip flaws because of specification errors.** 47% of designs with re-spin resulting from logic and functional flaws had incorrect or incomplete specification. 32% of designs with a re-spin resulting from logic and functional flaws had changes in specifications.

- **Problems with reused IP and imported IP.** 14% of all chips that failed had bugs in reused components or imported IP.

- **Effects of a re-spin.** Re-spins can cost a company up to $100,000. In addition, a respin delays product introduction and adds to expense due to failing systems that use these defective chips.

2.6 Conclusion

This chapter explains what verification is and some of the challenges and difficulties that are associated with it. One of the major problems mentioned in this chapter is the bottleneck within verification. To reduce the bottleneck in verification, companies have researched and developed solutions and ideas. Listed below are some of the ideas and solutions that companies have developed [7].

1. **Reduce chip complexity.** Practically, this is not possible because of customer demand for more functionality.

2. **Reduce the number of designs**. This solution affects a company's long-term goal of being profitable.

3. **Increase resources.** Another alternative is to increase the number of designers or verification engineers. This alternative works well to some extent, but does not meet today's demands for verifying exponentially complex chips with a limited amount of time.

4. **Increase productivity of designers.** Productivity gains have been achieved by improving compute power, using personal tools such as Microsoft Excel. While they have been of great help in capturing test and verification plans, the majority of the time is spent in coding the test cases, running them, and debugging.

5. **Increase verification productivity.** This has obvious potential for gains in productivity.

To increase verification productivity, the EDA industry has come up with a solution similar to what was used to solve the design bottleneck - the concept of abstraction. Higher-level languages such as Verilog and VHDL support chip verification using test benches. These languages include constructs such as tasks, threading (fork, join) and control structures such as "while." This provides more control to fully exercise the design on all functional corners. However, these constructs are not synthesisable and hence are not used by designers as a part of actual design code.

As complexity continues to grow, new verification languages have been created and introduced that try to verify complex designs at various levels of abstraction. Along with new verification languages, methodologies and tools have been developed to support them. One such verification language is SystemVerilog. Along with SystemVerilog the EDA companies have developed methodologies to support the language. The next chapter will discuss SystemVerilog and the different methodologies that support the language.

Chapter 3 Current Verification Languages, Methodologies and Trends

3.1Introduction

This chapter outlines the advantages SystemVerilog brings to the verification industry. SystemVerilog is a major extension of the established IEEE 1364 Verilog language [8]. These extensions are discussed, along with a review of the Verilog language. SystemVerilog also includes some features from the OpenVera verification language which is discussed in this chapter. Also explained are the different methodologies that have been developed by the major EDA companies to support SystemVerilog. Along with the methodologies the EDA companies are developing tools to support SystemVerilog.

3.2 SystemVerilog

SystemVerilog is the industry's first unified hardware description and verification language (HDVL). SystemVerilog combines the features of Hardware Description Languages such as Verilog and VHDL with features from Hardware Verification Languages such as OpenVera. SystemVerilog also includes features from C and C++. SystemVerilog became an official IEEE standard (IEEE 1800) in 2005 [9]. It was developed originally by Accellera to improve productivity in the design of large gate count, IP based, bus intensive chips.

Since its development SystemVerilog is finding practical use in the areas of concise and productive RTL coding, assertion based verification, and building coverage driven verification environments using constrained random techniques

SystemVerilog provides:

- Constrained random stimulus generation
- Functional Coverage
- Assertion based Verification
- Higher-level structures, especially Object Oriented Programming
- Multi-Threading and inter-process communication

The methods listed above help to improve the verification process. SystemVerilog also provides enhanced hardware modelling features that improve the RTL design productivity and simplify the design process [10]. Since its development SystemVerilog has been adopted by hundreds of semiconductor design companies and supported by more than 75 EDA, IP and training solutions worldwide.

3.3Verilog

SystemVerilog is a major extension of Verilog. Verilog is a hardware description language (HDL) used to model electronic systems. Verilog supports the design, verification and implementation of analogue, digital and mixed signal circuits at various levels of abstraction [11].

The designers of Verilog wanted a language with syntax similar to the C programming language. The language is case sensitive, it has a pre-processor like C, and the major control flow keywords such as "if" and "while" are similar to those in C.

Verilog differs from C in some fundamental ways. Verilog uses begin/end instead of curly braces to define a block of code. Verilog 1995 and 2001 do not have structures, pointers or recursive subroutines. However, SystemVerilog includes these capabilities. Also the concept of time, which is so important to a HDL, is not found in C.

The language differs from a conventional programming language in that the execution of statements is not strictly linear. A Verilog design consists of a hierarchy of modules [12]. Modules are defined with a set of input, output, and bi-directional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behaviour of the module by defining the relationships between component, ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. Concurrent statements and all begin/end blocks in the design are executed in parallel. A module can also contain one or more instances of another module to define sub-behaviour.

3.4 OpenVera

OpenVera is an interoperable, open hardware verification language developed by Synopsys [13]. The OpenVera language has been used as the basis for the SystemVerilog IEEE Standard. OpenVera remains a widely adopted and well supported hardware verification language.

OpenVera is an intuitive easy to learn language that combines the familiarity and strengths of HDLs, C++ and Java, with additional constructs targeted at functional verification making it ideal for developing test benches, assertions and properties [14] .

3.5 History of Verilog and SystemVerilog

The Verilog Hardware Description Language (HDL) was originally developed together with the Verilog XL simulator by Gateway design automation and was introduced in 1984 [15]. In 1989 Cadence Design Systems acquired Gateway, and with it the rights to the Verilog language and the Verilog XL simulator. In 1990 Cadence placed the Verilog language into the public domain. Open Verilog International (OVI) is a non-profit organisation tasked with making Verilog an IEEE standard, achieved in 1995 (IEEE 1364 -1995). In 2000, OVI combined with Very High Speed Integrated Circuit (VHSIC) Hardware Description Language better known as VHDL to form Accellera. In 2001 a revised version of Verilog, known as Verilog 2001 was released, with the IEEE standard 1364 – 2001. This version incorporates many useful improvements. In 2002 Accellera introduced further extensions to Verilog and these were released under the name SystemVerilog 3.0. These extensions offer a higher level of abstraction for modelling and verification. An extended version of SystemVerilog 3.1 and a further revision of SystemVerilog 3.1a formed the basis for SystemVerilog obtaining an IEEE standard in 2005 (1800 – 2005). In 2009, the SystemVerilog and Verilog (IEEE 1394-2005) standard merged, creating the current IEEE Standard 1800-2009. Figure 3.1 illustrates the timeline of Verilog and SystemVerilog development.

117086
GALWAY-MAYO INSTITUTE OF TECHNOLOGY
M.Eng 19
09 MAR 2011
LIBRARY

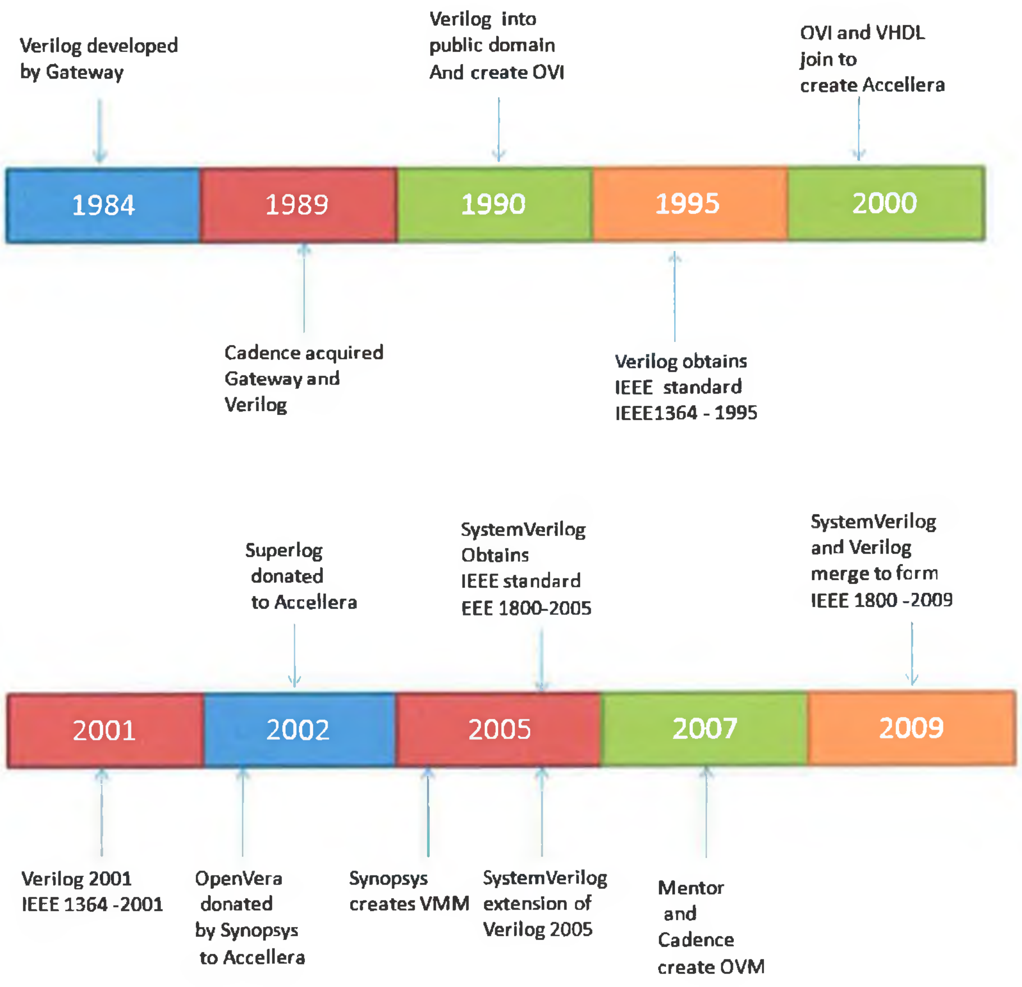

Verilog developed by Gateway
Verilog into public domain And create OVI
OVI and VHDL join to create Accellera
1984
1989
1990
1995
2000
Cadence acquired Gateway and Verilog
Verilog obtains IEEE standard IEEE1364 - 1995
Superlog donated to Accellera
SystemVerilog Obtains IEEE standard EEE 1800-2005
SystemVerilog and Verilog merge to form IEEE 1800 -2009
2001
2002
2005
2007
2009
Verilog 2001 IEEE 1364 -2001
OpenVera donated by Synopsys to Accellera
Synopsys creates VMM
SystemVerilog extension of Verilog 2005
Mentor and Cadence create OVM

Figure 3.1 Timeline of Verilog and SystemVerilog development

3.6 SystemVerilog Extensions to Verilog 2001

This section lists the major extensions and improvements and new features that SystemVerilog has made to Verilog 2001 [9].

The list of the extensions is included below along with a detailed explanation of each.

New Data Types

One of the first improvements in SystemVerilog from Verilog is the different C like data types. The new data Types include byte, shortint, int, longint, bit, logic, string, chandle, typedef, struct, union, tagged union, enum.

Dynamic Data Types

The next improvement in SystemVerilog is the use of dynamic data types. These include strings, classes, dynamic arrays, associative arrays including automatic memory management freeing users from de-allocation issues.

Dynamic arrays:

One dimensional arrays whose size can be set or changed at runtime. The space for a dynamic array does not exist until the array is explicitly created at runtime.

Associative Array:

When the size of the collection is unknown or the data space is sparse, an associative array is a better option than dynamic arrays. Associative arrays do not have the storage allocated until it is used, and the index expression is not restricted to integral expression, but can be of any type.

Classes:

Classes are defined to support object oriented programming within SystemVerilog,. Classes consist of properties and methods. Classes may be extended to form a new class. This is called single inheritance. So basically classes are user defined data types. Classes may be instantiated to create objects. To support directed random verification, Class Variables may be declared random. Classes may also include constraints, which direct the generation of random values.

Built in methods to extend the language.

The next improvement in SystemVerilog is the built in methods to extend the language, e.g. array manipulation methods and enum and string methods. SystemVerilog also includes C like jump statements such as return, break and continue.

Enhanced Task and Functions

SystemVerilog also enhances tasks and functions. In SystemVerilog, multiple statements can be written between the task declaration and end task, which means that begin / end, can be omitted. If begin / end is omitted, statements are executed sequentially. It is legal to have no statement at all.

Extensions to fork - join

SystemVerilog also has extensions to fork-join to model pipelines and for enhanced process control. The fork- join construct enables the creation of concurrent processes from each of its parallel statements. A Verilog fork-join block always causes the process executing the fork statement to block until the termination of all forked processes. With the addition of the join_any and join_none keywords, SystemVerilog provides three choices for specifying when the parent resumes execution.

Option	**Description**
join	The parent process blocks until all the processes spawned by this fork complete
join_any	The parent process blocks until any one of the processes spawned by this fork complete
join_none	The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start until the parent thread executes a blocking statement

Table 3.1 SystemVerilog extension to fork-join

Extensions to the Always Block

SystemVerilog provides extensions to the always block for modelling combinational, latched or clocked processes. In an always block which is used to model combinational logic, forgetting an else statement leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialised always_comb and always_latch blocks, which indicate the design intent to simulation, synthesis and formal verification tools. SystemVerilog also adds an always_ff block to indicate sequential logic.

Interface

SystemVerilog adds interfaces to improve encapsulated communication. Interfaces may effectively be a "bundle of wires" with a single name, but they are also capable of containing behaviour and so may be used by BFMs. Interfaces therefore support Transaction Level Modelling (TLM) and in particular the reuse of verification environments at different levels of abstraction.

Clocking Block

Clocking blocks allow control over the clocking behaviour of different parts of a testbench. Clocking blocks help the sampling of input/output data to be set up in a well-defined way [8].

Program Block

Also included in SystemVerilog is the program block for describing tests. The program block is the basic building block in Verilog. The program block can contain hierarchies of other modules, wires, task and function declarations, and procedural statements within always and initial blocks [16]. This construct works extremely well for the description of hardware. However, for the testbench, the emphasis is not in the hardware level details such as wires, structural hierarchy, and interconnects, but in modelling the complete environment in which a design is verified. A lot of effort is spent getting the environment properly initialised and synchronised, avoiding races between the design and the test-bench, automating the generation of input stimuli, and reusing existing models and other infrastructure.

The program block serves three basic purposes:

1 It provides an entry point to the execution of test-benches.

2 It creates a scope that encapsulates program-wide data.

3 It provides a syntactic context that specifies execution in the Reactive region.

Direct Programming Interface (DPI)

The SystemVerilog Direct Programming Interface (DPI) allows C functions to be called directly from SystemVerilog or vice versa without using the Programming Language Interface (PLI) [17]. The Direct Programming Interface (DPI) allows functions in a foreign language to be called as SystemVerilog functions. This is very useful as reference models are often written in C. They can then be used as a golden reference or as part of a checker in an automated testbench. The DPI allows this reuse in an efficient way.

SystemVerilog Assertions (SVA)

What is an assertion?

An assertion is basically a *"statement of fact"* or *"claim of truth"* made about a design by a design or verification engineer [18]. An engineer will assert or "claim" that certain conditions are always true or never true about a design. If that claim can ever be proven false, then the assertion fails.

Assertions essentially become active design comments. Assertions can be checked dynamically by simulation, or statically by a separate property checker tool i.e. a formal verification tool that proves whether or not a design meets its specification. Such tools may require certain assumptions about the design's behaviour to be specified.

What is a property?

A property is basically a rule that will be asserted to passively test a design [18]. The property can be a simple Boolean test regarding conditions that should always hold true about the design, or it can be a sampled sequence of signals that should follow a legal and prescribed protocol.

For formal analysis, a property describes the environment of the block under verification, i.e. what is the legal behaviour of the inputs.

Types of SystemVerilog Assertions:

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to the *if* statement. The difference is that an *if* statement does not assert that an expression is true, it simply checks that it is true.

```
if (A==B)  // Simply checks if A equals B
assert (A == B)  // Asserts that A equals B; if not an error
is generated
```

Figure 3.2 Example of Immediate Assertion

Concurrent Assertions:

The most valuable assertion style that can be used in design and verification environments is the concurrent assertion. Concurrent assertions are monitors that reside inside a block of code to periodically sample and test signals and to generate error messages if the assertion ever fails. Concurrent assertions are typically sampled once per clock period at the end of the clock cycle, just before the next active clock edge. Concurrent assertions require the assertion of a property, where a property is basically a design rule that should always be true. The simplest of concurrent assertions takes the form:

```
assert property (!(Read && Write));

// asserts that the expression Read && Write is never true at
any point during simulation.
```

Figure 3.3 Example of Concurrent Assertions

Properties may be checked using concurrent assertions. If an asserted property does not hold, the assertion is violated and an error is generated, Assertions may be checked dynamically during simulation, or statically using a property checker, which is a type of formal verification tool.

Coverage

Coverage serves two critical purposes throughout the verification process [19]. The first is to identify holes in the process by pointing to areas of the design that have not yet been sufficiently verified. This helps to direct the verification effort by answering the key question of what test to write next (directed test or constrained random test).

Coverage is also an indicator of when verification is thorough enough to tape out the design. Coverage provides more than a simple yes/no answer; incremental improvement in coverage helps to assess verification progress and thoroughness, leading to the point at which the development team has the confidence to tape out the design. In fact, coverage is so critical that most advanced, automated approaches implement coverage driven verification, in which coverage guides each step of the process.

Coverage is divided into two main categories: code coverage and functional coverage. Code coverage, in its many forms (line coverage, toggle coverage, expression coverage), is typically an automated process that tells whether all of the code in a particular RTL design description was exercised during a particular simulation run (or set of runs). Code coverage is a necessary but not sufficient component of a reliable verification methodology.

In contrast, functional coverage provides an explicit metric of how much of the desired functionality of the design has actually been exercised. Verification confidence can often be improved further by using cross-coverage techniques to measure combinations of coverage metrics [19]. Important functional coverage and cross-coverage points should be identified early in the project and preferably included in the written verification plan.

The process of filling the holes identified by the full range of coverage is the heart of the coverage driven verification process. When 100% coverage is achieved then the verification confidence is high enough to tape out the design. SystemVerilog provides coverage properties for lower level coverage points, covergroups for tracking higher level values and support for cross coverage.

Object Oriented

SystemVerilog provides an object-oriented programming model [20]. SystemVerilog classes support a single inheritance model. There is no facility that permits conformance of a class to multiple functional interfaces, such as the interface feature of Java. SystemVerilog classes can be type parameterised, providing the basic function of C++ templates. However, function templates and template specialization are not supported.

The polymorphism features are similar to those of C++: the programmer may specifically write a virtual function to have a derived class gain control of the function.

Encapsulation and data hiding is accomplished using the local and protected keywords, which must be applied to any item that is to be hidden. By default, all class properties are public.

SystemVerilog class instances are created with the new keyword. A constructor denoted by function new () can be defined. SystemVerilog supports garbage collection, so there is no facility to explicitly destroy class instances.

3.7 Methodologies and Tools

Since the release of SystemVerilog in 2002 the key EDA companies Synopsys, Mentor Graphics and Cadence have been developing methodologies to support the language. Synopsys were the first to develop a successful methodology for SystemVerilog called Verification Methodology Manual (VMM) in 2005. Mentor developed an open source methodology called Advanced Verification Methodology (AVM) in 2005. Cadence in 2006 also developed a methodology for SystemVerilog called Universal Reuse Methodology (URM). Since 2005 a number of changes and developments have been made to the methodologies. In 2007 Mentor and Cadence came together and formed a new methodology called Open Verification Methodology (OVM) that was also open source. Since OVM's development Synopsys have made VMM open source in 2008 and in 2009 released a new version of VMM (VMM 1.2) that has similar features to OVM. In late 2008 Mentor announced the availability of an open-source SystemVerilog solution (interoperability library) for users of OVM. The solution enables the easy and flexible reuse of VMM code within an OVM environment.

What is Interoperability?

VMM based verification components can now be seamlessly reused within an OVM environment [21]. In addition, entire VMM environments can be reused without modification within an OVM environment through the use of the Interoperability library that provides the data and semantic conversions between the old and new environments.

3.7.1 Verification Methodology Manual (VMM)

VMM is co-authored by verification experts from ARM and Synopsys. VMM describes how to use SystemVerilog to develop scalable, predictable and reusable verification environments [22]. VMM has become an important factor in increasing verification reuse, improving verification productivity and timelines. VMM consists of coding guide lines and base classes. VMM is focused on coverage driven verification (CDV).

VMM libraries consists of the following sub libraries

- VMM Standard Library
- VMM Register Abstraction Layer (RAL)
- VMM Hardware Abstraction Layer (HAL)

The VMM Standard Library provides base classes for key aspects of the verification environment, transaction generation, notification service and a message logging service.

VMM Component	**Description**
vmm_env	The class is a base class used to implement verification environments.
vmm_xactor	This class is to be used as the basis for all transactors, including bus functional models, monitors and generators. It provides a standard control mechanism expected to be found in all transactors.
vmm_channel	This class implements a generic transaction level interface mechanism. The transaction level interfaces remove the higher level layers from the physical interface details. Using vmm_channel, transactors can pass transactions from one to another.
vmm_data	This class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. The user must extend vmm_data to create a custom transaction.
vmm_log	This class provide a mechanism for reporting simulation activity to a file or a terminal. The vmm_log class ensures a consistent look and feel to the messages issued from different sources.
vmm_atomic_	vmm_atomic_gen is a macro. This macro defines an atomic generator for generating transactions which are derived from

gen	vmm_data.
vmm_scenario_gen	Defines a scenario generator class to generate sequences of related instances of the specified class.

Table 3.2 List of VMM Classes

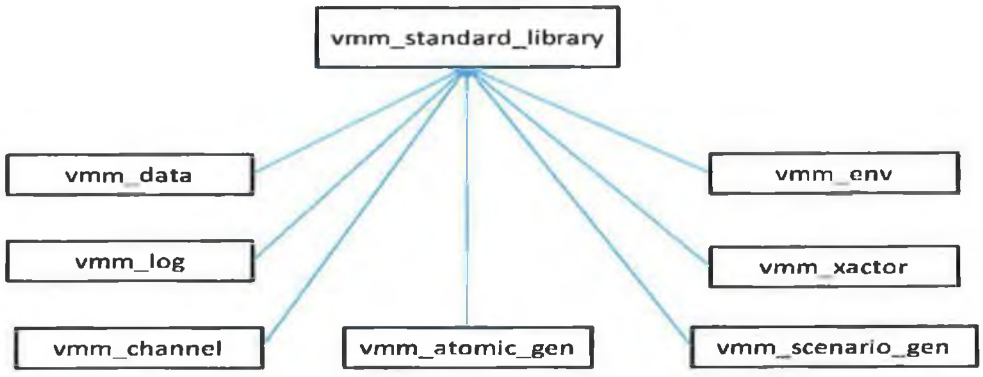


Figure 3.4 VMM Standard Library

3.7.2 Open Verification Methodology (OVM)

The OVM is the result of joint development between Cadence and Mentor Graphics. OVM is open source; it combines features from URM and AVM. The OVM Class Library [23] provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

The OVM library contains

- Component classes for building testbench components like generator, driver, monitor
- Reporting classes for logging,
- Factory for object substitution.
- Synchronisation classes for managing concurrent process.
- TLM Classes for transaction level interface.
- Sequencer and Sequence classes for generating realistic stimulus.
- Macros which can be used for shorthand notation of complex implementation.

Ovm Component	**Description**
ovm_driver	Driver takes the transaction from the sequencer using seq_item_port. This transaction will be driven to DUT as per the interface specification.
ovm_env	Used to create and connect the ovm_components like driver, monitors, sequencers etc. An environment class can also be used as sub-environment in another environment.
ovm_test	Using ovm_test provides the ability to select which test to execute using the OVM_TESTNAME command line option or argument to the ovm_root::run_test task.
ovm_sequencer	Sequencer generates stimulus data and passes it to the driver. All sequencers should be derived from the ovm_sequence base class directly or indirectly. ovm_sequence base call is parameterised for request and response item types.
ovm_monitor	ovm_monitor allows you to distinguish monitors from generic component types inheriting from ovm_component.

ovm_scoreboard	ovm_scoreboard will allow you to distinguish scoreboards from other component types inheriting directly from ovm_component.
ovm_transaction	The ovm_transaction class is the root base class for OVM transactions. Inheriting all the methods of ovm_object, ovm_transaction adds timing and recording interface.
ovm_sequence_ item	The ovm_sequence_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.
ovm_sequence	A sequence is defined by extending ovm_sequence class. This sequence of transactions should be defined in body() method of ovm_sequence class. OVM has macros and methods to define the transaction types
ovm_object	The ovm_object class is the base class for all OVM data and hierarchical classes. Its primary role is to define a set of methods for such common operations as create, copy, compare, print, and record. Classes deriving from ovm_object must implement the pure virtual methods such as create and get_type_name.

Table 3.3 List of OVM Classes

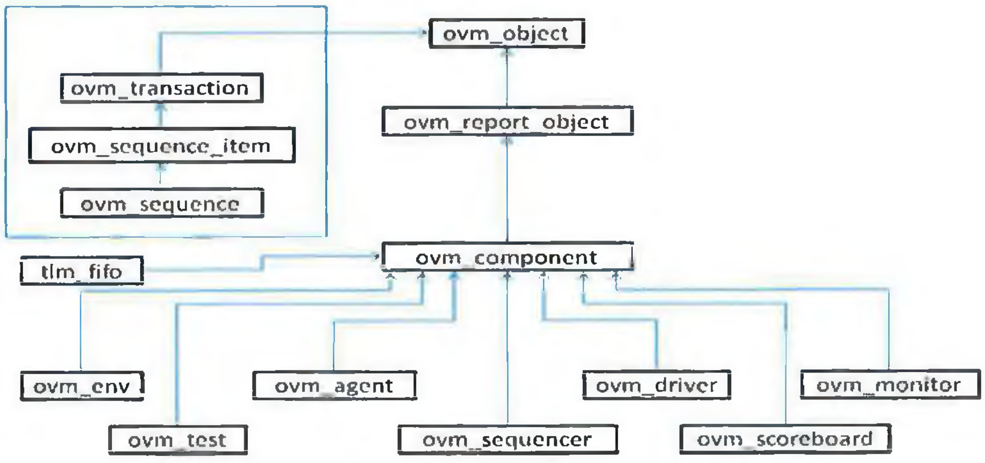


Figure 3.5 OVM Library

3.7.3 Synopsys VCS Comprehensive RTL Verification Solution

VCS [24] is an RTL verification product that provides advanced bug finding technologies with a built in debug and visualisation environment. VCS supports all popular design and verification languages including Verilog, VHDL, SystemVerilog and SystemC. VCS includes the full featured native testbench, complete assertions and comprehensive code and functional coverage making bug finding faster and easier. Also VCS Verification Library provides verification IP for today's most popular bus standards.

Key Benefits

- Supports SystemVerilog and OpenVera test benches.
- VCS also includes Synopsys VMM methodology
- VCS supports SystemVerilog assertions (SVA) and OpenVera assertions (OVA).
- Support for all popular design and verification language standards, including Verilog, VHDL, SystemVerilog and SystemC .

3.7.3.1 Discovery Visualisation Environment (DVE)

VCS also includes the Discovery Visualisation Environment (DVE) [25], which is a full featured debug and visualisation environment. DVE has been created to work with VCS and shares a common look and feel with other Synopsys graphical based analysis tools. DVE enables easy access to design and verification data along with drag and drop menu an icon driven environment. Its debug capabilities include: tracing drivers, waveform compare, schematic views, path schematics, and support for the highly efficient Synopsys compact VCD+ binary dump format.

It also provides SystemVerilog, VHDL and Verilog and SystemC/C++ language debugging windows along with assertion tracing capabilities. TCL support is provided for interaction or batch control and skin / menu customization. A unified command language is supported to provide a common set of commands for all tools, languages and environments making it easy to deploy new technology across design teams. These commands are logged for all actions in DVE and can be modified or replayed easily.

3.7.4 Mentor Graphics Questa

Questa offers built in support for testbench automation, coverage-driven verification, assertion-based verification (ABV), and transaction-level modelling (TLM) [26]. Questa provides support for SystemVerilog, VHDL, PSL, and SystemC in a single-kernel verification solution. Questa is targeted at mixed language flow. Questa enables designers to choose the languages that best address their needs.

3.7.5 Cadence Incisive Design Team Simulator

Cadence Incisive Design Team Simulator [27] provides testbench creation, reuse and analysis capabilities. These capabilities are used to verify designs from the system level, through RTL to the gate level. The environment supports a coverage driven methodology from verification planning to closure. Incisive Design Team Simulator's native compiler architecture speeds the simulation of behavioural, transaction level models, RTL, and gate level models, eliminating the performance reduction in traditional simulation. It also supports industry standard verification languages and is compatible with the Open Verification Methodology.

3.8 SystemVerilog Growth within the Verification Industry

While researching SystemVerilog a very useful survey was found [28]. The survey is interesting as it shows the growth that SystemVerilog has made in the design and verification industry since its development. The survey was performed at the DVCON conference [28], and was based on 665 responses given. The results of the survey are shown below.

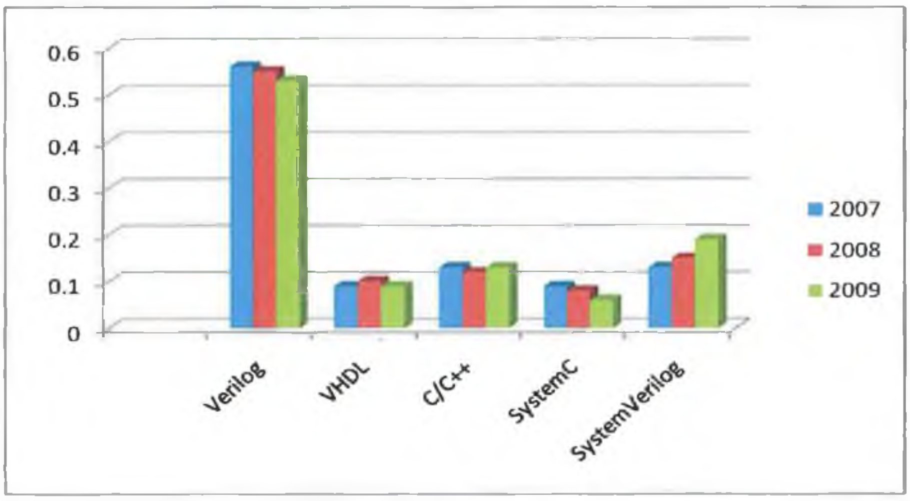


Figure 3.6 Design language usage

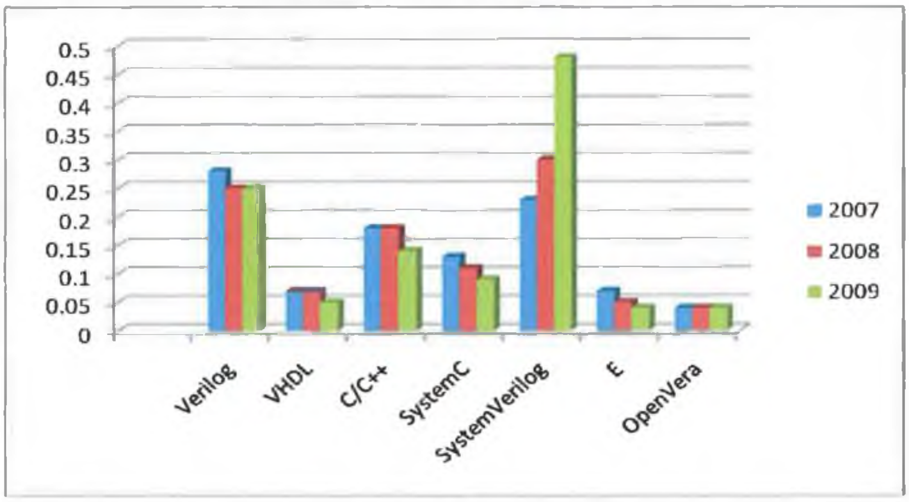


Figure 3.7 Verification language usage

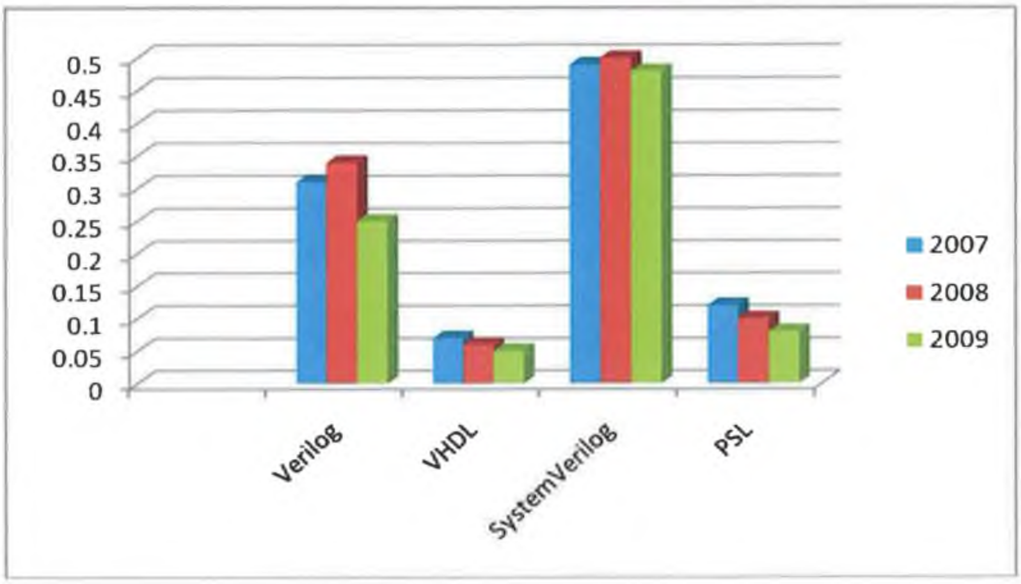


Figure 3.8 Assertions language usage

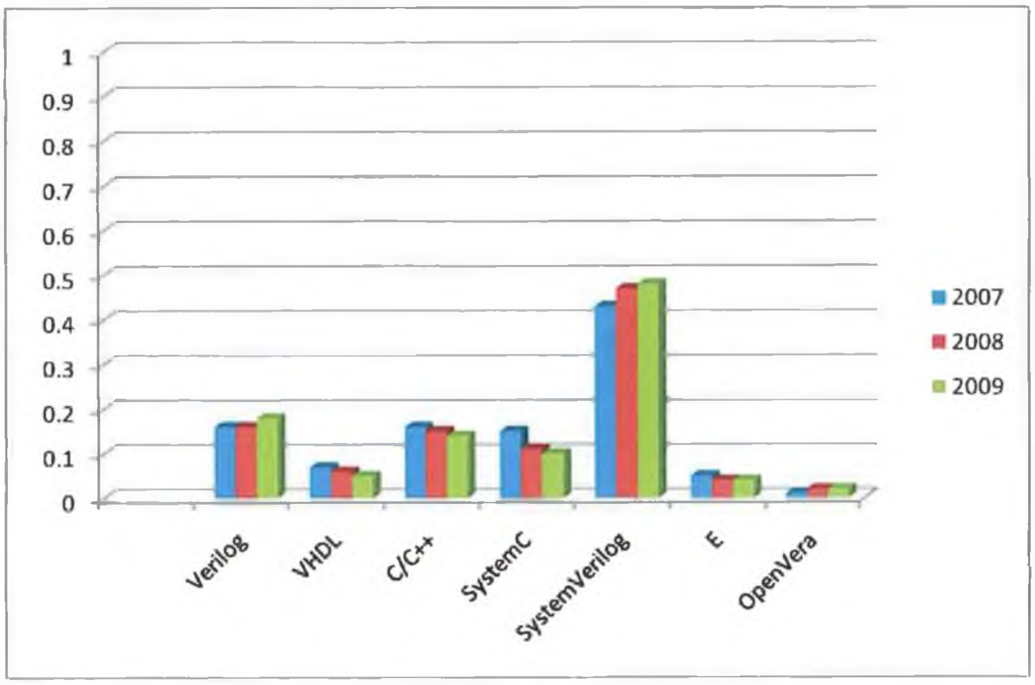


Figure 3.9 Predicted verification language engineers usage

The results of the survey indicate that usage of SystemVerilog has increased in both design and verification.

3.9 Conclusion

This chapter explained what SystemVerilog is and the extensions added to the Verilog language. Also described in this chapter are the methodologies that support SystemVerilog.

Along with significant growth in SystemVerilog, there have also been changes in the methodologies that support SystemVerilog.

- VMM released 2005
- AVM released 2005
- OVM released 2007
- VMM made open source
- VMM 1.2 released capturing many of OVM features
- Interoperability created for OVM/VMM

While SystemVerilog and the methodologies that support the language offer powerful techniques for verification, they don't however offer a standardised way of building test benches. For this reason the main motivation of the thesis was to develop a standardised test bench. The standardised testbench would incorporate the key features of both SystemVerilog and the methodologies.

- Assertions
- Constrained random test generation
- Functional Coverage
- Reference Model / Scoreboard
- Reporting Mechanism (URG)
- VMM planner

Chapter 4 VMM Based Verification Environment

4.1Introduction

In chapter three, SystemVerilog and the new features that it offers for verifying chip designs were discussed. Also VMM and the features that it provides for verification have also been addressed. This chapter will discuss building a verification environment using SystemVerilog and VMM. An IEEE-754 compliant floating-point adder model [28] has been used as a case study to implement the key features of both SystemVerilog and VMM. The reason for choosing a floating-point adder model is that it offers a robust design that can incorporate data intensive testing. The key features that were implemented are listed below and a brief description of each is given.

Interface: Provides the connections between the environment and DUT

Transaction: A transaction is an atomic data unit that is directly or indirectly applied to the DUT. Transactions are class derivatives of the vmm_data class. They extend the vmm_data class either directly or indirectly through various layers of class hierarchy. Figure 4.1 illustrates a floating-point transaction.

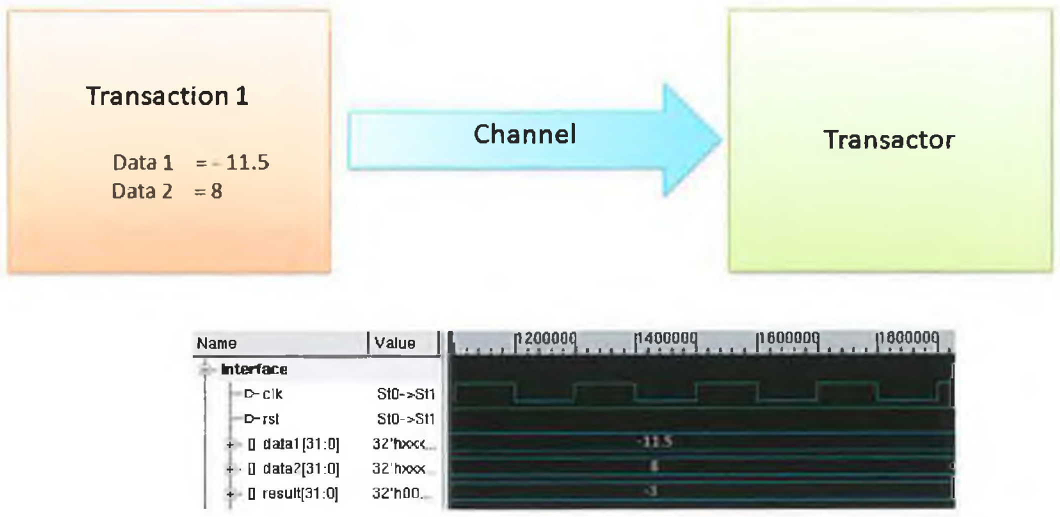


Figure 4.1 Example of a Floating-point Transaction

Transactor (BFM): A transactor is a static component of the verification environment. Transactors extend the vmm_xactor class either directly or indirectly. Transactors are components such as generators, drivers, monitors, checkers, and scoreboards, i.e. models that act on the data transactions that travel through the verification environment. The BFM is explained further in section 4.6.

