View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repository@USM

REUSABLE AUTOMATED AGENT FOR UNIVERSAL VERIFICATION METHODOLOGY SYSTEM TESTBENCH

By

R. LOGEISH RAJ S/O RAJUMANIKAM

A Dissertation submitted for partial fulfilment of the requirement for the degree of Master of Science (Electronic Systems Design Engineering)

AUGUST 2015

ACKNOWLEDGEMENT

First and foremost, I would like to record my appreciation and thanks to Dr. Rosmiwati binti Mohd Mokhtar. Dr. Rosmiwati has been ever present in guiding me to conduct this research successfully. Without her guiding hand, I would be utterly lost. I would also like to thank my manager who has been supportive throughout the course of my study and research.

I would like to express thanks to my parents, Raju Manikam and Usha. They laid the foundation for me to succeed in life. Last but not least, a special thanks to my beloved wife Sarani. She has been patient and understanding throughout this research. Without her support, encouragement and love, life would have been very tough.

TABLE OF CONTENTS

Acknowledgement ii

Table of Contents iii

List of Tables vii

List of Figures viii

List of Abbreviations xi

Abstrak xiii

Abstract xv

Chapter 1: Introduction

1.1 Research Background 1

1.2 Problem Statement 3

1.3 Objectives of Research 4

1.4 Scope of Research 5

1.5 Thesis Outline 6

Chapter 2: Literature Review

2.1 Introduction 8

2.2 SystemVerilog and UVM Verification 8

2.2.1 Functional Verification 9

2.2.2 SystemVerilog 20

2.2.3 Verification Methodologies for SystemVerilog 28

2.3 Universal Verification Methodology 31

2.4 UVM and Reusability in Verification 32

2.5 UVM Testbench Architecture 35

2.6 Architecture and Component Overview 39

2.6.1 Test 39

2.6.2 Environment 39

2.6.3 Agent 40

2.6.4 Sequencer 40

2.6.5 Driver 41

2.6.6 Monitor 41

2.6.7 Scoreboard 41

2.7 Chapter Summary 42

Chapter 3: Research Methodology

3.1 Introduction 43

3.2 Research Framework 46

3.3 DUT Evaluation 47

3.4 Evaluation of Existing Testbench Support 51

3.4.1 Sideband Directed Verilog Testbench 52

3.4.2 HMC UVM Testbench..56

3.5 Devising Sideband Testbench Support Strategy 60

3.5.1 Reuse of Sideband BFM 60

3.5.2 Develop and Reuse Verification Environments..62

3.5.3 Increase Automation and Test Abstraction through Autonomous Agent......63

3.5.4 Support Summary 64

3.6 Testbench Architecture Definition and Implementation...................................... 65

3.6.1 Sideband Agent Architecture with BFM 65

3.6.2 Sideband Environment Architecture..73

3.6.3 Sideband Scoreboard Architecture 75

3.6.4 HMC Environment Architecture 76

3.7 Autonomous Agent Approach Architecture Definition and Implementation 78

3.7.1 Driver Update for Autonomous Agent Approach...78

3.7.2 BFM Update for Autonomous Agent Approach...80

3.7.3 Scoreboard Update for Autonomous Agent Approach.................................83

3.8 Simulation and Qualification 85

3.9 Reusability Measurement and Comparative Analysis... 86

3.9.1 Reusability Measurement 86

3.9.2 Comparative Analysis...87

3.10 Chapter Summary 88

Chapter 4: Results & Discussions

4.1 Introduction 89

4.2 Simulation and Qualification 90

4.2.1 Main Band Tests 91

4.2.2 IO and MMR Mode Support 93

4.2.3 Sideband Scoreboard and Memory VIP support 97

4.2.4 Sideband Tests 99

4.2.5 Autonomous Agent Approach 100

4.3 Measurement and Comparative Analysis 102

4.3.1 Reusability Metric Measurement 103

4.3.2 Comparative Analysis 107

4.4 Chapter Summary 114

Chapter 5: Conclusion

5.1 Conclusion 115

5.2 Recommendation 117

REFERENCES

LIST OF TABLES

Table 3.1: Core Features to be supported by Testbench. 50

Table 3.2: Special JEDEC Protocol Requirements. 51

Table 3.3: Special DUT Requirements. 51

Table 3.4: Capabilities supported by BFM. 55

Table 3.5: HMC Testbench Strengths and Capabilities of Interest................................59

