International Journal of Scientific & Engineering Research, Volume 5, Issue 5, May-2014
ISSN 2229-5518

Pure SV Verification Environment Methodology for ASIC Verification

Dhara Gohel
Dept. VLSI & EMBEDDED SYSTEM
Ganpat University, Mehsana, India
Email: dharag91@gmail.com

Abstract— Nowadays there is ever-increasing density, development cost and turn-around time of VLSI chips. So it becomes increasingly important to have a design verification methodology which enables first-pass chips to be fully functional. The development time and effort can be reduced significantly by reusing design blocks from one project to the next. Verification consumes much more resources than design does in a typical design project, it would be of great value to build verification components that are modular and reusable. In this paper a verification environment is presented for ASIC verification which uses System Verilog (SV) as a Hardware Verification language (HVL) for its implementation.

Keywords—ASIC, Environment, HDL, HVL, SV, Verification, VLSI.

I. INTRODUCTION

As ASIC designs become more complex, it follows that the complexity of the verification environments for such designs increases dramatically as well. “Reuse” is a term that is frequently associated with verification productivity. When faced with writing a verification environment from scratch, or modifying an existing one, the choice will often be to stick with what’s familiar and already in existence. “Methodology” lays a foundation for a robust verification environment which is capable of handling complex verification needs and speed up the verification process. The basic strategy for development of verification model is discussed in this paper which exploits the best attributes of the traditional method of design verification.

There is a growing demand for guidelines and best practices to ensure successful ASIC verification. It is true that the verification problems did not change but the way the problems are approached and the structuring of the solutions, i.e. verification environments, depends much on the methodology. There are two key categories for ASIC verification guidelines: process, enabled by tools, and methodology. The process guidelines are about what we need to do and in what order, while the methodology guidelines are about how to do it.

When a verification environment is needed for a new design, or for a design revision with significant changes, it is important to objectively look at the shortcomings of the existing verification environment and expected productivity gain with the new methodology and determine the best solution. We need to find an optimum balance between re-usability of our legacy Verilog environment and the resource utilization along with limited timelines in adopting the new methodology. This can be accomplished by reusing the knowledge /legacy code from an earlier project along with an upgrade to a new methodology provided with the verification language, that is System Verilog.

Today in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System-on-a –Chip (SoC) designs, verification consumes anywhere between 50% and 75% [1] of the design resources (time and effort). The number of verification engineers is usually twice the number of RTL designers in any non-trivial project. When design projects are completed, the code base that implements the test benches and test cases frequently makes up about 80% of the total code volume [1-2]. This is the main reason design verification is currently the target of new tools and methodologies intended to accelerate the verification effort. These tools and techniques attempt to reduce the overall verification time by enabling parallelization of effort, higher levels of abstraction and automation. The current design trend is to reuse previously verified IP blocks in newer, bigger designs. In the same fashion, it makes sense to build verification components that can be reused in multiple chip design projects. However, the idea of reusable verification blocks is more recent than reusable design blocks [3-4].

II. ASIC VERIFICATION

A. *Importance of Verification*

Verification is not a testbench, nor is it a series of testbenches. Verification is a process used to demonstrate that the intent of a design is preserved in its implementation.

Verification effort has a very high development cost, it consumes most of the design resources, making verification the real limiting factor of time-to-market. Companies are forced to send the designs for fabrication because of the time to market constraints even though they are not close to the completion stage of verification of all the functional characteristics of the design. The overall verification time can be reduced by enabling parallelism of effort, higher abstraction levels and automation.

Verification time can be reduced through parallelism. If efforts can be parallelized, additional resources can be applied effectively to reduce the total verification time. To parallelize the verification effort, it is necessary to be able to write—and debug—testbenches in parallel with each other as well as in parallel with the implementation of the design.

Providing higher abstraction levels enables you to work more efficiently without worrying about low-level details. Higher abstraction levels are usually accompanied by a reduction in control and therefore must be chosen wisely. As

IJSER

the design abstraction gets higher, the verification challenge is higher.

Automation lets you do something else while a machine completes a task autonomously, faster and with predictable results. Automation requires standard processes with well-defined inputs and outputs. For specific domains, automation can be emulated using randomization. By constraining a random generator to produce valid inputs within the bounds of a particular domain, it is possible to automatically produce almost all of the interesting conditions. The automation process takes more computation time to achieve the same result, but it is completely autonomous, freeing valuable resources to work on other critical tasks.