Verification Environment: This is the entire top-level verification environment structure that embeds all transactors and any other components relevant to the verification environment for the DUT.

4.2 Device Under Test (Floating-point Adder Model)

The DUT is a single precision (32 bit) floating-point adder [29]. The DUT adds the two inputted numbers, returns a result and asserts the necessary status flags for that operation on each clock cycle

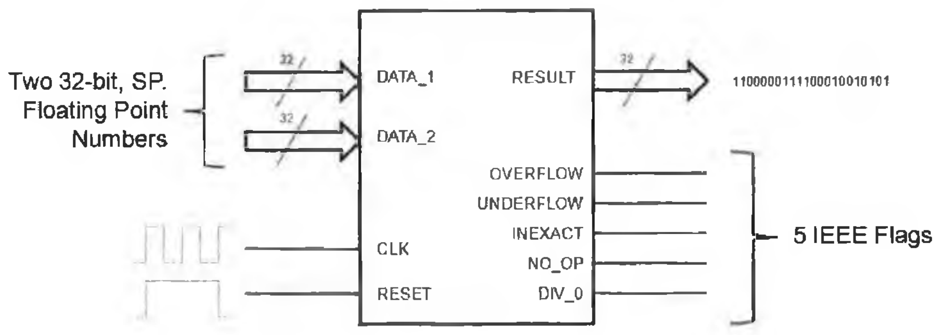


Figure 4.2 Single Precision Floating-point Adder Model

Figure 4.2 illustrates the single precision floating-point adder model or device under test in the testbench. A brief description of inputs and outputs of the model are listed below:

DATA_1 & DATA_2 –The two single precision floating-point numbers to be added.

RESULT – The outputted single precision floating-point number result.

CLK – Clock signal used to synchronise inputs and outputs through the device.

RESET – Active high reset, puts all zeros on result when active.

5 IEEE Flags – Exception flags, to show error conditions:

- **Inexact: Set** if the rounded result is not exact or if it overflows with no overflow trap.
- **Overflow: Set if the** operation produces a result that exceeds the range of the exponent
- **Underflow: Set if the** operation produces a result that is too small to be represented as a normal number.

- **Division by zero (DIV_0): Set** if the divisor is zero and the dividend is a finite non zero number

- **Invalid operation (NO_OP):** This flag is signalled if an operand is invalid for the operation to be performed. The result when the exception occurs is a quiet Nan [ref].

  Invalid operations are:

  - Any operation on a signalling Nan.
  - Addition or subtraction of infinities.
  - Multiplication of zero by infinity.
  - Division of $0 / 0$ and $\propto / \propto$
  - Remainder … x REM y where y is zero or x is infinite.
  - Square root if the operand is less than zero.
  - Conversion of a binary floating-point number to an integer or decimal format when infinity, overflow or Nan precludes a representation in that format and this cannot otherwise be signalled.
  - Comparison by way of predicates involving < or >, without ?, when the operands are unordered

4.3 Designing a Verification Environment

SystemVerilog offers many new features and methods for verifying chip designs. This section describes the design of a verification environment that can incorporate these features. Figure 4.3 illustrates such an environment. Sections 4.4 to 4.8 describe how each of the features illustrated are implemented within the verification environment.

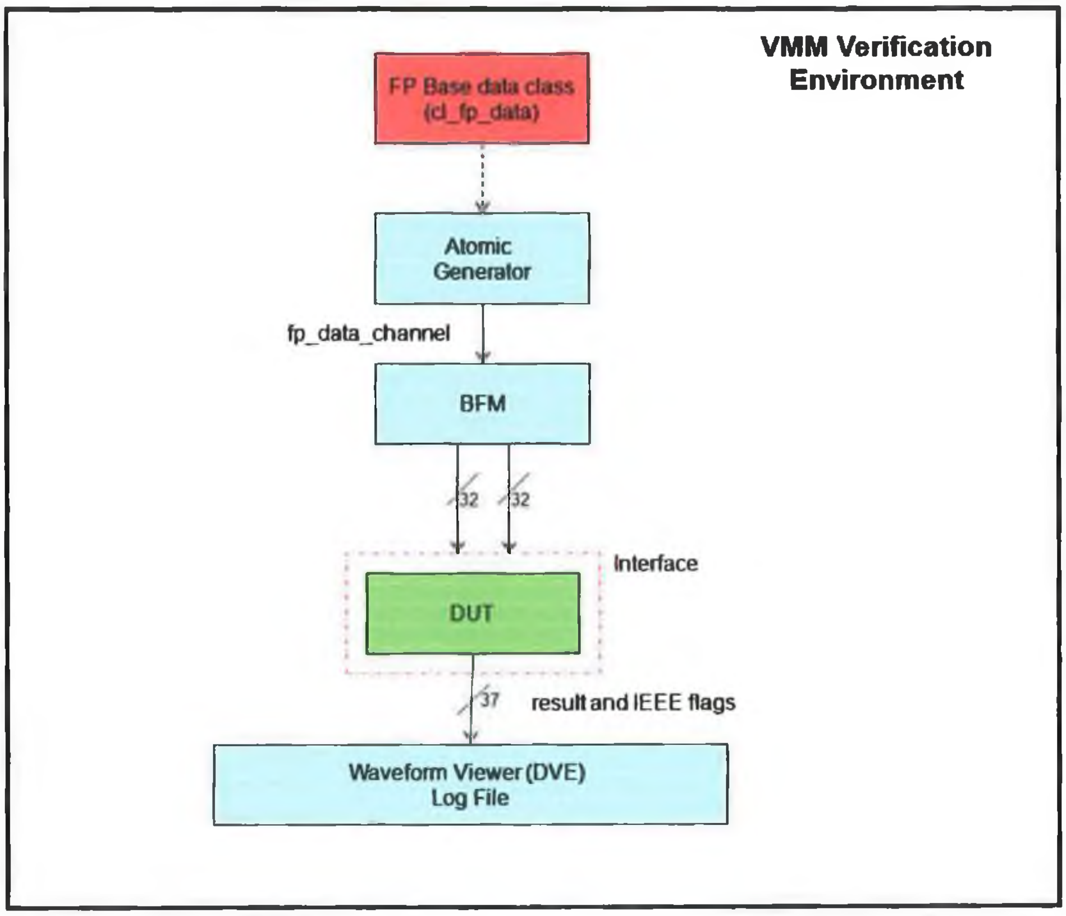

VMM Verification Environment
FP Base data class (cl_fp_data)
Atomic Generator
fp_data_channel
BFM
32
32
Interface
DUT
37
result and IEEE flags
Waveform Viewer (DVE) Log File

Figure 4.3: Basic Verification Environment

4.4 Floating-point Adder Interface

4.4.1 Introduction

SystemVerilog includes a powerful feature called the interface. The interface encapsulates the interconnection and communication between blocks. The signals declared within an interface can be input, output or inout in direction.

4.4.2 Building an Interface for the Floating-point Adder

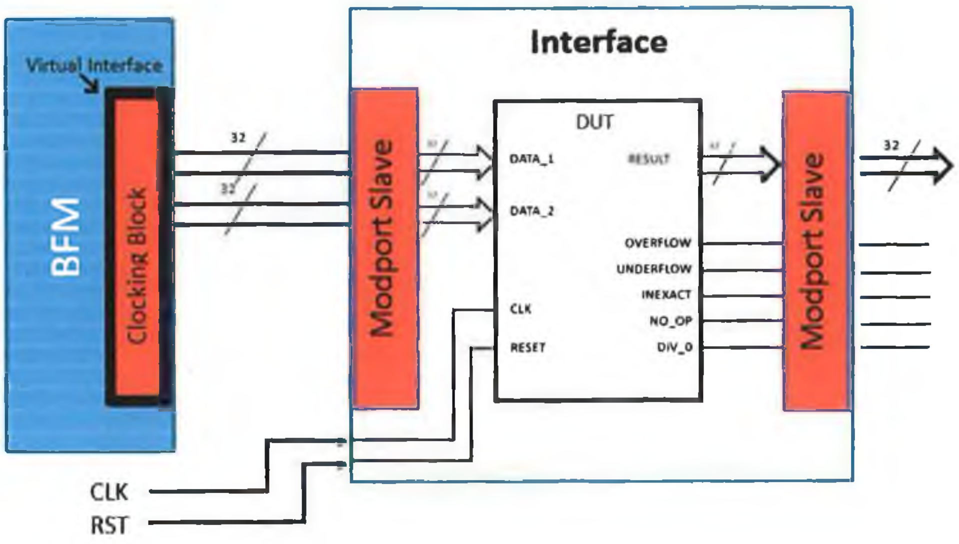


Figure 4.4 Interface for Floating-point Adder Testbench

In the first part of the interface the clock and the reset signals are declared. Figure 4.5 shows how this was done. The clock signal is used to synchronise all signals to and from the DUT. The reset is a master reset for the DUT and is low active.

```
interface fp_adder_if(input clk, input rst);
```

Figure 4.5 Declaring Clock and Reset

The next step in the interface is the declaration of the data signals and IEEE flags that are used within the DUT.

```
logic [31:0] data1;
logic [31:0] data2;
logic [31:0] result;
logic invalid_op_flag;
logic underflow_flag;
logic overflow_flag;
logic divide0_flag;
logic inexact_flag;
```

Figure 4.6 Declarations of Signals

Figure 4.6 illustrates that data 1 and data 2 are declared as logic 31 down to 0 (32 bit). These represent the 2 floating-point number inputs. The result signal is also a 32-bit number and represents the result of adding data 1 and data 2. The 5 IEEE flags are also declared; these deal with error handling for five exceptions that can occur within the floating-point standard [29].

4.4.3 What are Modports?

Modports are used to specify the direction of the signals with respect to a module that uses an interface rather than port lists. Modports restrict interface access within a module based on the direction declared.

4.4.4 Modports used in the Floating-point Interface

The modport implemented in the interface is shown in figure 4.7. The modport mp_slave is connected to the wrapper.

```
modport mp_slave (
                    input data1,
                    input data2 ,
                    output result,
                    output overflow_flag,
                    output underflow_flag,
                    output inexact_flag,
                    output divide0_flag,
                    output invalid_op_flag);
```

Figure 4.7 Modports within Interface

4.4.5 Wrapper

The wrapper is used to hook up the signals declared within the interface to the corresponding DUT pins.

4.4.6 Clocking Blocks

SystemVerilog interfaces also include clocking blocks. Clocking blocks are used to identify clock signals, and capture the timing and synchronisation requirements of the blocks being modelled. A clocking block assembles signals that are synchronous to a particular clock, and makes their timing explicit.

4.4.7 Implementing Clock Blocks within the Interface

Two clocking blocks, cb_master and cb_slave, are declared in the interface and are shown in figure 4.8.

```
 clocking cb_master @ (negedge clk);

     default input #setup_time output #hold_time;

     output data1, data2;

     input result, overflow_flag, underflow_flag, inexact_flag,
divide0_flag, invalid_op_flag ,data1_align,data2_align;

endclocking : cb_master

 clocking cb_slave @ (posedge clk);

     default input #setup_time output #hold_time;

     input data1, data2;

     output result, overflow_flag, underflow_flag, inexact_flag,
divide0_flag, invalid_op_flag,data1_align,data2_align;

    endclocking : cb_slave
```

Figure 4.8 Clocking blocks

Explanation of cb_master

The first line declares a clocking block called cb_master that is to be clocked on the negative edge of the signal clk. This clocking block is used in the BFM to drive information onto the DUT. The second line specifies that by default all signals in the clocking block use a 10ns input skew and a 2ns output skew by default. The next line adds two output signals to the clocking block data 1 and data 2. The last line adds input signals for the result and the 5 IEEE flags

4.4.8 Assertions

The last step in the interface is declaring assertions. Assertions are useful for verifying properties of a design that manifest themselves over time. An example of an assertion is shown in figure 4.9.

```
property p2;
      @(posedge clk) invalid_op_flag != 1;
endproperty

assert property (p2);
```

Figure 4.9 Example of a SystemVerilog Assertion

The assertion listed in figure 4.9 checks simply looks for a high signal on the invalid operation flag on the DUT and asserts if this condition is met.

4.5Data Class

4.5.1 Introduction

As mentioned in the introduction VMM has been used to build the floating-point adder test bench. Using this methodology a number of classes in the testbench are derived from VMM classes. The first of these classes is the data class.

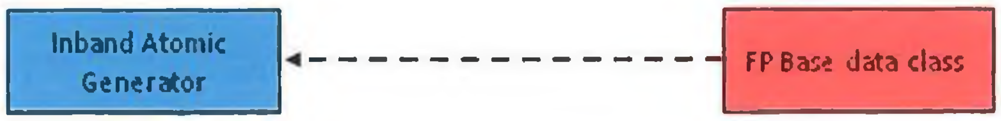


Figure 4.10 Data class

The data class (cl_fp_data) is derived from the vmm_data base class (figure 4.10). As a base class, it provides a set of methods and properties that is common to all derived transaction classes in the floating-point test-bench. This is an advantage, because it provides a consistent way of using common methods throughout the testbench.

4.5.2 Declaring Floating Data and Flags

The data class is responsible for declaring the floating-point numbers and flags. The floating-point numbers are declared using a struct. Using this struct it is possible to split the numbers into sign exponent and mantissa as shown in figure 4.11.

```
// define variables
rand struct packed {
      logic sign;
      logic [7:0] exp;
      logic [22:0] man;
      } data1 ;
```

Figure 4.11 Declaring Data 1 and Data 2

4.5.3 Using VMM Data Methods

VMM Log

The first method used from the vmm_data base class was vmm_log, as shown in figure 4.12. This is used because it provides message logging, message filtering and error counting facility.

```
static vmm_log log = new ("cl_fp_data", "class");
```

Figure 4.12 Declaring VMM log

Allocate Method

The allocate method (figure 4.13) is used to allocate an instance, which is of the same type as the object. It serves effectively as a virtual constructor, in contrast to the built in class constructor which is not a virtual method.

```
function vmm_data cl_fp_data::allocate();

 cl_fp_data i = new(); // Allocate a new object of this type
 allocate = i; // and return a handle to it

 endfunction
```

Figure 4.13 Allocate Method

psdisplay()

The psdisplay method (figure 4.14) displays the current value of the transaction or data described by this instance in a human-readable format on the standard output.

```
function string cl_fp_data::psdisplay(string prefix);
 $sformat(psdisplay, "data1 = %x, data2 = %x flags =
%x);
endfunction
```

Figure 4.14 ps_display() method

Copy method ()

The copy method (figure 4.15) is intended to implement a deep copy of the transaction. This includes a copy of all primitive as well as complex data members. The copy is returned from this function. The copy method takes a single argument of type vmm_data. The return type of the copy function is vmm_data. This implies that the return of the copy method may need to be cast back to the derived class type.

```
function vmm_data fp_data::copy(vmm_data to);

      fp_data cpy;

      if (to == null)
            cpy = new();

      else if (!$cast(cpy, to)) begin
            `vmm_error(this.log, "Cannot copy instance");
            copy = null;
            return;
      end

      copy_data(cpy);
      copy = cpy;

endfunction
```

Figure 4.15 Copy Method

4.5.4 VMM Channel

The next step inside in the data class is the creation of the vmm_channel. The vmm_channel provides the structure to store the transactions and also provide the support to process those transactions. One side of the channel is the generator putting the transaction onto the channel and the other is the transactor taking the transaction off the channel. Figure 4.16 illustrates is this process.

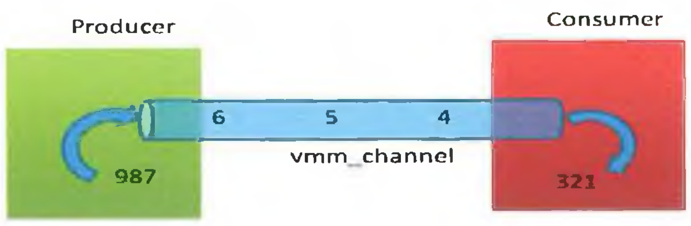


Figure 4.16 vmm_channel

To facilitate the implementation of the channel, VMM creates a derived class from cl_fp_data using the vmm_channel. The macro (figure 4.17) creates the class cl_fp_data_channel that provides a strongly typed queue to help prevent error coding.

```
`vmm_channel(cl_fp_data)
```

Figure 4.17 Creating Data Channel

4.5.5 Atomic Generator

The next step inside the cl_fp_data is creating the atomic generator. The reason for using the atomic generator is to randomise the transactions. The generator can accomplish this as follows:

- Instantiate a blueprint of the object to be generated
- Construct the object and randomise it
- Push the object to the output channel so that it can be taken off the channel later by a transactor which in the case of our testbench is the bus functional model (BFM).

Figure 4.18 illustrates the atomic generator.

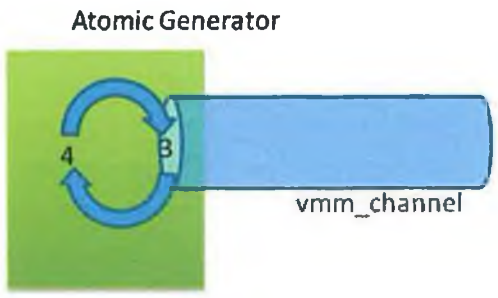


Figure 4.18 Atomic Generator

To support this feature in an automatic manner, vmm provides a macro vmm_atomic_gen (figure 4.19) for the creation of a generator class for atomic (purely random) generation of transactions. This macro creates the class fp_data_atomic_gen. When the atomic generator is started, the transactions are auto generated, randomised and put into the channel for extraction later by the BFM.

```
`vmm_atomic_gen(cl_fp_data, "Floating-point Atomic Generator")
```

Figure 4.19 Creating Atomic Generator

4.5.6 Declaring Constraints

Also within the data class, constraints are declared. An example of a constraint used in testing the floating-point adder is shown in figure 4.20.

```
constraint c1{
  data1 == 32'b11000001001110000000000000000000;  // -11.5
  data2 == 32'b01000001000000000000000000000000;  //   8
}
```

Figure 4.20 Constraint

The reason constraints are used, is to constrain the data that is being generated to a specific range of numbers. In the constraint above, data 1 and data 2 are constrained to -11.5 and 8 respectively. This constraint is used simply to check if the DUT can add 2 floating-point numbers correctly.

Figure 4.21 illustrates a constraint where the sign of data 1 and data 2 are constrained to be positive or negative for a certain number of transactions.

```
constraint con1{
            data1.sign dist {1'h0 := 10000, 1'h1 := 10000};
            data2.sign dist {1'h0 := 10000, 1'h1 := 10000};
}
```

Figure 4.21 Constraining Sign

4.6 Bus Functional Model (BFM)

4.6.1 What is a BFM?

The next class in the testbench is the BFM. The BFM is basically a transactor class and is derived from a VMM base class called vmm_xactor. All transactors in a testbench should be extended from vmm_xactor. The BFM is responsible for extracting the transaction from the channel and passing the transaction into the vector sequence used by the DUT. The transactors are the workhorses of a transaction based verification (TBV) environment, they perform the actual job of transferring the data to other units to perform a task such as driving the DUT pins, or driving the verification environment. The transactor needs to communicate the generated vectors onto signals. To do this the signals that are needed for the transactor class are defined into a virtual interface.

4.6.2 What is a Virtual Interface?

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design [31]. A Virtual interface allows the same subprogram to operate on different portions of a design, and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals.

4.6.3 Benefits of using Virtual Interface

1. The Virtual interface can be used to make the testbench independent of the physical interface. It allows development of the test component independent of the DUT port while working with the multi-port protocol.
2. With the virtual interface, it is possible to change references to the physical interface dynamically. Without virtual interfaces, all the connectivity is determined during compilation time, and therefore cannot be randomised or reconfigured.

4.6.4 Building the BFM

As with many VMM base classes there is a rich set of features available. The following are the methods that are most commonly used in vmm_xactor class derivatives.

New Method ()

The purpose of the constructor, or new method, is to allocate space for the various data structures used by the vmm_xactor class and, if necessary, to initialise these data structures with meaningful values. Figure 4.22 illustrates the new method within the floating-point adder BFM.

```
function cl_fp_bfm::new (cl_fp_cfg cfg,
                         string instance,
                         integer stream_id,
                         virtual fp_adder_if vi,
                         tp_kind kind,
                         cl_fp_data_channel in_chan);
```

Figure 4.22 vmm_xactor constructor

The constructor of the vmm_xactor class requires two string arguments and an optional integer stream_id. The two string arguments are used to set the name and instance of the vmm_log property while the stream_id is used to set the stream_id property of the vmm_xactor. Any extension of the constructor must start with a call to the constructor of the parent class (a call to super.new()).

The vmm_channel which was talked about earlier in the data class section is instantiated in the BFM and creates an object called in_chan that will be used throughout the BFM to refer to cl_fp_data_channel.

Main method ()

The main method holds the code associated with the main functionality of a specific vmm_xactor, typically one or more processes which run more or less constantly during a simulation. These processes may be temporarily paused by the user. The purpose of the start, stop and wait_if_stopped methods is to enable/disable this pausing which will be described further in section 4.7. The main method is a method that must be defined not as a matter of principle but as a matter of functionality: without an extended main method, the extended vmm_xactor will do nothing. Any derived class extension of the main method must have a fork-join_none call to the base class implementation.

do_master ()

The do_master task is responsible for putting the data onto the virtual interface. Also vmm_debug statements are created to identify individual transactions which will be printed out into log files during simulation. How this task was implemented can be seen in figure 4.23.

```
task fp_bfm::do_master(fp_data tr,int unsigned id_num);

  string st_2, st_3, st_4;

  vi.cb_master.data1  <=  tr.data1;

  vi.cb_master.data2  <=  tr.data2;

   $sformat(st_2, "DATA ID NUM :: %d ", id_num);
  `vmm_debug(log, st_2);
   $sformat(st_3, "        TR.data1 = %b ", tr.data1);

  `vmm_debug(log, st_3);
   $sformat(st_4, "        TR.data2 = %b ", tr.data2);

  @(vi.cb_master);

endtask
```

Figure 4.23 do_master task

4.7 The Environment

The final class in the testbench is the environment class. This class is used to implement the verification environment. The Verification environment is developed by extending the vmm_env class. VMM provides a single top-level container class to encapsulate and build the environment and to control the simulation run flow as seen in figure 4.24. This class is the vmm_env base class. The vmm_env base class provides a series of virtual methods each with a particular role in the construction and execution of the environment. The next section describes each of these methods and their intended usage.

4.7.1 Methods within VMM Env

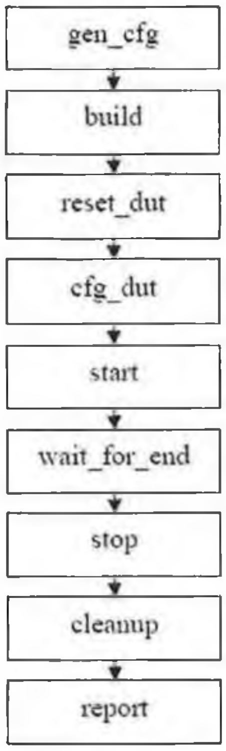


Figure 4.24 The Simulation Flow

virtual function void gen_cfg();

The 'gen_cfg' method (figure 4.25) is intended to generate a randomised environment configuration descriptor.

```
function void cl_fp_env::gen_cfg();

      super.gen_cfg() ;

endfunction
```

Figure 4.25 gen_cfg' method

virtual function void build();

The 'build' methods (figure 4.26) are used to instantiate and interconnect all the static components of our environment, i.e., the transactors, as per the environment configuration descriptor.

```
function void cl_fp_env::build();
      super.build() ;
      this.in_chan = new("Data channel","in_chan",100);
      this.gen_chan = new("Atomic Gen",1,in_chan);
      this.bfm_master = new("bfm_master", 200, dutif_fp,
      cl_fp_bfm::MASTER, in_chan);
      this.gen_chan.stop_after_n_insts = 10000;
endfunction
```

Figure 4.26 build method ()

`virtual task reset_dut();`

The 'reset_dut' method (figure 4.27) is used to reset the DUT so it is ready to be configured.

```
task automatic cl_fp_env::reset_dut();
      super.reset_dut();
      top.rst <= 1'b0;
      repeat (5) @(posedge top.clk);
      top.rst <= 1'b1;
      `vmm_note(log, "Reset has been toggled in the Environment
      file  ");
endtask
```

Figure 4.27 reset_dut() method

`virtual task cfg_dut();`

The 'cfg_dut' method (figure 4.28) is used to apply the DUT configuration descriptor to the DUT. This would generally be done by writing any relevant DUT registers and driving relevant pins required for DUT configuration.

```
task automatic cl_fp_env::cfg_dut();
      super.cfg_dut() ;
endtask
```

Figure 4.28 cfg_dut method

`virtual task start();`

The 'start' method (figure 4.29) is used to start the environment transactors. This task should typically call the vmm_xactor::start_xactor method of each transactor to start the transaction generation and execution flow throughout the environment. Sometimes it may be necessary to start some transactors before the start method, for example a driver to configure the DUT through a register interface.

```
task automatic cl_fp_env::start();

      super.start();
      // Start generators, now we run random !
      gen_chan.start_xactor();
      bfm_master.start_xactor();
endtask
```

Figure 4.29 start method

`virtual task wait_for_end();`

The 'wait_for_end' (figure 4.30) task is used to control when this test should end. Its completion signifies the end of the test and hence it should be designed to wait for the conditions that signify the end of the test. It is up to the user to model these end conditions. An example implementation may wait for the scoreboard to compare a configurable number of packets successfully, or for a predetermined timeout, whichever comes first.

```
task automatic cl_fp_env::wait_for_end();

      fork
      begin
            super.wait_for_end();

            this.gen_chan.notify.wait_for
                  (cl_fp_data_atomic_gen::DONE);
            #2000; // Time to run after the transactor stops
      end

      begin
            this.gen_chan.reset_xactor();
      end
      join
endtask
```

Figure 4.30 wait for end () method

`virtual task stop();`

The 'stop' method (figure 4.31) is used to stop the transactors in the environment as required in particular transactors like the atomic generators, scenario generator and drivers. Other transactors like monitors or the scoreboard need not be stopped but should be for good practice. Methods like 'wait_if_stopped' or 'wait_if_stopped_or_empty' will block and hence suspend the execution thread of its transactor while the transactor is stopped.

```
task automatic cl_fp_env::stop();

      super.stop();

      // Stop generators

      gen_chan.stop_xactor();
      bfm_master.stop_xactor();

      #300;  // Wait for all bus traffic
to end.

endtask
```

Figure 4.31 stop method ()

`virtual task cleanup();`

The 'cleanup' method (figure 4.32) is used to allow the simulation to end cleanly. This typically includes waiting for the DUT to finish processing transactions still in flight, tidying up scoreboards, etc.

```
task automatic cl_fp_env::cleanup();

      super.cleanup() ;

endtask
```

Figure 4.32 cleanup () method

`virtual task report();`

Last but not least, the 'report' method (figure 4.33) is used to report the success or failure of the simulation. The base class implementation makes a call to the log::report method which provides a PASS/FAIL status based on the presence of errors and warning messages issued throughout the simulation.

```
task automatic cl_fp_env::report() ;

      super.report() ;

endtask
```

Figure 4.33 report () method

4.8 Program and Top File

4.8.1 Program file

The Program block contains the instance of the testbench environment class. The program calls, the build () and run () methods of environment. Figure 4.34 illustrates the program file for the floating-point adder environment.

```
program pg_tb
             #(
                input  bit clk,          // clock
                output bit   rst,        // reset
                fp_adder_if adder_if
);

      cl_fp_env fp_env;       // TB Environment

      initial begin

            fp_env = new(top.fp_if);

            fp_env.build();   // Build the environment

            fp_env.run();       // Run all steps

      end

endprogram
```

Figure 4.34 Program File

4.8.2 Top File

The top-level module contains the design and testbench instance. The Top module also contains the clock generator. There is no need to instantiate the top module. The Testbench and DUT instances are connected in the top file. Figure 4.35 illustrates the top module for the floating-point adder testbench.

```
module top;

 fp_adder_if #(.setup_time(0),
            .hold_time(0)) fp_if(.*);

  logic rst, clk;

  pg_tb #fp_pgm (.clk(clk), .rst(rst), .adder_if (fp_if));

  initial begin
     clk = 1'b0;

     forever begin
        #100;
        clk = ~clk;
     end

  end

  // Connect up the RTL here!
  WrapperFP wrapper_i (.rst(rst), .clk(clk), .fp_if(fp_if));

endmodule
```

Figure 4.35 Top Module

4.9 Testing of Floating-point Adder Model

4.9.1Introduction

In testing the floating-point adder model a number of test cases have been written. These test cases range from simply adding two numbers, to producing thousands of random transactions. This section describes the test cases written. The VCS tool and the DVE environment described in section 3.7.3.1 have been used for testing the floating-point adder.

4.9.1.1 Test Case 1

The first test case written is the addition of 2 known numbers (-11.5 and 8). This has been done by constraining the data produced to the 2 numbers. Figure 4.36 illustrates how this was done. Also within the test case an assertion has been used to check for the result of adding the 2 numbers.

```
constraint c1{
      data1 == 32'b11000001001110000000000000000000;   // -11.5
      data2 == 32'b01000001000000000000000000000000;   // 8 = -3.5
}
```

```
property p1;
@(posedge clk) result == 32'b11000000011000000000000000000000;
endproperty
assert property (p1);
```

Figure 4.36 Constraint and Assertions

The result expected is (b11000000011000000000000000000000 or Hex C0600000). Figure 4.37 illustrates an output from the simulation seen in the DVE waveform window.

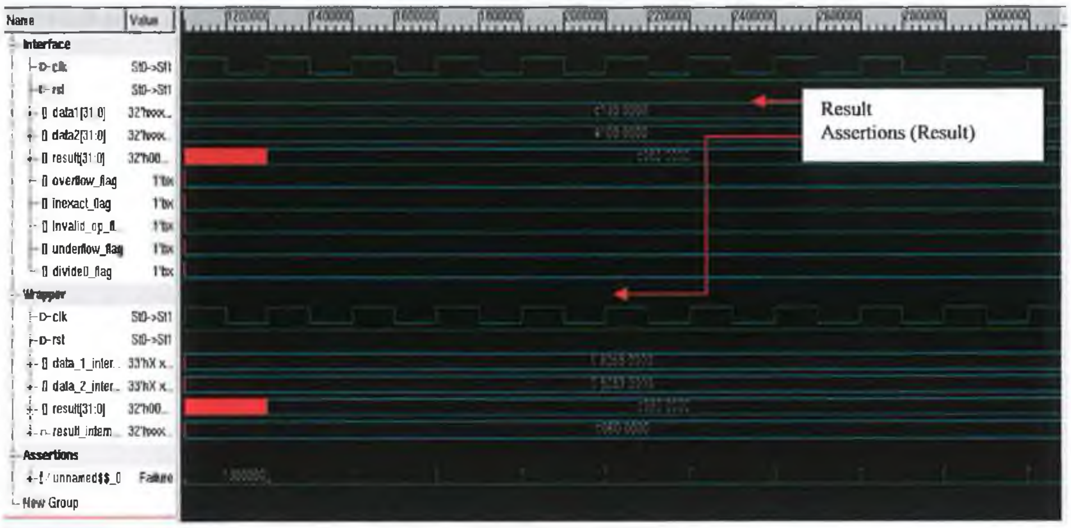


Figure 4.37 Output Testcase 1

4.9.1.2 Test Case 2

The second test case written is the addition of 2 NANs (not a numbers) together and checks for the invalid flag. Figure 4.38 illustrates the constraint and assertion used in this test case. The output of this test case is shown in figure 4.39.

```
constraint c2{
data1 == 32'b01111111110000010000000000000000;
// Checking for the invalid flag

data2 == 32'b11111111100100010010101010101010;
// by adding 2 NANs
}
```

```
  property p2;
    @(posedge clk) invalid_op_flag != 1;
  endproperty
assert property (p2);
```

Figure 4.38 Invalid Flag

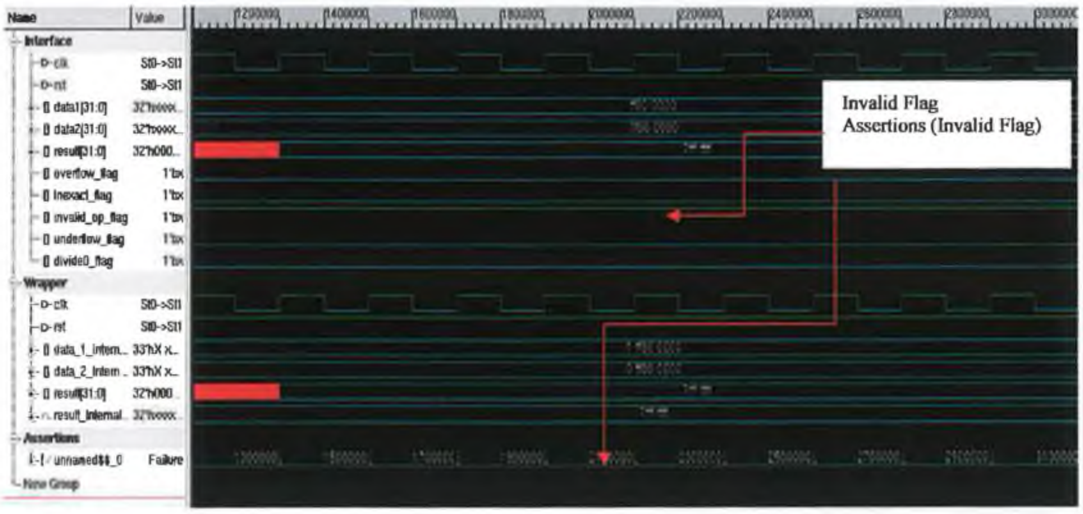


Figure 4.41 Output Testcase 3

4.9.1.4 Test Case 4

The fourth test case written produces constrained random data that is used to trigger the overflow flag. The appropriate assertion is written to check for the overflow flag and is illustrated in figure 4.42. Figure 4.43 illustrates an output from the simulation where the overflow flag and inexact flag are triggered high.

```
constraint c4{
      data1 == 32'h7e8a2ef7;
      data2 == 32'h7f3bb28c;
}
```

```
  property p4;
    @(posedge clk) overflow_flag != 1;
  endproperty
assert property (p4);
```

Figure 4.42 Overflow Flag

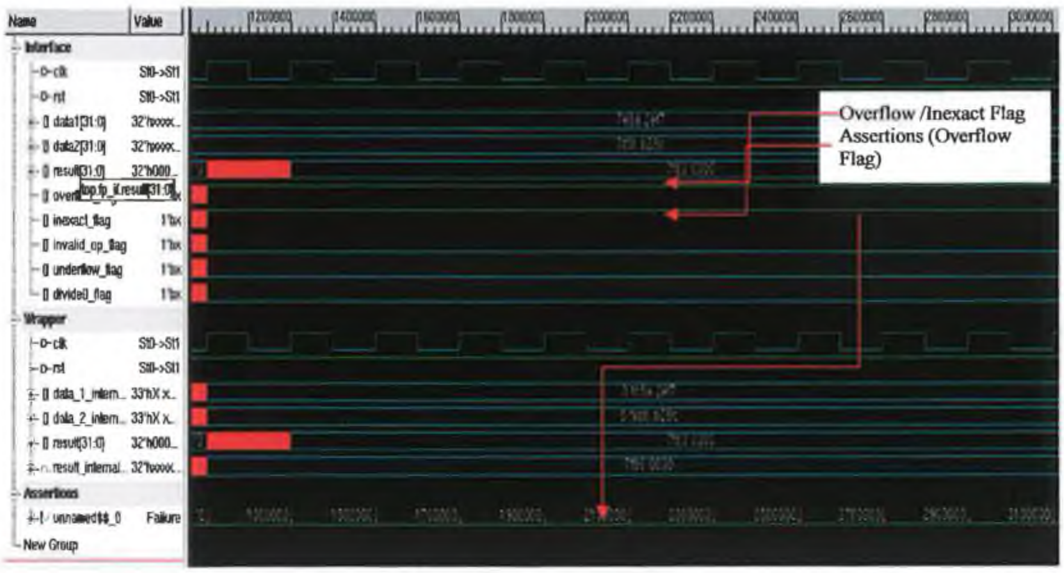


Figure 4.43 Output Testcase 4

4.9.1.5 Test Case 5

The fifth test case written produces constrained random data that would be used to trigger the underflow flag. An assertion has been used to check for the underflow flag as shown in figure 4.44 Figure 4.45 illustrates an output from the simulation, this time the underflow flag is high.

```
constraint c5{
      data1 == 32'h01562ef7;
      data2 == 32'h8143f059;
}
```

```
  property p5;
    @(posedge clk) underflow_flag != 1;
  endproperty
  assert property (p5);
```

Figure 4.44 Underflow Flag

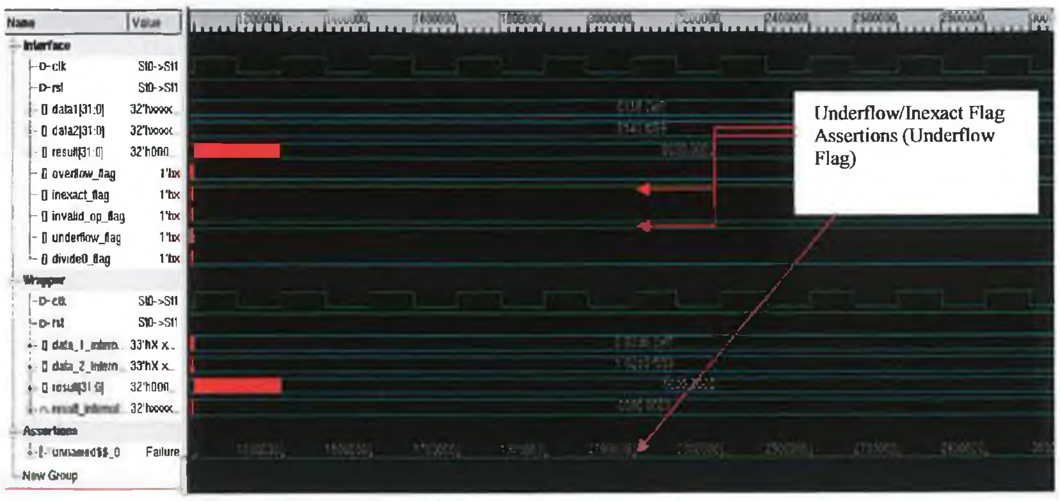


Figure 4.45 Output Testcase 5

4.9.1.6 Test Case 6

The final test case ran the simulation for 10000 transactions and records the results. As can be seen in figure 4.46 the inexact (x 2) and invalid flag(x 1) has been triggered high.

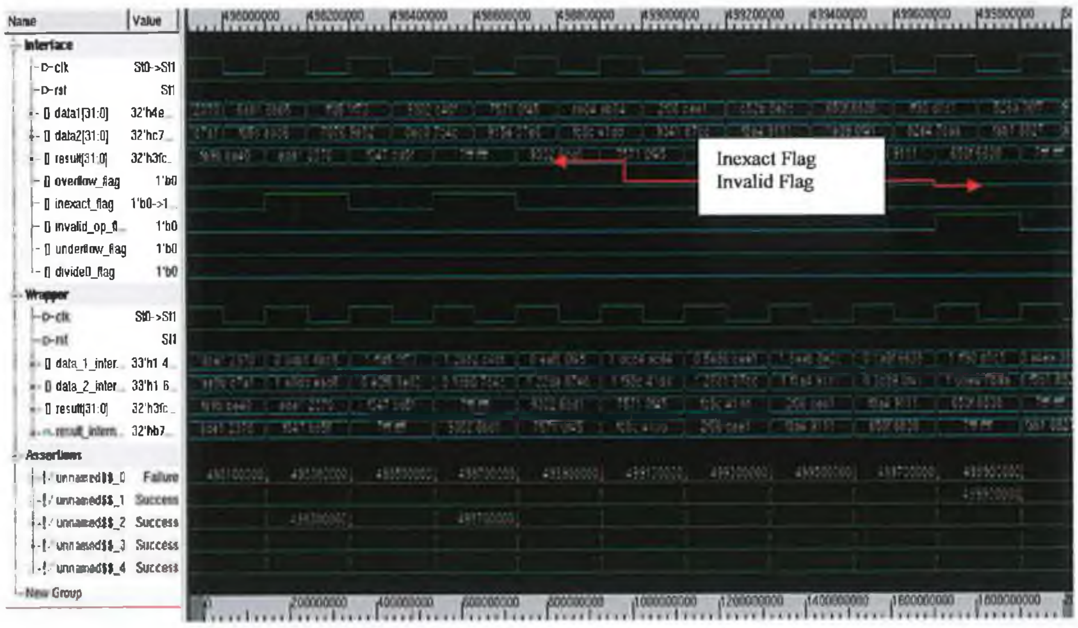


Figure 4.46 Output Testcase 6

4.10 Conclusions

The objective of this chapter was to build a SystemVerilog test bench using VMM techniques. This was accomplished using SystemVerilog and VMM features such as:

- Interface
- Data class
- Xactor class (BFM)
- Environment class
- Program and Top files
- Constraints and Assertions

Advantages of using SystemVerilog are as follows:

1) Executing large amount of random stimuli takes considerably less time than traditional tests.
2) The object-oriented nature of the SystemVerilog language, helps to quickly code up the test bench.
3) The ability to use assertions in SystemVerilog gives the user the ability to pin point exactly in simulation when assertions happen.