Table 4.1: Sideband Verification Solution Reuse Result..104

Table 4.2: Reuse For HMC Testbench..106

Table 4.3: Test Comparison For Regular And Autonomous Agent. 112

LIST OF FIGURES

Figure 2.1: ASIC Design Cycle (Onufryk, 1996). 10

Figure 2.2: Ad-Hoc Verilog Testbench. 11

Figure 2.3: Verilog testbench implemented by LSCC (1999). 13

Figure 2.4: Load Task used by LSCC (1999). 13

Figure 2.5: DUT response captured through $monitor task (LSCC, 1999) 14

Figure 2.6: Write and Read protocol of an IP (Iniguez, 2001). 16

Figure 2.7: Testbench Architecture employed by Iniguez (2001). 16

Figure 2.8: Breakdown of Effort in Design cycle (Evans et al., 1998). 18

Figure 2.9: Testbench Team Size (Evans et al., 1998). 19

Figure 2.10: Procedural Code VS OOP. 21

Figure 2.11: Tests Progress In A Project (Spear, 2008). 23

Figure 2.12: Signal Representation of Memory Write Transaction. 27

Figure 2.13: Transaction Based Test. 27

Figure 2.14: General testbench structure (Bergeron et al., 2006). 30

Figure 2.15: VMM Verification Environment (Bergeron et al., 2006). 30

Figure 2.16: UVM Base Classes for Testbench Development. 35

Figure 2.17: Example UVM Testbench Architecture. 38

Figure 3.1: Research Framework. 46

Figure 3.2: Simplified Illustration of the HMC. 48

Figure 3.3: Sideband Testbench. 53

Figure 3.4: REQ-ACK handshake of Sideband IO Interface. 53

Figure 3.5: Sideband BFM. 54
Figure 3.6: HMC UVM Testbench. 57
Figure 3.7: Typical Driver – DUT Communication. 61
Figure 3.8: High Level Architecture of Sideband Agent. 67
Figure 3.9: Sideband Agent Architecture with MMR support. 69
Figure 3.10: Re-Architect Sideband BFM. 72
Figure 3.11: Sideband Verification Environment. 74
Figure 3.12: Sideband Scoreboard Architecture. 76
Figure 3.13: HMC Environment with Sideband Environment Reused. 77
Figure 3.14: Updated Sideband Driver. 80
Figure 3.15: Updated BFM Architecture. 82
Figure 3.16: Updated Sideband Scoreboard Architecture. 84
Figure 3.17: Predictor Mechanism Algorithm. 84
Figure 4.1: Simulation and Qualification Steps. 91
Figure 4.2: Main Band Tests. 92
Figure 4.3: Main Band Tests Result. 92
Figure 4.4: Sideband Tests in IO Mode. 94
Figure 4.5: Simulation Waveform for IO Mode Tests. 94
Figure 4.6: DUT Response on Waveform for IO Mode Tests. 95
Figure 4.7: Sideband Tests in MMR Mode. 96
Figure 4.8: Simulation Waveform for MMR Mode Test. 96
Figure 4.9: Scoreboard Compare Result. 98
Figure 4.10: Sideband Tests. 99
Figure 4.11: Sideband Tests Results. 99

Figure 4.12: Back to Back Self Refresh Test. 100
Figure 4.13: Refresh insertion by Autonomous Agent. 101
Figure 4.14: Autonomous Agent's Simulation Waveform. 102
Figure 4.15: Sequence of Transactions for Short Calibration Command. 108
Figure 4.16: Short Calibration Test For Both Approaches. 108
Figure 4.17: Regular Agent Simulation Waveform For Case 1. 109
Figure 4.18: Autonomous Agent Simulation Waveform. 109
Figure 4.19: Sequence Of Transactions For One Self Refresh Cycle. 110
Figure 4.20: Self Refresh Test For Both Approaches. 111
Figure 4.21: Regular Agent Simulation Waveform For Case 2. 113
Figure 4.22: Autonomous Agent Simulation Waveform For Case 2. 113
Figure 5.1: Verification Progress With and Without Coverage Feedback. 118
Figure 5.2: Updated Memory Environment (mem_env). 120

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BFM	Bus Functional Module
DUT	Design Under Test
HMC	Hard Memory Controller
IC	Integrated Circuit
IP	Intellectual Property
LOC	Lines Of Code
OOP	Object-Oriented Programming
OVM	Open Verification Methodology
SOC	System On Chip
TBV	Transaction Based Verification
UVM	Universal Verification Methodology
VIP	Verification Intellectual Property
VL	Vector Language

VMM	Verification Methodology Manual
MPS	Maximum Power Saving
DPD	Deep Power Down

AGEN GUNA SEMULA BERAUTOMATIK BAGI MEJA UJIAN SISTEM METODOLOGI PENGESAHAN SEJAGAT

ABSTRAK

Proses pengesahan pra-silikon merupakan suatu perkara penting dalam aplikasi spesifik suatu kitaran reka bentuk cip bersepadu. Ia boleh dianggap sebagai salah satu perkara yang boleh melambatkan projek reka bentuk moden hari ini. Oleh itu, pengesahan kecekapan dan produktiviti telah menarik perhatian ramai sejak akhir-akhir ini dan ia menjadi faktor pemacu bagi penyelidikan yang dijalankan ini. Tujuan kajian ini ialah untuk membina suatu penyelesaian pengesahan yang secara aktif boleh mempromosikan guna semula dan inter-operasi bagi pengesahan komponen-komponen dan menambahbaik suatu automasi dalam kalangan penyelesaian pengesahan. Semua ini telah dikenalpasti sebagai suatu konsep yang penting dalam memperbaiki kecekapan pengesahan dan produktiviti. Seni penyelesaian pengesahan UVM (Metodologi Pengesahan Sejagat) yang berpusat kepada konsep ini dibina bagi modul jalur sisi sebuah pengawal memori keras. Pertama, keperluan pengesahan bagi modul jalur sisi akan disiasat. Kemudian, penyelesaian meja ujian sedia ada akan dinilai bagi mendapatkan keupayaaan guna semula. Ini diikuti dengan mencadangkan dan melaksanakan seni bina meja ujian yang boleh mengguna semula komponen-komponen pengesahan yang banyak dan boleh diguna semula secara sendiri. Seterusnya, seni bina

itu akan ditambahbaik bagi membenarkan automasi paras yang lebih tinggi dalam kalangan meja ujian. Penyelesaian pengesahan yang terlaksana itu kemudiannya akan diukur dan dianalisa berkaitan guna semula dan automasinya. Keputusan yang diperolehi menunjukkan pelaksanaan penyelesaian pengesahan mencapai tahap guna semula 21.70% dalam aras sistem meja ujian dan 49.67% di dalam persekitaran pengesahan jalur sisi tunggal. Sebagai tambahan, pendekatan agen berautomatik yang terlaksana di dalam seni bina telah mengurangkan beban ujian penulis sekurang-kurangnya 60% dan sehingga ke 78%.

REUSABLE AUTOMATED AGENT FOR UNIVERSAL VERIFICATION METHODOLOGY SYSTEM TESTBENCH

ABSTRACT

Pre-silicon verification process is an important cog in an application specific integrated chip design cycle. It is considered one of the biggest bottle-neck in modern day design projects. Thus, verification efficiency and productivity has gained a lot of attention lately and will be the driving factor of this research. The purpose of this research is to build a verification solution that actively promotes reusability and interoperability of verification components and improve the automation within the verification solution. These are identified as important concepts to improve verification efficiency and productivity. A state of the art UVM (Universal Verification Methodology) verification solution centered on these concepts is built for the sideband module of a hard memory controller. First, the verification requirements of the sideband module are investigated. Next, existing testbench solutions were evaluated for its reuse capabilities. This is followed by proposing and implementing a testbench architecture that highly reuses existing verification components and be reused friendly itself. Next, the architecture is improved to allow higher level of automation within the testbench. The implemented verification solution is then measured and analysed for its reusability and automation. The result obtained shows the implemented verification solution

achieves a reusability of 21.70% in a system level testbench and 49.67% in the standalone sideband verification environment. In addition, the autonomous agent approach implemented in the architecture reduces the test writer's burden by at least 60% and up to 78%.

CHAPTER 1

INTRODUCTION

1.1 Research Background

What is pre-silicon verification? Pre-silicon verification is part of a design cycle that is used to make sure if the design is meeting the specification and performs its functions as expected by the designer. Pre-silicon verification is conducted before a silicon prototype is available, thus enables the designer to correct and refine the design progressively right from the early stage (Wagner et al., 2011).