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code. To do this, efforts should be made to understand the input format, the transformation function, and the format of the output. There is always ambiguity in this interpretation, perhaps because of ambiguities in the original document, missing details, or conflicting descriptions. For verifying the DUT in a proper way, it is needed to know the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features[1-5].

To simulate a single design block, first of all tests are created that generates stimuli from all the surrounding blocks – a difficult chore. The benefit is that these low-level simulations run very fast. However, you may find bugs in both the design and testbench, as the latter will have a great deal of code to provide stimuli from the missing blocks. As you start to integrate design blocks, they can stimulate each other, reducing your workload. These multiple block simulations may uncover more bugs, but they also run slower.

At the highest level of the DUT, the entire system is tested, but the simulation performance is greatly reduced. Your tests should strive to have all blocks performing interesting activities concurrently. All I/O ports are active, processors are crunching data, and caches are being refilled. With all this action, data alignment and timing bugs are sure to occur. At this level you should be able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active.

Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. It is mandatory to see that the design can handle a partial transaction, or one with corrupted data or control fields. If the design faces some problems then there should be a solution to recover from those problems. Error injection and handling is the most challenging part of verification.

B. *Hardware Verification Language*

System Verilog (SV) is the first choice to be used since it is an IEEE standard as well as easy to learn, for those who are already familiar with Verilog. It provides some additional constructs for the randomization implementation and Object Oriented techniques for improving the Verification environment. Some of the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are [5]:

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially object-oriented programming
- Multithreading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create testbenches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

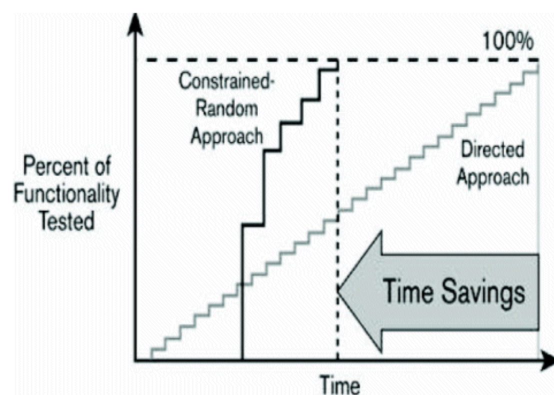
C. *Verification Approach*

This traditional approach of verifying the designs by writing the Verilog/VHDL testbench leads the designers to completely rely on developing a directed environment and hand-written directed test cases. These directed tests provide explicit stimulus to the design inputs, run the design in simulation, and check the behavior of the design against expected results. This approach may provide adequate results for small, simple designs but it is still a manual and somewhat error-prone method. In addition, directed tests are not able to catch obscure defects due to features that nobody thought of. Moreover these traditional methods have very limited and cumbersome random capability.

The solution is to create test cases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints.

Creating a CRT environment takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails. A CRT environment needs not only to create the stimulus but also to predict the result, using a reference model, transfer function, or other techniques.

Once this environment is in place, hundreds of tests can run without having to hand-check the results, thereby improving your productivity. Randomization also reduces the verification time by generating the stimuli and covering all test cases.



ISSN 2229-5518

Fig. 1. *Random v/s Directed Approach*[10]

With increase in complexity and size of design, there is higher and higher demand on exhaustive functional verification. These demands are necessitating the development of new verification technologies, such as, constrained random verification, score-boarding and functional coverage, to achieve exhaustive functional verification goal.

These development methods for reusable verification environment are much easier and helpful in constraining verification to find out the corner cases and hidden bugs which are left undetected with conventional directed approach.

D. Testing vs Verification

Testing is often confused with verification. The purpose of the former is to verify that the design was manufactured correctly. The purpose of the latter is to ensure that a design meets its functional intent. Verification makes the difference between a product that is seen as easy to use and one that repeatedly locks up.

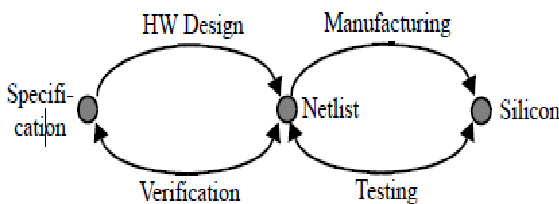


Fig. 2. *Testing v/s Verification*[1]

Basically testing is after silicon and verification is before silicon. Verification ensures that the design is as per the specifications and there are no bugs or flaws in the design.