Advantages of VMM are as follows:

1) VMM provides a set of base classes which describe the important elements of a testbench:
    - vmm_data for data objects and/or transactions
    - vmm_xactor for the functional blocks of the testbench (transactors)
    - vmm_channel for transactor communication
    - vmm_log and vmm_notify for recording and inter-process communication.
    - vmm_atomic_gen for stimulus generation
    - vmm_env to describe the structure of a particular testbench.
2) It provides definite methods on how to used base classes.

Useful information for learning SystemVerilog and VMM include: SystemVerilog books [1][5][8][9][32], online tutorials [16] [20] and example test benches [33] [34].

Chapter 5 Advanced Verification Environement

5.1 Introduction

The previous chapter describes how the basic floating-point test bench was built using SystemVerilog and VMM. The testbench did not accommodate all of the features associated with SystemVerilog and VMM as it only has the ability to pass two 32-bit floating-point numbers through the DUT. The results of adding these numbers could be viewed in the DVE. Also the test bench did not offer the ability to verify the DUT correctly. Listed below are some of the key features of SystemVerilog and VMM that are not included in the previous testbench.

- Functional coverage
- Scoreboard
- Reference Model
- Reporting Mechanism
- Direct and Random test case library

In this chapter a solution is provided which incorporates all of these features into one environment. This environment is used to verify the floating-point adder completely. The test bench is built in such a way that it offers a reusable directory structure and a test bench architecture, where a similar design can be verified. This testbench architecture is needed because up to 70% of chip development time is spent in the verification process [5]. Therefore, any method or technique that can improve this is extremely important.

The environment also offers the user the ability to run both random and directed tests within one testbench. In addition, the environment will incorporate a reference model and scoreboard as a comparison feature. Also within the environment is a functional coverage class, which is used to analyse how much of the DUT has been verified.

Finally, the test bench offers a Unified Report Generator (URG) reporting mechanism and a new feature from Synopsys called VMM planner, a reporting mechanism at top level. The design and building of this environment is discussed in this chapter also.

5.2 Directory Structure

In order to implement all the features associated with an advanced verification environment, it is important to create a directory structure that fits these requirements. Figure 5.1 below outlines such a structure.

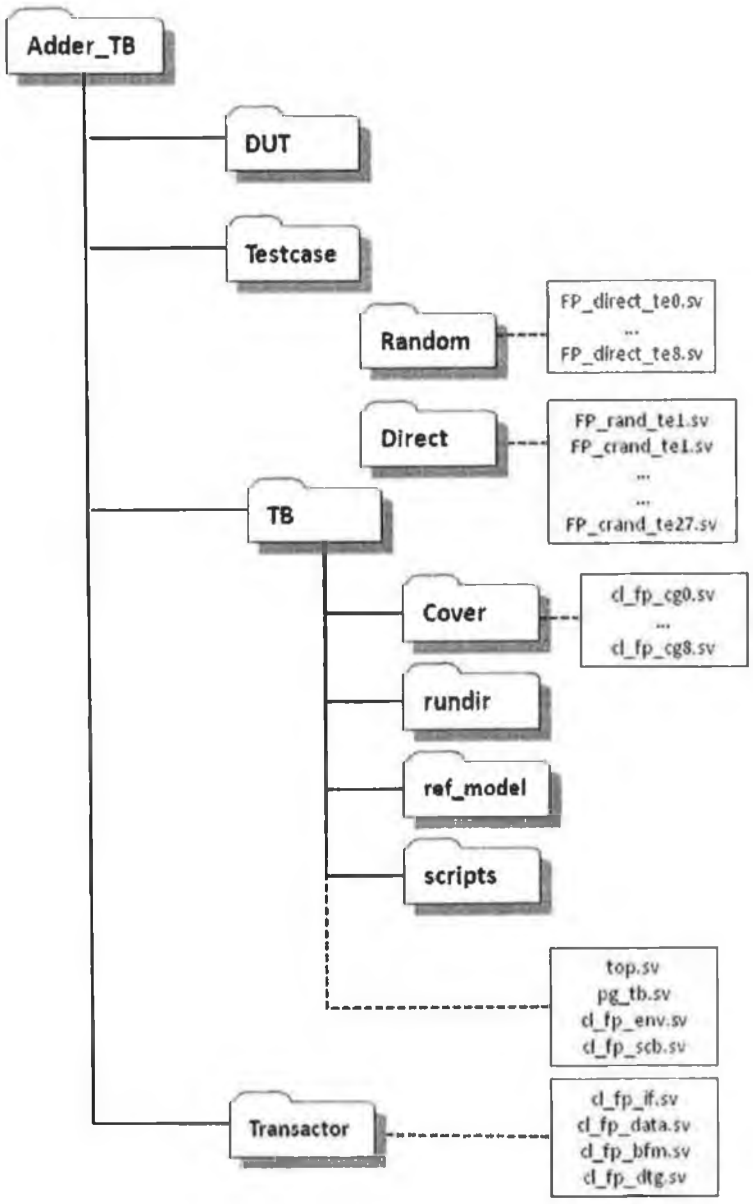


Figure 5.1 Directory Structure

The directory structure contains four main directories, the DUT, Test Bench (TB), Test Cases and the Transactor (BFM). These four main directories are sub divided into further directories and files. A description of each is given below.

DUT Directory

The DUT directory contains all files associated with the floating-point adder model discussed in section 4.2.

TB Directory

The TB directory is further subdivided into four directories:

- Coverage (functional coverage)
- Ref model (TLM reference model)
- Scripts (Scripts directory contains files that support the development of a regression flow that is needed to maximise the productivity of the solution.)
- Rundir (testbench execution directory)

The following files are also found within the TB Directory:

- Scoreboard (Compares Reference model with DUT)
- Environment class (Top level class which is used to encapsulate and build the environment and to control the simulation run flow.)
- Program File (Call methods to build the environment, also used to define types and parameters)
- Top file (Represents the top level of the test bench and instantiates the DUT)

Test Case Directory

The test case directory contains two libraries of test cases, random and direct. These test cases are written as classes and are implemented as extensions of the verification environment class (tb_env.sv).

Transactor Directory

The following files are found inside the Transactor Directory:

- Configuration class (Declares configurations)
- BFM (Extracts the transaction from the data channel and puts these transactions onto the interface connected to the DUT)
- Interface (Provides the connection between the environment and the DUT)
- Data Class (Basis for all transaction descriptors and data model)

This structure is designed in such a way that it provides the user with the ability to develop both random and directed test cases within the overall test bench architecture. This directory structure fits the requirements of the floating-point adder model but it is structured in a way that allows incorporation of another DUT where similar verification techniques are to be implemented upon it. It is a robust and reusable directory structure.

5.3 Test Bench Architecture

The testbench architecture has been created to implement the files and features within the directory structure. This architecture uses the verification environment file (tb_env.sv) to connect up these features. The test bench architecture contains each of the elements described in section 5.2. Figure 5.2 shows how each of these elements are integrated into the test bench architecture.

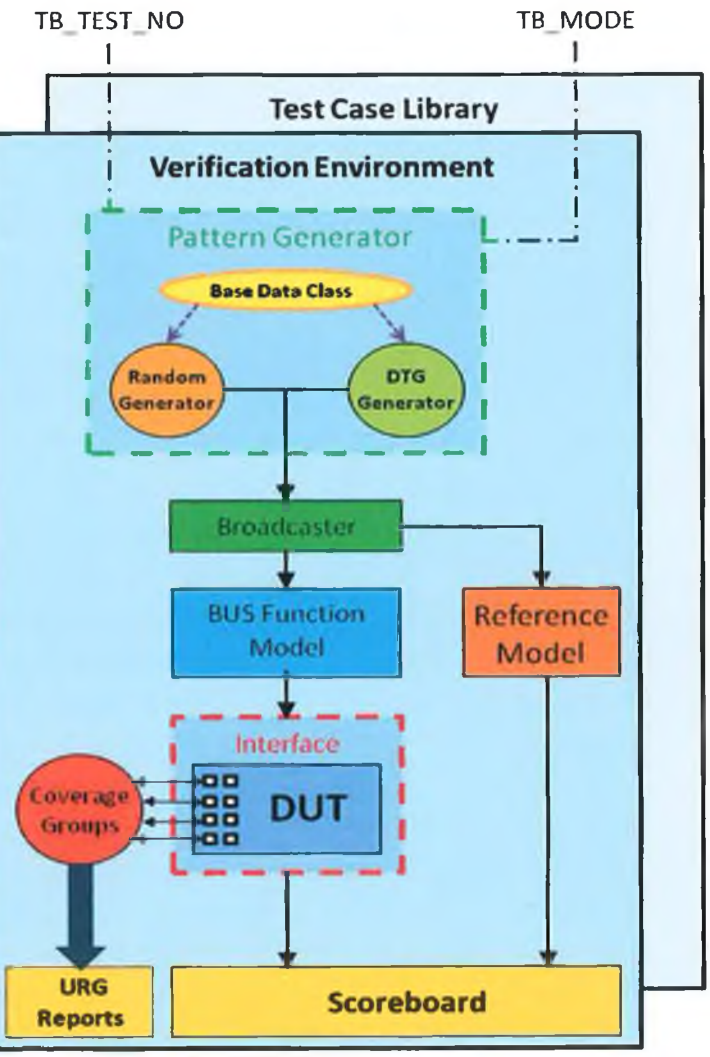


Figure 5.2 Test Bench Architecture

5.4 Reference Model

5.4.1 Introduction

A reference model is used to dynamically predict the response of the DUT. The reference model works directly with the same stimulus as the DUT and must thus produce an output in the same order as the design itself. The reference model for a particular DUT does not have to be pin accurate.

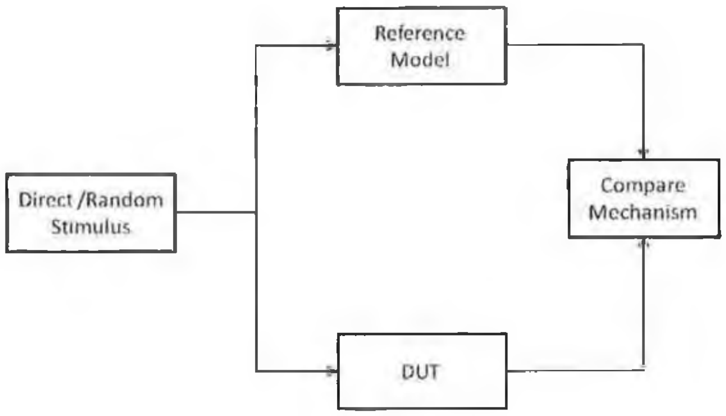


Figure 5.3 Traditional Reference model

5.4.2 Building the Reference Model

The reference model has been written in SystemVerilog. The following reasons are why SystemVerilog was chosen.

- To examine if it was possible
- To increase knowledge and experience of using SystemVerilog

In building the reference model the key functionality of the floating-point adder had to be recognised. Three files have been created to represent the floating-point adder: ref_adder.sv, add_detect.sv and ref_model.

Ref Adder

This model is responsible for implementing the major blocks of the floating-point adder.

- Swap
    - Determines the absolute value
    - Determines largest of data1 and data 2
    - Checks the sign of data 1 and data 2
- Shift
    - Shifts the mantissa of the numbers accordingly
    - Gets 2 complement if floating-point number have different sign
- Negate
    - If S is negative, replace it by its two's complement.
- Normalise
    - Tries to produce the normalised result
- Rounding
    - Represent the rounding mode of the adder

Add Detect

This file is responsible for implementing the detection system that is used within the floating-point adder. The main responsibility of the detection system, is to deal with difficult floating-point numbers.

Ref_model.sv

This file is responsible for taking transactions off the data channel. Once the data has been taken off the channel, the reference model adds the two floating-point numbers and also checks to see if any IEEE floating flags have being triggered. Once the result is calculated, the reference model sends the output back onto a data channel called ref_channel. Figure 5.4 illustrates how the result and flags are put back onto a ref_channel. This ref_channel is then connected to the scoreboard, which compares the output of the reference model with DUT.

```
tr.result          =      ref_add_detect_result[31:0];
tr.invalid_op_flag =      ref_add_detect_result[36];
tr.divide0_flag      =  ref_add_detect_result[35];
tr.overflow_flag     =  ref_add_detect_result[34];
tr.underflow_flag    =  ref_add_detect_result[33];
tr.inexact_flag      =  ref_add_detect_result[32];
```

Figure 5.4 Outputting results from Reference Model

5.43 Connecting the Reference Model up to the Test bench

To connect the reference model into the testbench, the environment (cl_fp_env.sv) file has to be modified. Figure 5.5 and figure 5.6 illustrate how this is done. A VMM feature called a broadcaster is used.

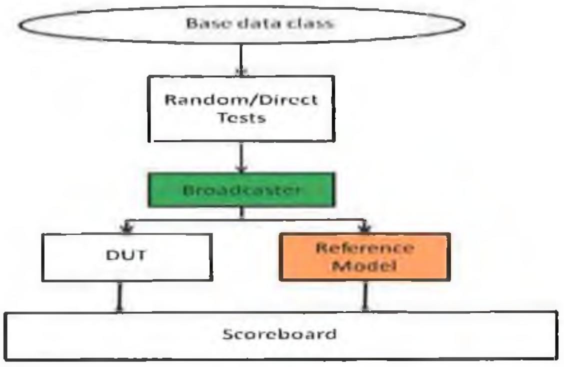


Figure 5.5 Reference Model within the Test Bench

```
// Reference model
cl_fp_ref_model ref_master;

// Broadcaster to split the data from the atomic_gen and sent on to
DUT and Ref Model

vmm_broadcast brc;

// Ref model instantiation
this.ref_master = new("ref model", 300, dutif_fp, null, null);
//  Construct the broadcaster

 brc = new ("Broadcaster","",in_chan,1);

brc.new_output(ref_master.in_chan);
```

Figure 5.6 Connecting up Reference model within Test bench Environment

5.4.4 Broadcaster

The broadcaster offers the ability to split the transactions being produced by the atomic generator [32]. In the case of testing the DUT the broadcaster is used to split the atomic generator channel and Direct Test Generator (DTG) channel between the BFM and the reference model.

5.5 Scoreboard

The Scoreboard is used to dynamically compare the output data and IEEE floating-point flags from the DUT to corresponding data from the reference model as seen in figure 5.7.

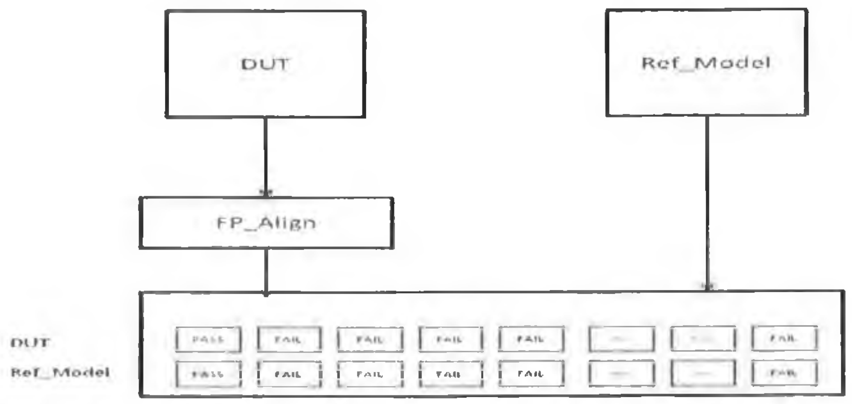


Figure 5.7 Scoreboard

This is done by a compare mechanism within the scoreboard that checks each transaction that is being processed by the DUT and the reference model. Figure 5.8 illustrates how this was implemented.

```
if(tr_dut.result == tr_ref.result) begin

      error_count = error_count; //Error count default to Zero

end
      else begin

            error_count = error_count ++;

      end
```

Figure 5.8 Compare Mechanisms within Scoreboard

The results of the compare mechanism within the scoreboard are sent to pass/fail log files for each transaction passing through the system.

This mechanism makes it easier for testing the floating-point adder model. In the case of a fail transaction, the time when the transaction failed and the number of the transaction are outputted to log files. The log files can be examined and used for debugging the DUT. When using the scoreboard it has been determined that there is an issue in the timing of the output from the reference model and the DUT. The reference model operates two clocks faster than the DUT in outputting the results. To solve this issue an alignment class has been written. The alignment class takes the output from the reference model and delays it by 2 clocks and sends it onto the scoreboard for comparison. The output of a log file from the scoreboard can be seen in figure 5.9

```
Transaction 11 failed
ref_data1 0110 ref_data2 1111 ref_result 1111
dut_data1 0110 dut_data2 1111 dut_result 1110
```

Figure 5.9 Outputs from Fail log File

5.6 Test Case Library

5.6.1 Introduction

This section highlights the ability of the testbench to use both constrained random and directed tests in the same verification environment. Also a description of random and directed tests is given.

5.6.2 Directed Tests

Traditionally, when faced with the task of verifying the correctness of a design, directed tests are written [1]. Using this approach, verification engineers examine the hardware specification and write a verification plan with a list of tests, each of which concentrates on a set of related features. The verification engineer writes stimulus vectors that exercise these features in the DUT. The DUT is simulated with these vectors and the engineer manually reviews the resulting log files and waveforms to make sure the design does what is expected. Once the test works correctly, it can be checked off the list of tests within the verification plan and the engineer moves onto the next test.

This incremental approach makes steady progress and is always popular with managers who want to see a project making headway. It also produces almost immediate results, since little infrastructure is needed in guiding the creation of a stimulus vector. Given enough time and staffing, directed testing is sufficient to verify many designs. Figure 5.10 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. Given enough time, an engineer can write all the tests needed for 100% coverage of the entire verification plan.

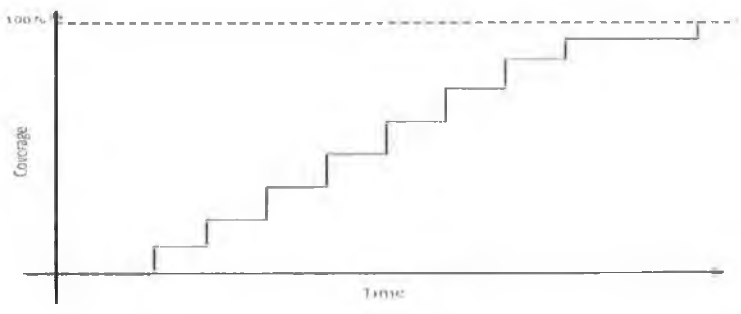


Figure 5.10 Directed Test Progress

5.6.2 Problem with Directed Tests

There are a number of problems with directed tests. If there is not enough time to carry out the directed testing approach then more engineers are required. If the design complexity doubles, managers should allow engineers twice as long to complete the process or again hire more engineers [1].

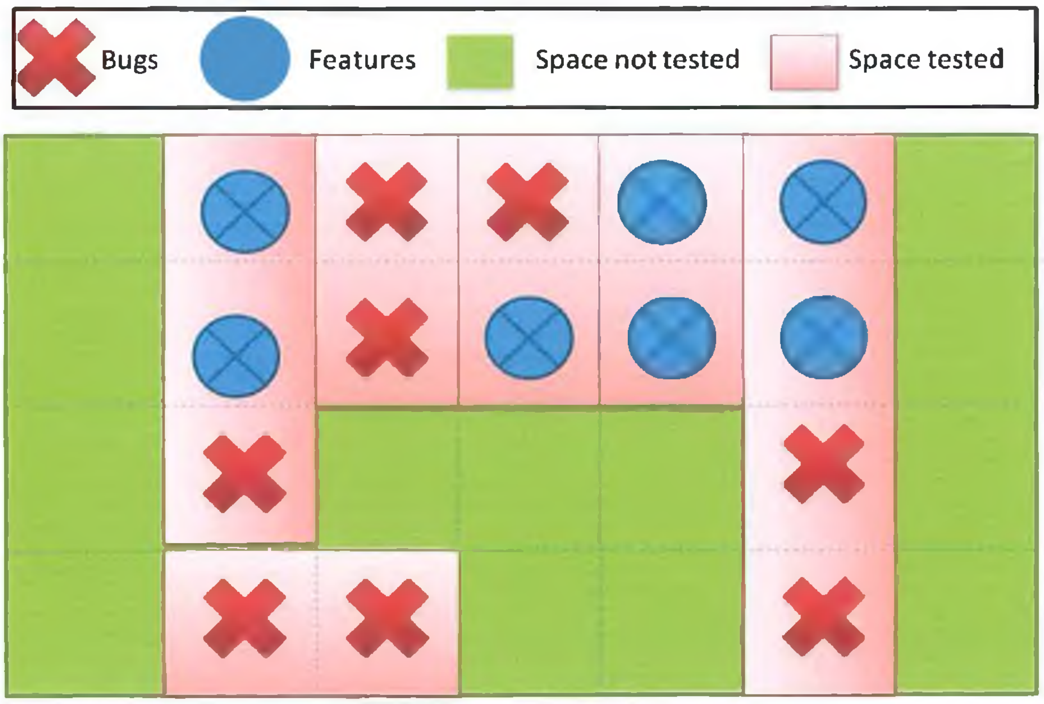


Figure 5.11 Directed Test Coverage

Figure 5.11 shows the total design space and the features covered by directed test cases. This space contains many features, some of which have bugs. The engineer must write tests that cover all the features to find the bugs. Neither of these situations is desirable. A methodology is required that finds bugs faster in order to reach the goal of 100% coverage.

5.6.3 Random Tests

Through the use of constrained random testing it is possible to greatly reduce the verification task. A random test will often cover a wider range of stimuli than a directed test [35]. A directed test finds the bugs an engineer expects to find in the design, while a random test can find bugs that he/she has never anticipated. When using random stimuli functional coverage is needed to measure the verification progress.

5.6.4 Combining Random and Directs Tests

When setting out to design the SystemVerilog testbench, the chief design goal is to facilitate the easy use of constrained random and directed tests within one environment. The verification environment developed allows the user to quickly write random or directed test cases with minimal code modifications and without any adjustment to the verification environment.

However, it is important to note that it is necessary to write a few directed test cases in order to target those cases not covered by any other constrained random tests. Figure 5.12 shows the paths to achieve complete functional coverage. It illustrates that most of the verification task is spent in the outer loop, making minimal code changes to add new constraints and only writing directed tests for the few features that are very unlikely to be reached by random tests [1].

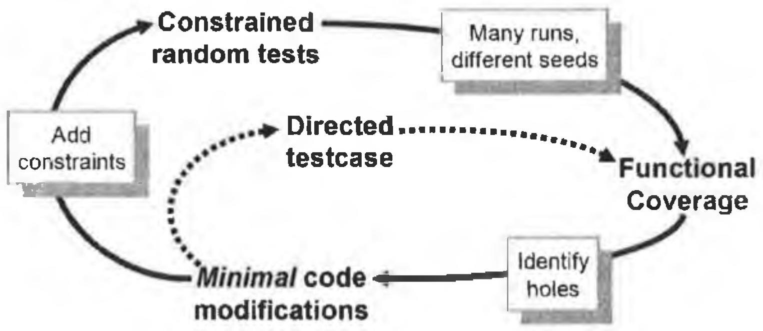


Figure 5.12 Coverage Path

5.6.5 Test case Library

In designing the verification environment a test case mechanism has to be integrated into the testbench. This allows the user to choose between either a random or directed test case. To implement this structure a directed test generator (DTG) has to be created and along with the atomic generator which provides constrained random stimuli for all random test cases. Figure 5.13 shows how this is implemented.

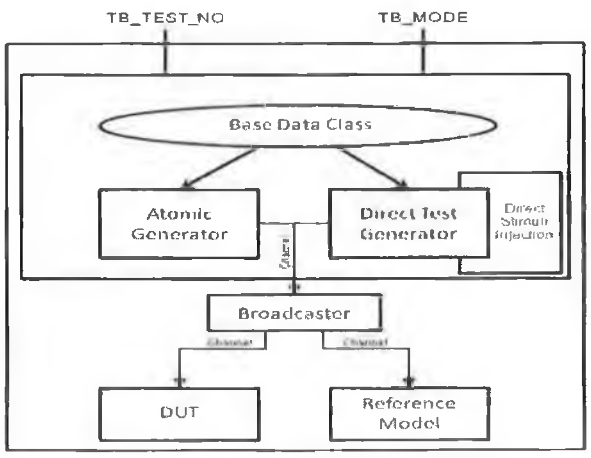


Figure 5.13 Test Case Mechanisms

Along with creating the DTG, other components are needed to implement the test case mechanism. A list of these components and an explanation is given below.

1. Test case directory
2. Random / Direct Test case
3. Fp_data_common
4. Fp_test_constr
5. Fp_testlist
6. Regression Flow

1. **Test case Directory**

In the test case directory, individual directories have been created for both random and directed tests. These directories contain the different test cases and the files associated with them.

2. **Random / Direct Testcases**

With the test case directory structure in place, the next task is to create the test cases

Test case extends environment (fp_env.sv)

```
class random_test_1 extends cl_fp_env

function new (virtual fp_adder_if
fp_if);
        super.new(fp_if);
endfunction
```

Figure 5.14 Testcase extends the environment

Figure 5.14 illustrates how the test case extends the environment and also the new method.

Methods used within the Test cases

build method ()

```
function void build();

  super.build();

  this.gen_chan.stop_after_n_insts = 20000;
// Determine the number of transactions allowed in
the test case

endfunction
```

Figure 5.15 Testcase build method

Figure 5.15 illustrates the build method within the test case. In a random test case the number of transactions are declared.

start Method ()

```
task automatic start();

  super.start();

  `vmm_debug(log , "running Random Testcase 2");

  gen_chan.start_xactor();

endtask

endfunction
```

```
  super.start();

  fork
    begin
     `vmm_note(log , "running Direct Testcase 0");

     fp_dtst.start_xactor();

     // Checking for rounding errors

     fp_dtst.fp_data_inject(100 ,32'h3DCCCCCD,
32'h3DCCCCCD);     //0.1 + 0.1

     #2000;
    end begin


    #1200;

    end join_any

endtask
```

Figure 5.16 Random and Direct Testcase

Figure 5.16 illustrates how the atomic generator (random) and DTG (direct) are called within the test case.

wait_for_end ()

```
task automatic wait_for_end();

  fork
    begin
     super.wait_for_end();

this.gen_chan.notify.wait_for(cl_fp_data_atomic_gen::DONE);
      #20000; // Time to run after the transactor stops
    end

    begin

        #4000600; // Time allowed for test to complete
    end
  join_any

endtask
```

Figure 5.17 wait_for_end method.

Figure 5.17 illustrates the wait for end method within the test case.

3. Fp_data_common

The fp_data_common file is used to constraint random stimuli that are produced by the atomic generator. The file is extended from the data class using SystemVerilog OO techniques. Using this file means no modifications are made to the data class itself. An example of a constraint within this file is illustrated in figure 5.18.

```
Constraint con
{
   data1.exp == 8'h00;
   data2.exp == 8'h00;
}
```

Figure 5.18 Constraint within fp_data_common

4. Fp_test_constr

This fp_test_constr file contains list of random/direct test cases. Each test case has been given a unique number. The reason for doing this is that when a user begins a simulation a parameter (TB_TEST_NO) is passed through the environment. This parameter matches the number of a test case written. Also within this file the environment and interface are extended.

5. Fp_testlist

This fp_testlist file contains the list of files within the directory and is linked up to the main file within the rundir directory.

6. Regression Flow

Regression flow is a means by which the verification environment can support and highlight certain information that can be viewed at a later point in time to determine different test case results. The regression tests can be merged together to organise different coverage and assertion results to indicate the overall degree of verification applied to the DUT.

To fully implement the test case structure, regression scripts are necessary. These regression scripts are written in TCL. A brief explanation of TCL scripts is given below.

These scripts allow the user to choose between random or directed test cases and the number of the tests he/she wants to run. Along with writing these scripts certain parameters have been written that inter connect with the regression scripts and the verification environment. These parameters have been written within the top file. Parameters that are included within the top file were as follows.

- TB_MODE allows the user to select between either a random test or a direct test
- TB_TEST_NO parameter allows the user to select the number associated with the test

A particular test and test mode (random/directed) may be selected at compile time by specifying parameters for both. Figure 5.19 illustrates how the test mode parameter (TB_MODE) switches the source of test generation within the verification environment.

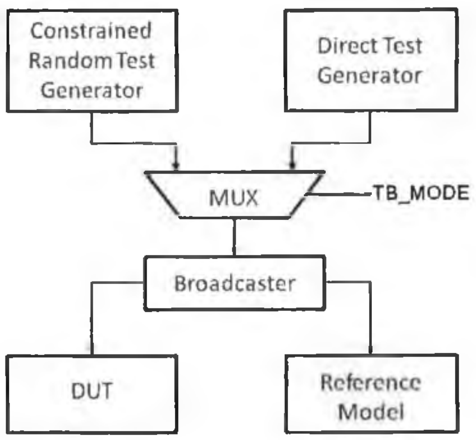


Figure 5.19 Chosen constrained random/directed stimuli

Associated with each test generator is a corresponding library of individual test cases. Each test case can be accessed by the test number parameter (TB_TEST_NO). Finally using regression scripts allows the user to select different seeds for constraint random testing.

TCL (Tool Command Language)

TCL is a scripting language developed for scientific and engineering applications TCL scripts are made up of commands separated by newlines or semicolons. It is often used for GUI, string-manipulation, testing, and integration of multiple components

The main concern in random stimulus is to avoid reproducing a simulation that has already created the same information. With this in mind, random seeds are used. Random seeds will generate different random stimulus when selected during simulation. When using regression scripts, a record should be kept of what seeds are used, in order not to run a simulation with the same seed twice.

5.7 Functional Coverage

5.7.1 Introduction to Functional Coverage

Functional Coverage is the determination of how much functionality of the design has been exercised by the verification environment [36].

The implementation of functional coverage consists of a number of steps. Firstly, code is added in the form of covergroups in order to monitor the stimuli being put on the DUT. The reactions and response to the stimuli are also monitored to determine what functionality has been exercised. Cover groups should be specified in the verification plan. Within a test case scenario, their usefulness is ascertained by analysing the RTL code and understanding the data they have recorded during the simulation.

Cover points become more powerful within a simulation when they are crossed together to identify greater levels of abstraction within a design [36]. Cover points provide a powerful mechanism in identifying areas of functional coverage within a design. Like assertions, they can be compiled into a detailed reporting structure using Synopsys's URG tool.

5.7.2URG Report

The Unified Report Generator (URG) generates combined reports for all types of coverage information [37]. The reports may be viewed through the design hierarchy, module lists, coverage groups, or through an overall summary "dashboard" for the entire design/testbench. The reports consist of a set of HTML or text files. The HTML versions 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 that supports CSS (Cascading Style Sheets).

5.7.3 Implementing Functional Coverage for the Floating-point Adder Model

To implement functional coverage for the DUT, the key functionality of the floating-point adder has been recognised. One of the key functionalities used in the DUT is the registers. A List of these registers is given in table 5.1.

Reg Name	**Function**	**Width**
Type 1	Floating-point number type data 1	3
Type 2	Floating-point number type data	3
Abs_d	Absolute value of the difference between the exponents	8
Complement	If signalled 2 number have different signs	1
Sadd	Results of significant [48] carry out bit	49
Shift_val	The amount of bits which negate has shifted right	5
Norm_temp	Describes the decision to choose underflow/overflow flag	1
Add_result	Contains L and G bits	56
Sticky	Sticky bit value	1
Sign_result	Single extended precision result from the normalisation block	56
product	Significant part of Round_exp1	24
Round_ov	Overflow flag relating to round result	1
Round_uv1	Underflow flag relating to round exp1	1
Round_ov1	Overflow flag relating to round exp1	1
Data 1	SP input with implicit bit included [24]	33
Data 2	SP input with implicit bit included [24]	33
Result	SP output value	32

Ov_flag	Overflow flag	1
Uv_flag	Underflow flag	1
X_flag	Inexact flag	1
Inv_flag	Invalid flag	1
Div_flag	Divide by 0 flag	1

Table 5.1 Registers used in Floating-point Adder

5.7.4 Building the Coverage Class

1. A module has been created where the signals needed by the coverage file are declared.
2. Coverage class is derived from the vmm_xactor.
3. Coverage groups and cover points have been created within the coverage class. Appendix 2 shows the cover group and cover points used in the coverage class. Figure 5.20 illustrates an example of a covergroups.
4. The final step in creating the coverage class is connecting up the signals that have been created in the coverage module to their corresponding signals in the DUT.

```
covergroup cg_fp_number_type_bin @ (clk_sample);

// Declare the bins to gather the info separately

cp_fp_type1_bin :  coverpoint cb_cover.fp_number_type1{

                        bins zero    = {3'b000};
                        bins inf     = {3'b001};
                        bins denorm  = {3'b010};
                        bins snan    = {3'b011};
                        bins qnan    = {3'b100};
                        bins regular = {3'b101};
}
cp_fp_type2_bin :  coverpoint cb_cover.fp_number_type2{

                        bins zero    = {3'b000};
                        bins inf     = {3'b001};
                        bins denorm  = {3'b010};
                        bins snan    = {3'b011};
                        bins qnan    = {3'b100};
                        bins regular = {3'b101};

}
// Cross Correlate the coverage information as needed

cross_fp_num_type_bins : cross cp_fp_type1_bin, cp_fp_type2_bin;

endgroup
```

Figure 5.20 Example of a cover group

5.8 VMM Planner

VMM Planner is an extension to Verification Methodology Manual. VMM Planner is a verification planning tool that allows an engineer to think about the verification process at a high level while working with the low level verification data [38] e.g. functional coverage. The VMM Planner application also allows an engineer to convert this low level data into useful information to track the progress of the verification project.

VMM Planner uses HVP (Hierarchical Verification Plan) language to describe the verification plan. HVP is a comprehensive language that allows an engineer to hierarchically describe a verification plan [39].

The VMM Planner application takes the HVP plan and a unified coverage database as inputs and links them together. This helps the verification engineer to track the verification plan completeness [40].

A typical HVP plan consists of attributes, metrics and mainly features.

- Attributes are named values specified in the plan.
- Metrics can be coverage information extracted from the coverage database after a simulation run. Metrics can also include project specific information, for example in the case of the floating-point adder project pass/fail.
- Each hierarchical section of a verification plan is called a feature. A feature may consist of the following:
    - Attribute value assignments
    - Metric measurement specifications
    - Sub features

VMM Planner enabled applications are:

- VMM Planner spread sheet annotator
- Unified Report Generator (URG)

The VMM Planner spread sheet Annotator enables XML formatted spread sheet to capture the verification plan and then back annotate it with coverage results to track progress throughout the project.

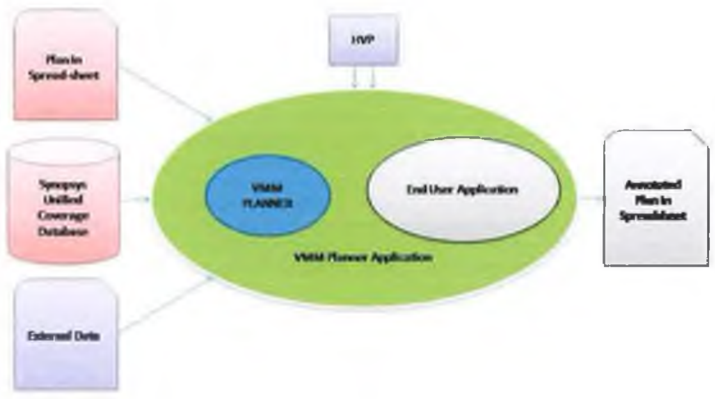


Figure 5.21 VMM Planner

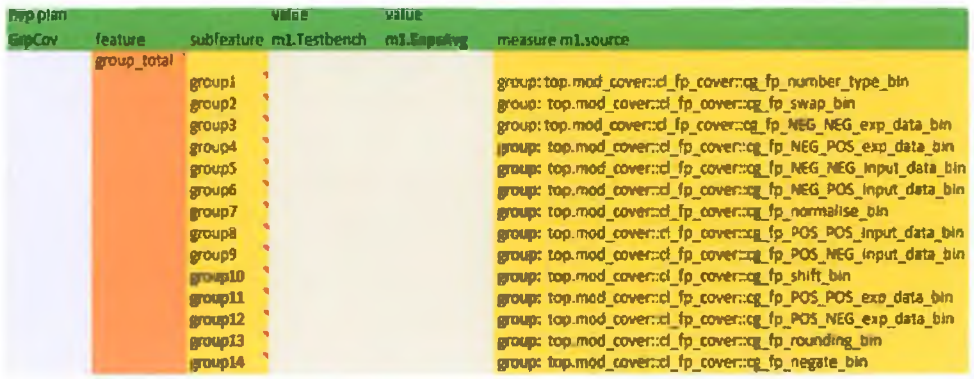
5.8.1 VMM Planner Application for the Floating-point Adder Model

In developing the VMM Planner application for the floating-point adder model a good deal of research has been required. This research included making contact with the Synopsys research and development team to discuss different issues encountered while using VMM planner. At the time of development the VMM Planner was in beta stage. To overcome the learning curve of using the planner application a number of examples have been executed that are included within the VMM library.

In developing the planner application a few problems have been encountered, as follows:

- Difficulties saving the XML file in open office and in Microsoft Excel. The VMM Planner application was not able to read the XML files when trying to annotate it. This problem was solved by using a text editor to edit the XML file.
- Difficulties opening the annotated XML in open office.

Once these problems had been overcome the planner application was developed. Figure 5.22 illustrates the XML file developed.



hvp plan			value	value	
GrpCov	feature	subfeature	m1.Testbench	m1.[illegible]	measure m1.source
	group_total				
		group1			group: top.mod_cover::cl_fp_cover::cg_fp_number_type_bin
		group2			group: top.mod_cover::cl_fp_cover::cg_fp_swap_bin
		group3			group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_exp_data_bin
		group4			group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_exp_data_bin
		group5			group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_input_data_bin
		group6			group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_input_data_bin
		group7			group: top.mod_cover::cl_fp_cover::cg_fp_normalise_bin
		group8			group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_input_data_bin
		group9			group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_input_data_bin
		group10			group: top.mod_cover::cl_fp_cover::cg_fp_shift_bin
		group11			group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_exp_data_bin
		group12			group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_exp_data_bin
		group13			group: top.mod_cover::cl_fp_cover::cg_fp_rounding_bin
		group14			group: top.mod_cover::cl_fp_cover::cg_fp_negate_bin

Figure 5.22 XML file before being annotated

An explanation of the XML file shown in figure 5.22 can be seen in table 5.2

Command	Descriptions
hvp *plan* | *metric* | *attribute* <identifier>	Required in cell A1. Use letters, numbers, "_"but no keywords or words beginning with a number.
Skip	Turns a column into a comment.
Feature	Required. Defines the level 1 feature… the top of the hierarchy for this plan.
value <source_name>.<metric_name>	This is the column that gets annotated with values for features that are associated with a measure source of the same name and type.
measure <source_name>.source	Must exist if there is a value column for that source_name.
goal	Allows for changes to the default goal (100%) for each feature (or subfeature).