In the past when integrated circuits (IC's) used to be made of few thousand gates, verification was not prominent. It used to be conducted fairly simply and quickly through custom implementation, where the implementation can vary from one team to another team within a company itself. It goes without saying that verification strategy in the semiconductor industry was highly fragmented with custom implementations.

However this verification strategy is no longer adequate. Transistor sizes have been shrinking. Transistor count in a single chip has been increasing rapidly. A single chip can be designed to perform multiple functions and the complexity of these designs is too great for a rudimentary verification approach. Wagner et al. (2011) also mentioned that, with the growing level of detail in design, the time and computational effort required to verify the design’s functionality has also increased. Besides that, the

verification approach had to be of high quality to produce bug free and high quality silicon. Therefore a great need for a better verification strategy arose. The industry responded to this need by creating SystemVerilog (IEEE Standard 1800, 2012).

SystemVerilog was mainly developed by Accellera Systems Initiative and became an IEEE standard. This standard has been revised multiple times and IEEE Standard 1800 (2012) shows the most recent revision. While Bromley (2013), specified that SystemVerilog creates a higher level of abstraction for modelling and verification compared to using Verilog hardware description language. Its main aim is to provide object–oriented programming (OOP) language that supports digital hardware verification.

With the advent of SystemVerilog, it was expected that verification approach can now be standardized across the industry by adopting best-practice verification techniques. Although system Verilog allows the creation of complex verification environment by a very skilled engineer, the language was too rich and powerful for widespread adoption. Therefore, verification methodologies such as Verification Methodology Manual (VMM) (Synopsys Inc. 2011), Open Verification Methodology (OVM) (Cadence Design Systems et al., 2008) and most recently Universal Verification Methodology (UVM) (Accellera.org, 2011) were introduced.

These verification methodologies contain guidelines, additional base class libraries, toolkits and macros to build verification environments in a structured way. It not only helped reduce the complexity of using system Verilog but also enabled reuse of verification components and sharing of verification ideas, solutions and industry best

practices among engineers. And according to Bromley (2013), besides the reuse of code across projects and among users, a standard methodology is beneficial for engineer's career prospect as it enables them to take with them the shared understandings and best practices with them from one place to another.

This research will be conducted using the UVM. This research intends to create a UVM based testbench architecture that will have a higher level of abstraction, highly automated and with high component reuse capability. It also intends to find a way to reuse older, module based verification component into this UVM testbench. This research exhibit high level of code reuse and reduce the test writers' burden by half.

1.2 Problem Statement

The adoption of UVM in verification has been widespread. With a standard methodology in place, reuse of UVM verification components, sharing of best practices in verification, reuse of third party VIP's and interoperability of UVM codes has never been easier. However, reusability of older non-UVM verification components too is important. Huge amounts of engineering effort need to be spent to re-develop, re-architect and re-evaluate the verification components from scratch. Thus it makes sense to reuse older non-UVM components wherever possible. This research will address the reuse of older non-UVM Bus Functional Module (BFM) with a UVM driver.

In the industry, designs are almost always never developed from scratch. Additional capabilities, be it small or large are added to the original design. When new capabilities are added to the design, the verification environment should follow such

reuse model as well. This research will strive to add verification capabilities without making many modifications to the older testbench especially when it involves a large verification feature.

The third problem is the lower level of automation in UVM testbenches. The UVM gives a lot of control to the users (or test writers) of the verification environment. Test writers have high degree of freedom to inject stimulus into the DUT as they please. Although this is good for a very advanced test writer who wants to do manual testing, it may not be attractive to the normal test writer. The testbench should be automated to drive and check the DUT without much human intervention.

1.3 Objectives of Research

The main aim of this research is to build an automated reusable verification solution for the sideband module of the hard memory controller (HMC) (Altera emi_rm, 2015) using the UVM system testbench. This is fulfilled by conducting the following objectives:

- To construct a unit level UVM verification environment to verify the sideband of a hard memory controller.
- To reuse a non-UVM module based sideband BFM, add new verification capability and integrate the newly developed sideband UVM environment into a system testbench.

- To develop sideband scoreboard to eliminate manual checking and introduce autonomous agent that makes the sideband UVM verification environment highly abstracted and automated.
- To measure the built verification solution’s reusability and compare the autonomous agent performance against regular UVM agent.

1.4 Scope of Research

This research will be based on the UVM. The UVM testbench development will be limited to exercising the sideband of a memory controller but will be reused and integrated into a system testbench that is tasked to verify the entire hard memory controller. HMC is used to control the DDR DRAM memory operations. The sideband of HMC is used to control some of the non-timing critical memory operations such as the refresh (REF) operation, self-refresh (SREF) operation, deep power down (DPD), maximum power saving (MPS), long calibration (ZQCAL_L), and short calibration (ZQCAL_S).

The verification solution must adhere to the concept of reusability and interoperability to improve verification efficiency and productivity. In addition, the verification solution should also strive to reduce the test writer burden by half through automation within the testbench. A state of the art UVM testbench will be developed to verify sideband of a hard memory controller (i.e. the DUT).

This testbench will be designed to comprehensively exercise the sideband of a memory controller and reuse previously available verification components and

environments. It will also be integration and reuse friendly, eliminate manual inspection of waveforms through scoreboarding, and uses an agent automation approach to reduce the burden of verification engineers (i.e. test writers). The autonomous agent approach should make the sideband UVM environment highly abstracted and automated, while being less dependent on test writer inputs to steer correct transactions into the DUT

Lastly the built verification solution's reusability is to be measured and a comparative analysis between the autonomous agent approach and the regular agent approach recommended by UVM to be conducted.

1.5 Thesis Outline

This thesis is divided into five chapters. These chapters are divided by the different phases undertaken to complete this research. Chapter one gives an overview of this research. It clearly spells out the problems that this research sets out to solve as well as the objectives of this research. It also defines the scope of this research.

Chapter two reviews the already available knowledge and previous works limited to the scope of this research. It also aims to show the importance of pre-silicon verification in an application specific integrated-circuit (ASIC) design cycle. In addition chapter two introduces key concepts very relevant to this research. It introduces the SystemVerilog language, the verification methodologies invented for it and the testbench architectures from different methodologies.

Chapter three, which is the Research Methodology, defines the steps taken to complete this research. These steps are clearly documented in the research framework laid out in this chapter. Each step taken, help solve the problems described in Chapter 1 and bring this research closer to completion. Figures and tables are heavily used to help describe what is being done for this research.