III. PURE SV VERIFICATION ENVIRONMENT

Verification is that are we building the product right or not. Through verification, we make sure the product behaves the way we want it to. The verification environment is architected based on layered testbench approach and designed to be reusable at gate level simulations and block level environments. Pure SV (System verilog) refers to the verification environment developed through SV as an HVL and the base architecture used is layered testbench architecture with some modifications.

For the purpose of Verification we will be using Pure SV environment which is having Generator, Driver, Monitor, Checker and Scoreboard as the components of the Testbench. All these classes are modeled inside Environment class. The test is at the top of the hierarchy, as it is the program that instantiates the Environment Class. Assertions are there which keeps us inform that the design doesn't violate from the protocol.

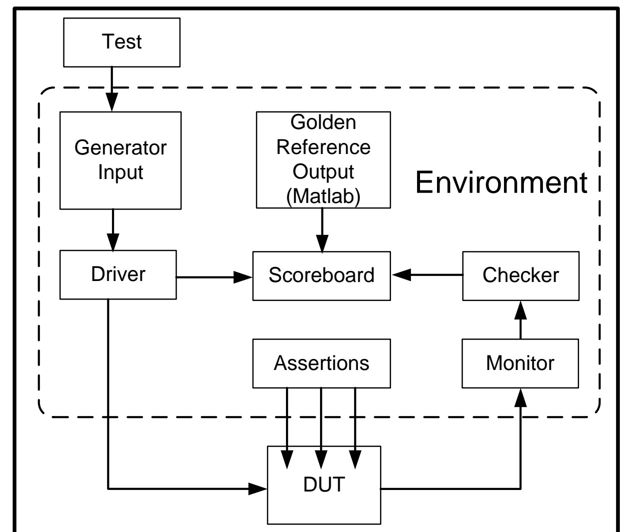


Fig. 3. *Pure SV Verification Environment*

Test: The top layer is the test layer which is shown by the box Test which generates constrained random stimulus needed for the test. The test is written as a program block which provides an entry point into the testbench environment.

Generator: It creates input at a high level of abstraction namely; as transactions like read write operation that is it provides different scenarios the DUT is capable of handling. It is therefore called the scenario layer.

Driver: Driver layer also called the command layer breaks the single commands into signal changes required to drive the DUT. Driver converts its input into actual design inputs, as defined in the specification of the designs interface.

Monitor: The DUT's outputs drives the monitor which takes signal transitions and groups them into commands required for the checker. Monitor reports the protocol violation and identifies all the transactions. It converts the pin level activities in to high level. The monitor only monitors the interface protocol. It doesn't check whether the data is same as expected data or not, as interface has nothing to do with the data. It is therefore called functional layer.

Checker: Checker passes the output of the monitor to the scoreboard. It converts the low level data to high-level data and validated the data. The comparison state is sent to scoreboard.

Scoreboard: The Scoreboard receives the commands from the Driver and predicts the results of the transaction. These are compared with the checker input to the scoreboard to see if the results match. Scoreboard also keeps track of how many transactions were initiated, how many finished and how many are pending and whether a given transaction passed or failed.

Assertions: They cross the Command/signal layer (Driver, DUT) as they look at changes of the individual signal across the entire command and flag an error if the signal does not behave correctly. If Critical signal is —asserted at the wrong cycle, the assertion is triggered. Assertions capture design intent and can be incorporated with the design. Assertions detect errors closer to their source, speeding removal.

Interface: It encapsulates the communication between the testbench and DUT which may include:

- Connectivity using named bundle of wires

IJSER

ISSN 2229-5518

- One or more bundles to connect
- Can be reduced for different tests(like all interfaces may not be needed for a particular test case
- Directional Information (using modport)
- Timing (clocking blocks)
- Functionality (routines, assertions, initial /always blocks)

Mathworks Matlab is used to generate the golden reference output for the DUT (Design under Test) to be verified. Input is randomized as per the constraints of the specifications which would cover all the test cases.

Scoreboard compares the output of the DUT with the golden reference output (Matlab) and sees that the data matches or not. So, the golden reference should also be perfect for this purpose. For some ASICs, this golden reference may be provided by the client itself.