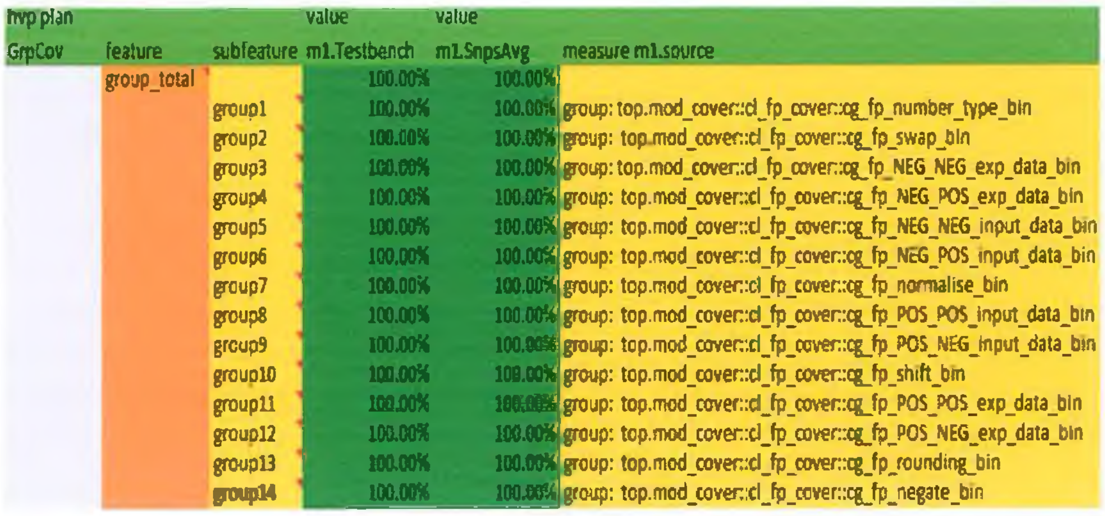
Table 5.2 Features of VMM Planner

Table 5.3 lists some of the commands used to run the planner application. Figure 5.23 illustrates the annotated version of the XML shown in figure 5.22.

usage: hvp annotate -plan planfile [-plan_out annfile] [-dir covdbpath] [-userdata userdata|-userdatafile txtfile] [-userdata_out outvedata] [-metric_prefix prefix] [other switches]

Commands	Description
-h	show this help message and exit
-plan <planfile>	spreadsheet, doc XML or HVP file for your verification plan, mandatory.
-plan_out <annfile>	output annotated spreadsheet, doc XML file. if -plan_out is not given, <original file>.ann.xml file will be generated.
-userdata <userdata files>	userdata file path, could be multiple
-userdatafile <txtfile>	a text file which contains list of userdb file path
-dir <covdb_dirs>	Synopsys coverage db(cm, vdb) path, could be multiple
-userdata_out <out userdata file >	dump annotated score in all measures into userdata file

Table 5.3 VMM Planner Run Commands



hvp plan			value	value	
GrpCov	feature	subfeature	m1.Testbench	m1.SnpsAvg	measure m1.source
	group_total		100.00%	100.00%	
		group1	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_number_type_bin
		group2	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_swap_bin
		group3	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_exp_data_bin
		group4	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_exp_data_bin
		group5	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_input_data_bin
		group6	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_input_data_bin
		group7	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_normalise_bin
		group8	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_input_data_bin
		group9	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_input_data_bin
		group10	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_shift_bin
		group11	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_exp_data_bin
		group12	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_exp_data_bin
		group13	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_rounding_bin
		group14	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_negate_bin

Figure 5.23 Annotated XML File

5.8.2 Recommendations for VMM Planner

Hyperlink to URG Report

When a coverage group produces a low score, to determine the cause of the low score a hyperlink pointing to the problem in the URG report should be accessible within the XML file.

Display Testcases with XML File

In verifying the floating-point adder model, multiple random test cases and directed test cases have been developed. A recommendation that could be useful for the planner application is that these test cases could be displayed within one XML document. VMM planner would use the information provided within the XML to automatically generate a verification plan for each test case within the XML file. Another option that possibly should be looked at is developing a GUI interface for VMM planner so that a third party application such as Microsoft Excel does not need to be used.

5.9 Testing the Floating-point Adder

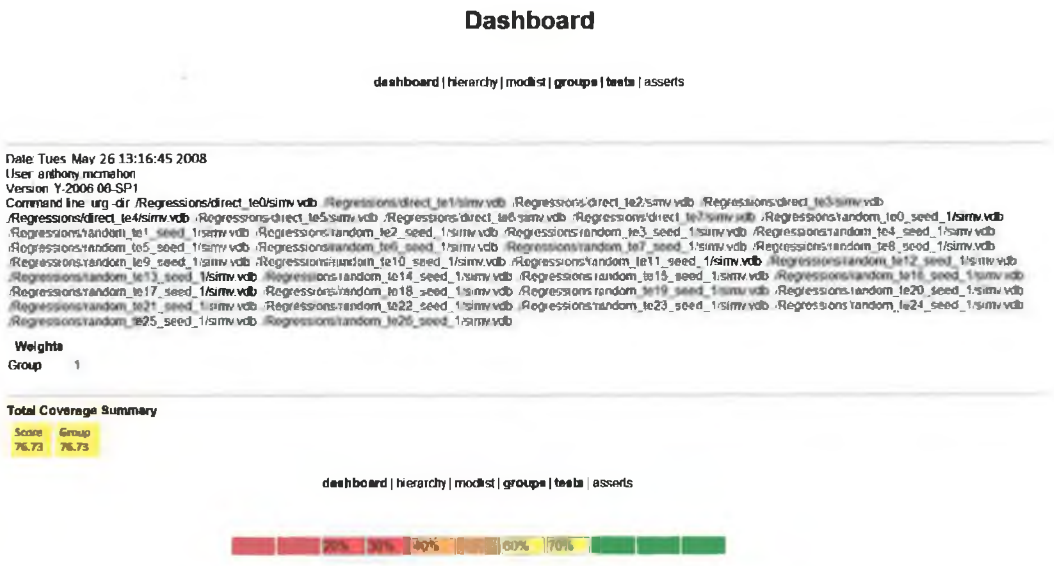
A library of test cases were written to test the floating-point adder using the advanced verification environment.

5.9.1 Testcase 1

The first test case written produced 10,000 random test transactions. This test checks to see if the URG reporting mechanism, VMM planner application and the regression flow mechanism within the environment are working correctly.

5.9.2 Testcase 2

This test case consists of two million transactions (random stimulus) through the environment. Using URG and the VMM planner 76.7 % functional coverage has been achieved. The reports produced by VMM planner and URG are illustrated in figure 5.24 and figure 5.25.


Dashboard

dashboard | hierarchy | modlist | groups | tests | asserts

```
Date: Tues May 26 13:16:45 2008
User: anthony mcmahon
Version: Y-2006.06-SP1
Command line: urg -dir /Regressions/direct_te0/simv.vdb /Regressions/direct_te1/simv.vdb /Regressions/direct_te2/simv.vdb /Regressions/direct_te3/simv.vdb
/Regressions/direct_te4/simv.vdb /Regressions/direct_te5/simv.vdb /Regressions/direct_te6/simv.vdb /Regressions/direct_te7/simv.vdb /Regressions/random_te0_seed_1/simv.vdb
/Regressions/random_te1_seed_1/simv.vdb /Regressions/random_te2_seed_1/simv.vdb /Regressions/random_te3_seed_1/simv.vdb /Regressions/random_te4_seed_1/simv.vdb
/Regressions/random_te5_seed_1/simv.vdb /Regressions/random_te6_seed_1/simv.vdb /Regressions/random_te7_seed_1/simv.vdb /Regressions/random_te8_seed_1/simv.vdb
/Regressions/random_te9_seed_1/simv.vdb /Regressions/random_te10_seed_1/simv.vdb /Regressions/random_te11_seed_1/simv.vdb /Regressions/random_te12_seed_1/simv.vdb
/Regressions/random_te13_seed_1/simv.vdb /Regressions/random_te14_seed_1/simv.vdb /Regressions/random_te15_seed_1/simv.vdb /Regressions/random_te16_seed_1/simv.vdb
/Regressions/random_te17_seed_1/simv.vdb /Regressions/random_te18_seed_1/simv.vdb /Regressions/random_te19_seed_1/simv.vdb /Regressions/random_te20_seed_1/simv.vdb
/Regressions/random_te21_seed_1/simv.vdb /Regressions/random_te22_seed_1/simv.vdb /Regressions/random_te23_seed_1/simv.vdb /Regressions/random_te24_seed_1/simv.vdb
/Regressions/random_te25_seed_1/simv.vdb /Regressions/random_te26_seed_1/simv.vdb
```

Weights

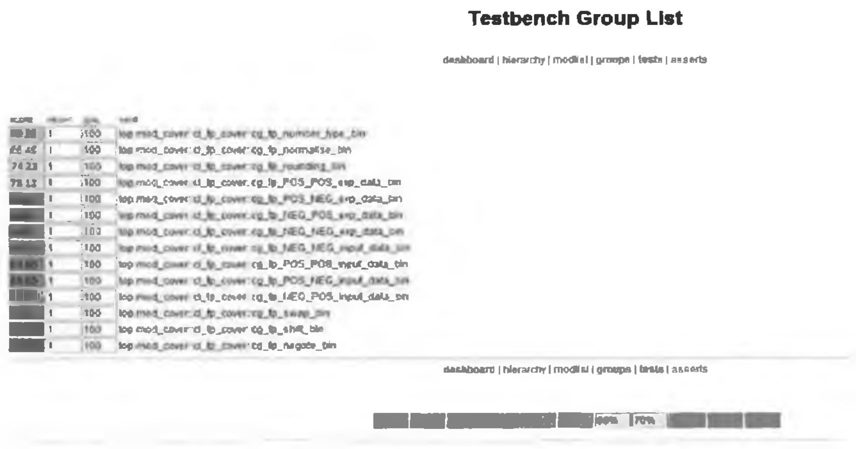
Group	1

Total Coverage Summary

Score	Group
76.73	76.73

dashboard | hierarchy | modlist | groups | tests | asserts

Figure 5.24 Output from URG Dash Board



Testbench Group List

dashboard | hierarchy | modlist | groups | tests | asserts

score	weight	goal	name
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_number_type_bin
66.46	1	100	top.mod_cover.cl_fp_cover.cg_fp_normalise_bin
74.23	1	100	top.mod_cover.cl_fp_cover.cg_fp_rounding_bin
78.12	1	100	top.mod_cover.cl_fp_cover.cg_fp_POS_POS_exp_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_POS_NEG_exp_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_NEG_POS_exp_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_NEG_NEG_exp_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_NEG_NEG_input_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_POS_POS_input_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_POS_NEG_input_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_NEG_POS_input_data_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_swap_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_shift_bin
[illegible]	1	100	top.mod_cover.cl_fp_cover.cg_fp_negate_bin

dashboard | hierarchy | modlist | groups | tests | asserts

Figure 5.25 Output of URG (Group List)

5.9.3 Testcase 3

2 million random test transactions were created using a different random seed. The result of using a different seed makes little or no difference to the functional coverage score (76.9%).

5.9.4 Testcase 4

The fourth test case written was for 5 million vectors, this showed a significant difference in the functional coverage score (80.3 %).

5.9.5 Testcase 5, 6, 7

Testcases produced 10, 20, and 40 million random test transactions respectively. The results of these testcases are listed in table 5.3.

5.9.6 Testcase 8, 9

The DUT was tested for 50 and 60 million transactions. Less than half a percent change in coverage score was achieved. It was concluded that 40 million transactions would be used as a basis for any more tests. Using 40 million vectors, a functional coverage score of 88% was achieved. Once testing using pure random stimuli was exhausted, constrained random tests would have to be written.

5.9.7 Testcase 10 Constrained random:

Example of a Constraint

Data 1 would be positive for a certain number of transactions and would be negative for same amount of transactions. The same would be applied to data 2. This test case would add positive and negative numbers together. This will check to see if there is an issue in adding positive and negative floating-point numbers.

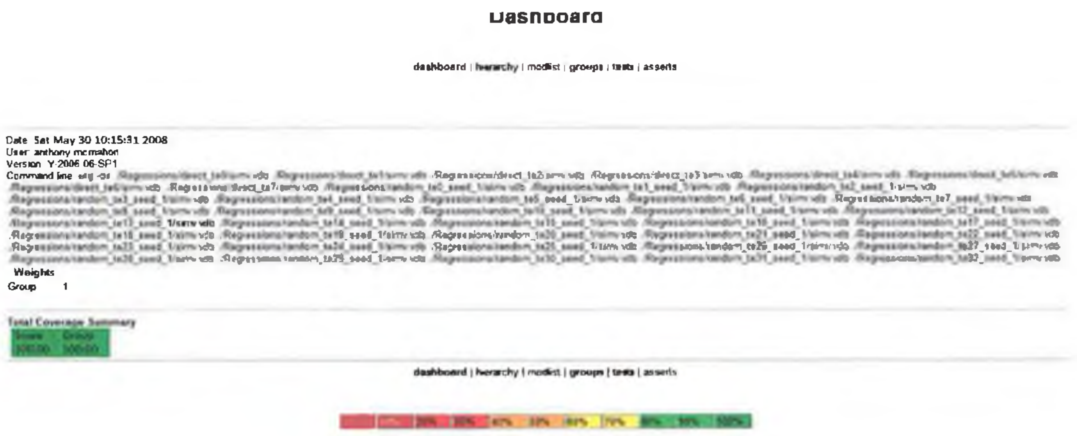
Testing the DUT with pure random stimuli and constrained random stimuli together achieved a functional coverage score of 89%.

5.9.8 Testcase 11, 12, 13

In all, 27 constrained random tests have been written. A description of each is included in table 5.2. Using pure random stimuli and constrained random stimuli, 94.2 % test coverage was achieved. Therefore to close out the verification process and achieve 100% coverage, some directed tests are required.

5.9.9 Testcases 13 – 18

In all, nine directed tests have been created. The final test consists of one pure random test case, 27 constrained random test cases and 9 direct test cases. The output from the URG report and VMM planner can be seen in figure 5.26.



Dashboard

dashboard | hierarchy | modlist | groups | tests | asserts

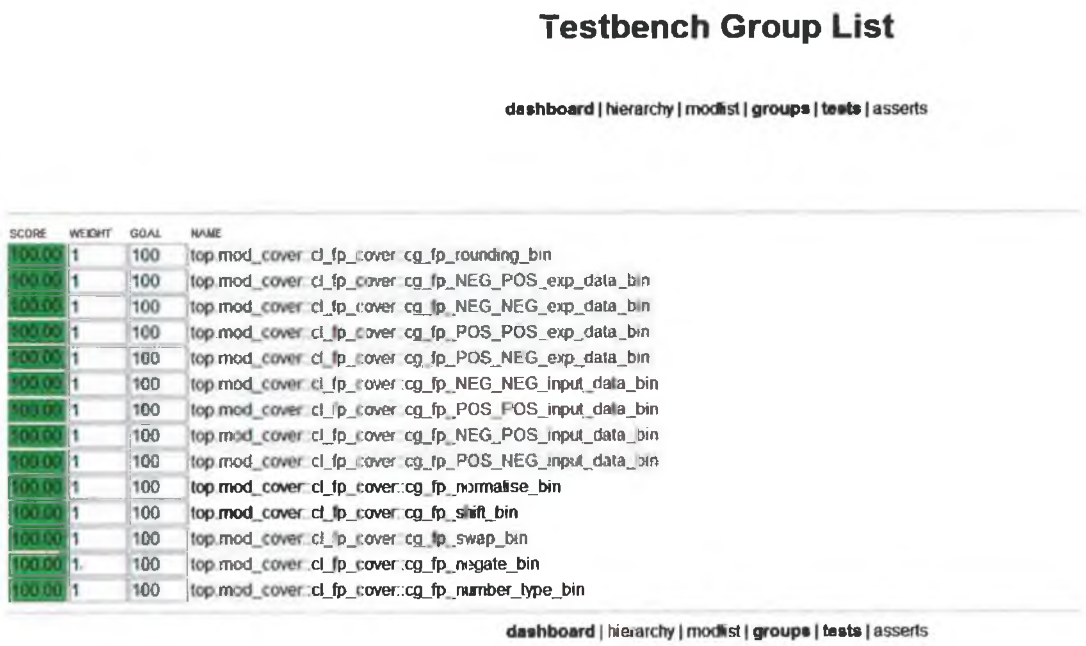
```
Date Sat May 30 10:15:31 2008
User anthony mcmahon
Version Y-2006 06-SP1
Command line urg -dir [illegible]
 Weights
Group      1
```

Total Coverage Summary

Score	Group
100.00	100.00

dashboard | hierarchy | modlist | groups | tests | asserts

Figure 5.26 Output from URG Dash Board



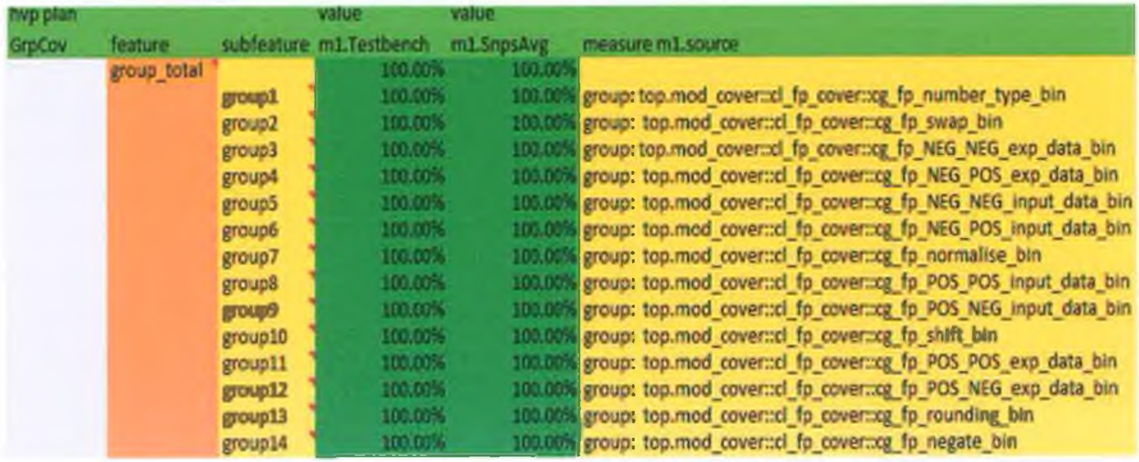
Testbench Group List

dashboard | hierarchy | modlist | **groups** | **tests** | asserts

SCORE	WEIGHT	GOAL	NAME
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_rounding_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_exp_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_exp_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_POS_POS_exp_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_exp_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_input_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_POS_POS_input_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_input_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_input_data_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_normalise_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_shift_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_swap_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_negate_bin
100.00	1	100	top.mod_cover::cl_fp_cover::cg_fp_number_type_bin

dashboard | hierarchy | modlist | **groups** | **tests** | asserts

Figure 5.27 Output of URG (Group List)



hvp plan			value	value	
GrpCov	feature	subfeature	m1.Testbench	m1.SnpsAvg	measure m1.source
	group_total		100.00%	100.00%	
		group1	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_number_type_bin
		group2	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_swap_bin
		group3	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_exp_data_bin
		group4	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_exp_data_bin
		group5	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_NEG_input_data_bin
		group6	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_NEG_POS_input_data_bin
		group7	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_normalise_bin
		group8	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_input_data_bin
		group9	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_input_data_bin
		group10	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_shift_bin
		group11	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_POS_exp_data_bin
		group12	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_POS_NEG_exp_data_bin
		group13	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_rounding_bin
		group14	100.00%	100.00%	group: top.mod_cover::cl_fp_cover::cg_fp_negate_bin

Figure 5.28 Show the final Output from VMM planner

5.10 Test Cases and Test run

Name of Test Case	**Description of Testcase**
Random_te0.sv	Pure random stimulus
Random_te1.sv	Constraint random stimuli (sign 1 or 0 for data 1 and data 2)
Random_te2.sv	Constraint random stimuli (exp all 1's or 0's for data 1 and data 2)
Random_te3.sv	Constraint random stimuli (mant all 1's or 0's for data 1 and data 2)
Random_te4.sv	Constraint random stimuli (sign , exp , mant together for all 1's or 0's)
Random_te5.sv	Constraint random stimuli (exp for certain ranges)
Random_te6.sv	Constraint random stimuli (exp for certain ranges)
Random_te7.sv	Constraint random stimuli (exp for certain ranges)
Random_te8.sv	Constraint random stimuli (mant for certain ranges)
Random_te9.sv	Constraint random stimuli (mant for certain ranges)
Random_te10.sv	Constraint random stimuli (mant for certain ranges)
Random_te11.sv	Constraint random stimuli (mant for certain ranges)
Random_te12.sv	Constraint random stimuli (mant for certain ranges)
Random_te13.sv	Constraint random stimuli (mant for certain ranges)
Random_te14.sv	Constraint random stimuli (mant for certain ranges)
Random_te15.sv	Constraint random stimuli (mant for certain ranges)
Random_te16.sv	Constraint random stimuli (mant for certain ranges)
Random_te17.sv	Constraint random stimuli (mant for certain ranges)
Random_te18.sv	Constraint random stimuli (mant for certain ranges)
Random_te19.sv	Constraint random stimuli (mant for certain ranges)
Random_te20.sv	Constraint random stimuli (mant for certain ranges)
Random_te21.sv	Constraint random stimuli (exp and mant for certain ranges)
Random_te22.sv	Constraint random stimuli (exp and mant for certain ranges)
Random_te23.sv	Constraint random stimuli (exp and mant for certain ranges)
Random_te24.sv	Constraint random stimuli (exp and mant for certain ranges)
Random_te25.sv	Constraint random stimuli (exp and mant for certain ranges)
Random_te26.sv	Constraint random stimuli (sign, exp and mant for certain ranges)
Random_te27.sv	Constraint random stimuli (sign ,exp and mant for certain ranges)

Direct_te0.sv	Add data 1 + data 2 (0 + 0)
Direct_te1.sv	Add different number types
Direct_te2.sv	Add different number types
Direct_te3.sv	Add different number types
Direct_te4.sv	Checking for rounding errors
Direct_te5.sv	Check for right hand swap
Direct_te6.sv	Check for left hand swap
Direct_te7.sv	Check for shift
Direct_te8.sv	Check for normalised shift

Table 5.4 List of Test Cases

Type of Test	Number of Transaction	Coverage Score
Pure Random (default test)	10000	34.23%
Pure Random	2,000,000	76.61%
Pure random different seed	2,000,000	76.65%
Pure random	5,000,000	81.66%
Pure random	10,000,000	85.23%
Pure random	20,000,000	86.59%
Pure Random	40,000,000	88.42%
Pure Random	50,000,000	88.86%
Pure Random	60,000,000	89.08%
Pure Random and 5 constraint random	40,000,000 +50,000	90.16%
Pure Random and 10 constraint random	40,000,000 + 100000	91.24%
Pure Random and 18 constraint random	40,000,000 + 180000	92.06%
Pure Random and 27 constraint random	40,000,000 + 270000	93.31%
Pure Random and constraint random and 1 direct tests	40,000,000 + 270000	93.42%
Pure Random and constraint random and 4 direct tests	40,000,000 + 270000	94.76%
Pure Random and constraint random and 6 direct tests	40,000,000 + 270000	95.60%
Pure Random and constraint random and 7 direct tests	40,000,000 + 270000	97.60%
Pure Random and constraint random and 8 direct tests	40,000,000 + 270000	98.87%
Pure Random and constraint random and 9 direct tests	40,000,000 + 270000	100 %

Table 5.5 List of Tests run

5.10 Conclusion

The goal of this chapter has been to create a reusable test bench environment. This has been implemented by incorporating key SystemVerilog and VMM features.

While developing the advanced verification environment has taken a significant amount of time, this yielded a robust reusable testbench structure. The time spent in research and development is not required for another project as it encompasses a complete SystemVerilog verification flow. Figure 5.29 illustrates the time spent in the research and development of the advanced verification. The graph does not represent the initial learning curve needed to learn SystemVerilog. Figure 5.29 also represents the research and development time and the time taken to verify the floating-point adder once a good understanding of SystemVerilog was known.

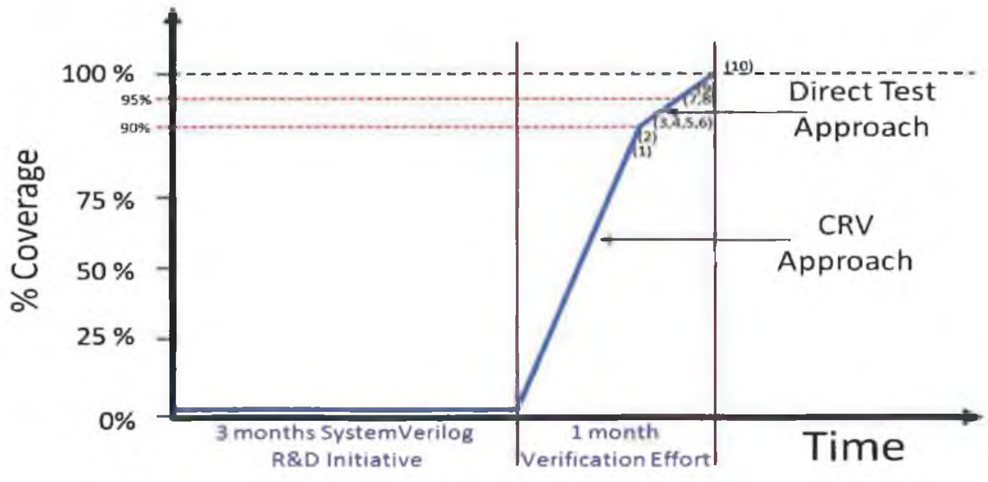


Figure 5.29 R&D of Verification Environment

The constrained random verification approach that implements functional coverage, assertions and score boarding, together with the ability to use random and directed tests in a single environment has proven to be highly productive for future implementations where this verification environment is used.

Thus, reusing this process will relieve an engineer from one verification task and allow them to spend more time actually figuring out how to verify the DUT.

Another goal of this chapter has been to verify the floating-point adder model fully. This has been done by creating a library of testcases for both random and directed tests.

The floating-point adder has been verified by using functional cover groups, assertions and also by incorporating a scoreboard and reference model into the testbench. The scoreboard and the reference model have been used as a compare feature within the testbench.

Finally the floating-point adder has been verified by using a single pure random test that consists of running 40 million random vectors though the adder model. Also a suite of 27 constrained random tests has been created and to finish off the verification process 5 direct tests were written.

The results of building the advanced verification environment discussed in this chapter have been published in a paper at the ISSC 2008 (appendix 1). The paper does not represent a complete solution of the advanced verification environment as more work has been conducted after the paper was published. This included integrating the VMM planner into the advanced verification environment. Further work has also been done on the cover-groups and the regression flow has been developed for the advanced verification environment discussed in this chapter.

Chapter 6 Advanced BFM (I2C)

6.1 Introduction

Chapter 5 described how an advanced verification environment for a floating-point adder model has been built. One of the main objectives of building the standardised test bench has been to create a reusable environment and in turn this would help to reduce the time spent in verifying a DUT. This means the environment could be used to verify another DUT without having to build a complete environment from scratch. The environment has been built in such a way that minimum modifications would have to be made. The only part of the environment in which major changes have to be made is the BFM. This is because the BFM implements the protocol being tested. So, to test the reusability of the test bench an advanced protocol has been chosen. The protocol chosen is the Inter Integrated Circuit (I2C).

The structure of this chapter is as follows; a background and an overview of the I2C protocol are given, before describing how the test bench has been built for the I2C. Also described is how the I2C protocol has been tested for both Master and Slave mode.

6.2 History of I2C

The I2C bus [41] was developed by Philips Semiconductors in the early 1980's. I2C's original purpose was to provide an easy way to connect a Central Processing Unit (CPU) to peripheral chips in televisions. In embedded systems, peripheral devices are often connected to the microcontroller as memory mapped devices, using the microcontroller's parallel address and data bus. The result is lots of wiring on the PCB's to route the address and data lines, not to mention a number of address decoders and glue logic needed. In mass production items such as TVs, VCRs and audio equipment, this is not acceptable. In these items, every component that can be saved means increased profitability for the manufacturer and more affordable products for the end customer [42].

The research done by Philips to overcome these problems has resulted in a 2 wire communication bus called the I2C bus. Its name literally explains its purpose, to provide a communication link between Integrated Circuits.

Today, the I2C bus is used in many other applications than just audio and video equipment. The I2C bus has been adopted by several leading chip manufacturers such as Intel, ST Microelectronics, Infineon Technologies, Texas Instruments, Maxim, Atmel, Analog Devices and others.

Since its release, the I2C bus has had a number of different versions. These versions have added new features to the bus. The operating speed of the bus has been increased. Listed below are the different versions and also a list of the different features added.

6.2.1 Version Summary

The first I2C specification dates back to 1982, operating at Standard mode (up to 100 kbit/s) and allowing for 7-bit addressing.

Version 1.0 – 1992 included the following modifications:

- Programming of a slave address by software has been omitted. The realisation of this feature was rather complicated and had not been used.
- The "low-speed mode" has been omitted. This mode was, in fact, a subset of the total I2C-bus specification and did not need to be specified explicitly.
- The Fast-mode was added. This allows a fourfold increase of the bit rate up to 400 kbit/s.
- Fast-mode devices are downwards compatible i.e. they can be used in a 0 to 100 kbit/s I2C-bus system.
- 10-bit addressing was added. This allows 1024 additional slave addresses.
- Slope control and input filtering for fast mode devices was specified to improve the EMC behaviour.

Version 2.0 - 1998

As the I2C bus became a world standard implemented in over 1000 different ICs and licensed to more than 50 companies. An update version became necessary as many of the newer applications required higher bus speeds and lower supply voltages. This version 2.0 of the I2C bus met those requirements and included the following modifications:

- The High-speed mode (Hs-mode) was added. This allows an increase in the bit rate up to 3.4 Mbit/s. Hs-mode devices can be mixed with Fast- and Standard-mode devices on the one I2C-bus system with bit rates from 0 to 3.4 Mbit/s.

- The low output level and hysteresis of devices with a supply voltage of 2 V and below has been adapted to meet the required noise margins and to remain compatible with higher supply voltage devices.
- The 0.6 V at 6 mA requirement for the output stages of Fast-mode devices has been omitted.
- The fixed input levels for new devices were replaced by bus voltage-related levels.
- After a repeated START condition in Hs-mode, it is possible to stretch the clock signal SCLH.

Version 2.1 - 2000

Version 2.1 of the I2C bus specification includes the following minor modifications:

- After a repeated START condition in Hs-mode, it is possible to stretch the clock signal SCLH.
- Some timing parameters in Hs-mode have been relaxed.

6.22 Bus Speeds

Originally, the I2C-bus was limited to 100 kbit/s operation. Over time there have been several additions to the specification so that there are now 4 operating speed categories. All devices are downward-compatible and any device may be operated at a lower bus speed.

- **Standard-mode,** with a bit rate up to 100 kbit/s
- **Fast-mode,** with a bit rate up to 400 kbit/s
- **High-speed mode (Hs-mode)**, with a bit rate up to 3.4 Mbit/s.

6.3 I2C Bus Hardware

The I2C bus physically consists of 2 active wires and a ground connection [41]. The active wires called SDA and SCL are both bi-directional. SDA is the serial data line, and SCL is the serial clock line. This means that in a particular device, these lines can be driven by the microcontroller itself or from an external device. To do this both SDA and SCL are open collector or open drain outputs.

The bus interface is built around an input buffer and an open drain or open collector transistor. When the bus is idle, the bus lines are in the logic high state, external pull-up resistors are necessary for this. To put a signal on the bus, the chip drives its output transistor, thus pulling the bus to a low level. The pull-up resistor in the devices is actually a small current source or even non-existent. The advantage of this system is that it makes it easier to control multi masters on the bus [41]. If the bus is occupied by a device that is sending a 0, then all other device lose their right to access the bus.

6.4 I2C Bus Protocol

In the I2C bus protocol every device connected up to the bus has its own unique address, whether it is a microcontroller, memory, or ASIC. Each of these chips can act as a receiver or transmitter, depending on the functionality.

The I2C bus is a multi-master bus. This means that more than one device is capable of initiating a data transfer on the bus. The I2C specification states that the device that initiates a data transfer on the bus is considered the bus master and all the other devices are regarded to be bus slaves. An example of a master communicating with a slave device can be seen in figure 6.1.

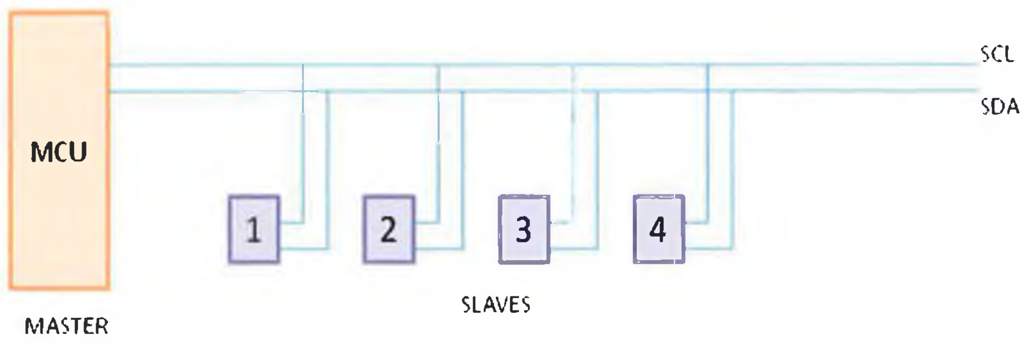


Figure 6.1 Master communicating with Slave

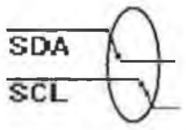
Using the example given above, if the microcontroller wants to send data to one of the slaves, then

1. First the microcontroller will issue a start condition. This acts as an attention signal to all of the connected devices on the bus that will be listening for incoming data.
2. The microcontroller sends the address of the slave device it wants to communicate with, along with a read/write bit determining if the master is going to read from or write to slave.
3. Having received the address, all devices will compare it with their own address. If it doesn't match, they simply wait until the bus is released by the stop condition.
4. If the address matches however, the slave will produce a response called the acknowledge signal.
5. Once the microcontroller receives an acknowledge, it can start sending or receiving data depending on whether a read or write operation was selected. In this case the microcontroller will transmit data and wait for an acknowledge from the slave
6. When the transmission is done, the microcontroller will issue a stop condition. This is a signal that the bus has been released and that the connected devices may expect another transmission to start any moment.

There are several key states on the bus for example: Start, Address, Acknowledge, Data and Stop. These states will be important in the design of the BFM for the I2C protocol.

6.4.1 Start Condition

Before any transaction can happen on the bus, a start condition needs to be generated on the bus. The start condition acts as a signal to all connected IC's that something is about to be transmitted on the bus. As a result, all connected chips will listen to the bus. Only in the start and stop conditions may the SDA line change while SCL is high. In transmitting data the SDA line must change while SCL is low as data is read @ posedge of SCL.



Start Condition is generated by master, first pulls the SDA line low while SCL is high, and next pulls the SCL line low.

6.4.2 Transmitting an address to a slave

Once the start condition has been sent, a byte can be transmitted by the master to the slave. This first byte after a start condition will identify the slave on the bus (address) and will also determine read/write communication.

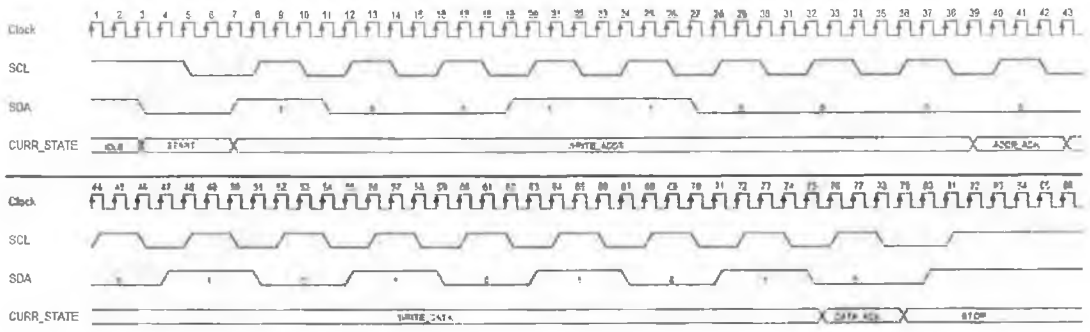
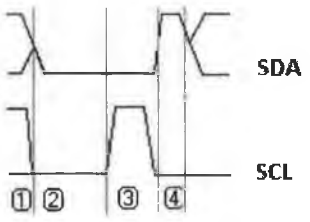


Figure 6.2 Master Write with Slave Acknowledge

6.4.3 Slave Acknowledging Data

When an address or data byte has been transmitted onto the bus then this must be acknowledged by the slave. In the case of address state, slaves check their addresses with the address sent and only the slave matching this address will respond with an acknowledge. The Slave that has been addressed will again respond with an acknowledge when the data byte is sent to it. The slave that is going to give an ACK pulls the SDA line low immediately after reception of the 8th bit transmitted.

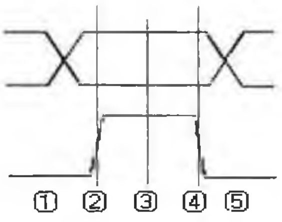


- The master pulls SCL low to complete the transmission of the bit (1),
- SDA will be pulled low by the slave (2).
- The master now issues a clock pulse on the SCL line 9^{th} clock (3).
- The slave will release the SDA line upon completion of this clock pulse (4).

The slave will keep the SDA low, until the master has generated a clock pulse (3) on the SCL line. The bus is free (4) again for the master to continue sending data or to generate a stop condition. In case of data being written to a slave, this cycle must be completed before a stop condition can be generated.

6.4.4 Receiving a byte from a slave

Once the slave has been addressed and the slave has acknowledged this, a byte can be received from the slave if the read/write bit has been set to 1. The protocol is the same as in transmitting a byte to a slave, except that now the master is not allowed to drive the SDA line. Prior to sending the 8 clock pulses needed to clock in a byte on the SCL line, the master releases the SDA line. The slave will now take control of this line. The line will then go high if it wants to transmit a '1' or, if the slave wants to send a '0', remain low.

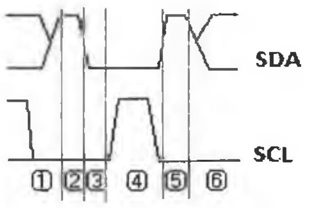


- Master generates a rising edge on the SCL line (2)
- Read the SDA line (3)
- Then generates a falling edge on the SCL line (4)
- Slave may change the state of SDA line during (1) and (5)

The slave will not change the data during the time that SCL is high. Otherwise a Start or Stop condition might accidently be generated. This sequence is performed 8 times to complete the data byte. Bytes are always transmitted with most significant bit (MSB) first.

6.4.5 Master acknowledging data from Slave

Once the master has received a data byte from slave it must decide to acknowledge it or not. The master is in full control of the SDA and the SCL line.



- After transmission of the last bit to the master (1) the slave will release the SDA line.
- The SDA line should then go high (2).
- The Master will now pull the SDA line low (3).
- Next, the master will put a clock pulse on the SCL line (4).
- After completion of this clock pulse, the master will again release the SDA line (5).
- The slave will now regain control of the SDA line (6).

An Acknowledge of a byte received from a slave is always necessary, except on the last byte received. If the master wants to stop receiving data from the slave, it must be able to send a stop condition.