Chapter four presents the results of the methodology undertaken in Chapter three. For the Simulation and Qualification step, waveforms and simulation logs are shown to prove the functionality of the verification solution that has been designed and explained throughout Chapter three. The verification solution is then measured for its reusability and comparative analysis carried out between autonomous agent approach and the regular approach.

Chapter five concludes this research. It summarizes the research findings and recommends future work on this area.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter intends to introduce UVM and the readily available methodology to create a working testbench from scratch. The major components of a testbench will be explained in detail with the aid of diagrams. This chapter will also discuss existing works on increasing reusability of verification environments and components. At the end of this chapter, a basic knowledge of the workings of a UVM testbench and the challenges as well as the increasing importance reuse in verification will be highlighted.

2.2 SystemVerilog and UVM Verification

Functional verification is important in producing quality products. SystemVerilog is the language designed to aide functional verification while UVM is the state of the art verification methodology that is based on SystemVerilog.

2.2.1 Functional Verification

Functional verification is the art of making sure an ASIC design or system on chip (SOC) is functioning according to the designer's expectations and fulfilling the design requirements. In the past when ASIC design was still in its infancy, verification was not a prominent idea. As ASIC design grew and SOC's become a household name, verification started to get very important, as it allows design bugs to be caught very early in design cycle.

Figure 2.1 shows a typical ASIC design flow (Onufryk, 1996), where functional verification is done early in the design cycle. Any failures in this step would mean RTL code need to be revised to fix the bug caught in the process. Functional verification is important as bug fixes can be done quickly and easily in the early stage. RTL bugs found at this early stage will not be costly to fix.

In the beginning when verification was still gaining importance, it was done in an ad-hoc fashion (Mintz et al., 2007), where simulations are inspected visually and directed tests cases were used to verify the design. Ad-hoc verification means test cases and testbenches are built for immediate use without much planning. In an ad-hoc testing approach, designers usually employ any means necessary to find a bug. Therefore, ad-hoc testing is never standard or structured.

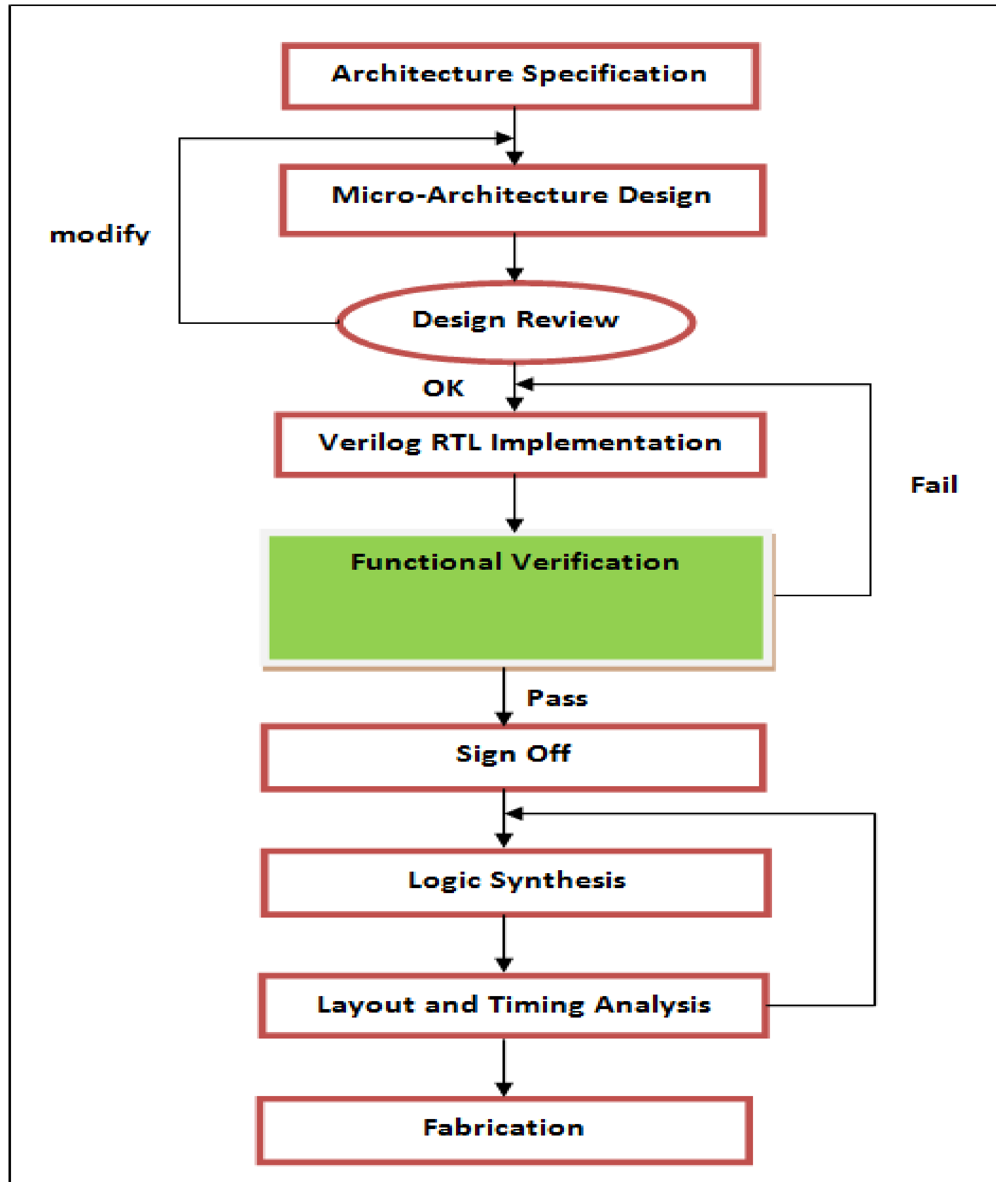

Figure 2.1: ASIC Design Cycle (Onufryk, 1996)

Figure 2.2 shows a typical ad-hoc testbench. The blue cloud on the left holds the code to drive the stimulus into the DUT and the green cloud on the right holds the checking code to check the DUT output. The designers stimulate the DUT using directed

stimuli to check its functionality. The method to drive and check the DUT's response is application specific as designers may apply any means necessary to check the DUT. For example, some may opt to drive and check the DUT using Verilog tasks and functions, some may use simulator waveforms while some other may use more advanced technique such as Bus Functional Module (BFM) to drive and monitors to observe the DUT.