If there's mismatch in the data compared, first thing is to see if there are any timing violations. If there is Timing violations it needs to be solved. Secondly, for which test cases output is not matching. This confirms that there are RTL design bugs in DUT which needs to be solved by RTL developer. Bugs report will be made and given to the designer to fix the bugs. After bug fixing again the test cases will be passes through the design. This process will iterate till there are no bugs and the design seems to be fully functional according to the specifications.

Lastly, if the design has passed all the corner cases successfully then the design is verified by applying the negative cases to the design. Also, coverage is kept in mind and attempts are made to achieve 100 percent functional coverage. If there are some areas where the design doesn't enter then some directed cases can be written and the design functionality can be checked. This may result in achieving good code coverage.

This is the way to verify a DUT through the pure SV verification approach, which is faster than the traditional way for verification and productivity becomes higher.

IV. IMPLEMENTATION

Pure SV verification environment is implemented on a Differential Encoder - Decoder design. Differential encoding/ decoding is used to resolve an initial phase ambiguity for PSK schemes.

All the reusable modules are coded using the system verilog language. Tool used is Synopsys VCS – Verilog Code Simulator. VCS is a high-performance, high-capacity Verilog simulator that incorporates advanced, high level abstraction verification technologies into a single open native platform [8]. There are three main steps that is analyze compile and simulate.

Scripting is done as per the pure sv environment and the verification is carried out on the VCS tool by synopsys. Assertions are checked and it is verified that the design is properly verified or not. All the three steps are done on VCS. The modules scripted are by keeping in mind the environment and the methodology to be implemented for verifying the design. Modules are reusable and can be used for design to design by making minor modifications.

All modules are scripted as mentioned in Section III of this paper. Modules incorporates the randomization and assertions in their development.

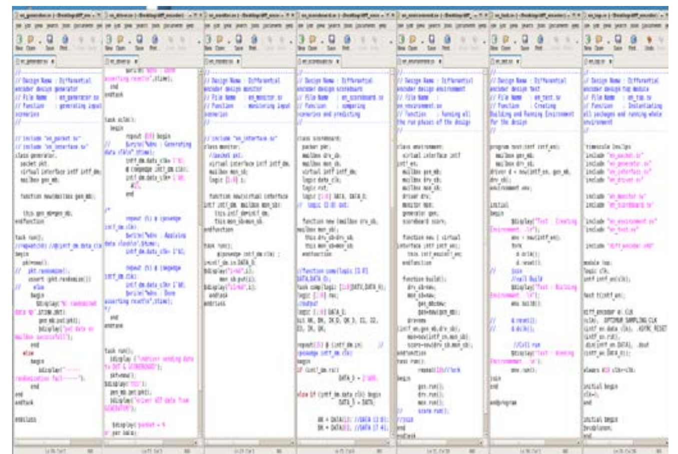


Fig. 4. *Coding of reusable modules in SV*

A. *Analyze*

The DUT is in the VHDL format, so I have to integrate VHDL with System Verilog. For integrating and analyzing it the process is as below:

1. Make the directory name work

   mkdir work

2. Create the synopsys_sim.setup file and map the logical and physical libraries to default work directory.

3. Next analyse all the vhdl files

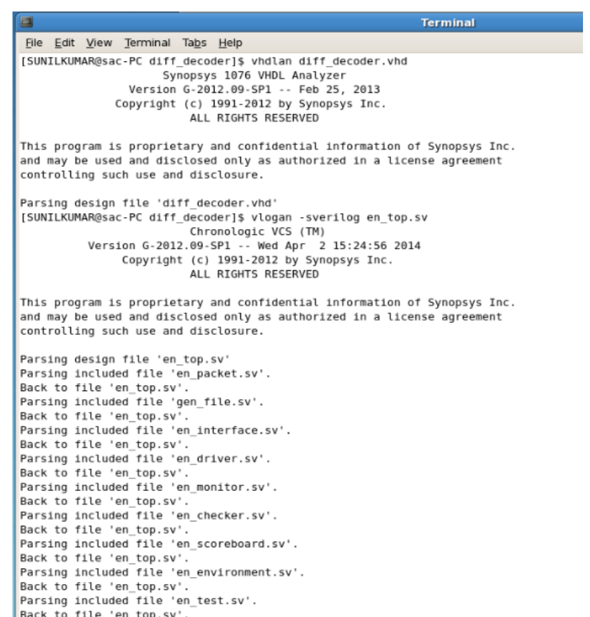
   vhdlan name of vhdl files

4. Analyze the system verilog files that is the environment created

   vlogan –sverilog name of the top module of sv

Here, the switch used is –sverilog for compiling the system verilog modules.