Since the slave regains control of the SDA line after the ACK, this can lead to problems. If the next bit sent to the master is a 0. The SDA line would be pulled low by the slave immediately after the master takes the SCL line low. The master now attempts to generate a stop condition on the bus. It releases the SCL line first and then tries to release the SDA line, which is held low by the slave. Therefore, no Stop condition has been generated on the bus. This condition is called a Not Acknowledge (NACK). Do not confuse this with No acknowledge:

Condition	**Can only occur :**
Not Acknowledge (NACK)	After a master has read a byte from a slave
No Acknowledge	After a master has written a byte to a slave

6.4.6 No Acknowledge (from slave to master)

If after transmission of the 8th bit from the master to the slave the slave does not pull the SDA line low, then this is considered a No Acknowledge condition and is illustrated in figure 6.3.

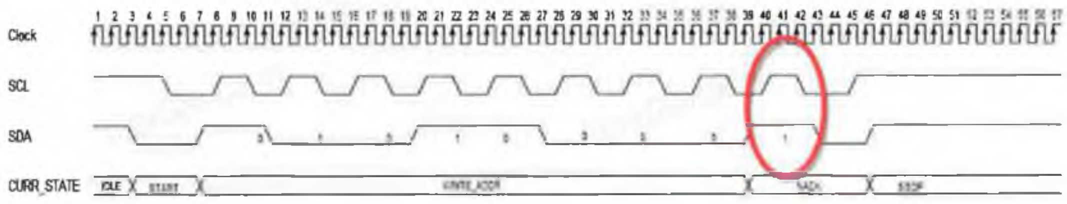


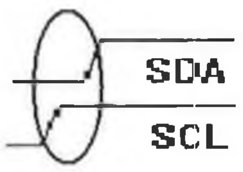
Figure 6.3 No Acknowledge

Reasons why this might happen include:

- The slave is not there
- The slave missed a pulse and got out of sync with the SCL line of the master.
- The bus is stuck, one of the lines could be held low permanently.
- In any case the master should abort by attempting to send a stop condition

6.4.7 Stop conditions

After a message has been completed, a stop condition is generated. This is the signal for all devices on the bus that the bus is available again.



- Stop Condition is generated by the master releasing the SDA line (low to high transition) while the SCL is high and then releasing the SCL line.
- A Stop condition always signals the end of a transmission of data. Even if it is issued in the middle of a transaction or in the middle of a byte. It is good practice for a chip to disregards the information sent and resumes the listening state, waiting for a new start

6.4.8 Repeated Start Condition

The master lets the SCL line go high and the device pulls SDA low to acknowledge. The slave will release the SDA line when it detects that SCL is low. Next the master generates a stop condition and immediately afterwards the master generates a start condition which generates the repeated start condition seen in figure 6.4. The main reason that the repeated start exists is in a multi master configuration where the current bus master does not want to release its master ownership of the bus. Using the repeated start keeps the bus busy so that no other master can grab the bus.

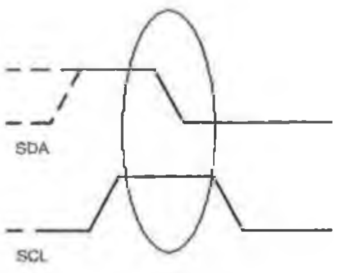


Figure 6.4 Repeated Start Condition

6.4.9Arbitration

The I2C bus was originally developed as a multi master bus. This means that more than one master can try to control the bus at the same time. When using only one master on the bus there is no real risk of corrupted data, except if a slave device is malfunctioning or if there is a fault with the SDA/SCL bus lines. This situation changes with 2 microcontrollers:

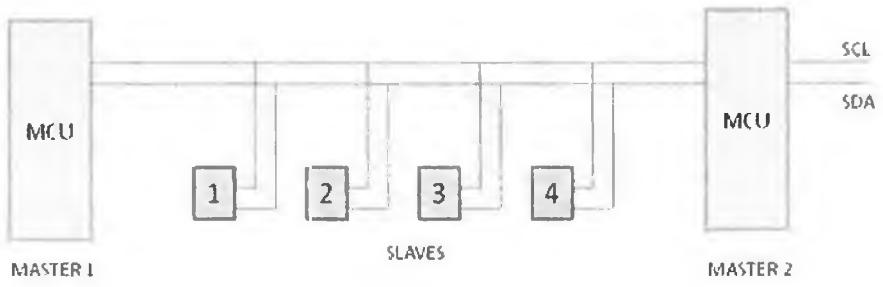


Figure 6.5 I2C Multi Master Arbitration

When microcontroller 1 issues a start condition and sends an address, all slaves will listen including master 2 which at that time is considered a slave as well. If the address does not match the address of master 2, this device has to hold back any activity until the bus becomes idle again after a stop condition. As long as the two masters monitor what is going on the bus, when start and stop are generated and as long as they are aware that a transaction is going on, there is no problem. Assuming one of the masters misses the start condition and still thinks the bus is idle, or it comes out of reset and wants to start talking on the bus, this could lead to problems.

Determining which Master will control the Bus

Since the bus structure is a wired AND if one device pulls a line low it stays low, you can test if the bus is idle or occupied. When a master changes the state of a line to high, it must always check that the line really has gone to high. If line stays low then this is an indication that the bus is occupied and some other device is pulling the line low. Therefore if a master cannot get a certain line to go high, it loses arbitration and needs to back off and wait until a stop condition is generated before making another attempt to start transmitting.

This kind of back off condition will only occur if the two levels transmitted by the two masters are not the same. Figure 6.6 illustrates an example where two masters start transmitting at the same time:

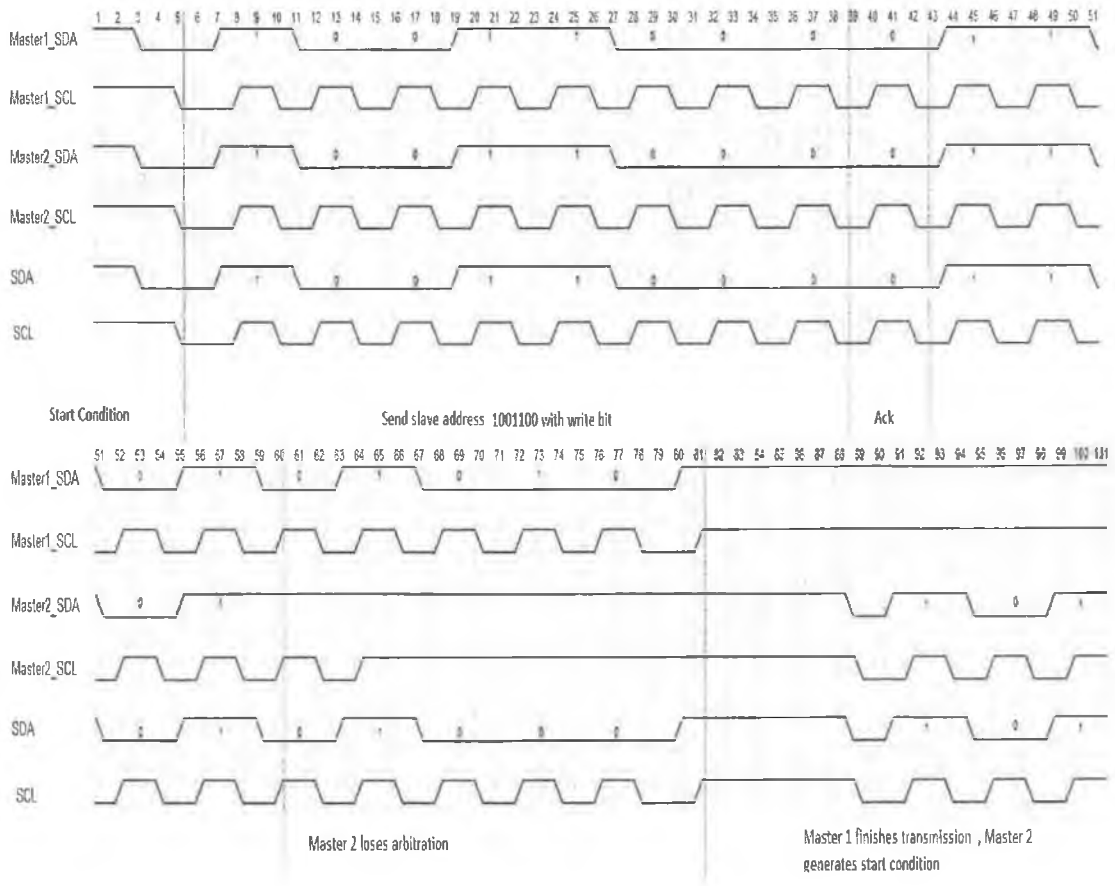

Master1_SDA
Master1_SCL
Master2_SDA
Master2_SCL
SDA
SCL
Start Condition
Send slave address 1001100 with write bit
Ack
Master 2 loses arbitration
Master 1 finishes transmission , Master 2 generates start condition

Figure 6.6 I2C Multi Master Arbitration

The two masters sending address 1001100 to slave with read/write bit set to '0'. The slave acknowledges this. Both masters are under the impression that they own the bus. Now master 1 wants to transmit 11010100 to the slave, while master 2 wants to transmit 11011100 to the slave. The moment the data bits do not match anymore because what one of the masters' sends is different than what is present on the bus, one of them loses arbitration and backs off. Obviously, this is the master which did not get its data on the bus. For as long as there has been no stop present on the bus, it won't touch the bus and leave the SDA and SCL lines alone. The moment a stop condition is detected; master 2 can attempt to transmit again.

Therefore the master that is pulling the line low wins the arbitration. The master which wanted the line to go high but is being pulled low by the other master loses the bus. This is called loss of arbitration or a back-off condition. The Master that has lost arbitration has to wait for a stop condition to appear on the bus before it can attempt to try and transmit on the bus.

6.4.10 Clock Synchronisation

All masters generate their own clock on the SCL line to transfer messages on the I2C bus. Data is only valid during the high period of the clock. A defined clock is therefore needed for the bit by bit arbitration procedure to take place.

Clock synchronisation is performed using the wired AND connection of the I2C SCL line. This means that a high to low transition on the SCL line will cause the devices concerned to start counting off their low period and, once a device clock has gone low, it will hold the SCL line in that state until the clock high state is reached. However, the low to high transition of this clock may not change the state of the SCL line if another clock is still within its low period. The SCL line will therefore be held low by the device with the longest low period. Devices with shorter low periods enter a high wait state during this time.

When all devices concerned have counted off their low period, the clock line will be released and go high. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their high periods. The first device to complete its high period will again pull the SCL line low. In this way, a synchronized SCL clock is generated with its low period determined by the device with the longest clock low period, and its high period determined by the one with the shortest clock high period. Figure 6.7 illustrates clock synchronisation.

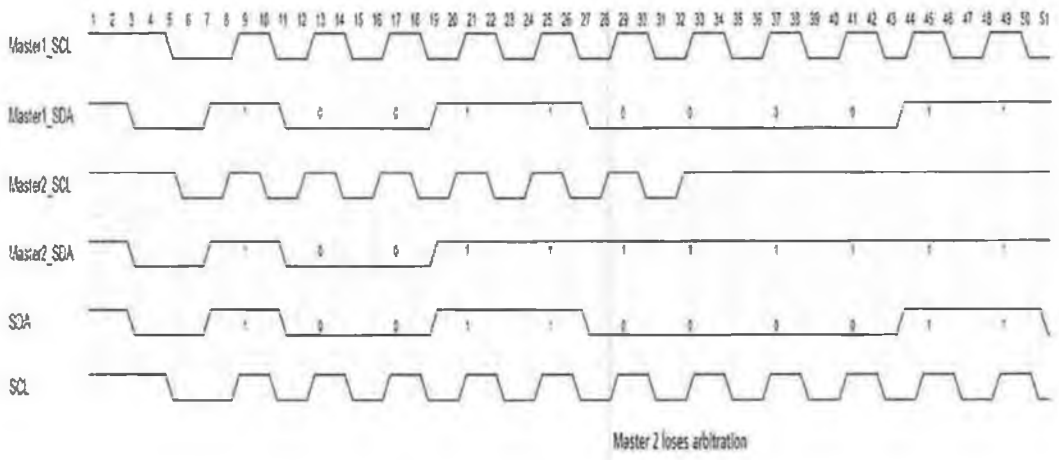


Figure 6.7 Mulit Master Clock Synchronisation

1. Master 1 and Master 2 try to generate start
2. Master 1 and Master 2 start counting off their low periods , Master 2 has the longest low periods , so Master 1 goes into wait state
3. Master 2 is finished counting it low periods and pulls SCL high
4. Master 1 and master 2 start counting their high periods , Masters 1 has the shortest low period and pulls SDA low
5. Master 1 and 2 repeat step 3 and 4 until one of them loses arbitration
6. Master 2 loses arbitration, therefore Master1 takes over control of SDA and SCL lines

6.5 Building the I2C Environment

The previous sections describe the history of I2C protocol and gave an overview of what the I2C bus is. The next few sections describe how the advanced verification environment was built for the I2C protocol.

6.5.1 Test Bench Structure:

As in the case of the floating-point adder model the same directory structure would be used for the I2C test bench. Figure 6.8 illustrates the I2C directory structure.

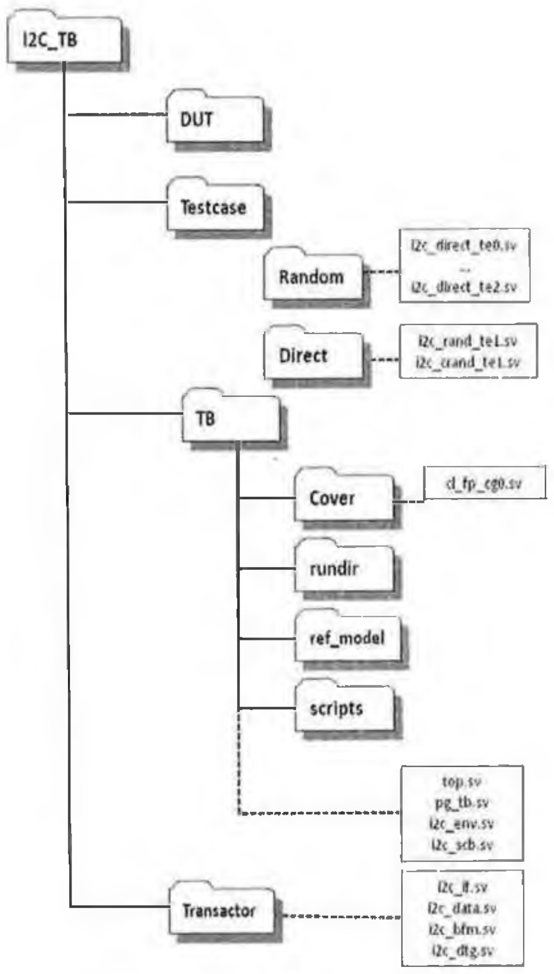


Figure 6.8 I2C testbench Structure

- DUT Directory: Floating-point adder was replaced with I2C master and slave core.
- Testcase Directory : Random and direct testcases were modifield for I2C specification
- TB Directory: The environment file ,top file and program file were modified. The regression scripts files were also modifield.
- Transactory Directory: The main modifications of thedirectory structure were made in the transactor directory. The data file and the interface had minor modification made to them. However, in the BFM major modifications were made.

6.6 Building the I2C Testbench

As mentioned in section 6.5 the same testbench structure has been used as the floating-point adder. However some modifications have to be made. This section describes these modifications.

6.6. 1 Interface

In modifying the interface (figure 6.9) only slight changes have to be made.

1) SCL and SDA declared
2) Clocking block and mod ports block modified to include SDA and SCL as inout.
3) The current state and the next state have been declared to make it possible to view them as a waveform

```
tri scl;                                // SCL I/P
tri sda_i;                             // SDA I/P

logic [3:0]  curr_state, next_state;

modport mp_slave (
                inout sda_i,
                inout scl
);

modport mp_master (
                  inout sda_i,
                  inout scl
);
clocking cb_slave @ (posedge clk);
      default input #setup_time output #hold_time;
      inout sda_i;
      inout scl;
endclocking : cb_slave

clocking cb_master @ (negedge clk);
      default input #setup_time output #hold_time;
      inout sda_i;
      inout scl;
      input  curr_state, next_state;
      input slave_curr_state, slave_next_state;
endclocking : cb_master
```

Figure 6.9 12C Interface

6.6.2 Data File

In the data file the only modification that had to be made was declaring address and data as seen in figure 6.10.

```
rand bit [7:0] addr;
rand bit [7:0] data;
```

Figure 6.10 Data and Address declared within data file

6.6.3 Environment File

In the environment file only minor changes have to be made. These are the instantiation of the different classes used in the testbench. The build function within the environment is the only element where significant changes have been made and can be seen in figure 6.11

```
function void cl_i2c_env::build() ;

  super.build() ;

//  Constructor args

this.in_chan = new("Data channel","in_chan",100);
this.gen_chan = new("Atomic Gen",1,in_chan);

// BFM instantiation
this.i2c_bfm = new("i2c_bfm", 300, null, null, dutif_i2c,
svvc_ctrl,null,null);


// Direct Test Case Xactor instantiation
this.i2c_bfm = new(null, "Atomic_Gen", 500, i2c_bfm.in_chan);

  // Direct Test Case Xactor instantiation
this.i2c_dtst = new(null, "DTST_Gen", 501, i2c_bfm.in_chan);

endfunction
```

Figure 6.11 Build method within environment

6.6.4 Top File

Within the top file very few changes have to be made. These are connecting up the master and slave core to the test bench and also declaring pull-up resistors for SDA and SCL line. The modifications that were made can be seen in figure 6.12.

```
// Connect up the RTL here!

I2Cslave i2c_slave_core
(.clk_in(clk),.clr_in(rst),.scl_in(i2c_if.scl),.sda_io(i2c_if.sda_i));


 pullup (i2c_if.sda_i);       // Pullup resistors

 pullup (i2c_if.scl);
```

Figure 6.12 Top File

6.7 Building the I2C Bus Functional Model (BFM)

In creating the BFM for the I2C protocol two separate BFMs have been built, one for the master mode and another for the slave mode. The reason for doing this is that makes it easier for testing and debugging the testbench. Also at the time of development more knowledge is required to incorporate both master and slave into one BFM. Figure 6.13 illustrates a block diagram showing how the I2C environment has been connected up.

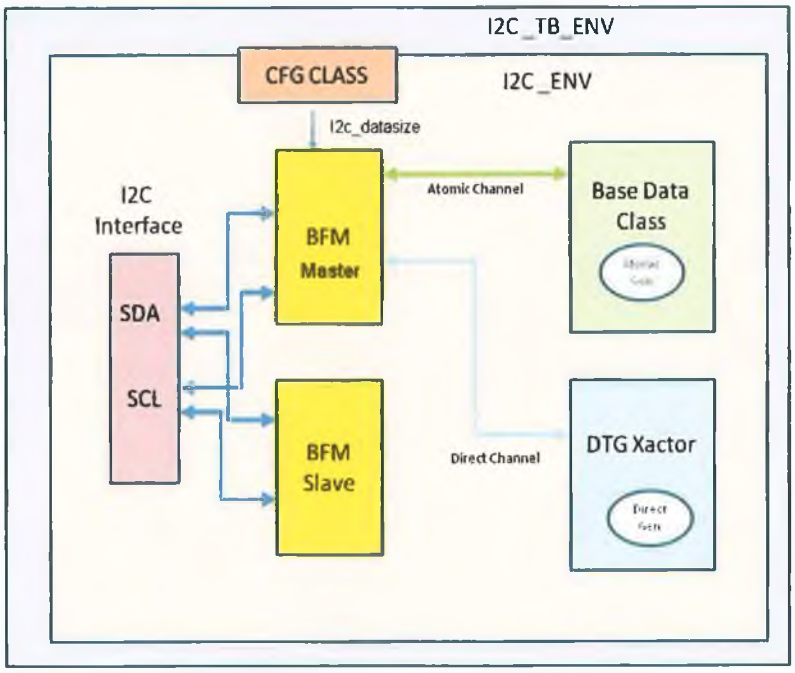


Figure 6.13 Block Diagram of Basic I2C Test bench

In building the two BFMs for the I2C protocol a list of features to be implemented has been created. The list created is given below.

Master

- Generate start condition
- Send slave address and look for acknowledge
- Send invalid address look for no acknowledge
- Send a block of data
- Read a block of data

- Generate Stop condition and repeated start condition
- Insert Slave Core
- Implement multi master features (arbitration and clock synchronisation)

Slave

- Detect Start Condition
- Receive address and acknowledge appropriately
- Receive data and acknowledge appropriately
- Send data and wait for acknowledge
- Detect stop
- Insert Master Core

6.8 Master BFM

6.8.1. Start Condition

The first step in the building the master BFM was generating the start condition. The start condition is a high to low transition on the SDA line while the SCL line remains high. Figure 6.14 illustrates how the start condition was generated and the resulting waveform can be seen in figure 6.15.

```
vi.cb_master.sda_i   <= 1'b0;
vi.cb_master.scl     <= 1'bZ;
repeat (cfg.i2c_clks_divider / 4) @ (vi.cb_master);
```

Figure 6.14 Generating Start condition

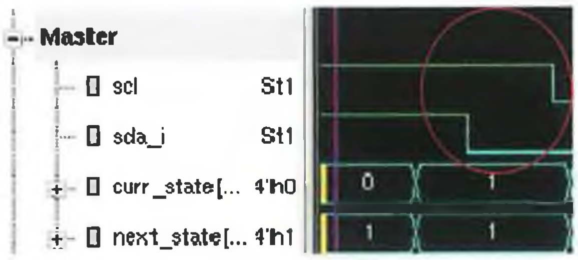


Figure 6.15 Start Condition

6.8.2 Transmitting an address to a slave

In this state the master must first take the address data off the data channel (i2c_data_channel). Next the master pushes the address along with the read/write bit onto the interface, to do this the most significant bit had to be sent first. Figure 6.16 illustrates how the transmit address state was implemented and figure 6.17 shows the outputted waveform.

```
 if(curr_state == cl_i2c_bfm::WRITE_ADDR) begin
        for(int j= cfg.i2c_datasize-1; j>=0; j--) begin
             this.in_chan.get(tr);    // Get transaction from channel
             slave_address <= tr.addr; // Address data equal to
slave_address
             vi.sda_i <= slave_address[j]; Pull address onto SDA
             repeat (cfg.i2c_clks_divider) @ (vi.cb_master);
       end
 end
```

Figure 6.16 Sending Address Data

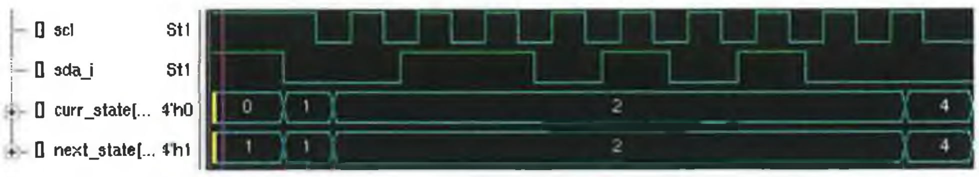


Figure 6.17 Slave Address with Acknowledge

6.8.3 Checking for Acknowledge

Once the address and the read/write bit have been sent successfully, the master waits for acknowledge. If a slave device matches the address sent, then that slave will put the SDA line low on the 9th SCL clock of the address transmission. If however no slave matches the address sent, this is considered a no acknowledge and the SDA line will stay high. Figure 6.18 illustrates how this was implemented.

```
else if ( vi.sda_i == 1'b0 && vi.scl == 1'b1) begin
      ack_bit = 1'b1;
      i2c_fsm_write(tr, curr_state, next_state, id_num);
      repeat (cfg.i2c_clks_divider / 4) @ (vi.cb_master);
      curr_state = next_state;
end
```

Figure 6.18 Checking for Acknowledge

6.8.4 Send Data

With acknowledge received from the slave, the master now moves to the send data state. This state is very similar to send address state, except this time the master takes data from the i2c_data_channel. The data is then pushed onto the interface in the same way as before. Once the master is finished transmitting data to the slave, it again waits for acknowledge from the slave.

6.8.5 Stop conditions

When the master finishes communicating on the bus it produces a stop condition. The stop condition is generated when the SDA line goes from a low to a high while SCL stays high. Once a stop condition has been generated the master goes back into idle state. Figure 6.19 illustrates the stop condition generated.

```
If (curr_state == cl_i2c_bfm::STOP) begin
      vi.cb_master.sda_i  <=1'b0;
      i2c_fsm_write(tr, curr_state, next_state, id_num);
      repeat (cfg.i2c_clks_divider) @ (vi.cb_master);

      curr_state = next_state;
end
```

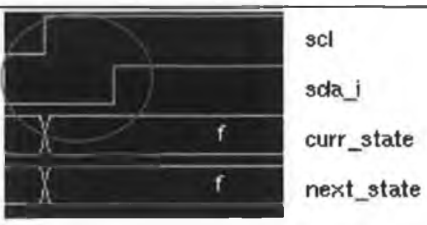


Figure 6.19 Stop Condition

6.8.6 Repeated Start Condition

Another condition that can occur is a repeated start condition. In this state the master generates a stop condition and immediately afterwards the master generates a start condition which generates the repeated start condition. Figure 6.20 illustrates how the repeated start condition was generated.

```
if(curr_state == cl_i2c_bfm::REP_START) begin

      `vmm_debug(log, " Master  is in repeated start");

            vi.cb_master.sda_i  <=1'bz;

            repeat (cfg.i2c_clks_divider / 4) @ (vi.cb_master);

            vi.cb_master.sda_i  <= 1'b0;
end
```

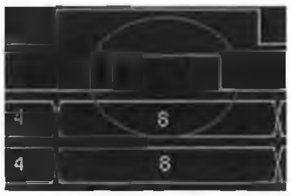


Figure 6.20 Repeated Start Condition

6.8.7 Generating SCL Clock:

SCL clock is generated within a task of its own. Depending on the states selected the SCL line will perform differently, figure 6.21 illustrates how this was done.

```
while(1)begin
      this.wait_if_stopped();
      if(curr_state == i2c_bfm::START || curr_state == _bfm::STOP )
            begin
            vi.cb_master.scl <= 1'bZ;
            end
      else if(curr_state == bfm::WRITE_ADR||curr_state ==i2c_bfm::ACK);
            begin
             vi.cb_master.scl  <= ~vi.cb_master.scl;
            end
            repeat (cfg.i2c_clks_divider / 2 ) @ (vi.cb_master)  ;
      end
```

Figure 6.21 Generating SCL Clock

6.9 Slave BFM

Two separate BFMs were built for the I2C test bench. This section explains how the slave BFM has been created

6.9.1 Detect Start

This task detects if a start condition has been generated. The slave remains in an idle state until a start condition is detected. The slave checks for the start condition by monitoring the SDA line every top clock. The top clock in this case is the main clock for the test bench. If the SDA line goes from a high to low transition while SCL remains high, this can be considered a valid start condition.

6.9.2 Received Address / Acknowledge State

During the address state the slave checks the address sent by the master at the posedge of the SCL clock. The slave repeats this process seven times to represent the 7-bit address. Once the complete address is received the slave then checks the address sent with its own and responds accordingly. If the address matches, the slave acknowledges by pulling the SDA low during the 9th clock of the transmission. On the other hand if the address does not match, no acknowledge occurs and the SDA line goes high. Finally the 8 bit sent determines if the master is going to read or write to the slave

```
for(int j= cfg.i2c_datasize-1; j>=0; j--) begin
            address [j] = vi.sda_i ;
            @(posedge vi.scl);
      end
   if(address [7:1] == 7'b0110101) begin
       vi.sda_i  <=1'b0;
       if (address [0] == 0) begin
            give_ack_bit = 1'b1;
            slave_read_bit = 1'b1;
            i2c_give_ack ();
      end
```

Figure 6.22 Acknowledge

6.9.3 Read Data / Acknowledge State

This state is very similar to the receive address state except this time the master is sending a block of 8 bit data and the slave acknowledges in the same way as before.

6.9.4 Send Data / Get Acknowledge state

In the case of write bit selected, the slave writes data to the master BFM. The slave writes to the master, the same way as the master writes to the slave except that this time the master cannot touch the SDA line until the slave has finished transmitting data. Once the byte of data is sent, the slave moves into the get acknowledge state. In the acknowledge state, the slave checks the SDA line at the 9^{th} clock of transmission to determine if the master has acknowledged the data or not.

6.9.5 Detect Stop

As with the start condition, the stop condition is generated by the master, so the slave needs to have some facility to check for this. The same as the start condition the slave checks the SDA and SCL signals every top clock to see if the stop condition is generated. The Stop condition is generated by low to high transition of the SDA line while SCL remains high.

6.10 State Machine for BFMs

To control the states within the BFMs, two state machines have been implemented. To develop the state machine, key states have been recognised within the I2C protocol. As described in the I2C protocol section the key states are listed as: Start, Address, Acknowledge, Data and Stop. The state machine diagrams could be drawn up and can be seen in figures 6.23 and 6.24. In all, 2 diagrams have been developed master read / write and slave read/write.

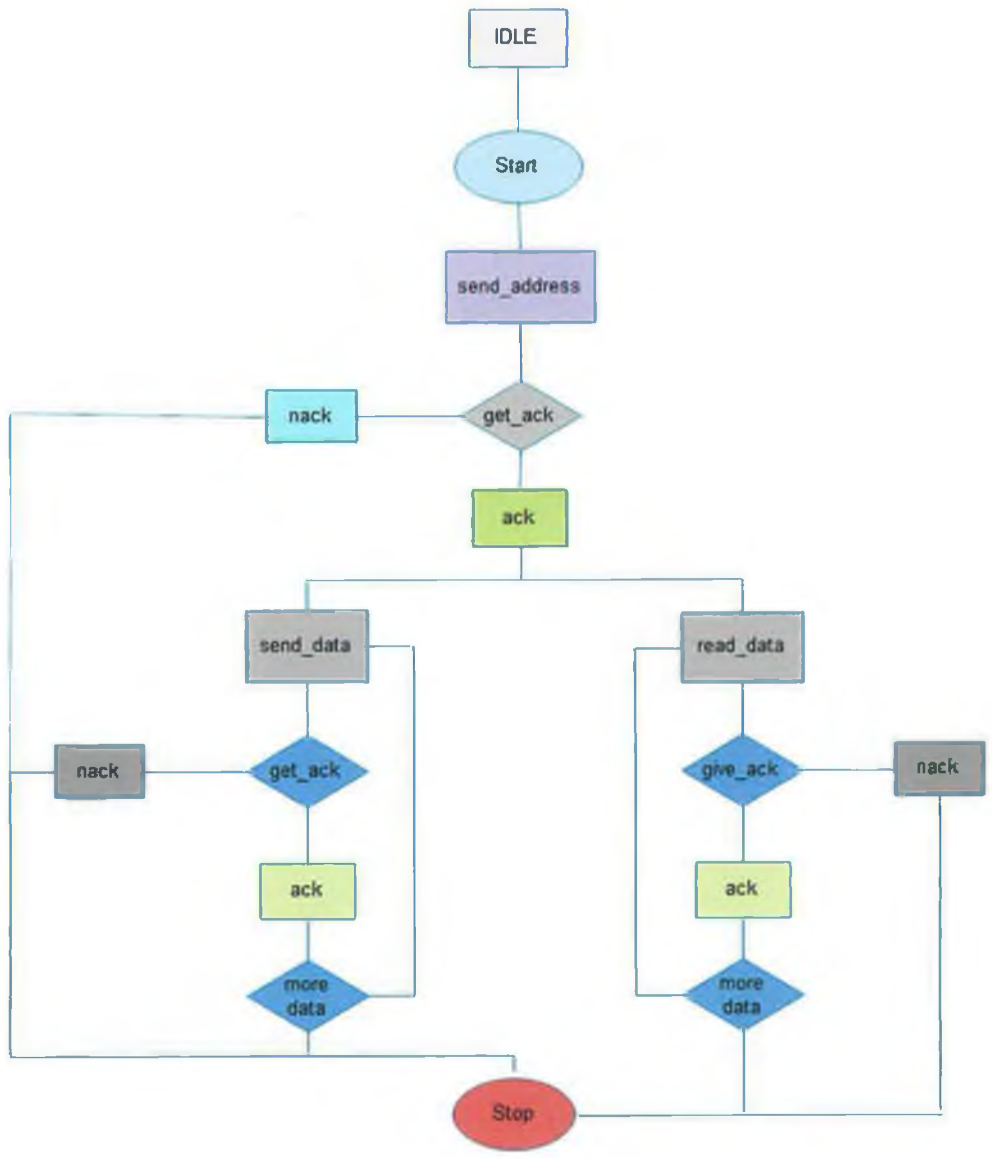


Figure 6.23 Master FSM

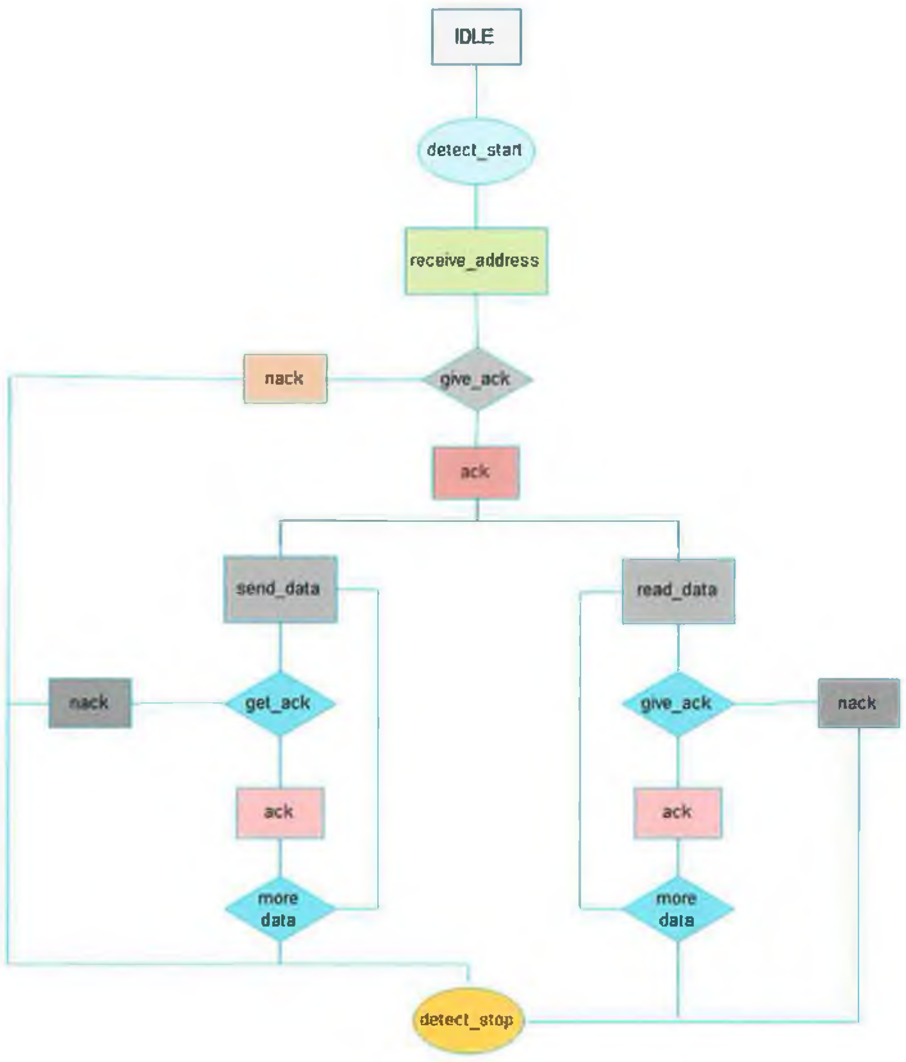


Figure 6.24 Slave FSM

6.10.1 Implement FSM

To implement the FSM the states used in the FSM are declared first. Figure 6.25 illustrates the states used for both master and slave. With master and slave states declared, the next step is to declare the current state and the next state. Figure 6.25 illustrates the control bits used to control the FSM.

```
// FSM State process
typedef enum logic [3:0]
{
IDLE                  = 4'b0000,
START                 = 4'b0001,
WRITE_ADR             = 4'b0010,
WRITE_DATA            = 4'b0011,
ACK                   = 4'b0100,
READ_DATA             = 4'b0101,
NACK                  = 4'b0110,
STOP                  = 4'b0111,
CHECK_ACK             = 4'b1000,
SETUP                 = 4'b1001,
MASTER_ACK            = 4'b1010,
MASTER_NACK           = 4'b1011,
REP_START             = 4'b1110,
SUB_ADDR              = 4'b1111
} i2c_fsm_kind;


i2c_fsm_kind curr_state,
next_state;
// Slave FSM State process
typedef enum logic [3:0]
{
SLAVE_IDLE             = 4'b0000,
DETECT_START           = 4'b0001,
READ_ADR               = 4'b0010,
SLAVE_READ_DATA        = 4'b0011,
SENT_ACK               = 4'b0100,
SLAVE_WRITE_DATA       = 4'b0101,
SENT_NACK              = 4'b0110,
DETECT_STOP            = 4'b0111,
DETECT_REP_START       = 4'b1000,
GET_ACK                = 4'b1001,
NOT_RESET              = 4'b1010,
RESET                      =4'b1011
} i2c_fsm_slave;


i2c_fsm_slave slave_curr_state,
slave_next_state;
```

```
// Local variables for Master

logic idle_bit;
logic start_bit ;
logic address_bit ;
logic setup_bit ;
logic sub_address_bit;
logic ack_bit;
logic nack_bit;
logic master_ack_bit;
logic master_nack_bit;
logic check_bit;
logic read_bit;
logic write_bit;
logic address_read_bit;
logic address_write_bit;
logic stop_bit;
logic rep_start_bit;

//Local variables for Slave

logic slave_idle_bit;
logic detect_start_bit ;
logic read_address_bit ;
logic give_ack_bit;
logic give_nack_bit;
logic get_ack_bit;
logic slave_read_bit;
logic slave_write_bit;
logic detect_stop_bit;
logic nack_stop_bit;
```

Figure 6.25 Master and Slave states & control bits

The FSM has been created using a task called fsm_i2c(). Within the task fsm_i2c a list of different if statements has been created. Figure 6.26 illustrates how the start state is chosen using the FSM and Figure 6.27 shows how the start state has been created within the generate start task.

```
else if(curr_state == cl_i2c_bfm::START) begin
      if(setup_bit == 1'b1) begin
            next_state = SETUP;
      end
      else begin
            next_state = IDLE;
      end
end
```

Figure 6.26 State Machine (Start state)

```
(1) start_bit = 1'b1;
(2) i2c_fsm_write(tr, curr_state, next_state); // Implement i2c FSM
(3) @ (vi.cb_master);
(4) curr_state = next_state;
```

Figure 6.27 Implementing State Machine and selecting states

1. Set start bit to 1, this selects the start state from within state machine
2. Call state machine task
3. States are updated at next top_clock
4. Current state is updated with next state , in this case start state

6.11 Further Implementation of I2C Protocol

6.11.1 Integrating Master and Slave BFM

In the original design of the testbench, the master and slave sides of the I2C protocol have been implemented as two separate BFMs. Once a better understanding of the I2C protocol and better understanding of SystemVerilog was obtained, it was decided to integrate the two BFMs into one BFM. Figure 6.28 illustrates the new I2C environment also shown in the diagram Master and Slave core connected up.