Although these methods are fast to find few initial bugs (except for BFM development), it will not be able to comprehensively test a DUT, as the designers can only test what they can think of. While these methods are acceptable for small designs, it will not be sufficient for designs with thousands of design variables and vectors.

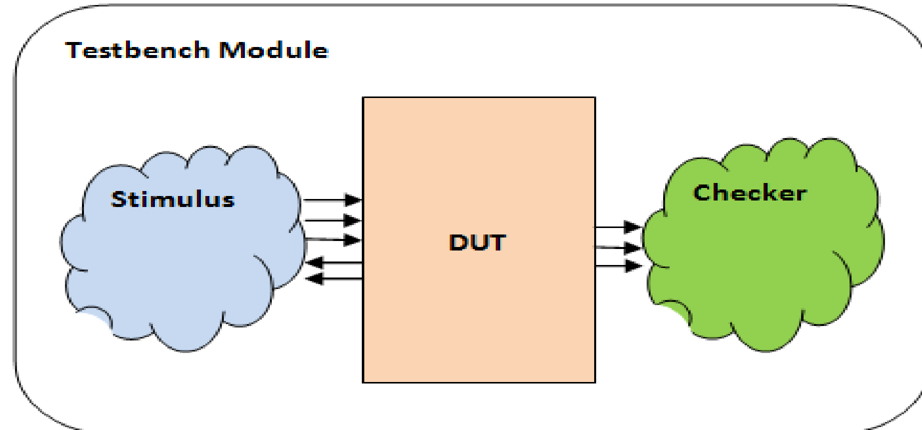

Figure 2.2: Ad-Hoc Verilog Testbench

In addition, as mentioned earlier the implementation to drive the stimulus and to check the DUT's output can vary greatly from one team to another. This is evident in the

testbench solutions employed by LSCC (1999) and Iniguez (2001). LSCC (1999) describes a Verilog testbench designed by engineers of Lattice Semiconductor to verify a four bit asynchronous reset counter with load and count enable that will reside in their Vantis CPLD in 1999.

The testbench implementation described by LSCC (1999) has been summarized into Figure 2.3. Verilog Always blocks, Initial blocks and custom Tasks were used to drive stimulus into the DUT. The DUT's responses were analyzed manually through waveform viewer and Verilog built-in system tasks.

Initial and Always blocks are Verilog constructs that contain procedural statements that execute sequentially (Khalil, 2007). An Initial Block executes only once at simulation time zero, while an Always block executes repeatedly. In this testbench the Initial block is used to initialize the clock, reset and other input pins of the DUT. The Always block is used to generate a stable clock for the DUT.

The custom Verilog task is used to define repetitive or related commands and can be called from an Initial or Always block. The custom task in this testbench loads a vector into the Count_in pin of the DUT at the negative edge of the clock and controls the DUT's load signal appropriately. Figure 2.4 is a snippet of Verilog testbench code implemented by LSCC (1999) and the function of this task is as summarized earlier. The response of the DUT is captured through Verilog system task called $monitor and also checked manually through waveform viewer. Figure 2.5 shows the output captured by the '$monitor tasks' for manual debugging.

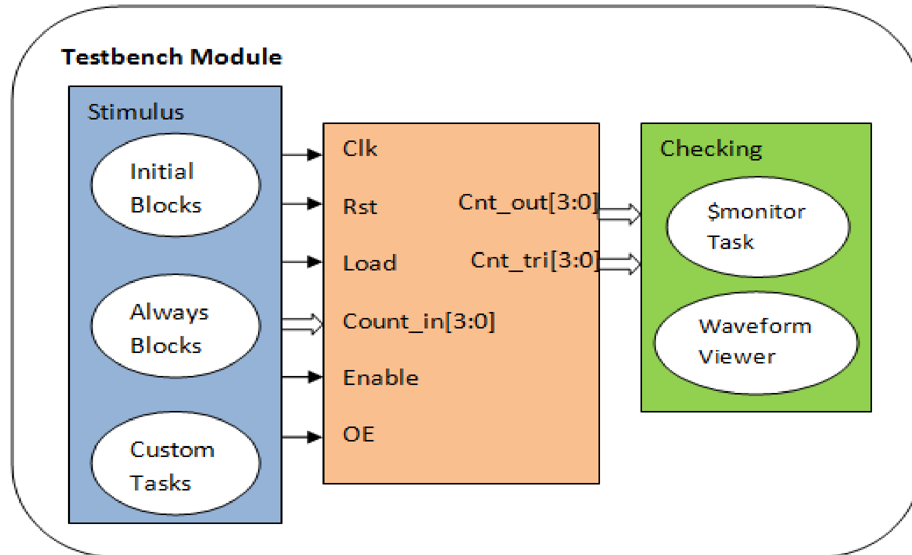

Figure 2.3: Verilog testbench implemented by LSCC (1999)

```
//------------------------------------------------------------
// The load_count task loads the counter with the value passed

task load_count;
   input [3:0] load_value;
   begin
    @(negedge clk_50);
    $display($time, " << Loading the counter with %h >>", load_value);
    load_l = 1'b0;
    count_in = load_value;
    @(negedge clk_50);
    load_l = 1'b1;
   end
endtask //of load_count
```