If the DUT is in verilog the first three steps are not required as verilog is the default option in the simulator.



ISSN 2229-5518

Fig. 5. *Analyzing modules of VHDL and SV in VCS*

After all the modules are analyzed and passed successfully the next step is to compile the design.

B. *Compile*

Now the compilation process is to be done which is by a simple command.

Compile the main module using vcs command

vcs module name

```
[SUNILKUMAR@sac-PC diff_decoder]$ vcs -debug_all top
Doing common elaboration
.                        Chronologic VCS (TM)
         Version G-2012.09-SP1 -- Wed Apr  2 15:24:59 2014
              Copyright (c) 1991-2012 by Synopsys Inc.
                         ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

Top Level Modules:
       top
TimeScale is 1 ns / 1 ps

Note-[SIMU-RESOLUTION] Simulation time resolution
  Simulation time resolution is 1 PS

Notice: Ports coerced to inout, use -notice for details
Starting vcs inline pass...
4 modules and 0 UDP read.
        However, due to incremental compilation, no re-compilation is necessary.
ld -r -m elf_i386 -o pre_vcsobj_1_1.o --whole-archive pre_vcsobj_1_1.a --no-whole-archive
ld -r -m elf_i386 -o pre_vcsobj_1_2.o --whole-archive pre_vcsobj_1_2.a --no-whole-archive
ld -r -m elf_i386 -o pre_vcsobj_1_3.o --whole-archive pre_vcsobj_1_3.a --no-whole-archive
if [ -x ../simv ]; then chmod -x ../simv; fi
g++  -o ../simv -melf_i386 -m32 -Wl,-E -lncurses -Wl,-lpthread   -Wl,-whole-archive  vh/_vhdl_ar_1.a vh/_vhdl_ar_2.a vh/scscomm.o
 vh/scsFilelist.o   -Wl,-no-whole-archive   SIM_l.o 5NrI_d.o 5NrIB_d.o    pre_vcsobj_1_1.o pre_vcsobj_1_2.o pre_vcsobj_1_3.o  rma
pats_mop.o rmapats.o       /usr/synopsys/VCS/linux/lib/libnplex_stub.so /usr/synopsys/VCS/linux/lib/librterrorinf.so /usr/synopsy
s/VCS/linux/lib/libsnpsmalloc.so     /usr/synopsys/VCS/linux/lib/libvirsim.so /usr/synopsys/VCS/linux/lib/libvcsfnew.so  /usr/syn
opsys/VCS/linux/lib/vcs_main.o /usr/synopsys/VCS/linux/lib/libvcsmx.so /usr/synopsys/VCS/linux/lib/libreader_common.so /usr/synop
sys/VCS/linux/lib/libBA.a /usr/synopsys/VCS/linux/lib/libuclinative.so          /usr/synopsys/VCS/linux/lib/vcs_save_restore_ne
w.o /usr/synopsys/VCS/linux/lib/ctype-stubs_32.a -ldl -lm  -lc -lpthread -ldl
../simv up to date
CPU time: .228 seconds to compile + .022 seconds to elab + .167 seconds to link
[SUNILKUMAR@sac-PC diff_decoder]$
```

Fig. 6. *Compiling top module of SV in VCS*

After all the modules are compiled successfully without any errors the next step is to simulate the design. It will indicate that simv up to date. This simv is the executable file generated after the compilation.

C. *Simulate*

The simv file which is auto generated is the executable file. This file is executed to simulate the design.