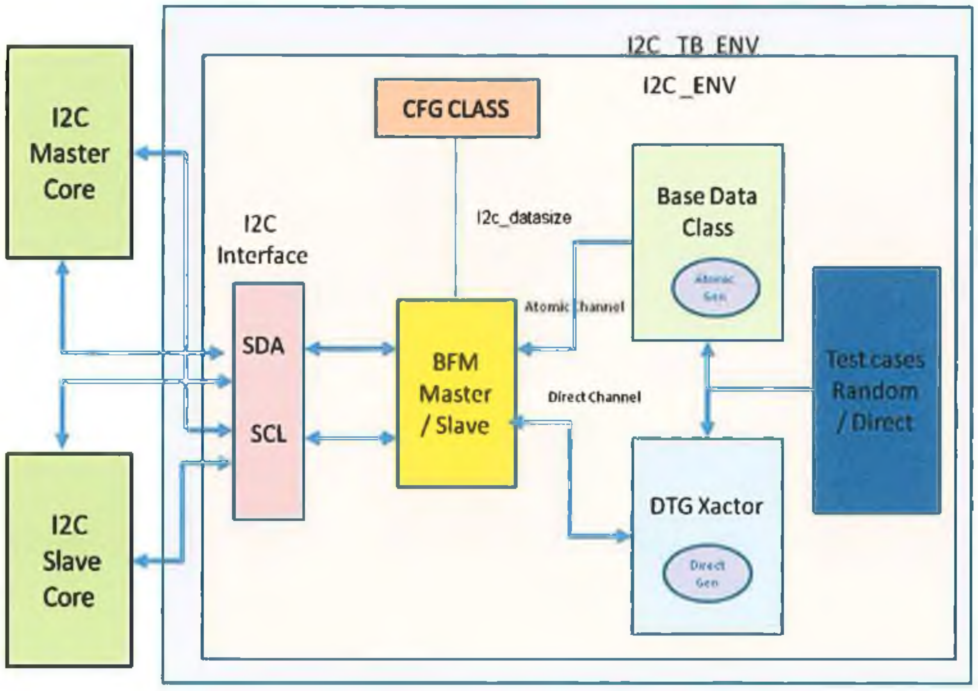


Figure 6.28 Complete Block Diagram of I2C Testbench

To integrate the two BFMs into one, the configuration class had to be modified. The reason the configuration class was modified was that it would allow the user to select either master or slave operations or both.

The new configuration would be called from either random or direct test case depending on which was selected. Figure 6.29 illustrates how the configuration class interacts with the chosen test case and BFM.

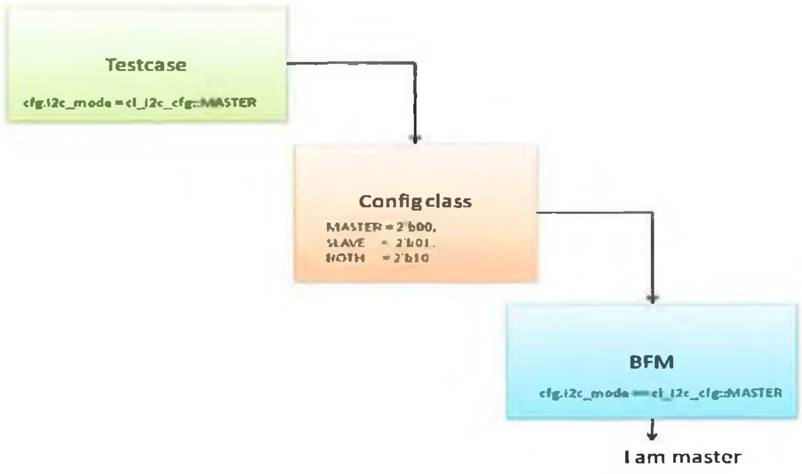


Figure 6.29 Configuration selection

After the BFM integration and testing the DUT has been placed into the testbench for testing. The next section describes this process.

6.11.2 Master / Slave Core

OpenCores is a community that enable engineers to develop open source hardware, with a similar ethos to the free software movement. Currently the emphasis is on digital modules called 'cores' or 'IP Cores' [43].

Through searching Opencores two suitable cores were found that could be used to test the BFM. Unfortunately at the time of searching Opencores, there was no core that had both master and slave integrated into the one design.

6.11.3 Slave Core

The first core that was connected up onto the testbench was the slave core. The slave core was used to test the master functionality of the BFM.

- Generate start condition
- Send slave address
- Send sub address (location of register that the master want to write to)
- Wait for acknowledge

- Send data or read data back from register master had written to
- Generate repeated start condition
- Generate stop condition

6.11.4 Master Core

Once the master functionality of the BFM was tested satisfactory the master core was added to the testbench and used to test the slave functionality of the BFM.

- Detect start
- Receive address
- Acknowledge address
- Receive data or send data
- Detect start

As well as testing the slave functionality of the BFM, the slave also tests the functionality of the master core.

6.12 Multi Master Features

Multi master functionality, multi master arbitration and clock synchronisation of the I2C protocol has been included in the test bench..

6.12.1 Arbitration

The first feature implemented is the multi master arbitration; two masters can control the bus at the same time and can complete a transaction without problem if they are the same. If however they are not then there needs to be a mechanism to determine who can control the bus.

To do this both masters monitor the SDA line at every SCL clock. Example code can be seen in figure 6.31. The maser that tries to put the SDA line high while it is low loses the arbitration and backs off and waits for a stop condition and tries again and this process can be seen in figure 6.30.

Firstly to implement arbitration a state machine diagram has to be drawn up. Figure 6.30 illustrates the state machine diagram. How this state machine was implemented can be seen in figure 6.31.

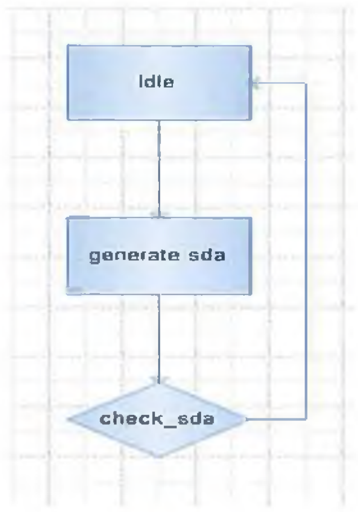


Figure 6.30 Arbitration

```
if ( curr_state == check_sda) begin

      if( internal_sda == vi.sda) begin

            drive_sda_bit = 1'b1;

            i2c_arbit_fsm (curr_state , next_state)

            curr_state = next_state;
      end

      else if( internal_sda == 1'bz && vi.sda == 1'b0) begin

            wait_state_bit = 1'b1;

            i2c_arbit_fsm (curr_state , next_state)

            curr_state = next_state;

            i2c_wait_state();

      end
end
```

Figure 6.31 Checking the SDA line

6.12.2 Implementing Clock Synchronisation

The last feature included into the test bench is clock synchronisation. The feature counts the high periods and low periods of master clocks and checks to see if they match the SCL line accordingly.

1. Both master start with their clocks high, and the SCL line high,
2. Master 1 clock goes low and pulls the SCL line low. Therefore master 2 will have to react and pull it's clock low.
3. Both masters start counting off their low periods. Master 1 finishes counting off it's low period first and tries to pull the SCL line high. It cannot get SCL to go high because another device is keeping the SCL line low ,
4. Master 1 goes into a "wait state" until the SCL line goes high
5. Both masters start counting off their high periods. The master with the shortest high period pulls the SCL line low
6. The other master reacts by pulling it's clock low as well
7. This sequence continues until one of the master loses arbitration

To implement the multi master synchronisation, a state machine diagram has been drawn up (Figure 6.32). Figures 6.33 and 6.34 show how the low count off and high count off were implemented.

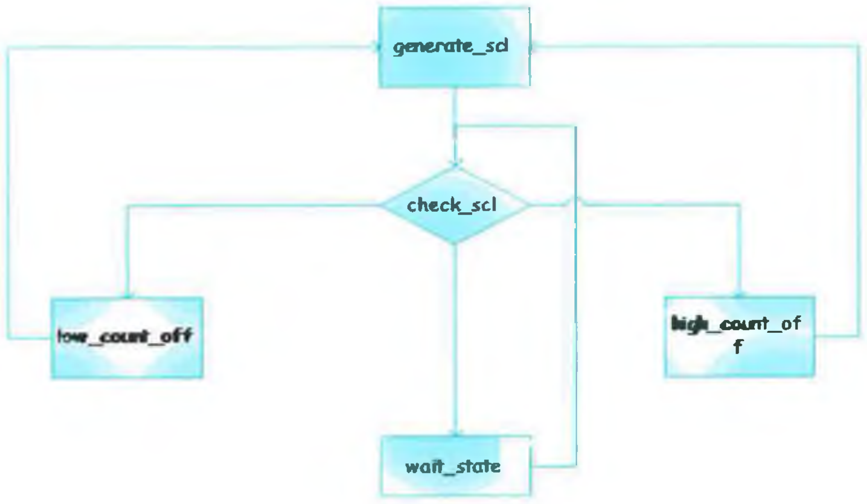


Figure 6.32 Clock Synchronisation

```
task automatic cl_i2c_bfm::i2c_low_count_off();

if ( curr_state == low_count_off) begin


      for (i = low_period ; i <=0; i --) begin

            internal_scl ==1'b0;

            vi.scl = internal_scl [i];

      end
end
```

Figure 6.33 Low Count off

```
if (curr_state == high_count_off) begin


            for (i = high_period ; i <=0; i --) begin

                  internal_scl ==1'b1;


                  vi.scl = internal_scl[j];
            end
end
```

Figure 6.34 High count off

6.13 Testing of I2C Protocol

6.13.1 Testcase 1

The first test performed is to check if the master BFM could control the SDA and SCL line. This is done by the master pulling the SDA and SCL line high. Then the master generates a start condition. The master pulls the SDA line low while SCL is still high and then pulls SCL low. Figure 6.35 illustrates the waveform generated.

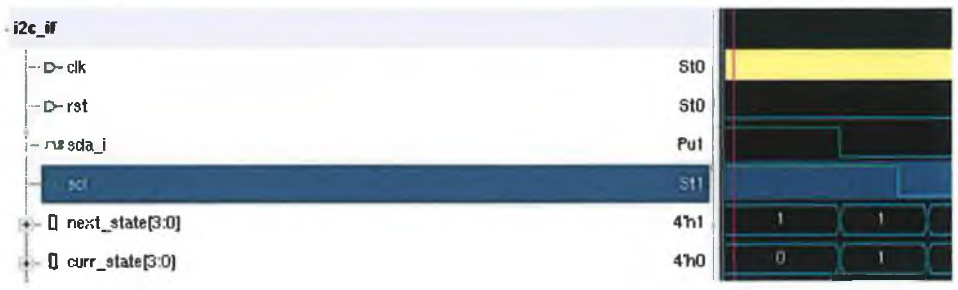


Figure 6.35 Start Condition

6.13.2 Testcase 2

The next test created is a check of the main functionality of the BFM (master and slave). The functionality is described below.

- Master generates start condition
- Slave detects start condition
- Master sends slave address with write/ read bit
- Slave receives the address and checks it against its own and acknowledges accordingly (in this case slave address matches). Another test is performed where an invalid address is send.
- Master sends a block of 8 bit data to slave
- Slave reads data and again acknowledges accordingly.
- Once the master receives the acknowledge it generates a stop condition

Figure 6.36 and 6.37 show the waveforms generated

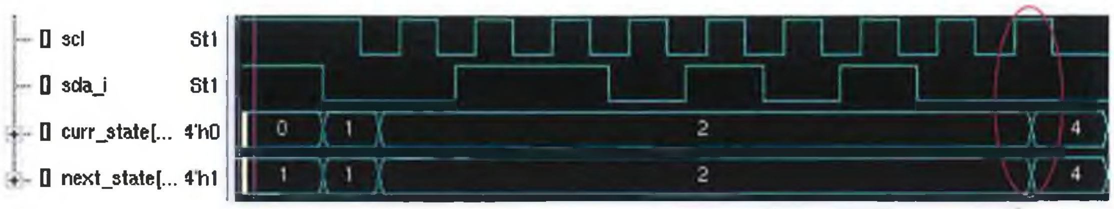


Figure 6.36 Test BFM Functionality (Acknowledge)

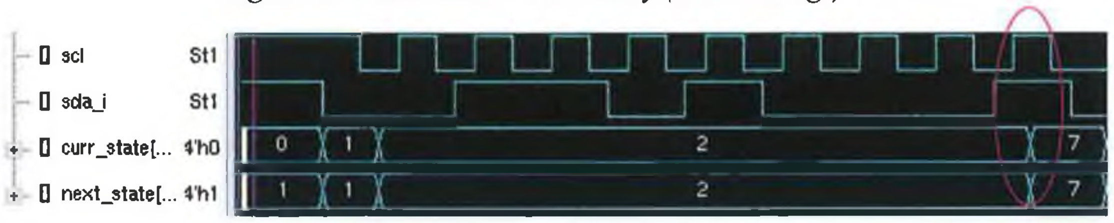


Figure 6.37 Test BFM functionality (No Acknowledge)

6.13.3 Testcase 3

This test case tests for the same functionality as test case two, but this time the master is reading data from the slave and must generate an acknowledge or no acknowledge accordingly. Figure 6.38 illustrates the waveform generated.

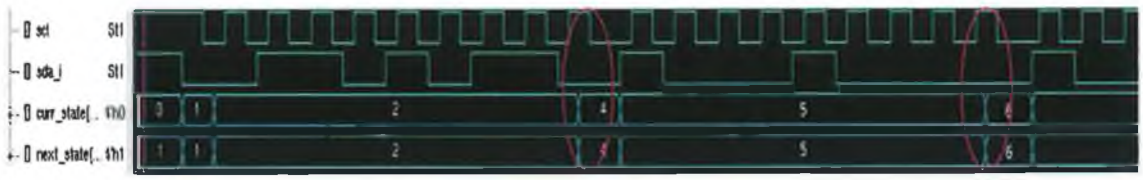


Figure 6.38 Slave write

6.13.4 Testcase 4

This test checks for a repeated start condition. Figure 6.39 shows the resulting output waveform.

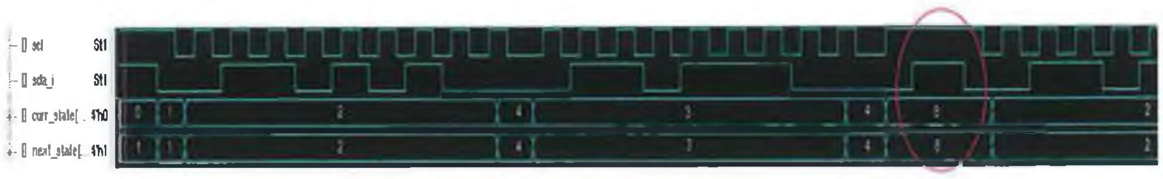


Figure 6.39 Repeated Start Condition

6.13.5 Testcase 5

Master and slave BFM integrated into one BFM. The previous test cases (2, 3, and 4) are run to check for functionality of the new BFM and the resulting output is illustrated in figure 6.40.

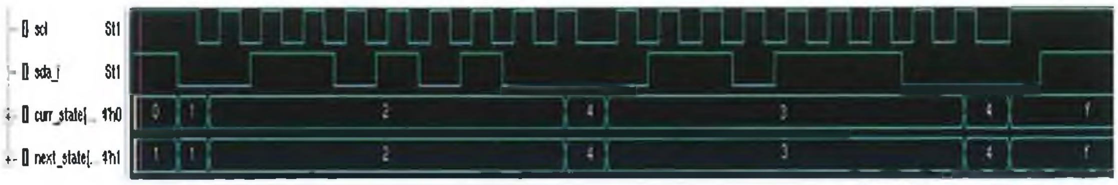


Figure 6.40 Integrated BFM

6.13.6 Testcase 6

To test the slave core the same tests are performed as in test case (2, 3 and 4) and the resulting output is illustrated in figure 6.41.

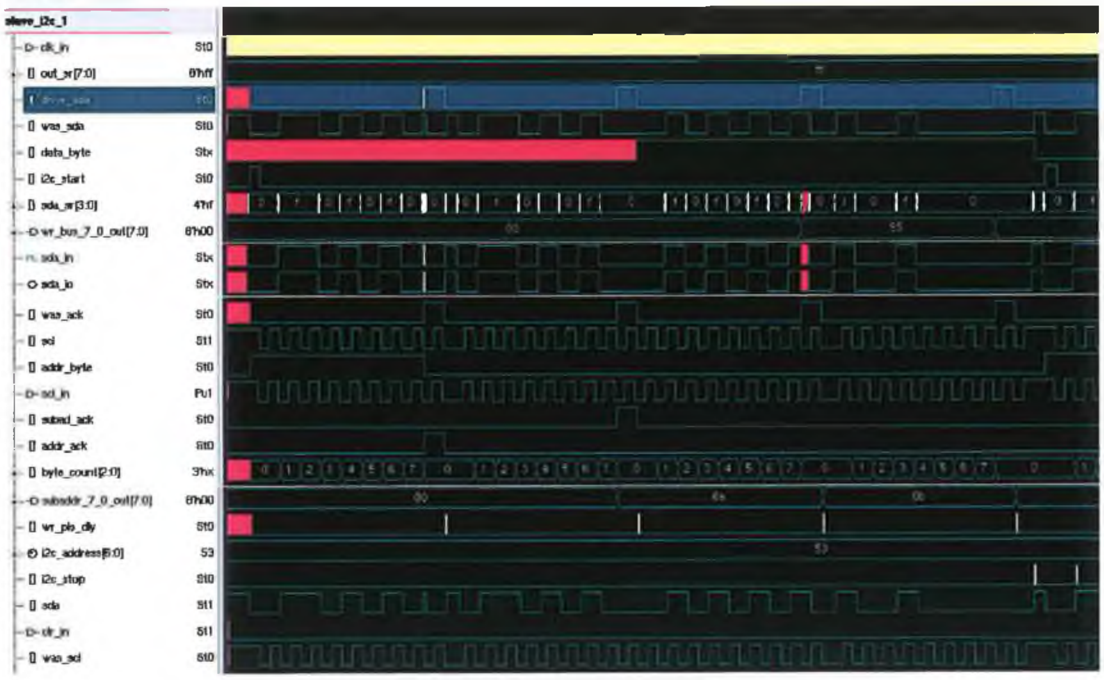


Figure 6.41 Slave Core

6.13.7 Testcase 7

To check the slave side of the BFM, the master core generates the start condition and the slave BFM waits to detect this. Then the master sends the address and waits for acknowledge. Once acknowledge is received the master sends data to the slave, another test will be performed where the slave writes to the master core. The resulting waveform from this test is illustrated in figure 6.42.

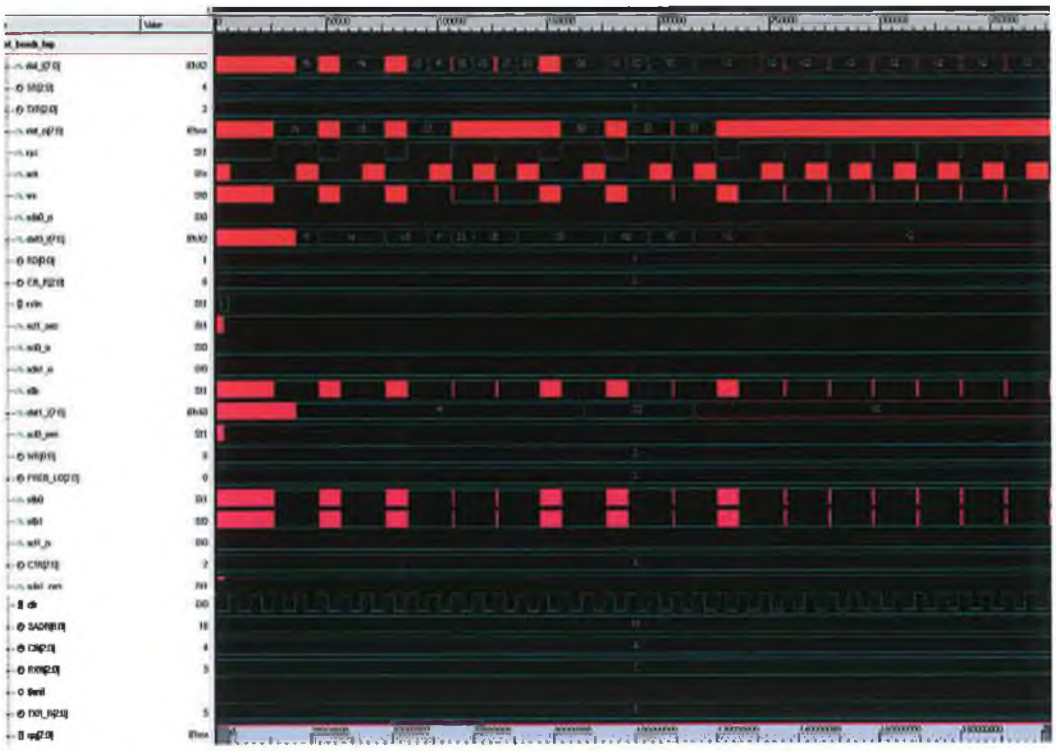


Figure 6.42 Master Core

6.13.8 Testcase 8

This test checks the multi master functionality of the BFM. In this case the test is checking the multi master arbitration. To test for the multi master feature of the BFM, a second BFM was created (a copy of the original BFM). The BFM is then tested against the master core. Listed below are some of the main aspects of the test. Figure 6.43 illustrates the waveform generated.

- Both Masters generate a start condition
- Masters 1 and 2 try to send address 0110101 with read bit
- Masters 1 and 2 wait for an acknowledge
- Master 1 tries to send data 01101110 while Master 2 tries to send 01101110, Master 2 fails to send data; therefore it goes into idle state and waits for the master 1 to finish transmitting data.

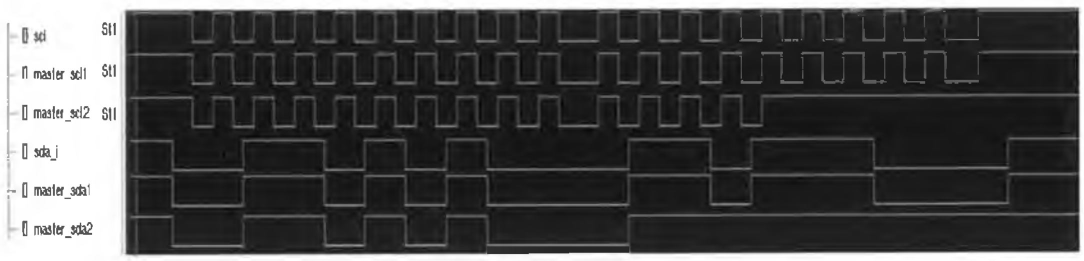


Figure 6.43 Multi Master Arbitration

6.13.9 Testcase 9

This test checks multi master clock synchronisation. Listed below is the test sequence, while figure 6.44 shows the waveform generated.

- Master 2 generates a start condition and pulls SDA low; Master 1 tries to generate a start condition and sees SDA has gone low so it pulls it SDA line low.
- Master 1 and 2 start counting off their low periods. Master 2 counts off it's low period first and goes into wait state as it cannot get SDA to go high. Master 1 finishes counting off it's low period and pulls SDA high. Master 1 and 2 start counting off their high periods Master 2 is first to finish counting off it's high period and pulls SDA low. This process continues until one of the masters loses arbitration, in this case master 2 loses arbitration. Both masters can finish a complete transmission if their data matches.

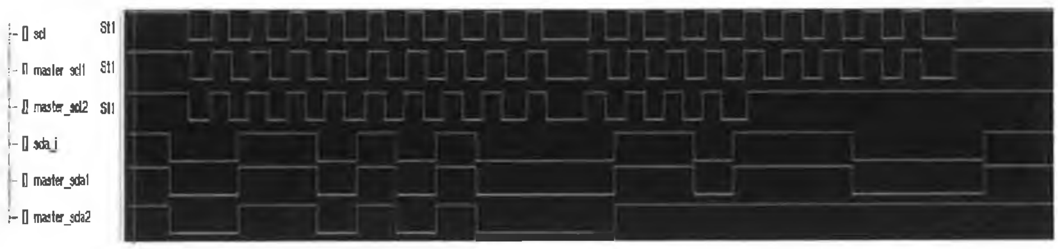


Figure 6.44 Clock Synchronisation

6.14 Conclusion

The main objective of the work described in this chapter is to build an advanced verification environment for the I2C protocol. The reason for building an advanced verification environment was to test the re-usability of the advanced verification environment described in chapter 5. The following conclusions have been drawn from building the advanced verification environment for the I2C protocol:

1. In converting the floating-point testbench directory structure little or no changes were made
2. In converting the testbench only minor changes were made.
3. Only in the BFM were major changes made, this was because the BFM represents the protocol being tested
4. The ability to use either random or direct tests or both was extremely useful in verifying the I2C protocol
5. The initial development time of the testbench has been reduced from 3 months in the case of the floating-point adder down to a couple of weeks for the I2C protocol. This has been achieved by using the advanced verification environment described in chapter 5. However, this reduction in time has been offset by the development of the more complex BFM. The development time for the BFM has taken a couple of months. Figure 6.45 illustrates the time taken to develop the advanced verification environment of the I2C protocol and also the time taken to verify the BFM.

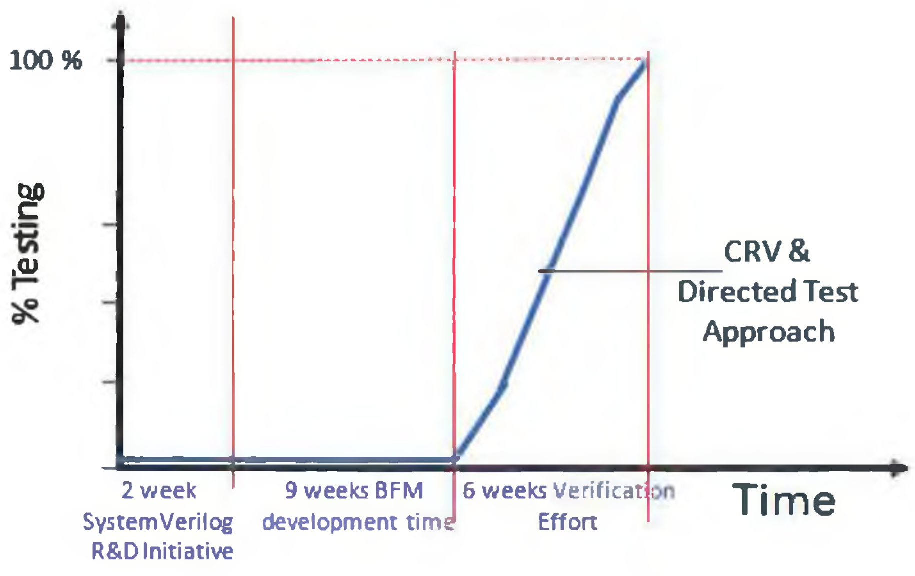


Figure 6.45 I2C BFM Development time

6. The development time for creating BFM could be reduced by creating a standard approach for creating BFMs.

    - Common approach for creating task and functions within the BFM

    - Consistent approach for creating state machines for the BFMs

However some difficulties have been encountered while building the advanced verification environment for I2C protocol. Listed below are some of the difficulties encountered

- The learning curve of understanding the I2C specification
- Integrating the master and slave BFMs into a single BFM
- Implementing the functionality of the protocol, especially the multi-master functionality
- Difficulties finding master and slave cores suitable for testing

Chapter 7 Formal Verification

7.1Introduction

In the previous chapters functional verification has been discussed. This chapter will investigate and review another verification method called formal verification. The reason for reviewing formal verification is that as chips have grown from tens of thousands of gates to millions today, simulation run times have stretched from a few hours to days, or even weeks. The volume of stimuli required to check every logic function and timing path has threatened to overwhelm even the best funded projects. As a result, reliance on simulation as the sole source of verification has proven impractical. This has meant that the major EDA and ASIC companies have looked at other solutions and ideas for verification. One such method is formal verification.

Formal hardware verification attempts to overcome the weakness of functional simulation by proving the similarities between some abstract specification and the design in hand.

There are 3 different techniques of formal hardware verification namely:

- Deductive Methods (Theorem Proving)
- Equivalence Checking
- Model Checking

This chapter will also investigate the different formal verification tools that are available on the market at the moment.

7.2 What is Formal Verification?

Formal verification means proving or disproving the correctness of a design with respect to a certain formal specification or property using formal methods or mathematical proofs [44]. The promise of formal verification is proving in the sense of mathematical proof, in contrast to traditional simulation and tests, which can only tell that nothing went wrong on the specific test tried. Formal verification can be viewed as giving the same effect as of exhaustive simulation.

Figure 7.1 illustrates the formal verification flow.

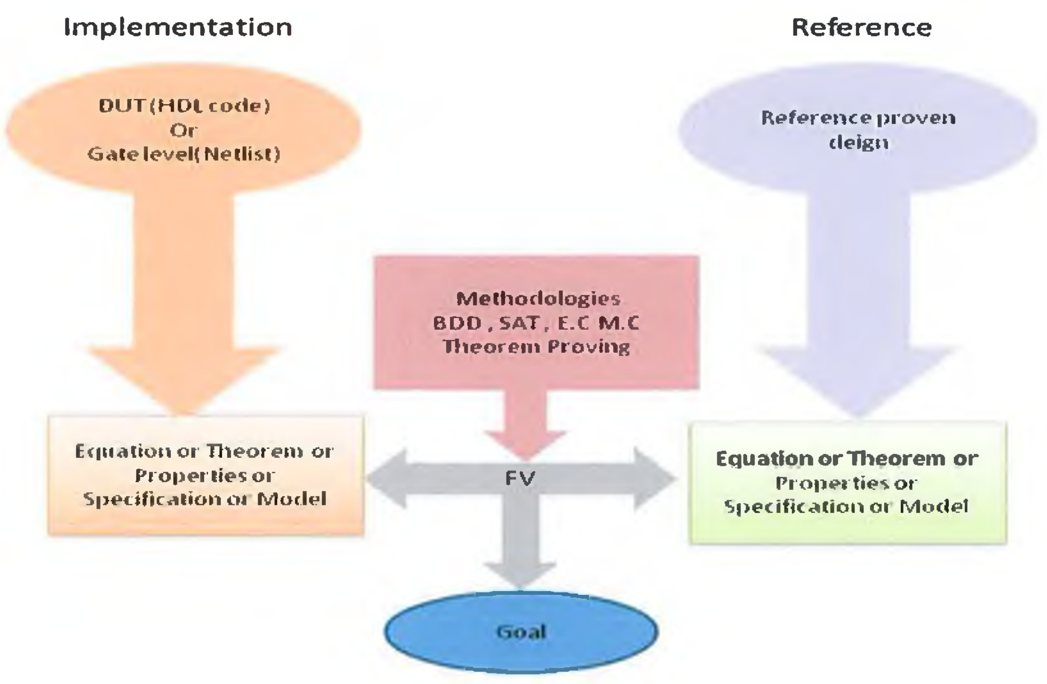


Figure 7.1 Formal Verification

A difficulty within formal verification is specifying the property to prove and creating an accurate model of the design. It is near impossible to prove complete system correctness:

- Is the design specification correct?
- Is the model accurate?
- Is the verifier correct?
- Is the computer used to run the verifier correct?

Ideally the model being verified is as close to the actual hardware as possible. The problem is that for complicated designs an abstracted model is usually needed to simplify the verification process.

7.3 Why Formal Verification?

The research into formal verification of computer hardware and software has been going on for decades. The focus traditionally had been on approaching the ideal of proving a system correct. These verification methods require considerable time and expertise to verify even simple systems [45]. In hardware design projects, hiring or training formal verification experts will result in delaying a product.

A major factor in the current interest in formal verification is a different approach that clearly recognises economic demands. The main factors include new verification techniques that support this emphasis and the high design complexity, short design cycles and strain on current validation methods [45]. What this all means is that basically bugs cost money, especially the hard to find bugs that surface late in the design cycle. The results of finding these bugs are that an extra spin of silicon may be needed that will delay a product launch and may even result in a massive product recall. Any technique that finds these bugs earlier is of enormous value. So instead of trying to certify correctness, formal verification is used as a powerful debugging tool. Therefore if the time and effort invested in formal verification is less than the time and effort saved by uncovering difficult bugs earlier, then formal verification is a winner, regardless of whether or not any claims about proving the system is correct.

7.4 History of Formal Verification

This section gives an overview of the history of formal verification. Listed below are some of the main events in the history of formal verification.

- Logic Theorist (1957)
- Resolution Theorem Proving (1965)
- Model Checking (Clarke and Emerson 1981)
- Practical Model Checking (McMillan 1992)
- Memories of FDVI (June 1994)
- Oin founded (1996 , formal verification tool , later purchased by Mentor)
- Verplex founded (1997 , formal verification tool, later became Cadence Conformal)
- Standardising Assertions (2003 to present)

7.41 Logic Theorist

In 1956 Allen Newell, J. C. Shaw and Herbert Simon introduced the first AI program, the Logic Theorist, to find the basic equations of logic as defined in Principia Mathematica [46] by Bertrand Russell and Alfred North Whitehead. It was the first program engineered to mimic the problem solving skills of a human being. The Logic Theorist would soon prove 38 of the first 52 theorems in Russell and Whitehead's Principia Mathematica and was even able to find new and better proofs for some.

7.42 Resolution theorem proving

Resolution theorem proving [47] was introduced by John Alan Robinson in 1965. Resolution is a rule of inference that provides theorem proving methods for proofing propositional and first order logic. In other words repeatedly applying the resolution rule in a suitable way allows the user to tell whether a propositional formula is satisfied and for proving that a first-order formula is not satisfied.

7.43 Model Checking

In 1981, Edmund Clarke and Allen Emerson were working in the U.S., and Joseph Sifakis working independently in France with J.P. Queille, authored seminal papers [48] that founded what has become the highly successful field of Model Checking. This verification technology offers two advantages. It provides an algorithmic means of determining whether an abstract model such as a hardware or software design satisfies a formal specification, such as a temporal logic formula. In addition it identifies the counter examples that show the source of the problem, which must be addressed should the stated property specification not hold.

7.44 Floating-point Divide (FDIV) Bug

The Pentium FDIV bug was a bug in Intel's original floating-point unit [49]. When a certain floating-point division operation was performed within the processors, it would produce incorrect results. According to Intel, there were a few missing entries in the lookup table used by the divide operation algorithm. The flaw was independently discovered by Professor Thomas Nicely, then at Lynchburg College, in October 1994.

Although encountering the flaw was extremely rare in practice (1 in 9 billion), Intel's initial handling of the matter was heavily criticized. Intel ultimately recalled the processors.

7.45 Standardising Assertions

Assertions are constructs, in the form of a group of statements, a task, or a function, that check for certain conditions on signals or variables in a design over a span of time. If the condition is violated, an error message is issued identifying the location of the occurrence. Assertions have been used in software and hardware design for quite some time, but only recently have there been coordinated efforts to standardise the use of assertions. Assertions benefit design and verification by removing ambiguity from specifications, finding bugs sooner and allowing fewer bugs through to production.

7.5 Deductive Reasoning

7.5.1 Introduction

Deductive reasoning is one of the two basic forms of valid reasoning [50]. While inductive reasoning argues from the particular to the general, deductive reasoning argues from the general to a specific instance. The basic idea is that if something is true of a class, this truth applies to all valid members of that class. The key is to be able to properly identify members of the class. Misinterpreting these members will result in invalid conclusions.

Deductive reasoning argues that if certain premises **(P)** are known or assumed, a conclusion **(C)** can be drawn from them. Deductive arguments are said to be valid or invalid, never true or false.

One of the most common and useful forms of deductive reasoning is the syllogism. The syllogism is a specific form of argument that has three easy steps.

- Every X has the characteristic Y.
- This thing is X
- Therefore, this thing has the characteristic Y.

Simple example of Deductive Reasoning is as follows:

- **P:** All men are mortal.
- **P:** Socrates is a man.
- **C:** Socrates is mortal.

7.5.2 Axioms and Deductive Reasoning

The first Axioms were developed by Euclid who was a mathematician in Alexandra in 300 BC. Euclid came up with 10 assumptions. Five of them were specific to geometry and 5 are not specific. These axioms can be found in Appendix 3.

Together, these common notions and postulates represent the axioms of Euclid's geometry. An axiom is a logical principle which is assumed to be true rather than proven, and which can be used as a premise in a deductive argument.

7.5.3 Deductive Reasoning in Verification

With deductive reasoning explained from an everyday point of view, this section explains how deductive reasoning is used for verification. Deductive reasoning is a verification methodology using axioms and proof rules to establish reasoning. The goal of deductive reasoning is to provide an automatic mathematical (logical) reasoning. When given a set of axioms and a set of interference rules, the proof implemented is semi auto constructed and mechanically checked by a theorem prover (PVS, ACL2, and HOL). To construct the proofs required it usually needs a great expertise in mathematics and logic. Finally with enough human involvement any theorem can eventually be proven.

7.5.4 Implementing Deductive Reasoning

To implement deductive reasoning the following steps should be followed:

- Implementation represented by a logical formula I (example Hoare logic [51])
- Specification represented by a logic formula S
- Correctness: Implementation $\rightarrow$ Specification (implication) or Implementation $\leftrightarrow$ specification (Equivalence)
- Proof is carried out at the synaptic level

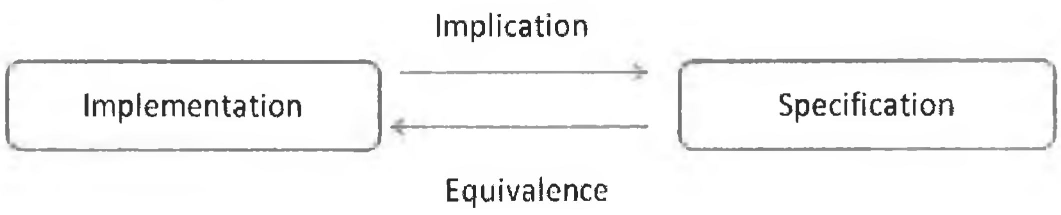


Figure 7.2 Implementing Deductive Reasoning

7.5.4 Theorem Proving System

A theorem proving system [52] is a program that mechanises a proof system. A calculus or proof system C for logic L consists of a set of axioms A and a set of inference rules R. The axioms are formulas of L and are generally "elementally" in the sense that they capture the basic properties of the logic's operators. The general form of an inference rule is:

$$\boxed{\frac{\alpha 1 \ldots\ldots \alpha k}{\beta}}$$

The formulas α1 ....., αk are called premises of the rule while β is called the conclusion.

There are three aspects which are particular to mechanised proof systems:

1. It can mechanically check a proof. This means that it verifies that a given sequence of formulas is indeed a deduction. This is usually not a very difficult task, all that is required is for formula to syntactically match the inference rules against the appropriate premises in the sequence and verify the formula. This is obtained though application of the rule.

2. It can assist in the construction of a proof. Given a set of assumptions and a goal, heuristic search techniques may be able to find a deduction $\beta = \hat{\partial}n$.

3. It permits the use of decision procedures. A decision procedure is an algorithm which decides the validity of a class of formula. BDDs provide a practical decision procedure for propositional logic. BDD will be explained in section 8.5.

A number of theorem proving systems have been implemented and many have been used for hardware verification, including HOL, ISABELLE and ACL2. These systems are well known by among other aspects, the proof style used, mathematical logic used, the way automatic decision procedures are integrated into the system and the user interface. Proof styles are often characterised as *forward* or *backward*.

- A *forward* proof starts with the axioms and assumptions then inferences are applied until the desired theorem has been proven.

- A *backward* proof starts with the theorem as a goal and applies the inverses of inference rules to reduce the theorem to simpler intermediate goals.

7.5.5 Types of Theorem Proving Systems

7.5.5.1 Higher Order Logic (HOL)

HOL is one of a family of theorem provers that share a common design called the Logic for Computable Function (LCF) architecture [53]. The LCF architecture is so named because its first use was in a theorem prover called LCF. The users of LCF architecture tools prove theorems in logic supported by the tool, which is the object language. The user does this by writing programs to construct a proof using a programming language (meta-language). The object language of the HOL system is a classical higher-order logic based on Church's simple theory of types [54].

HOL supports both forward and goal directed backward proof in a natural deduction style calculus. The HOL system is designed to support interactive theorem proving in higher order logic. The formal logic is interfaced with the programming language ML in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. The version of higher order logic used in HOL is predicate calculus with terms from the typed lambda calculus. This was originally developed as a foundation for mathematics [55]. The primary application of HOL was initially intended to be the specification and verification of hardware designs. However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

7.5.5.2 Prototype Verification System (PVS)

The specification language of PVS [56] is built on higher-order logic. It was developed at the Computer Science Laboratory of SRI International, California USA.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, simplification using decision procedures for equality and linear arithmetic, data and predicate abstraction, and symbolic model checking. The implementations of these primitive inferences are optimised for large proofs: for example, propositional simplification uses. The user defined procedures can combine these primitive inferences to yield higher level proof strategies.