Figure 2.4: Load Task used by LSCC (1999)

```
ModelSim/Vantis - [Transcript]
File  View  Library  Project  Run  Signals  Options  Window  Help
VCOM  VLOG  VSIM  RUN  CONT  BREAK  STEP  OVER
#      60 clk_50=0, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=a, oe_l=0, count_tri=a
#      70 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=a, oe_l=0, count_tri=a
#      80 << Turning ON the count enable >>
#      80 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=a, oe_l=0, count_tri=a
#      90 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=a, oe_l=0, count_tri=a
#      91 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=b, oe_l=0, count_tri=b
#     100 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=b, oe_l=0, count_tri=b
#     110 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=b, oe_l=0, count_tri=b
#     111 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=c, oe_l=0, count_tri=c
#     120 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=c, oe_l=0, count_tri=c
#     130 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=c, oe_l=0, count_tri=c
#     131 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=d, oe_l=0, count_tri=d
#     140 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=d, oe_l=0, count_tri=d
#     150 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=d, oe_l=0, count_tri=d
#     151 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=e, oe_l=0, count_tri=e
#     160 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=e, oe_l=0, count_tri=e
#     170 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=e, oe_l=0, count_tri=e
#     171 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=f, oe_l=0, count_tri=f
#     180 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=f, oe_l=0, count_tri=f
#     190 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=f, oe_l=0, count_tri=f
#     191 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=0, oe_l=0, count_tri=0
#     200 clk_50=0, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=0, oe_l=0, count_tri=0
#     210 clk_50=1, rst_l=1, enable_l=0, load_l=1, count_in=a, cnt_out=0, oe_l=0, count_tri=0
#     211 << count =  1 - Turning OFF the count enable >>
#     211 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=0, count_tri=1
#     220 clk_50=0, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=0, count_tri=1
#     230 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=0, count_tri=1
#     240 clk_50=0, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=0, count_tri=1
#     250 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=0, count_tri=1
#     251 << Turning OFF the OE >>
#     251 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=1, count_tri=z
#     260 clk_50=0, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=1, count_tri=z
#     270 clk_50=1, rst_l=1, enable_l=1, load_l=1, count_in=a, cnt_out=1, oe_l=1, count_tri=z
#     271 << Simulation Complete >>
# Break at D:/projects/counter/tb_src/cnt16_tb.v line 75
VSIM>
Now: 271 ns   Delta: 0      Line 4689 of 4724
```

Figure 2.5: DUT response captured through '$monitor task' (LSCC, 1999)

The testbench development method shown by LSCC (1999) is very quick and simple to implement and it is recommended for really small and simple DUTs. But with the highly sophisticated and complex DUT or ASIC's of today, this method is no longer viable.

While LSCC (1999) uses simple Verilog capabilities to quickly build a working testbench, Iniguez (2001) employs much more advanced verification techniques to solve harder verification problems. The author proposed a verification methodology for intellectual property (IP) cores. It uses Verilog based coding style called the Vector Language (VL) and a bus functional model (BFM) to verify the write and read protocol

of an IP. Figure 2.6 shows the write and read protocol of an IP that the proposed testbench intend to verify.

A BFM is a Verilog model that emulates the bus protocol shown in Figure 2.6. The BFM will be the only component in this testbench that has direct connection with the DUT's signals. It will contain task and functions that will be invoked by the VL to drive the DUT. VL is the tests that will have tasks and functions calls to the BFM. Basically the VL will invoke the BFM's tasks and the BFM will drive the DUT's signals based on the invoked tasks.

Figure 2.7 shows the testbench architecture described earlier. The DUT's response will be monitored by a built-in monitor inside the BFM. The monitor will dump out a log file that will be used to debug the DUT. The verification technique used Iniguez (2001) to drive the DUT using a BFM is good. In fact, this technique to use a BFM still exists today and even this research uses a BFM to drive the DUT. While the technique to use a BFM is still relevant today, this testbench is insufficient as it uses manual checking and directed test cases to verify the DUT.

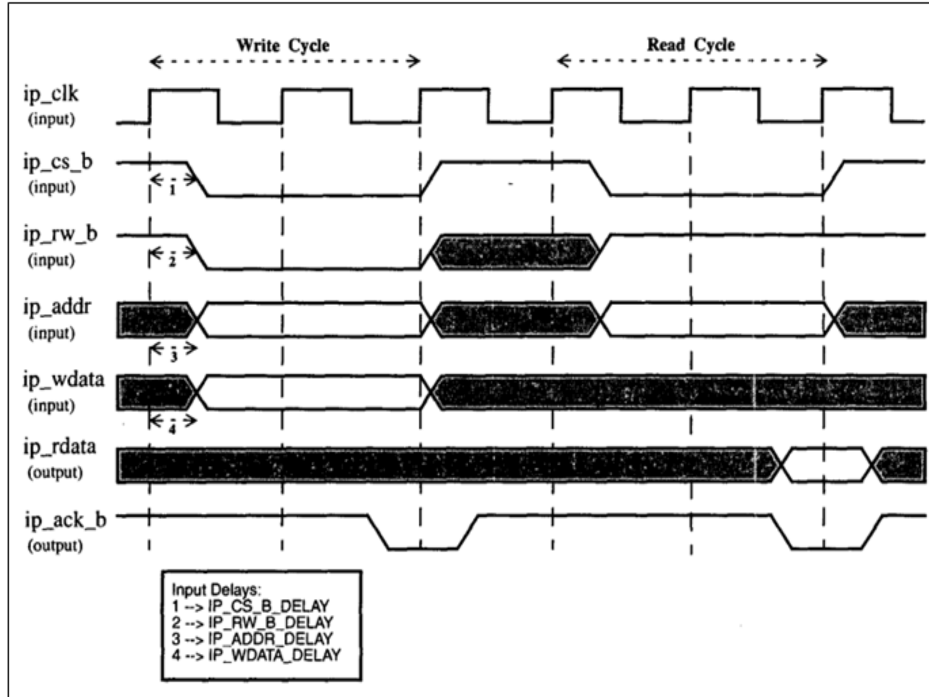

Figure 2.6: Write and Read protocol of an IP (Iniguez, 2001)

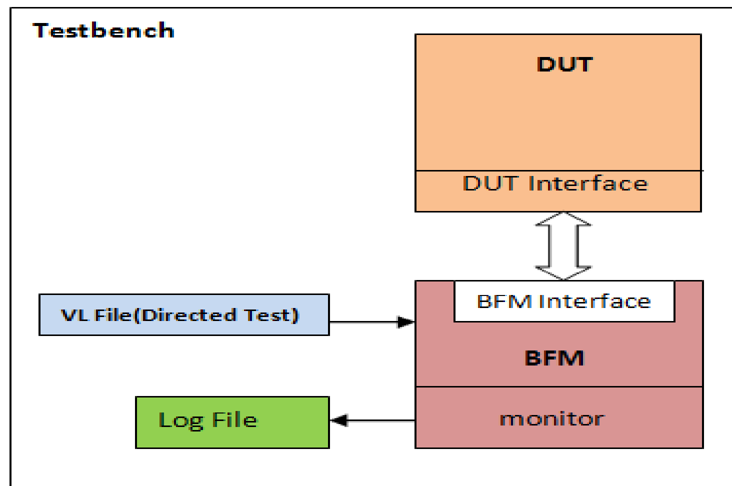

Figure 2.7: Testbench Architecture employed by Iniguez (2001)

These old verification approaches as shown by LSCC (1999) and Iniguez (2001) are insufficient today as they are not random enough to capture more bugs, require manual checking, too ad-hoc, limited reusability, limited automation and most importantly do not have a standard verification structure in place. Bergeron (2006), points out that the old verification approach of writing targeted test cases and manual checking was reaching its limit and a new verification approach is needed to keep up with Moore's Law.

