International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org
Volume 3, Issue 4, April 2014 ISSN 2319 - 4847

Performance Enhancement of Processor Functional Verification by Script Based Approach

Jitendra Kumar Sao[1], Amarish Dubey[2] & Atul Kumar Dewangan[3]

[1] Department of Electronics & Telecommunication, KIT Raigarh C.G., INDIA
[2]Department of Electronics and Communication Engineering, RIET Kanpur U.P., INDIA
[3]Department of Electrical Engineering, KIT Raigarh C.G., INDIA

Abstract

To create revenue, design must be functionally correct and provide benefits to the customers. 70% of ASIC / IP / SoC design efforts and time spend in verification. So, new methodologies and automation is needed to reduce the time of verification. Verification performance is measure of how effective a verification environment or technique is at achieving the coverage goals in the least amount of simulation time.

We can increase the verification performance by reducing the simulation time. Simulation time can be reduced by reducing the redundancy in test-case generation. Decrementing the randomness in test-case generation decreases redundant test-cases. This reduces the number of test-cases for achieving certain coverage goals and hence improves the verification performance.

Keywords: Functional Verification, Coverage, Holes, DUT (Design under Test).

1.INTRODUCTION

Functional verification is widely recognized as the bottleneck of the hardware design cycle. Proper functional verification demonstrates trustworthiness of the design. The Coverage-Driven Verification approach makes coverage the core engine that drives the whole verification flow, which enables reaching high quality verification in a timely manner.

Verification performance is measure of how effective a verification environment or technique is at achieving the coverage goals in the least amount of simulation time. [5] It can be defined using the equation:

$$Verification\ Performance = \frac{\text{Coverage goals}}{\text{Simulation Time}}$$

The two types of verification techniques generally used for verifying digital designs are:

- Formal Verification
- Functional Verification

Formal property verification is a formal technique used to verify formal properties. The specifications themselves are specified mathematically in the form of a formal property specification. The technique mathematically verifies the implementation against the formal property specifications. Equivalence checking and model checking are the examples of formal verification. [6]

In Functional verification or Simulation-based verification, a variety of test cases are fed to the DUT (Design under test), simulated and tested against expected behavior. But here, the verification is confined to those areas that are encountered during simulation. On completion of application of test cases, coverage metric is provided based on the constraints provided.

1.1 Functional Verification

Functional verification, in electronic design automation, is the task of verifying that the logic design conforms to specification. In everyday terms, functional verification attempts to answer the question "Does this proposed design do what is intended?" This is a complex task, and takes the majority of time and effort in most large electronic system design projects.

Functional verification environments have been designed many ways but all of them abstractly follow the block diagram in Figure 1.1. A test-case is used to apply a series of input vectors to both the inputs of the simulated logic design and a golden model (Golden Model is a model of the correct operation of the design under test based on that design's specification). The responses of the golden model, called the expected values, and the responses of the design under test, called the actual values, are then compared. A discrepancy between the two responses indicates either a bug in the design or a flaw in the implementation of the golden model. An agreement between the two responses indicates that the design is functioning correctly or that both the golden model and the design under test share a common design flaw.[7] These shared flaws are improbable, yet possible.

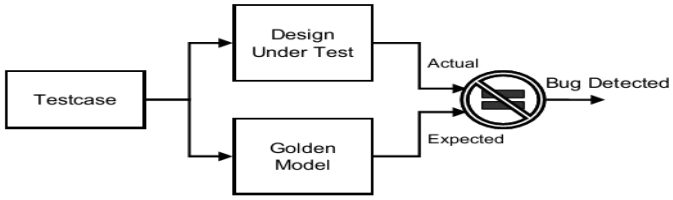

Figure 1 Abstract Functional Simulation Environments

2. COVERAGE

A coverage measure that considers additional verification qualities can significantly extend the verification accuracy. Functional coverage makes all legal input stimuli tested at any moment and finds out functional faults. Now, random stimulus generation technique is used to automatically generate test Vectors.[1]

The purpose of functional coverage driven verification is to measure the credibility or trustworthiness of DUTs by calculating coverage. When the bug appears, the position of bug can be located. Checking constrained-random stimuli whether are defined correctly and design whether exists faults. [1]-[3] When coverage reaches to the expected rate, verification over, otherwise, constraint-random stimulus is modified and then coverage is calculated again until satisfying the need of coverage.