Finally PVS includes a BDD based decision procedure for the relational mu-calculus and thereby provides an experimental integration between theorem proving and CTL model checking.

7.5.5.3 Boyer-Moore (Nqthm)

The Boyer-Moore logic is a first order, quantifier free logic, recursive function with equality and mathematical induction [57]. Nqthm has been reengineered and extended has been released under the name ACL2.

ACL2 contains axioms of primitive data types such as numbers and lists. ACL2 combines backward and forward methods to prove theorems. In the backward phase, a pending obligation is chosen, and the prover attempts to discharge it by applying a sequence of more and more general proof techniques. First, a simplifying stage applies conditional and congruence-based rewrites, a BDD-based propositional decision procedure, a linear arithmetic decision procedure and other simplification techniques to the obligation. If this step fails to reduce the obligation to true, ACL2 tries other proof techniques, the most general of which attempts to discover an induction scheme for the obligation. The forward aspect of ACL2 is that most of the proof techniques are rule driven. Previously proved theorems can be turned into rules which are added to a rule database and can be used by later proofs.

7.5.6 Advantage of Deductive Reasoning

The advantage of deductive reasoning is that it can be used for reasoning about an infinite state system. This task can be automated to a limited event. Deductive reasoning provides a high abstraction level where possible, expressive notation, powerful logic and reasoning.

7.5.7 Disadvantage of Deductive Reasoning

The major disadvantage of deductive reasoning is that it is often time consuming and is known to be labour intensive and require considerable time to learn and use [59]. In addition, there is no guarantee that the proving procedure will terminate. Even if a tool called a theorem prover has been developed to provide a certain degree of automation, its inherent characteristic makes it difficult to be used widely for verifying recent network protocol stacks.

However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof. An interactive and deep understanding of design and high order logic is required.

7.6 Model Checking

7.6.1 Introduction

Model checking is a means of checking that the model being verified satisfies the specification [57]. The analysis is performed algorithmically by searching the state space of the model. Today the term applies more generally, models need not be finite state and requirements can be written in a variety of other languages.

Model checking can be expensive even if limited attention is used to simple requirements. The problem is rooted in the fact that the number of states of a system grows exponentially with the number of variables used to describe it.

Given a model and a requirement as input, a model checker does not simply answer Yes or No. Rather, when the model does not satisfy the requirement, it produces a counter example, an evidence for the failure. This diagnostic information is extremely useful for debugging purposes. Model checking is an incremental process. The designer starts with a model of the system, checks a variety of requirements, and uses the feedback to modify the model.

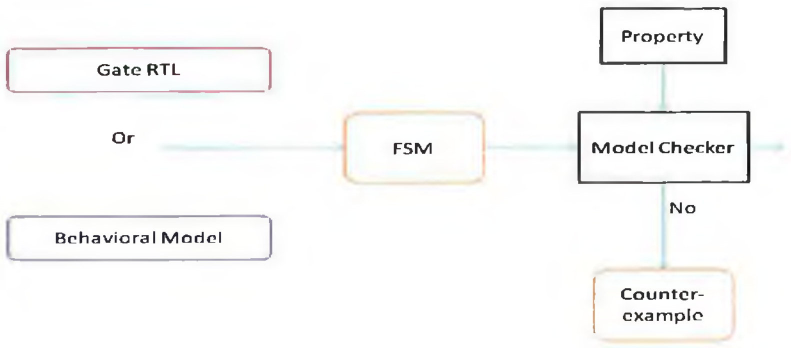


Figure 7.3 Model Checking

7.6.2 Model Checking in the Hardware Industry

Model checking has been most successful in hardware verification [58]. In 1992, the model checker SMV was used to pinpoint logical errors in the cache coherence protocol described in the IEEE Future bus and standard. This and numerous other case studies attracted a lot of attention from the hardware industry, who were eager to enhance capabilities of design automation tools. Model checking seems suitable to debug complex aspects of microprocessor designs. Today semiconductor companies such as Lucent, IBM, Intel, Motorola, and Siemens, have internal verification groups aimed at integrating formal verification into the design flow. Electronic Design Automation (EDA) companies such as Cadence and Synopsis are exploring ways to add verification capability to design tools. An important reason for the success of model checking in hardware verification is the ease with which it fits into the existing design methodology.

7.6.3 Model Checking and Kripke Structure

A **Kripke structure** [57] is a model used to give semantics (definitions of when a specified property holds) for modelling temporal logic. In the model checking domain, a Kripke structure is a graph having the states of the system as nodes and state transitions of the system as edges. It also contains labelling of the states in the structure with properties that hold in each state. The following is the formal definition.

Kripke Structure Let AP be a non-empty finite set of atomic propositions that denotes the properties of individual states we are interested in. A Kripke structure is a four tuple M = (s, s0, R, L), where

- s is a finite set of states,
- s0 $\in$ S is the initial state,
- R $\subseteq$ S $\times$ S is a transition relation,
- $L : S \longrightarrow 2AP$ is a function that labels each state with the set of atomic proposition true in this state

7.6.4 Types of Model Checking

- Computational Tree Logic (CTL)
- Linear Time Logic (LTL)
- Computational Tree Logic* (CTL*)
- Symbolic Model Checking

7.6.5 Computational Tree Logic (CTL)

CTL is a propositional, branching-time, temporal logic proposed by Clarke and Emerson in 1981 [48]. CTL is expressed as a tree like structure in which the future is not determined; there are different paths in the future, any one of which might be the actual path. CTL, one of the most popular logics for practical model checking, uses atomic propositions as the basis, and formulas are constructed from logical operators, temporal operators and path quantifiers to make statements about the Kripke structure. It is used in formal verification of software or hardware programs, typically by software applications known as model checkers who determine if a software or hardware programs possesses safety or liveliness properties. It is in a class of temporal logic that also includes LTL. CTL is a restricted subset of CTL* where each of the temporal operators must be immediately preceded by a path quantifier.

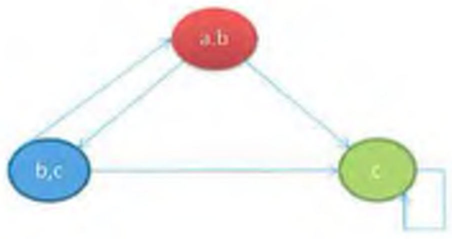


Figure 7.4 State transition graph of Kripke Model

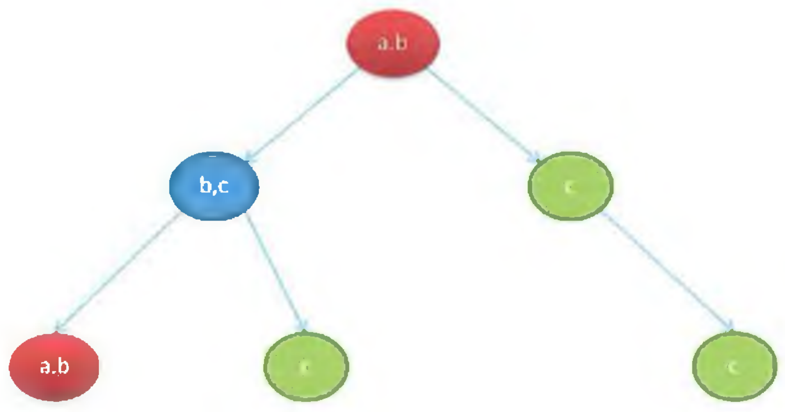


Figure 7.5 Infinite Computation Tree

Syntax of CTL

Syntax gives the rules to write correct formulae. Each CTL operator is a pair of symbols. The syntax of CTL is given in appendix 4.

Semantics of CTL

Semantics gives a meaning to well-formed formulas. Semantics is used to decide whether or not a given well-formed formula is true or false.

If f is a CTL state formula, and M is a Kripke structure, the formula M,s |= f denotes that f holds at states in the Kripke structure M. The semantics of CTL can be found in appendix 4.

7.7.6 Linear temporal logic (LTL)

LTL is a modal temporal logic with modalities referring to time. In LTL, one can encode formulas about the future of paths such as that a condition will eventually be true and that a condition will be true until another fact becomes true. [60]

LTL is built up from a set of propositional variables, logic connectives and the following temporal model operators:

- **X** for next (Next state.);
- **G** for always (globally);
- **F** for eventually (in the future);

- **U** for until;
- **R** for release.

The first three operators are unary, so that **X f** is a well formed formula (w.f.f) whenever f is a well-formed formula. The last two operators are binary, so that p **U** q is a well-formed formula whenever p and q are well-formed formulas.

Semantics

The semantics of LTL formulas is defined with respect to computation paths of a transition system. The syntax of LTL is given in appendix 4.

7.7.7 Computational Tree Logic *

CTL * [61] is a logic which combines the expressive powers of LTL and CTL , by dropping the CTL constraint that every temporal operator (X, U , F , G) has to be associated with a unique path quantifier (A,E). The syntax and semantics of CTL * can be found in appendix 4.

7.7.8 State Explosion Problem

State explosion is a major problem in Model Checking [62]. The number of global states of a concurrent system with many processes can be enormous. The asynchronous composition of n processes, each having m states, may have m^n states. A similar problem occurs with data. The state-transition system for an n-bit counter will have 2^n states. All Model Checkers suffer from this problem.

Fortunately, steady progress has been made over the past 27 years for special types of systems that occur frequently in practice. In fact, the state explosion problem has been the driving force behind much of the research in Model Checking and the development of new Model Checkers.

7.7.9 Symbolic Model Checking

One of the big problems with model checking is state space problem. To tackle these problems McMillan [62][63] while working on his PHD introduced a new method of model checking called Symbolic Model Checking in 1992. Symbolic model checking is a powerful formal verification technique that, contrary to theorem proving, requires no user assistance. It is able to verify that an implementation, modelled as a labelled finite state transition graph, satisfies its specification, given as a set of terms in some temporal logic.

In the fall of 1987, McMillan, then a graduate student at Carnegie Mellon, realised that by using a symbolic representation for the state transition graphs, much larger systems could be verified. The new symbolic representation was based on ordered binary decision diagrams (OBDDs). OBDDs provide a canonical form for Boolean formulas that is often considerably more compact form than normal formulas, and very efficient algorithms have been developed for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states , many orders of magnitude larger than could be handled by the explicit state algorithms. By using the original CTL Model Checking algorithm of Clarke and Emerson with the new representation for state transition graphs, it became possible to verify some examples that had more than 1020 states. Since then various refinements of the OBDD based techniques by other researchers have pushed the state count up to more than 10120.

7.7.11 What are BDDS

BDDs are a representation for Boolean formulas, which is canonical once an order on the variables has been established. A Boolean formula is a compact representation of the set of the states represented by the assignments which make the formula true. Similarly, the transition relation can be expressed as a Boolean formula in two sets of variables, one relative to the current state and the other relative to the next state.

This makes it possible to represent predicate transformers and fix points as BDDs. The basic Boolean operations are handled by means of standard algorithms for computing Boolean connectives with BDDs, and fixed point algorithms can be easily implemented in terms of basic BDD operations [45].

7.7.12 Advantage and Disadvantage of Symbolic Model Checking

Symbolic model checking has been used to verify a large variety of systems hardware descriptions, software and protocols. In practice, symbolic model checking is well suited for the verification of the control components of a system [63]. However it performs poorly with data parts. The reason is that BDDs are ill suited to represent arithmetic expressions or data intensive operations. Practically this means that symbolic model checking cannot be used to uncover bugs such as the one found in the Pentium chip floating-point division unit. An approach to verify this type of system has been to combine model checking using other data structures rather than BDDs to represent the data parts of the system under verification.

7.7 Equivalence Checking

7.7.1 Introduction

Modern circuit design flows increasingly employ formal verification techniques in order to ensure quality and reduce time to market by avoiding bug related design iterations. Equivalence checking has become a regular step in the flow. Modern tools for equivalence checking are capable of verifying designs with millions of gates in very short times.

The great success of equivalence checking technology is due to numerous research advancements in this field during the past decade. An important idea on which a typical equivalence checker is based is to exploit structural similarity between the two circuit models being compared [64]. Structurally similar circuits contain a lot of internal nodes that implement equivalent circuit functions. These internal equivalences, sometimes called cut points can be used to efficiently break the verification problem down into smaller ones.

7.7.2 What is Equivalence Checking?

Equivalence checking provides a solution where given two system models are asked whether these systems are equivalent with respect to some notion of conformance, or functionally similar with respect to their input/output behaviour [64]. The verification of the systems can be based on specific properties like transient or steady state response properties, in the time domain or the frequency domain. To find if there is a relationship between two designs exhaustive testing is done to prove two expressions are equivalent. This can be a difficult task for any reasonably large circuit. Instead, symbolic reasoning methods can prove or disprove equivalence using decision procedures over the whole range of inputs described symbolically. Therefore, it is possible to compare circuits on the same level of abstraction as well as on different levels.

The goal of Equivalence Checking is to ensure the equivalence of two given circuit descriptions.

These circuits might be given on different levels of abstraction, i.e. register transfer level or gate level. The main steps of an equivalence checker are as follows:

1. Translate both designs to an internal format.

2. Establish the correspondence between the two designs in a matching phase.

3. Prove equivalence or not equivalence.

4. In the equivalence case a counter-example is generated and the debugging phase starts.

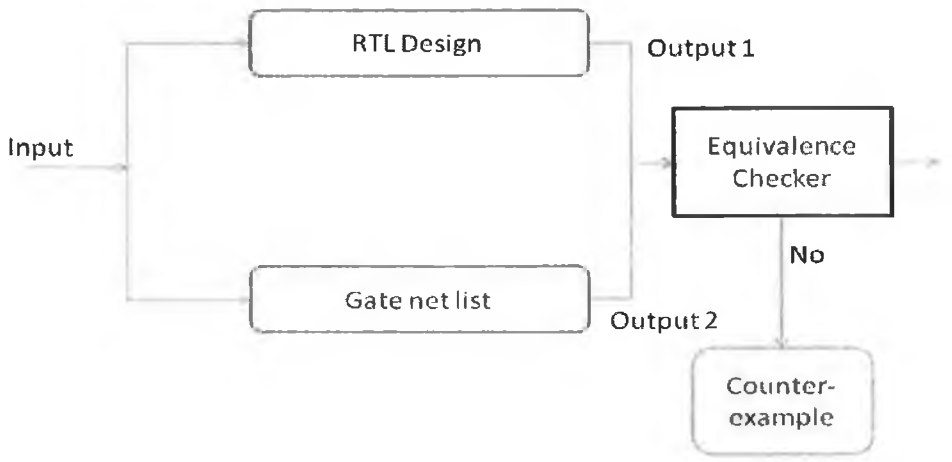


Figure 7.6 Equivalence Checking

7.7.3 Types of Equivalence Checking

- Combinational Equivalence Checking
- Sequential Equivalence Checking

7.7.4 Combinational Equivalence Checking (CEC)

Combinational equivalence checking (CEC) [65] plays an important role in EDA. The goal of combinational equivalence checking is to check whether two combinational circuits are functionally equivalent. This mean checking for all possible inputs, both combinational circuits have the same outputs. In a typical scenario, there are two structurally different implementations of the same design, and the problem is to prove their functional equivalence.

CEC is a framework commonly used for validating that logic synthesis does not alter the functionality of a design. For such applications, CEC operates on two versions of a design: the first is pre-synthesis, often an RTL version of a design which is used for functional verification; the second is post-synthesis, often a gate- or transistor level version of the design. CEC operates by correlating primary inputs and latch points between the two designs, and proving that this pairing guarantees equivalence of all primary outputs and next-state functions. CEC has become by far the most commonly used form of formal verification throughout the industry, due to ease of use and scalability

7.7.5 Combinational Equivalence Checking Issues

While powerful, CEC is for the most part limited in applicability to designs with 1:1 state element pairings [66]. CEC tends to become ineffective if significant sequential transformations are performed on a design, e.g., by optimisations such as retiming or FSM re-encoding, addition of power-saving logic such as clock-gating.

Due to the growing demands of hardware design, however, such transformations tend to comprise an increasing portion of the modifications performed during the life-cycle of a product. With a purely CEC-based methodology, this means that sequential transformations often require a full regression of the functional verification process, which is often time-consuming and uses simulation and can be incomplete. This means that performing sequential optimisations becomes somewhat of a bottleneck in design cycles, and that certain design sub-optimality are instead tolerated to avoid the risk of late introduced bugs [66].

7.7.6 Sequential Equivalence Checking (SEC)

Sequential equivalence checking (SEC) [66] is an equivalence checking method that can help to offset the limitations of CEC. SEC performs a true sequential check of input/output equivalence, hence is not limited to operation on designs with 1:1 state element pairings. The benefits of SEC are multiple. For example, one may use SEC to efficiently prove the correctness of sequential transformations that preserve design functionality, without a need to re-run lengthier often pointless functional verification regressions. This enables resource savings during the design cycle. This eliminates the risk associated with sequential transformations enabling more aggressive testing than otherwise would be tolerated, especially late in the design phase. Therefore a methodology based upon SEC implies that sequential transformations can be automated by synthesis tools. This allows more abstract reference models to be the basis of verification and synthesis.

In practice, SEC enables a more flexible set of applications than direct input/output equivalence, including white box functional checks. SEC can check against designs that are not even strictly equivalent but can be made so by disabling initialization/test/debug logic.

7.7.7 Sequential Equivalence Checking Issues

Unfortunately with SEC this benefit does not come without a price. SEC is much more computationally expensive than CEC. CEC often assumes not only 1:1 latch correspondence, but also 1:1 design hierarchy equivalence. This enables CEC to perform checks up to even the largest chip-level designs. SEC in comparison often does not assume latch or hierarchy equivalence. This often limits SEC in applicability to only smaller design units, mandating in cases a fair amount of user experience to verify a larger design and specify corresponding boundaries to yield a higher level equivalence proof [66]. In many cases, the lack of scalability of SEC results in an incomplete application of the technology.

7.7.8 Equivalence Checking Issues

Although equivalence checking technology has matured greatly during the last few years and designs with millions of gates can be handled, some specific problems remain to be difficult. Formal verification of arithmetic circuits, especially if multiplication is involved, is one of these problems. The problem occurs when an RTL specification of a circuit must be compared against a gate level implementation, e.g., when verifying the correctness of the logic synthesis step [64]. Figure 7.7 illustrates this case. The verification engines in a typical equivalence checker all operate on gate-level circuit models. Hence, in order to compare an RTL specification with a gate-level implementation, the frontend of the verification tool first has to generate a gate-level representation of the specification. The process is similar to the logic synthesis step that produced the implementation.

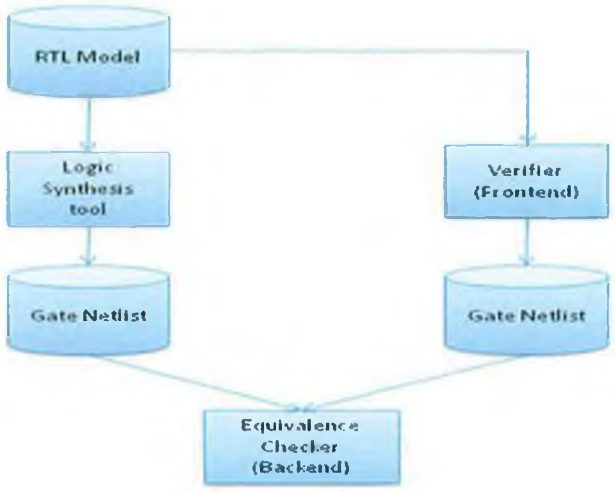


Figure 7.7 Equivalence check of a RTL model against gate net list

The two gate net lists can then be compared by the backend engines to verify equivalence or produce a counterexample. When the design contains arithmetic functions, this approach is bound to fail. The problem is that the two gate net lists hardly contain any structural similarity at all. The reason for this lies in the great flexibility when implementing arithmetic functions.

7.8 Formal Verification and Assertions

Assertions are a key ingredient to today's property based formal verification environment. Industry standard assertion languages such as SystemVerilog Assertions (SVA) and Property Specification Language (PSL) have very strong formal friendly assertion constructs that help the user to express their complex design behaviour at a high level of abstraction [67]. Although the language permits the user to express the complex temporal assertion behaviour in a concise way, it often yields complex logical behaviour at the lower level depending on the coding style used to write the assertions. Writing efficient assertions, especially formal friendly assertions, usually involves a longer learning curve not only to interpret the assertion constructs correctly but also to use them in a more effective and accurate ways.

In formal verification, assertions are used to define the property to be proven by formal methods as well as to define the constraint environment for the DUT [67]. The effectiveness and completeness of Formal Verification greatly relies on the accuracy of these assertions. A major part of the formal verification effort is spent on creating a robust set of properties and constraints. This development often slows down the easy and useful adoption of formal technology into the design verification process.

7.8.1 Property Specification Language (PSL)

PSL was the first hardware assertion language to receive IEEE standard (IEEE 1850 – 2005) in 2005 [68]. PSL offers many useful macros for expressing assertions; this means it can be used in VHDL, Verilog, SystemVerilog and SystemC. PSL also incorporates many temporal operators found in formal verification and model checking, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), it therefore can be used for both formal verification and simulation.

PSL is designed to capture design intent in an executable, formal, unambiguous manner. It is developed as a more "evolutionary" language than re-inventing the wheel. It uses many of the underlying HDL operators and expression syntax to build the Boolean expressions in properties rather than defining its own syntax and semantics for the same. However, wherever required, it defines its own syntax to build complex temporal relationship among the Boolean expressions.

7.8.2 SystemVerilog Assertions (SVA)

SVA are an integral part of the SystemVerilog language, which has IEEE standard (1800- 2005) [69]. Verification Methodology Manual (VMM) and Open Verification Methodology (OVM) contain guidelines for writing effective assertions. As with PSL, SVA also incorporates Temporal Operators that can be used for formal verification and simulation. Unlike PSL though SVA doesn't not have flavour macros for supporting HDL's and so is mostly used in SystemVerilog. Figure 7.8 illustrates an example of assertions used in the floating-point adder model.

```
//********************  look for invalid operation
property p2;
    @(posedge clk) invalid_op_flag != 1;
  endproperty

assert property (p2);

//********************  look for inexact

  property p3;
    @(posedge clk) inexact_flag != 1;
  endproperty
```

Figure 7.8 Assertions used in floating-point adder

7.9 Formal Verification Tools

The list of Formal Verification tools is shown in table 7.1

Supplier	Tool Name	Class of Tool	HDL	Design Level
Commercial Tools				
Chrystalis	Design Verifier	Equiv Checking	VHDL / Verilog	RTL /Gate
Synopsys	Formality	Equiv Checking	VHDL / Verilog	RTL /Gate
Cadence	Affirma	Equiv Checking	VHDL / Verilog	RTL /Gate
Compass	V Formal	Equiv Checking	VHDL / Verilog	RTL /Gate
Verysys	Tornado	Equiv Checking	VHDL / Verilog	RTL /Gate
Abstract Hardware LTD	Checkoff - E	Equiv Checking	VHDL / Verilog	RTL /Gate
IBM	BoolsEye	Equiv Checking	VHDL / Verilog	RTL /Gate
Cadence	FormalCheck	Model Checking	VHDL / Verilog	RTL
Abstract Hardware LTD	Checkoff - M	Model Checking	VHDL / Verilog	RTL /Gate
IBM	RuleBase	Model Checking	VHDL	RTL
Abstract Hardware LTD	Lambda	Theorem Proving	VHDL / Verilog	RTl /Gate
Public Domain Tools				
CMU	SMV	Model Checking	Own language	RTL
Cadence	Cadence SMV	Model Checking	Verilog	RTL
UC Berkeley	VIS	Model /Equiv Check	Verilog	RTL /Gate
Standford U	Murphy	Model Checking	Own language	RTL
Cambridge U	HOL	Theorem Proving	SML	Universal
SRI	PVS	Theorem Proving	LISP	Universal
UT Austin / CTI	ACL2	Theorem Proving	Lisp	Universal

Table 7.1 Formal Verification Tools

7.10 Future Trends of Formal Verification

In the future, progress will take place on the key topics of limiting state explosion, improved abstractions, better symbolic representations, broader parameterised reasoning techniques, and the development of temporal formalisms specialised to particular application domains [70].

Other areas where work needs to be done are:

- *Composition.* Understand how to compose methods, specifications, models, theories, and proofs.
- *Decomposition.* Develop more efficient methods for decomposing a computationally demanding global property into local properties whose verification is computationally simple.
- *Abstraction.* Real systems are difficult to specify and verify without abstractions. Identify different kinds of abstractions, perhaps tailored for certain kinds of systems or problem domains, and we need to develop ways to justify them formally, perhaps using mechanical help.
- *Reusable models and theories.* Rather than defining models and theories from scratch each time a new application is tackled, it would be better to have reusable and parameterized models and theories.
- *Combinations of mathematical theories.* Many safety critical systems have both digital and analog components. These hybrid systems require reasoning about both discrete and continuous mathematics. System developers would like to be able to predict how well their system will operate in the field.
- *Data structures and algorithms.* To handle larger search spaces and larger systems, new data structures and algorithms are used.

7.11 Conclusion

The main focus of this chapter is to review and summarises formal verification and the three main techniques within it; deductive reasoning, equivalence checking, model checking and to list the advantages and disadvantages of each. The main advantages and disadvantages of each are summarised in table 7.2.

Formal Methods	**Advantages**	**Disadvantages**
Model Checking	• No proofs needed • Fast • Produces Counterexamples • Temporal Logic	• State space explosion • Writing specification is difficult
Equivalence Checking	• Exhaustive testing	• Doesn't catch design errors at HDL level
Deductive Reasoning	• High abstraction and powerful logic expressiveness • Powerful reasoning • Unrestricted applications	• User intervention and guidance is necessary • Requires expertise for efficient use • Automated for narrow classes of designs

Table 7.2 Advantage & disadvantage of formal methods

Another reason for investigating formal verification is to see if it could be used within the advanced verification environment. Formal verification methods such as equivalence checking could be used to verify the floating-point adder model. This could be done by using equivalence checking tools like Synopsys Formality or Cadences Affirma. However, without more research into formal verification tools, it is not possible to determine if formal verification tools could be integrated into the advanced verification environment described in chapter 5.

Chapter 8 Conclusion and future work

8.1 Thesis Summary

The aim of this thesis has been to research SystemVerilog and the different methodologies that support the language and to develop and test an advanced verification environment that supports the major features of both SystemVerilog and VMM.

Chapter 2 investigates verification and the various verification techniques. The challenges associated with verification and the bottleneck in verification (up to 70 % of the time spent developing a chip can be spent in verification) is also discussed. Chapter 3 introduces and describes SystemVerilog a new verification language that has been created to help with the difficulties within verification. Also examined were the other verification languages that are available. The methodologies that support SystemVerilog and the tools that support SystemVerilog have been explained.

Chapter 4 describes the development of a basic test bench using SystemVerilog and VMM. This helped to gain knowledge of SystemVerilog. The main features incorporated in the testbench are the VMM classes vmm_data, vmm_xactor, vmm_env and some of the key SystemVerilog features including the interface, Assertions, program block, top module and constrained random verification.

Chapter 5 describes the development of the advanced verification environment, to create a reusable structure allowing similar DUT to be tested. This environment includes features not included in a basic verification environment such as functional coverage, reference model, scoreboard and a reporting mechanism. The environment also includes a unique way of incorporating both contained random tests and directed test within one environment. Using the advanced verification environment it was possible to verify the floating-point adder model fully.

Chapter 5 describes the VMM planner reporting mechanism. The VMM planner is a useful reporting tool as it gives a verification manager a top level view of the verification progress on a spread sheet. This verification progress can include the following: functional coverage scores, assertions and pass/fail information.

Chapter 6 discusses the development and testing of an advanced verification environment for the I2C bus. The environment built uses the same testbench structure and testbench architecture as described in chapter 5. To test the reusability of the advanced verification environment and also develop a BFM for an advanced protocol.

Chapter 7 reviews and summarises formal verification, the different techniques within formal verification and also the different tools that are available.

8.2 Conclusions

The following conclusions can be drawn from researching and developing test benches using SystemVerilog and VMM

- SystemVerilog is a very powerful verification language. This conclusion can be backed up by the survey in chapter 3 showing the steady growth that SystemVerilog has made within the verification industry.
- Some of the main SystemVerilog features such as assertions, functional coverage, interfaces, program block and clocking blocks help the verification engineer to quickly write test benches.
- Since the project started in early 2008 a large number of advances in SystemVerilog have been made.
    - OVM released
    - OVM / VMM interoperability library released
    - VMM 1.2 released
    - SystemVerilog and Verilog merged to created P1800-2009
    - RVM is been developed by Accellera to create a unified methodology
- The major EDA companies have been developing methodologies to support SystemVerilog. These methodologies are very useful in the creation of test benches. They help to make SystemVerilog a very powerful verification language.
- While these methodologies provide very powerful techniques for developing test benches, they do not offer a standardised way of creating verification environments. This research has created a standardised test bench environment.
- The advanced verification environment that was described in the thesis has been developed in such a way that it can be used to verify other DUTs with similar verification techniques. This has been achieved by creating a directory structure

that fits the requirements of a reusable environment. The environment has been built so that (apart from the BFM) only minimum modifications have to be made to the test bench to verify another DUT.

- The environments incorporate a unique way of having both random and direct testcases within the same testbench. Using regression scripts it is possible to run both random and direct tests together. This makes it possible to quickly verify the floating-point adder model.
- As well as offering a reusable environment the advanced verification environment includes key SystemVerilog features such as assertions and functional coverage. The verification environment also includes a reference model and scoreboard to complete the verification process of the floating-point adder.
- The verification environment includes VMM planner. At the time of development the VMM Planner application was in beta release stage. The VMM Planner application gives the Verification engineer a quick view of the functional coverage, code coverage, assertions and pass/fail report progress in a simple XML file that can be viewed in any office application.
- In creating the advanced verification environment of the I2C protocol the development time for the directory structure and the testbench has been significantly reduced.
- Formal verification will play a major part in verification of ASIC devices in the future.

8.3 Further Development

The advanced verification environment could be further enhanced with the following developments

- Use the interoperability library with the advanced verification environment so that both VMM and OVM can be used within it.

- The advanced verification environment described in this thesis uses VMM 1.0. A newer version of VMM 1.2 was released late 2009. Research should be done to check if they are any differences between VMM 1.0 and VMM 1.2. If differences are found they should be implemented within the verification environment.
- At the time of writing this thesis a unified methodology (Unified Verification Methodology (UVM)) is being developed by Accellera. This methodology will combine features from OVM and VMM. The advanced verification environment could be converted to the new methodology once available.
- The floating-point adder could be tested with code coverage.
- The advanced verification environment could be built for another protocol. The reason for building another verification environment would be to try and develop a standard approach for creating BFMs. Thus in turn would reduce the time spent in creating BFMs.
- More work could be done with VMM Planner. At the time of implementation VMM Planner was in beta stage.
- Formal verification could be integrated into the advanced verification environment.

References

[1]. Chris Spear "SystemVerilog for Verification" Volume I 2005 pp. 1-5

[2]. Srivatisa Vasudevan "Effective Functional Verification: Principles and processes" Volume I 2005 pp. 2-6

[3]. Andreas Meyer "Principles of Functional Verification" Volume I 2004 pp. 1-10

[4]. San Iman "Step by step Functional Verification with System and OVM" Volume I 2008 pp. 3-7

[5]. Janick Bergeron "Writing Testbenches Using SystemVerilog" Volume I 2006 pp. 1-10

[6]. A Molina O Cadenas "Functional Verification: Approaches and Challenges" Latin American Applied Research 2007

[7]. Tom Fitzpatrick "Design for verification methodology allows silicon success" EEdesign 2003

[8]. Doulos Ltd. "SystemVerilog Golden Reference Guide" Volume II 2003 pp. 1-11

[9]. Accellera "SystemVerilog 3.1 final" 2005 [online] [Accessed: Feb.05,2008]

[10]. Abhiram Rao "What is SystemVerilog" 2002 [online] http://electrosoft.com/systemverilog/introdution.html [Accessed: Feb.05,2008]

[11]. Donald E Thomas "Verilog Hardware Description Language" 2002 Volume V pp.2-7

[12]. Doulos Ltd "Verilog Golden Reference Guide" Doulos Ltd. 2002 Volume I pp.6-7

[13]. Synopsys "What is OpenVera" 2002 [online] http://www.openvera.com [Accessed: Jan.22, 2010]

[14]. Synopsys "Synopsys Commitment to EDA tool Interoperability Expands to Verification" 2001

[15]. Stuart Sutherland "Verilog 2001 What's New and Why You Need It" 2000

[16]. Kacper Technologies Pvt. Ltd. "Program Block" 2004 [online] http://systemverilog.in.program-block.php [Accessed: Jan.22, 2010]

[17]. Doulos "SystemVerilog Direct Programming Interface Tutorial" 2002 [online] http://doulos.com/knowhow/systemverilog/tutoral.dpi [Accessed: Feb.05, 2008]

[18]. Clifford E Cumming Sunburst Design LTD "SystemVerilog Assertions Design Tricks and SVA Bind" SNUG 2009

[19]. Thomas Anderson Janick Bergerson Eduard Cerny EEtimes "SystemVerilog reference verification methodology: introduction" 2006

[20]. Mike Mintz, Robet Edendathl "Hardware Verification with SystemVerilog" 2007

[21]. Gabe Moretti "Mentor supports VMM code in OVM environment" EDA Design line 2009

[22]. J.Bergeron, E.Cerney, A.Hunter, A.Nightingale, "Verification Methodology Manual for SystemVerilog" 2005 pp. 2-9

[23]. Mark Glasser "Open Verification Methodology Cookbook" Springer 2009 pp. 1-10

[24]. Synopsys "VCS Datasheet" 2001

[25]. Synopsys "Discovery Visualisation Environment" 2005

[26]. Mentor Graphics "Questa Product Guide" 2003

[27]. Cadence "Cadence Incisive Design Team Simulator" 2005

[28]. "Design and Verification Conference Attendee Questionnaire Results" 2007, 2008, 2009"

[29]. "IEEE Std. 754-1985, Binary Floating-Point Arithmetic", IEEE Press, Piscataway, N.J., 1985

[30]. K Keane, D McNamara, Ferghal Morgan, C Ryan "Floating-point versus Fixed Point Implementation of DSP Algorithms" ISSC 1999

[31]. ASIC World "SystemVerilog Interface" [online] http://www.asic-world.com/systemverilog/interface8.html 2001 [Accessed: Feb.22, 2010]

[32]. Synopsys "VMM: Introduction to Design Verification, A Quick start guide" 2008

[33]. Ben Cohen "VMMing "A SystemVerilog Testbench" SNUG San Jose 2005

[34]. Amre Sultan "VMM For Dummies" SNUG Boston 2006

[35]. Charles Li "Simplify the Setup of Constrained- Random-Test Environments" Chip Design Magazine 11-2007

[36]. Leena Singh "Advanced Verification Techniques" 2004 Volume I pp. 40

[37]. Synopsys "Unified Coverage Reporting User Guide" 2005

[38]. Synopsys "VMM Planner User Guide" 2008

[39]. Nancy Pitt "Are we there Yet?" IBM SNUG San Jose 2008

[40]. SocCentral "Synopsys Extends VMM Methodology for Higher Functional Verification Productivity" 2007

[41]. NXP "UM10204 I2C BUS Specification" 2007

[42]. Esacademy "I2C BUS" [online] 2001 http://www.esacademy.com/en/library/technical-articales-and-documents/miscelleous/i2c-bus.html [Accessed: Nov.03, 2008]

[43]. "Opencores" http://opencores.org 2003 [Accessed: Mar.07, 2009]

[44]. Jean P Mermet "Fundamentals and standards in hardware description languages" ISBN 0-7923-2513-3 Kluwer 1993

[45]. AJ HU "Formal Hardware Verification with BDDS: an Introduction" IEEE Pacific Rim Conference on Communication, Computer and Signal Processing 1997

[46]. Alfred Whitehead Bertrand Russell "Principia Mathematica to *56" Cambridge University Press 1997 volume II pp. 1-4

[47]. John Alan Robinson Andrei Voronkov "Handbook of Automated Resolution" 2001 volume I pp.1-3

[48]. Edmund Clarke and Allen Emerson Joseph Sifakis * JP Queille "Model Checking Algorithmic Verification and Debugging" 2009

[49]. Dusko Koncaliev Earlham College "Pentium FDIV BUG" [online] http://www.cs.earlham.edu/~dusko/cs63/fdiv.html 2001 [Accessed: Sep.07, 2009]

[50]. Mary Elizabeth "Painless Speaking" 2003 volume I pp. 206 -207

[51]. C.A.R Hoare "An Axiomatic Basis for Computer Programming" 1969

[52]. W.W Bledsoe " Contemporary Mathematics Automated Theorem Proving: after 25 years" American Mathematical Society 1984 Volume I pp. 89- 98

[53]. T.F Melham "Higher Order Logic & Hardware Verification" Cambridge`Press 1993 volume VII pp.1-17

[54]. P.Andrews Stanford Encyclopedia of Philosophy "Church Type Theory" 2001

[55]. J.Joyce "Hol Theorem Proving System and Its Applications" 1993 Volume VI pp 174 - 178

[56]. S.Owre, Dr.J.Rushby, Dr.N. Shankar, "Prototype Verification System" 11th International Conference on Automated Deduction 1992

[57]. University Texas "Boyer Moore Theorem Prover" 2001

[58]. E.M Clarke, O.Grumberg, D.Peterel, University Texas "Model Checking" 1999 Volume I pp. 1-10

[59]. Christoph Kern Mark Greenstreet "Formal Verification In Hardware Design: A Survey" 2002

[60]. Rajeav Alur Thomas Heinziger "Computer Aided Verification LTL" 2002

[61]. F Raimodi "Computation Tree Logic and Model Checking" UCL 2007

[62]. E .Clarke O Grunberg "Lecture Notes in Computer Science, Progress on the State Explosion Problem in Model Checking" 2001 Volume 2000/2001 pp.176 - 194

[63]. Ken McMillian "Symoblic Model Checking" 1993 Volume I pp. 25 - 33

[64]. Rolf Drechsler "Advanced Formal Verification" 2004 Volume I pp.77 - 81

[65]. Randal E. Bryant James H Kukula "Formal Methods for functional Verification" 2003

[66]. Jason Baunjauler Hari Mony Viresh Paruthi "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations" 2006

[67]. Manoj KumarGaura Gupta Mandor Munishwar "Developing Assertion IP for Formal Verification " 2009

[68]. Marc Boule Zeljko Zilic "Generating Hardware Assertions Checkers" 2008 Volume I pp. 13 -30

[69]. Abhiram Rao "Assertions in SystemVerilog" [Online] http://electrosoft.com/systemverilog/assertions1.html 2005 [Accessed: Nov.16, 2009]

[70] O.Grumberg, D Veith, E.Allen Clarke Carnegie Mellon University "25 Years of Model Checking , The birth of Model Checking" 2008 pp.1-27

Appendix 1 Publication

The Development of Advanced Verification Environments using System Verilog

Martin Keaveney†, Anthony McMahon†, Niall O'Keeffe*, Kevin Keane†, James O'Reilly†

**Department of Electronic Engineering*
GMIT, Dublin Road, Galway

†Chipright Ltd.
IiBC, GMIT, Dublin Road, Galway

†e-mail: martin.keaveney@chipright.com
†e-mail: anthony.mcmahon@chipright.com
**e-mail: niall.okeeffe@gmit.ie*
†e-mail: kevin.keane@chipright.com
†e-mail: james.oreilly@chipright.com

Abstract — **This paper describes a System Verilog Verification Methodology Manual (VMM) test bench architecture that is structured to gain maximum efficiency from both constrained random and directed test case development. We specify how a novel form of directed traffic can be implemented in parallel to a complete random traffic generator inside a reusable directory structure which takes full advantage of coverage and assertion techniques. The paper uses an IEEE-754 compliant Floating-Point adder model as part of a case study that illustrates a complete set of results from using this test bench solution**

Keywords **– System Verilog, VMM, Test bench, Verification.**

I INTRODUCTION

System Verilog is the industry's first unified Hardware Description and Verification Language (HDVL). It became an official IEEE standard (IEEE 1800™) in 2005 under the development of Accellera [1]. The language can be viewed as an extension of the Verilog language with the added benefit of supporting Object Orientated (OO) constructs that have been used within the software development industry for many years. This is the first time that OO constructs have been made available to both digital design and verification engineers. As a result, engineers need to consider which OO constructs are applicable to their requirements.