In 1965, G. Moore made an observation, where the number of transistors per chip was doubling every eighteen months since 1959 (Moore, 1965). By extrapolating this trend on a semi-log scale for a decade, Moore predicted that the number of transistors in a square inch of silicon would double every two years. Later, Moore's prediction became the goal that drove the industry (Hence came the term Moore's Law). While keeping up with Moore’s Law, today’s ASIC's have increased greatly in size and complexity.

The great increase in gate count and complexity of ASIC's, has now made verification a major challenge and a serious bottleneck (Dhodhi et al., 1999), where 50% to 60% percent of efforts in the design cycle goes to verification. For instance, the engineers at Nortel, Canada analyzed the breakdown of effort for design and verification of three of Nortel's largest ASIC's in Evans et al. (1998). These ASIC's were fabricated in 0.25 micron process and had gate count of 482K, 824K and 635K gates. Evans et al. (1998) studied the bottleneck in functional verification, as the system become more complex. Figure 2.8 shows the breakdown of effort from the start of the design stage to the start of the layout stage, averaged for all three ASIC's.

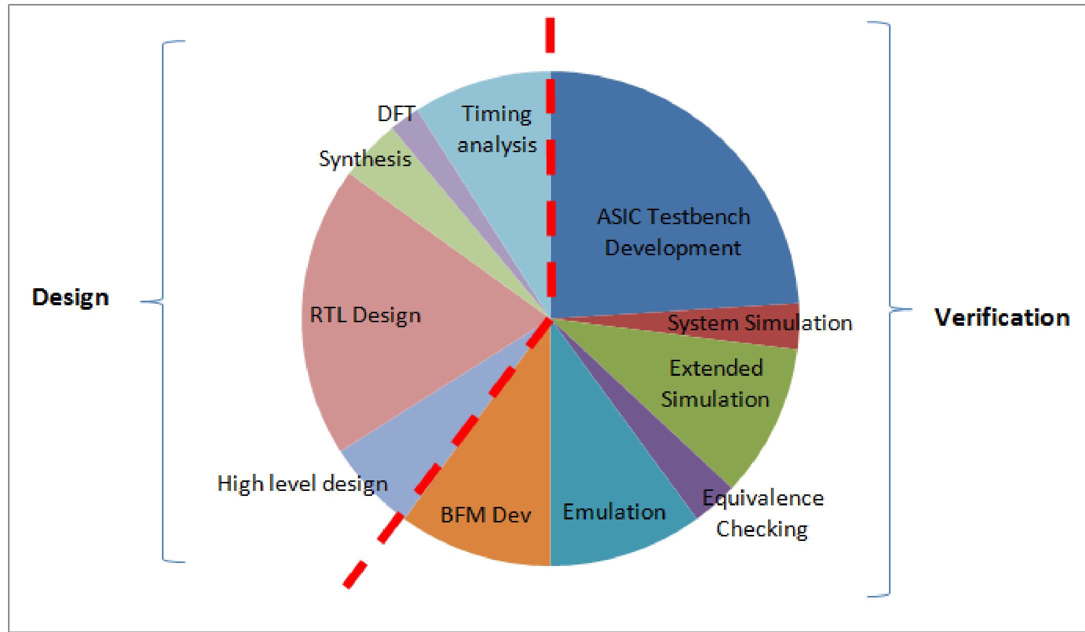

Figure 2.8: Breakdown of Effort in Design cycle (Evans et al., 1998)

Based on Figure 2.8, the study noted that almost two third of the entire design cycle effort is spent on verification and one third of the total effort is spent on testbench's development and simulation. Testbench development and debugging is identified as the bottleneck in the functional verification. In addition, testbench development difficulty increases as the DUT becomes more complex.

Figure 2.9 by Evans et al. (1998) supports this notion, where it shows the overall testbench team size for 824K gate ASIC is more compared to 625K gate ASIC over the full project period. The study concludes that functional verification represented the largest task in the design to layout interval, while testbench development is identified as the critical path in functional verification. However the paper never mentioned anything about the reduction of 824K gate team size in the latter months of the project. The paper

also failed to mention the exact percentage of breakdown of the effort in design cycle in Figure 2.8. It only points out that verification takes more than 50 percent of the total effort.

These previous studies further strengthen the point that verification is lagging behind design, and engineers require advanced verification techniques and methodologies to overcome the verification challenges and gaps. SystemVerilog addresses this verification gap.

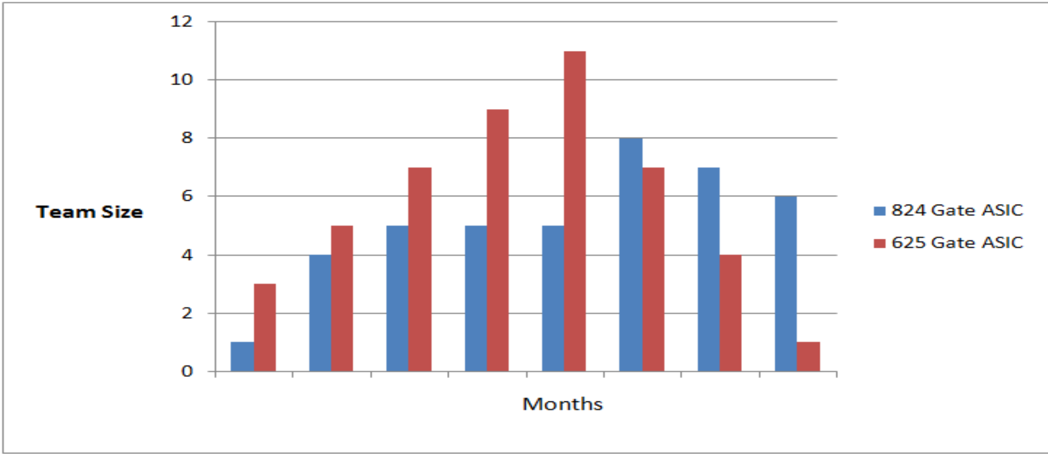

Figure 2.9: Testbench Team Size (Evans et al., 1998)

2.2.2 SystemVerilog

SystemVerilog is an IEEE language standard that provides for advanced verification techniques. It was originally invented for both designers and verification engineers. However, with C++ like features such as object oriented programming (OOP) and transaction level modelling, constrained random stimulus generation, functional coverage creation, event control (Bromley, 2013), dynamic arrays and queues and assertion support has made SystemVerilog popular among verification engineers.