Then run the simv file and to open gui use the switch

./simv –gui

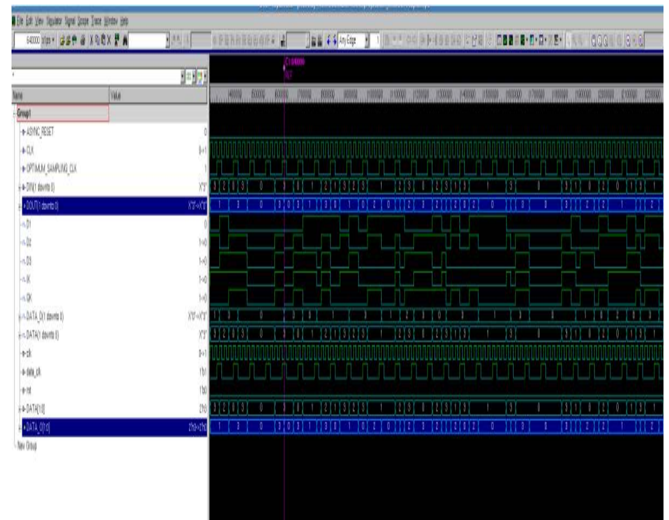


Fig. 7. *Simulating top module of SV in VCS*

After gui is invoked add the signals you want to monitor and verify through assertions. Assertions will fail if the design is violating the rule.

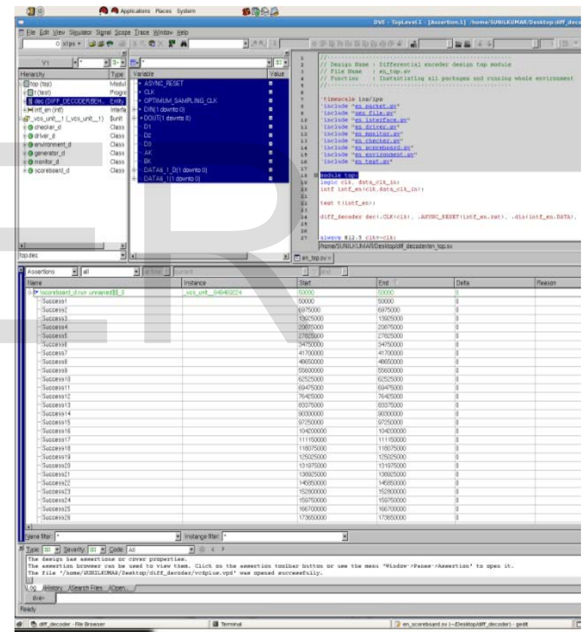


Fig. 8. *Assertions view in VCS*

If some of the assertions fail then you need to go back and iterate the process again and solve the flaws in the design. After solving the bugs again the same thing is verified. But the challenge here is that you define the assertion in a perfect way.

V. CONCLUSION

The implemented pure SV verification environment has many pros. It speeds up verification and results in early tape out of the chip. Less man power is required, by which the overall cost of the project will be low. Environment can be reusable with minimal modifications this results in achieving time to market goal. Also, easy tracking of verification progress can be done and bugs can be found out and resolved quickly.

IJSER

ACKNOWLEDGMENT

The author wishes to express her gratitude to her supervisor Mr. Pinakin Thaker from SAC-ISRO who was helpful and offered invaluable support, guidance and facilities. The author would also like to convey thanks to her internal guide Assoc. Prof. Himanshu Patel, Ganpat University, who rendered his help and assistance. The author wishes to express her love and gratitude to her beloved families; for their endless love and understanding, through the duration of her studies.

REFERENCES

[1] J. Bergeron, Writing Testbenches: Functional Verification of HDL *Models*, Kluwer Academic Publishers, 2000.

[2] H. Foster et al., Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog, Kluwer Academic Publishers, 2001.

[3] Petlin, O., et al., “Methodology and Code Reuse in the Verification of Telecommunications SOCs,” Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, Washington, D.C., USA, September 2000.

[4] Y. Lu, “Design Verification Concepts,” Proceedings of the 4th International Conference on ASICs,” Shanghai, China, October 2001.

[5] Chris Spear, “System Verilog for Verification”, Springer publication, 2008.

[6] “IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language”, IEEE Computer Society.

[7] “System Verilog 3.1a Language Reference Manual”, Accellera’s Extensions to Verilog.

[8] Link: http://users.ece.utexas.edu/handouts/vcs.pdf

[9] A. Sagahyroon, G. Lakkaraju and M. Karunaratne, “A Functional Verification Environment”, IEEE Conference Publications Circuits and Systems, 48th Midwest Symposium on 7-10 Aug. 2005.

[10] Link: http://www.design-reuse.com

IJSER