Historically, the digital design community has focused much of its attention towards developing languages and tools primarily for use in designing an ASIC device. Today, an ASIC design flow can be viewed as being almost compliant amongst design houses. However the same cannot be said about verification flows. More recently, the key EDA tool vendors have researched new verification methodologies and languages, namely Specman E, System Verilog, VMM, AVM and the emerging OVM. These methodologies implement functional code coverage, assertion based coverage and constrained random techniques inside a complete verification environment.

The EDA vendors have recognised that a standard verification approach is required and they support the structures needed to build an advanced verification environment. However, they do not specify how the test bench environment solution should be built. This remains the responsibility of the verification manager, who must also create a test bench which is re-usable for future chip sets.

Whilst current synthesis tools do not yet support all the System Verilog features, verification engineers can take advantage of the OO constructs to develop high-level test scenarios that are both scalable and reusable.

One goal of verification tool designers is in reducing the complexity of the test bench environment. This paper describes a standardised test bench structure that engineers can quickly and easily use to increase verification productivity. This paper describes a System Verilog Verification Methodology Manual (VMM) test bench using a Single Precision floating-point adder model as a case study. The VMM defines a set of rules and recommendations to be used in the design of System Verilog test benches [2]. It utilises OO techniques to build pattern generators, transactors, scoreboards and an environment which can be extended to yield a library of test cases.

Developing test benches that utilise a complete random approach often require additional directed test cases in order to verify all elements defined within the verification plan.

A survey of current VMM based test bench architectures highlights the need for a directed test mechanism to be considered at the start of the verification process in order to yield an efficient verification environment [3].

This paper outlines a directory structure and test bench architecture that provides a straightforward way to constrain and manipulate data direct test case scenario that is extrapolated using a Verilog. We implement functional coverage, score boarding and assertions to verify the Design under Test (DUT).

The structure of this paper is as follows: Section II describes the rationale for developing a standardised test bench structure. Section III outlines the essential components of an advanced verification environment. Section IV describes the directed test case method and its implementation. The direct test case scenario is extrapolated using a combined data member class. Its use in maximising the efficiency of the test bench is outlined. Section V outlines the floating-point test bench case study. Integration of random and direct tests, code coverage, assertions and a scoreboard together in one standard verification environment is described. Section VI concludes the paper. Results are highlighted and indicate the point in the verification effort where maximum efficiency is achieved together with a productivity improvement over previous efforts to verify the same device.

II THE NEED FOR A STANDARDISED TEST BENCH

At present there is no standardised test bench architecture that is described within System Verilog or within the verification industry. The different verification methodologies developed by the major EDA companies have very powerful verification techniques but they do not describe how a targeted verification environment should be built.

In developing a chip, it is plausible that engineers within a single organisation working in different teams will develop varying strategies for verifying their IP blocks. This scenario leads to an organisation having to support and integrate multiple test benches within a single chip development. Ultimately, this will lead to problems at the system level at the point where much energy is spent in getting the device through tape-out. There is a clear need for organisations (regardless of size) to develop and maintain a strategy that supports block level, sub-system level and system level scenario's.

It is the responsibility of the verification manager and verification engineers to develop solutions that will support the development of a chip from start to finish. To achieve this, they need to: -

1. Produce a test bench architecture that will support the different functional verification levels right through to gate level simulation
2. Develop a structure that can be easily built upon, supported, documented and extended to yield a library of test cases
3. Provide their managers with detailed reporting mechanisms that can be extrapolated from the environment speedily and make a decision as to when the device is fully verified

These tasks are typical requirements placed upon verification engineers today. The EDA developers have identified these tasks and empowered their tools to help the engineer develop solutions to their problems. Yet, different strategies are still required depending on the type of chip being developed, e.g. a network chip requiring a lot of block intensive traffic to pass through it or a SoC level chip that requires a degree of connectivity testing as part of its verification process.

It is possible to build these differing types of verification solutions using System Verilog and VMM. In our case study, we use a block intensive approach that can be extended to support the development of directed test case mechanisms – often used in providing a solution to a connectivity focused test bench. We leverage our test bench solution with the Synopsys VCS simulation tool to provide a platform that an engineer can use to generate the verification result speedily.

III THE ESSENTIAL TEST BENCH COMPONENTS

The key elements within a System Verilog and VMM style verification environment are:

- Classes
- Assertions
- Functional Coverage
- Scoreboards
- Random Pattern Generators
- Transaction Level Model
- Regression capability

The key features of these elements are briefly described here:

Classes are used to develop transactor components, score boards, cover points and data structures that are used to manipulate the stimulus provided to them by pattern generators. Classes are written in System Verilog code and utilise the same software techniques as would be present in a C++ style environment.

Assertions are used to verify properties of a design that manifest themselves over time. An example of an assertion is shown in Figure 1.

```
//look for invalid operation
property p2;
  @(posedge clk) invalid_op_flag != 1;
Endproperty

assert property (p2);
```

Figure 1: Example Assertion Code

An assertion is a property that should hold true at all times. Assertions are used to specify assumptions and help identify design faults on the DUT entity [2]. Within the verification environment, all assertions are declared in the interface file description – the point where the pins on the DUT are specified. Assertions provide a powerful pinpointing mechanism in highlighting unexpected behaviour on an interface and can be compiled into a detailed reporting structure in the Synopsys Unified Report Generation (URG) tool.

Functional Coverage is a means of measuring what has been verified in a design in order to check off items in the verification plan [2].

The implementation of functional coverage consists of a number of steps. Firstly, code is added in the form of cover groups in order to monitor the stimuli being put on the DUT. The reactions and response to the stimuli are also monitored to determine what functionality has been exercised. Cover groups should be specified in the verification plan. Within a test case scenario, their usefulness is ascertained by analysing the RTL code and understanding the data they have recorded during the simulation.

Cover points become more powerful within a simulation when they are crossed together to identify greater levels of abstraction within a design. Cover points provide a powerful mechanism in identifying areas of functional code coverage within a design. As with assertions, they can be compiled into a detailed reporting structure in the URG tool.

Scoreboards are built from classes and are used to perform checks between the DUT data and the expected data. For the solution in this paper, the expected data comes from a reference model. Within a VMM style verification environment, data is usually communicated to the scoreboard via channels or mailboxes as an efficient means of transmission inside a test bench. Scoreboards can operate dynamically during a simulation and they provide the final piece of information when determining whether a test case has passed or not. Pass / fail log files are used to identify this information when running regression suites of test cases.

Random Pattern Generators are used to create stimuli that fit the data members described within a transactor's base class. The random pattern generator creates complete random data to be applied to a Bus Functional Model (BFM) that communicates with the DUT. In some cases the data is constrained so as to limit the amount of non-useful data generated.

Transaction Level Model (TLM) is a term used to identify a reference model that implements the required functionality at a very high level of abstraction. Given the RTL code is written at a low level of abstraction, TLM's are usually written more quickly and efficiently capture the expected function of the device at an approved degree of abstraction.

Regression capability is a means in which the verification environment can support and highlight certain information that can be extrapolated at a later point in time to determine different test case results. Regression runs are merged together to collate different code coverage results to indicate the overall degree of verification applied to the DUT.

IV DIRECTED TEST CASE MECHANISM

The verification engineer must design the architecture of the verification environment at the outset to achieve the ability to support both directed and constrained random test cases in an efficient manner [3].

In designing the Floating-Point test bench solution, our goal is to develop a robust and easy to use mechanism that facilitates the development of test cases with minimal impact to the test bench code. Figure 2 illustrates the key elements on which the architecture is based:

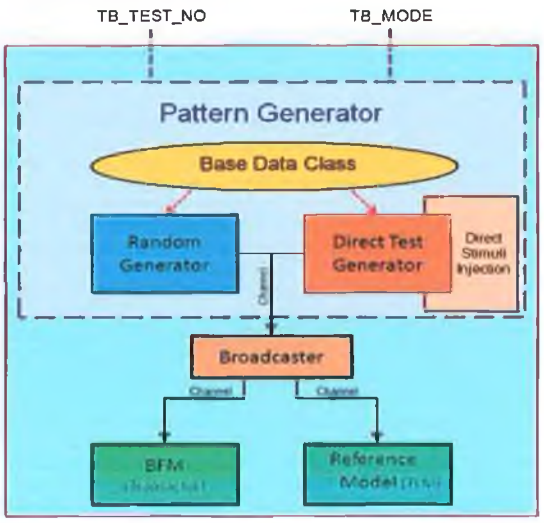


Figure 2: TB Architecture

Figure 2 illustrates that the base data class (containing the Floating-Point data member constructs) provides a basis for which the two pattern generators create stimulus. The question is how to select the required pattern generator within a test case and how to focus traffic through the Directed Test Generator (DTG). The solution is as follows:

Firstly, parameters are used on the top level of the test bench to determine which generator is initiated (TB_MODE) and a second parameter to identify the test case (TB_TEST_NO) to be implemented. To achieve this, the following coding hierarchy is implemented:

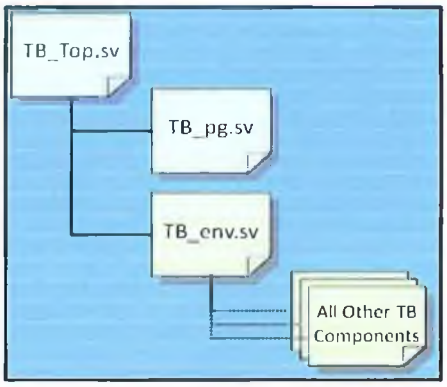


Figure 3: Test Bench Coding Hierarchy

The parameters are inserted into the tb_top.sv file and replicated down through the hierarchy of the design to the environment level (tb_env.sv). The test bench transactors, scoreboards and other key elements are instantiated inside the environment file.

Secondly, tasks and functions are constructed inside the DTG to put known stimuli out onto a channel for transmission to a broadcaster component. The broadcaster facilitates the transmission of the data received to multiple blocks efficiently. These tasks and functions are the commands and instructions that a user of the test bench can utilise to program test case scenarios quickly and efficiently. Using this structure, the implementation of class extensions within the environment facilitates the development of directed test cases.

By implementing this solution, it is possible to develop different test case scenario's to support both low and high levels of abstraction quite easily. The architecture enables each pattern generator to be uniquely associated with a corresponding library of individual test cases. Each test case is accessed by the test number parameter (TB_TEST_NO). This format is aligned to the test bench with scenario layer diagram [4] that illustrates varying test case abstraction levels.

Figure 4 below illustrates an approach that verification engineers can use when implementing a constrained random approach to fulfil their task of verifying a device. It supports the use of developing a set of test cases that use multiple seeds to implement several verification runs. The constrained random approach differs from the traditional verification flow whereby the engineer maximises the use of functional coverage to close off the task rather than build specific test cases to gradually close off on the verification task. Traditionally the verification engineer will use the verification plan to write directed test cases that exercise the various features of the DUT, thereby making steady incremental progress.

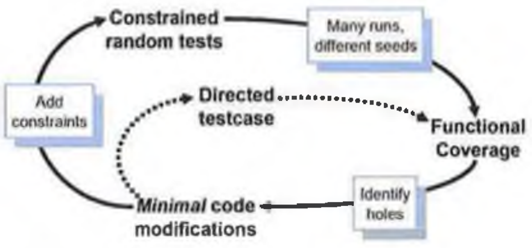


Figure 4: The Paths to close off the verification task using Functional Coverage

There is a downside to implementing the traditional approach. Specifically, it is necessary to write 100% of all stimuli being transmitted to the DUT. The technique is extremely time-consuming and each test is targeted at a very specific part of the design. Given that up to 70% of an ASIC lifecycle is spent in the verification task [5] [6], any improvement in reducing that figure is warmly received within the industry. This quantifies the effort undertaken by the EDA tool development community in creating and supporting such methodologies.

Through the use of constrained random testing it is possible to reduce the time required to fulfil the verification task. A random test will often cover a wider range of stimuli than a directed test. It can be manipulated by changing the seed on which the stimuli is created to yield a greater result from a single test scenario. However, there is still a need to write some directed test cases in order to target areas of a chip that a random test case may have difficulty in verifying [7]. Figure 4 shows the paths to achieve complete code coverage. It illustrates the greatest amount of time is spent in the outer loop, making minimal code changes to add new constraints and only writing directed tests for the few features that are very unlikely to be reached by random tests.

V IEEE-754 FLOATING-POINT TEST BENCH CASE STUDY

The IEEE-754 standard [8] describes both single precision and double precision formats in detail with a description of invalid behaviour. We use a single precision adder model as a DUT for the case study. The model supports the 5 IEEE exception flags associated with the standard.

Firstly, in order to implement all the features associated with an advanced System Verilog test bench, it is important to first align on a directory structure that fit's the requirement. Figure 5 below outlines the structure used.

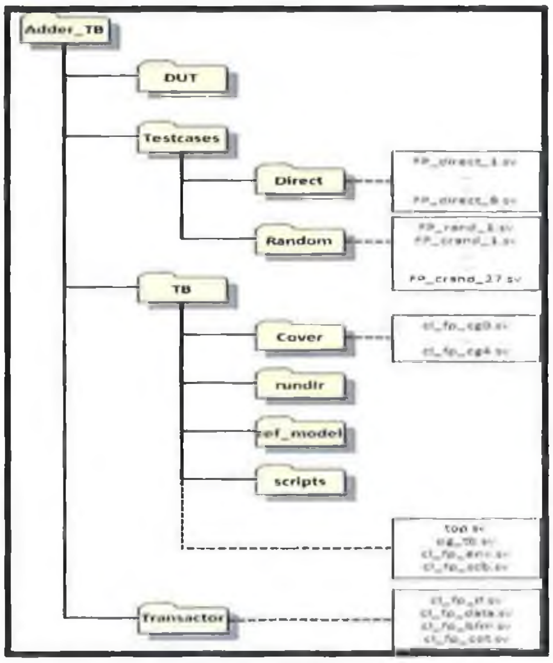


Figure 5: Test Bench Directory Structure

The directory structure contains four main directories, namely the DUT, Test Bench (TB), test cases and the transactor (BFM) component. The TB directory is further subdivided to contain the TLM reference model, coverage points, associated TB scripts and the run directory where the test bench is executed. The test cases directory contains two libraries of test cases, namely random and direct. These test cases are written as classes and are implemented as extensions of the verification environment class. The scripts directory contains files that support the development of a regression flow that is needed to maximise the productivity of the solution.

Collectively, this structure facilitates the development of test cases within the overall test bench architecture, providing the user with a programmable method of developing both random and directed test cases.

This directory structure fits the requirements of the floating-point adder component but it is structured in a way that would incorporate another DUT where similar verification techniques are to be implemented upon it. It is a robust and re-usable directory structure.

Secondly, in implementing the features, using the pattern generation technique described previously, the verification environment file (tb_env.sv) is used to connect up other components. Figure 6 below illustrates the floating-point adder test bench architecture.

The test bench architecture contains each of the elements described in section III and it can be seen how each element is tied with the directory structure as shown in Figure 5.

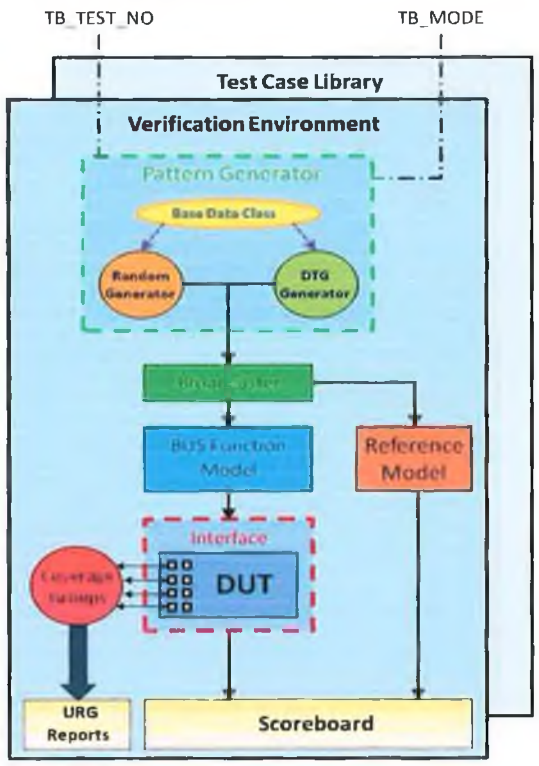


Figure 6: Test Bench Architecture

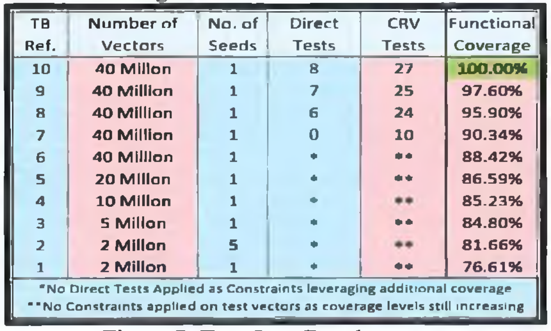
The test case library consists of both random and directed test cases. A limited number of Constrained Random Vectors (CRV) needed to be developed - given the ability to run the same random test case with multiple seeds. For the directed tests, the use of additional constraints enabled the development of scenarios that led to reaching the target of 100% functional code coverage.

Regression scripts written in Tcl are linked to the test case library and are used to fully implement the test-case scenarios. The scripts allow the user to choose between random or directed test cases and contain parameters that link into the verification environment.

VI RESULTS AND CONCLUSIONS

The initial time spent in developing the flow yielded an efficient solution that overcame and aided the development of directed test cases, as identified as an important step in [3]. This R&D time would not be required for another project as it encompasses a complete SystemVerilog verification flow. Thus, reusing it will alleviate an engineer from one verification task and allow them to spend more time actually figuring out how to verify the DUT.

Figure 7 illustrates the number of test case vector scenarios applied by the verification environment. It illustrates that the complete random approach yielded a 90% coverage result by applying 40 million random test vectors. Increasing the number of random vectors above this figure had no impact on the coverage levels and so CRV and directed tests were required to reach the goal of 100% functional coverage. This aligned t the flow illustrated in Figure 4.



TB Ref.	Number of Vectors	No. of Seeds	Direct Tests	CRV Tests	Functional Coverage
10	40 Millon	1	8	27	100.00%
9	40 Million	1	7	25	97.60%
8	40 Million	1	6	24	95.90%
7	40 Million	1	0	10	90.34%
6	40 Million	1	*	**	88.42%
5	20 Millon	1	*	**	86.59%
4	10 Millon	1	*	**	85.23%
3	5 Millon	1	*	**	84.80%
2	2 Millon	5	*	**	81.66%
1	2 Millon	1	*	**	76.61%

*No Direct Tests Applied as Constraints leveraging additional coverage
**No Constraints applied on test vectors as coverage levels still increasing

Figure 7: Test Case Development

We found that using a single seed for the constraint solver in VCS with a large number of test vectors proved to be more productive than applying several seeds with an equivalent number of test vectors in yielding a coverage result.

Test scenario 7 in Figure 7 above (40 million vectors with a single seed) yielded the greatest degree of productivity from the CRV approach without running 100+ million vectors. This point was chosen as the starting point from which to develop directed test cases. Had there been no time invested into developing the test bench structure at the start of the process, it would have resulted in a longer time frame to close off the verification task at this stage.

We developed a suite of 36 test cases to achieve 100% coverage. This consisted of one completely random test, 27 constrained random tests and 8 directed tests. It took 1 months' time to verify the DUT using this method. Given the same device was verified previously using a standard Verilog flow that took 2 months of verification effort, this results in a productivity gain of 1 month.

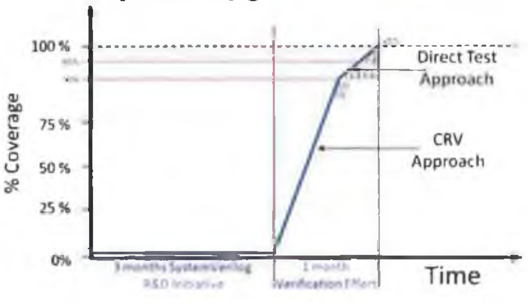
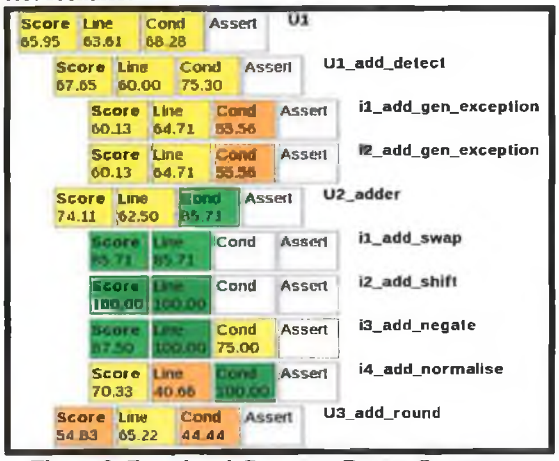


Figure 8: Coverage Process

Figure 8 illustrates a graph depicting the level of functional coverage achieved as per the test scenario's outlined in Figure 7. It highlights the usefulness of the CRV approach up to a certain point after which the directed tests were required to close off the verification task.

Multiple cover points were identified inside the DUT and they were grouped together to yield cross coverage points that encapsulated the functionality of the DUT. Leveraging the solution with the Synopsys URG tool provided ample report mechanisms to indicate when the task was complete.

Figure 9 depicts a typical report derived from the Synopsys URG tool. It illustrates a sample functional and code coverage report from running a test scenario.



Score	Line	Cond	Assert	
Score 65.95	Line 63.61	Cond 68.28	Assert	U1
Score 67.65	Line 60.00	Cond 75.30	Assert	U1_add_detect
Score 60.13	Line 64.71	Cond 55.56	Assert	i1_add_gen_exception
Score 60.13	Line 64.71	Cond 55.56	Assert	i2_add_gen_exception
Score 74.11	Line 62.50	Cond 85.71	Assert	U2_adder
Score 85.71	Line 85.71	Cond	Assert	i1_add_swap
Score 100.00	Line 100.00	Cond	Assert	i2_add_shift
Score 87.50	Line 100.00	Cond 75.00	Assert	i3_add_negate
Score 70.33	Line 40.66	Cond 100.00	Assert	i4_add_normalise
Score 54.83	Line 65.22	Cond 44.44	Assert	U3_add_round

Figure 9: Functional Coverage Report Summary

VII ACKNOWLEDGEMENTS

We would like to recognise the support of Damien Melville in Chipright for valuable feedback in the early phases of the project.

VIII REFERENCES

[1] IEEE Standard for System Verilog, IEEE Std 1800™ - 2500
[2] Verification Methodology Manual for System Verilog by, Janick Bergeron. Eduard Cerny, Andrew Nightingale and Alan Hunter, Springer US, ISBN: 978-0-387- 25538-5
[3] Richard McGee, Paul Furlong, Silicon and Software Systems, "VHDL to System Verilog, constrained Random verification of USB 2.0 host controller sub-system", SNUG 2006, Europe
[4] System Verilog for Verification by Chris Spears, Springer US, ISBN 978-0-387-27038-3
[5] Motorola Russia, "Using System Verilog for IC verification", SNUG 2005, Europe
[6] Effective Functional Verification, Springer US, ISBN: 978-0-387-28601-3
[7] Torstein Hernes Dybdahl, "How much System Verilog coverage do I need?", SNUG 2007, Europe
[8] ANSI / IEEE Std 754 - 1985, IEEE Standard for Floating-Point Arithmetic, IEEE Computer Society Press, Los Alamos, California, 19

Appendix 2 Functional Coverage Cover Groups

The following cover groups have been created for the testing of the floating-point adder model.

Type	Group/Cross Name	Coverpoint Name	Reg Used	Bins Created	Description
Cover Group 1	**cg_fp_number_type_bin**	cp_fp_type1_bin (1)	type1	6	Data 1 floating-point number type.
		cp_fp_type2_bin (2)	type2	6	Data 2 floating-point number type.
Cross Bins	fp_number_type_binss	Cross 1 and 2		36	Cross covers all 36 possible number types which can be added in the DUT
Group 2	**cg_fp_swap_bin**	cp_fp_swap_bin (1)		5	List of data 1 and data 2 exponent greater and equal to each other
		cp_fp_data1_exp_bin (2)		10	Covers different exponent ranges for data 2
		cp_fp_data2_exp_bin (3)		10	Covers different exponent ranges for data 1
Cross	cross_fp_swap_bins	Cross 1 , 2 ,3		136	Not all cross bins are possible, any illegal bins are ignored
Group 3	**cg_fp_shift_bin**	cp_fp_abs_D_bin (1)	ABS_D	26	List of absolute values
		cp_fp_comp_bin (2)	comp	2	complement == 0 complement == 1
Cross	cross_fp_shift_bins	Cross 1 and 2		5.	Crosses values of abs_D and values of complement
Group 4	**cg_fp_negate_bin**	cp_fp_comp_bin (1)	comp	2	complement == 0 complement == 1
		cp_fp_MSB_bin (2)	S_add[47]	2	S_add[47] == 0 S_add[47] == 1
		cp_fp_carry_out_bin (3)	S_add[48]	2	S_add[48] == 0 S_add[48] == 1
Cross	cross_fp_negate_bins	Cross 1 ,2 ,3		negate no_negate	The 'negate' cross bin crosses comp_bin(comp) MSB_bin (one) and carry_out_bin (no_carry).
Group 5	**cg_fp_normalise_bin**	cp_fp_norm_temp_bin (1)	temp	3	temp_case_11 cannot exist

		cp_fp_Shift_Val_bin (2)	Shift_val	25	Each bin accounts for each state of the variable shft_val. Even shft_val 24-30
Cross	cross_fp_normalise_bins	Cross 1 and 2		26 bins	Not all cross bins are possible, any illegal bins are ignored
Groups 6	**cg_fp_POS_POS_input _data_bin**	cp_fp_data1_sign_bin (1)	data1 [31]	1	Data 1 positive
		cp_fp_data2_sign_bin (2)	data2 [31]	1	Data 2 positive
		cp_fp_data1_exp_bin (3)	data1 [30:23]	10	Data 1 exponent ranges
		cp_fp_data2_exp_bin (4)	data2 [30:23]	10	Data 2 exponent ranges
		cp_fp_data1_mant_bin (5)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (6)	data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_number_type_b ins	Cross 1 ,2,3,4,5,6		10000	Cross Data 1 and Data 2 for different ranges
Groups 7	**cg_fp_NEG_NEG_inpu t_data_bin**	cp_fp_data1_sign_bin (1)	data1 [31]	1	Data 1 negative
		cp_fp_data2_sign_bin (2)	data2 [31]	1	Data 2 negative
		cp_fp_data1_exp_bin (3)	data1 [30:23]	10	Data 1 exponent ranges
		cp_fp_data2_exp_bin (4)	data2 [30:23]	10	Data 2 exponent ranges
		cp_fp_data1_mant_bin (5)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (6)	data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_number_type_b ins	Cross 1 ,2,3,4,5,6		10000	Cross Data 1 and Data 2 for different ranges
Groups 8	**cg_fp_POS_NEG_input _data_bin**	cp_fp_data1_sign_bin (1)	data1 [31]	1	Data 1 positive
		cp_fp_data2_sign_bin (2)	data2 [31]	1	Data 2 negative
		cp_fp_data1_exp_bin (3)	data1 [30:23]	10	Data 1 exponent ranges
		cp_fp_data2_exp_bin (4)	data2 [30:23]	10	Data 2 exponent ranges
		cp_fp_data1_mant_bin (5)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (6)	data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_number_type_b ins	Cross 1 ,2,3,4,5,6		10000	Cross Data 1 and Data 2 for different ranges
Groups 9	**cg_fp_NEG_POS_input _data_bin**	cp_fp_data1_sign_bin (1)	data1 [31]	1	Data 1 negative
		cp_fp_data2_sign_bin (2)	data2 [31]	1	Data 2 positive
		cp_fp_data1_exp_bin	data1	10	Data 1 exponent

		(3)	[30:23]		ranges
		cp_fp_data2_exp_bin (4)	data2 [30:23]	10	Data 2 exponent ranges
		cp_fp_data1_mant_bin (5)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (6)	data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_number_type_bins	Cross 1 ,2,3,4,5,6		10000	Cross Data 1 and Data 2 for different ranges
Group 10	**cg_fp_POS_POS_exp_data_bin**	cp_fp_abs_D_bin (1)	abs_D	256	All bins will show each value of the variable abs_D
		cp_fp_data1_sign_bin **(2)**	data1 [31]	1	Data 1 positive
		cp_fp_data2_sign_bin (3)	data2 [31]	1	Data 2 positive
		cp_fp_data1_mant_bin (4)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (5)	Data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_absd_mant_bins	Cross 1,2,3,4,5	25600		Cross covers ranges of abs_D with ranges of mantissas data 1 and dat 2
Group 11	**cg_fp_NEG_NEG_exp_data_bin**	cp_fp_abs_D_bin (1)	abs_D	256	All bins will show each value of the variable abs_D
		cp_fp_data1_sign_bin (2)	data1 [31]	1	Data 1 negative
		cp_fp_data2_sign_bin (3)	data2 [31]	1	Data 2 negative
		cp_fp_data1_mant_bin (4)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (5)	Data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_absd_mant_bins	Cross 1,2,3,4,5		25600	Cross covers ranges of abs_D with ranges of mantissas of data1and data2
Group 12	**cg_fp_POS_NEG_exp_data_bin**	cp_fp_abs_D_bin (1)	abs_D	256	All bins will show each value of the variable abs_D
		cp_fp_data1_sign_bin (2)	data1 [31]	1	Data 1 positive
		cp_fp_data2_sign_bin (3)	data2 [31]	1	Data 2 negative
		cp_fp_data1_mant_bin (4)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (5)	Data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_absd_mant_bins	Cross 1,2,3,4,5		25600	Cross covers ranges of abs_D with ranges of mantissas of data1and data2
Group	**cg_fp_NEG_POS_exp_**	cp_fp_abs_D_bin (1)	abs_D	256	All bins will show

13	**data_bin**				each value of the variable abs_D
		cp_fp_data1_sign_bin (2)	data1 [31]	1	Data 1 negative
		cp_fp_data2_sign_bin (3)	data2 [31]	1	Data 2 positive
		cp_fp_data1_mant_bin (4)	data1 [22:0]	10	Data 1 mantissa ranges
		cp_fp_data2_mant_bin (5)	Data2 [22:0]	10	Data 2 mantissa ranges
Cross	cross_fp_absd_mant_bins	Cross 1,2,3,4,5		25600	Cross covers ranges of abs_D with ranges of mantissas of data1and data2
Group 14	**cg_fp_rounding_bin**	cp_fp_G_bin (1)	Add_result[23]	2	Add_result[23] == 0 Add_result[23] == 1
		cp_fp_S_bin (2)	sticky	2	sticky == 0 sticky == 1
		cp_fp_round_Ov1_bin (3)	Round_Ov1	2	Round_Ov1 == 0 Round_Ov1 == 1
		cp_fp_round_Uv1_bin (4)	Round_Uv1	2	Round_Uv1 == 0 Round_Uv1 == 1
		cp_fp_L2_bin (5)	new_product [23]	2	new_product[23] == 0 new_product[23] == 1
		cp_fp_check_exp_bin (6)	check_and_expl	2	check_and_exp1 == 0 check_and_exp1 == 1
		cp_fp_round_Ov_bin (7)	Ov	2	Ov == 0 Ov == 1
		cp_fp_LGS_bin (9)	Add_result[24] Add_result[23] sticky	5	5 bins created represent the five cases in the truth table below
		cp_fp_Shift_Val_bin (10)	shft_val	25	Each bin accounts for each state of the variable shft_val.
	cross_GS_bins	4,5,1,2,3,6,7,8		11	Cross bins represent rounding logic in the DUT.
	cross_LGS_bins	9,10		125	Cross bins represent all LGS rounding parameters with the normalise block shift value.

Appendix 3 Axioms in Deductive Reasoning

5 Assumption relating to geometry called common postulates are:

1. It is possible to draw a straight line from any point to any point.
2. It is possible to produce a finite straight line continuously in a straight line.
3. It is possible to describe a circle with any centre and distance.
4. All right angles are equal to one another.
5. If a straight line falling on two straight lines makes the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which are the angles less than the two right angles.

5 Assumptions not relating to geometry are: and are called common notions:

1. Things which are equal to the same thing are also equal to one another.
2. If equals be added to equals, the wholes are equal.
3. If equals be subtracted from equals, the remainders are equal.
4. Things which coincide with one another are equal to one another.
5. The whole is greater than the part.

Appendix 4 Model Checking Types

The following are the syntax and semantics of CTL, LTL and CTL *. They are different types of model checking.

Syntax of CTL

- $f1 ::= \top \mid \bot \mid P \mid (\neg f1)$
- $(f1 \wedge f2) \mid (f1 \vee f2) \mid (f1 \rightarrow f2)$
- $AX f1 \mid EX f1$
- $A[f1 U f2] \mid E[f1 U f2]$
- $AG f1 \mid EG f1$
- $AF f1 \mid EF f1$
- $P ::= p1 \mid p2 \mid p3 \mid$

CTL Operators

They are eight basic CTL operators, which are listed below.

- AX f: on **A**ll paths, f is true in the ne**X**t state
- EX f : on som**E** path, f is true in the ne**X**t state
- AF f : on **A**ll paths, in some **F**uture state f is true
- EF f : on som**E** path, in some **F**uture state f is true
- AG f : on **A**ll paths, in all future states (**G**lobally) f is true
- EG f: on som**E** path, in all future states (**G**lobally) f is true
- AU($f1$, $f2$) : on **A**ll paths, $f1$ is true **U**ntil $f2$ is true
- EU($f1$, $f2$) : on som**E** path, $f1$ is true **U**ntil $f2$ is true

Semantics of CTL

1. $M, s \models \top$ and $M, s \not\models \bot$ for all states s.
2. $M, s \models p \Leftrightarrow p \in L(s)$.
3. $M, s \models \neg f1 \Leftrightarrow M, s \not\models f1$.
4. $M, s \models f1 \wedge f2 \Leftrightarrow M, s \models f1$ and $M, s \models f2$.
5. $M, s \models f1 \vee f2 \Leftrightarrow M, s \models f1$ *or* $M, s \models f2$.
6. $M, s \models f1 ! f2 \Leftrightarrow M, s \not\models f1$or $M, s \models f2$.
7. $M, s \models AX f1 \Leftrightarrow M, s' \models f1$ for all states s0 with s ! s0.

Equivalences of CTL

- $AX f1 \equiv \neg EX(\neg f1)$
- $AG f1 \equiv \neg EF(\neg f1)$
- $AF f1 \equiv \neg EG(\neg f1)$
- $A[f U g] \equiv \neg E[\neg g\, U \neg f \wedge \neg g] \wedge \neg EG\, \neg g$

LTL Semantics

- $\pi \models p \leftrightarrow p \in L(s)$, where s is the first state of π.
- $\pi \models \neg f1 \quad \pi \neq f1$.
- $\pi \models f1 \wedge f2 \leftrightarrow \pi \models f1$ and $\pi \models f1$.
- $\pi \models X f1 \leftrightarrow \pi\ 2 \models f1$.
- $\pi \models G f1 \leftrightarrow$, for all $i \geq 1$, $\pi\ i \models f1$.
- $\pi \models F f1 \leftrightarrow$, for some $i \geq 1$, $\pi\ i \models f1$.
- $\pi \models f1\ U f2 \leftrightarrow$ f there is some $i \geq 1$, such that $\pi\ i \models f2$ and for all j with $1 \leq j < i$, $\pi\ j \models f1$.
- $\pi \models f1\ W f2 \leftrightarrow$ either there is some $i \geq 1$, such that $\pi\ i \models f2$ and for all j with $1 \leq j < i$, $\pi\ j \models f1$; or else $\pi\ k \models f1$, for all $k \geq 1$.
- $\pi \models f1 R f2 \leftrightarrow$ either there is some $i \geq 1$, such that $\pi\ i \models f1$ and for all j with $1 \leq j \leq i$, $\pi\ j \models f2$; or else $\pi\ k \models f2$, for all $k \geq 1$.

LTL Equivalences

One can easily establish various equivalences for LTL, including the following:

- $G f1 \equiv \neg F \neg f1$
- $X f1 \equiv \neg X \neg f1$
- $F(f1 \vee f2) \equiv F f1 \vee F f2$
- $G(f1 \wedge f2) \equiv G f1 \wedge G f1$
- $F f1 \equiv \top U f1$
- $G f1 \equiv \bot R f1$
- $\neg(f1 U f2) \equiv \neg f1 R \neg f2$
- $\neg(f1 R f2) \equiv \neg f1 U \neg f2$
- $f1 W f2 \equiv f1 U f2 \vee G f1$
- $f1 W f2 \equiv f2 R (f1 \vee f2)$
- $f1 R f2 \equiv f2 W (f1 \wedge f2)$

Another important equivalence is: $f1 U f2 \equiv \neg(\neg f2 U (\neg f1 \wedge \neg f2)) \wedge F f2$, which holds for all LTL formulas $f1$ and $f2$.

CTL * Syntax

If f is a CTL* state formula, it must be constructed by one of the following rules:

- f is an atomic proposition.
- f = $\neg$g, where g is a state formula.
- f = g $\vee$ h, where g, h are state formulas.
- f = **A**p, where p is a path formula.
- f = **E**p, where p is a path formula.

If p is a CTL* path formula, it must be constructed by one of the following rules:

- p is a state formula.
- p = $\neg$q, where q is a path formula.
- p = q $\vee$ r, where q, r are path formulas.
- p = **X**q, where q is a path formula.
- p = **G**q, where q is a path formula.
- p = **F**q, where q is a path formula.
- p = q**U**r, where q, r are path formulas.

CTL * Semantics

1. *If* $f1 \in AP$*, then* $M,s \models f1 \Leftrightarrow f1 \in L(s)$.
2. $M,s \models \neg f1 \Leftrightarrow M,s \not\models f1$.
3. $M,s \models f1 \vee f2 \Leftrightarrow M,s \models f1$ *or* $M,s \models f2$.
4. $M,s \models A(p1) \Leftrightarrow$ *for all paths* π *starting with s, such that* $M,\pi \models p1$.
5. $M,s \models E(p1) \Leftrightarrow$ *there exists a path* π *starting with s, such that* $M,\pi \models p1$.
6. $M,\pi \models f1 \Leftrightarrow$ *s is the first state of* π *and* $M,s \models f1$. *i.e., each state formula is also a path formula.*
7. $M,\pi \models \neg p1 \Leftrightarrow M,\pi \not\models p1$.
8. $M,\pi \models p1 \vee p2 \Leftrightarrow M,\pi \models p1$ *or* $M,\pi \models p2$.
9. $M,\pi \models \mathbf{X}p1 \Leftrightarrow M,\pi1 \models p1$.
10. $M,\pi \models \mathbf{G}p1 \Leftrightarrow$ *for all* $k \geq 0$, $M,\pi k \models p1$.
11. $M,\pi \models \mathbf{F}p1 \Leftrightarrow$ *there exists a* $k \geq 0$, $M,\pi k \models p1$.
12. $M,\pi \models p1\,\mathbf{U}p2 \Leftrightarrow$ *there exists a* $k \geq 0$ *such that* $M,\pi k \models p2$ *and for all* $0 \leq j < k$, $M,\pi \models p1$.