SystemVerilog has become the language of choice for verification as pointed out by Mintz et al. (2007). To appreciate SystemVerilog for verification, it is imperative to understand some the advanced features described earlier. The following sub-sections are intended to give an overview of some of these features.

2.2.2 (a) Object Oriented Programming (OOP) and Transaction Level Modelling

OOP is a programming paradigm that tries to solve a problem in terms of objects. To understand OOP more, it can be compared with procedural programming. OOP approaches a problem by decomposing it into a bunch of data types before applying action (methods) onto the data type. Whereas procedural programming decomposes a problem into a series of actions (procedures: functions and tasks) and apply the step by step actions onto the data type. Figure 2.10 shows the basic difference between these two paradigms. In OOP, data and the action is merged together to form an object (the basic building block of OOP).

In procedural programming the procedure and data are viewed as separate concept and thus kept separate. The key benefit of merging the data and the action is the ability to model complex behaviours with lesser amount of code. With OOP, SystemVerilog can be used to create complex data types and tie them together with the routines that work with them (Spear, 2008). It allows creation of models that work at a higher level of abstraction known as transactions instead of signals.

Abstraction of information, which is the main objective of OOP in SystemVerilog, improves verification productivity. It helps testbench developer to abstract away lower level details from the test writer, resulting in simpler test cases that do not need to deal with ones and zeroes. OOP in SystemVerilog can be considered the greatest contributor to transaction-based verification (TBV) methodology. TBV will be explained in detail in latter sections.

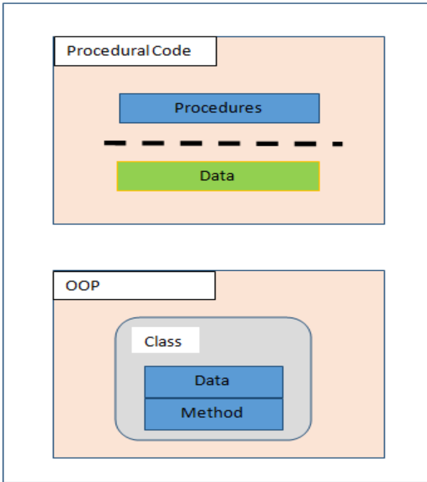

Figure 2.10: Procedural Code VS OOP

2.2.2 (b) Constrained Random Stimulus Generation

As the title suggest, SystemVerilog allows generation and injection of random stimulus into the DUT. However the stimulus will be constrained so that the DUT can be exercised in meaningful states. Spear (2008) argues that by randomizing the input vectors, the test can hit multiple scenarios faster and may even exercise the DUT in a way never thought of by the designers. Figure 2.11 from Spear (2008) shows the difference in progress between a constraint random test and a directed test. Directed testing is fast to execute, producing almost immediate results.

On the other hand, constraint random tests require more upfront work and investment before producing results. However over the longer run, random test is more beneficial as it hits hundred percents coverage faster. On a side note, Figure 2.11 is more of an ideal case where both random and directed test cases achieved hundred percents coverage on its own. In practice where random test is preferred, a few directed tests is required to hit hundred percents coverage.

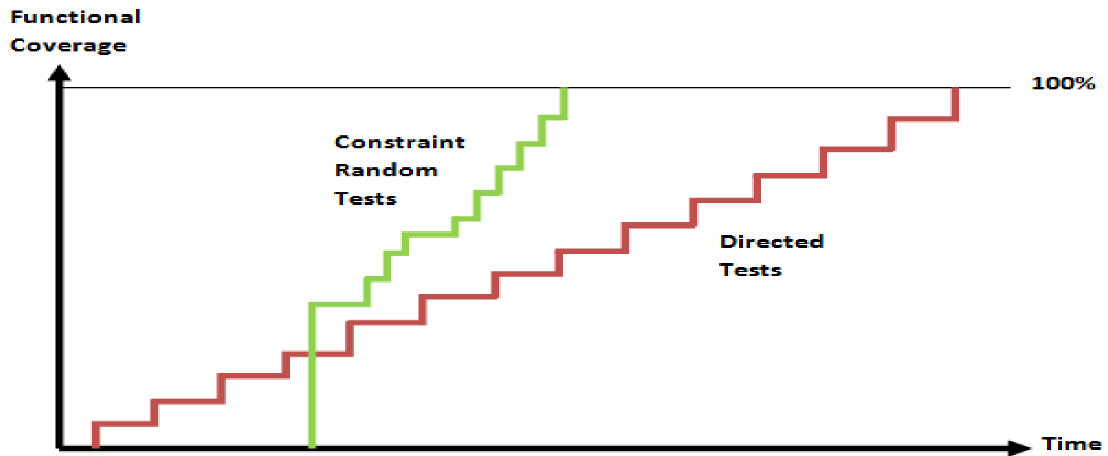

Figure 2.11: Tests Progress In A Project (Spear, 2008)

2.2.2 (c) Functional Coverage

In constraint random testing of today where designs have thousands of vectors, some states of the design will never be hit. To produce quality verification, it is important to measure what has been verified, so that directed tests can be written to hit untouched design spaces. Functional coverage is a measure of the progress and quality of a verification based on the verification plan document (test plan). It is used check off items in the verification plan (Spear, 2008). SystemVerilog provides provision to monitor the stimulus going into the DUT, record the DUT's response as simulation data, and combines data from all the tests into a functional coverage report.

2.2.2 (d) Improved Data Types

Compared with Verilog, SystemVerilog offers more advanced data types. SystemVerilog introduced two state data type for better performance and reduced memory usage, queues, dynamic arrays, unions, packed structures, strings with built-in support, enumeration and most importantly classes, which are the basis for abstract data structures.

2.2.2 (e) Advance Communication Mechanism

SystemVerilog introduced the concept of interfaces to replace signal to signal connectivity between modules. SystemVerilog interface encapsulates all the interconnects between separate modules. However this physical interface can only be used in a static environment such as in a module. A class based environment which is very common for verification need extra handling to use this SystemVerilog physical interface.

A class is dynamic in nature as it based on OOP and operates in run time. To handle the communication between the module world (static) and class world (dynamic), virtual interface is introduced. A virtual interface is an abstract model of the actual physical interface. It allows the user to access the physical interface from a class based environment dynamically.