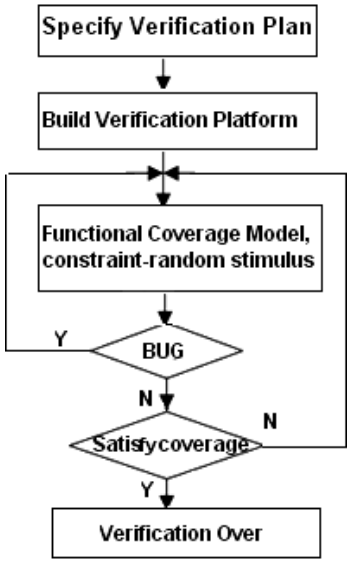

Figure 2 Flow of Functional Coverage Driven Verification [1]

There are mainly two types of coverage:

- Functional coverage
- Code coverage

Functional Coverage is determination of how much functionality of the design has been exercised by the verification environment. The verification environment monitors the function coverage at all times during the process of generating instructions and the test generation engine takes the coverage metrics as the goal for vectors generation. If this verification environment finds that the coverage has meet the goal which is set at the beginning of verification, the environment will stop to run.

Code Coverage checks how your test-bench covers the codes means Statements, Expressions, Conditions, branches etc. it also gives information about how many lines are executed, how many times expression, condition, branches are executed.

The set of features provided by code coverage tools usually includes line/block coverage, arc coverage for state machines, expression coverage, event coverage and toggle coverage. More recent code coverage tools also include capabilities that automatically extract finite state machines from the design and simplify the task of ensuring complete states and arc coverage.

2.1 Functional Coverage

The purpose of measuring functional coverage is to measure verification progress based on perspective of the requirements of functional specification and design specification. Coverage metrics, which is well accepted in the

industry due to its usability of simulation progress, can not only ensure optimal use of simulation resources and measure the completeness of validation, but also direct simulations toward unexplored areas of the design.[9] Therefore, it is very important to design the model of abstracting test-cases because it can be easy to create useful stimulus and provide sufficient level of control.

Functional coverage perceives the design from a user's or a system point of view. Have you covered all of your typical scenarios? Error cases? Corner cases? Protocols? Functional coverage also allows relationships, "OK, I've covered every state in my state machine, but did I ever have an interrupt at the same time? When the input buffer was full, did I have all types of packets injected? Did I ever inject two erroneous packets in a row?"

Proper execution of each test in a test suite is measure of functional coverage. Each test is created to check the particular functionality of a specification. Therefore, it is natural we assume that if it were proven that each test is completed properly, and then the entire set of functionality is verified. This assumes that each test has been verified to exercise the functionality for which it was created. In many cases this verification is performed manually. This type of manual checking is both time consuming and error prone. There appears to be confusion in the industry what constitutes a functional coverage.

The bulk of low-level details may be hidden from the report reviewer. Functional coverage elevates the discussion to specific transactions or bursts without overwhelming the verification engineer with bit vectors and signal names. This level of abstraction enables natural translation from coverage results to test plan items. Specman Elite's functional coverage engine is an example of a powerful functional coverage tool. [8]

2.2 Coverage Holes

Uncovered scenarios are known as holes of coverage report. One of the most important of functional verification is to identify the coverage holes in the coverage space the goal for any successful verification is to achieve the specified target goals with least amount of simulation cycle.

3. THE TEST GENERATION GENESYS TOOL

Genesys is a directed-random test-program generator that based on the Model Based Test Generation. Tool enables the creation of test programs ranging from completely deterministic to totally random. The system consists of three basic interacting components: a generic, architecture independent test generator which is the engine of the system, an external specification (the model) which holds a formal description of the targeted architecture, and a behavioral simulator which is used to predict the results of instruction execution. The user can control test generation by specifying desired biasing towards special events. This biasing can also be saved to a user directives file or Template test-case file. [4]

The external specification model also allows incorporation of testing knowledge. It is employed in order to generate probing test cases and allows expert users to add knowledge to the system in a local and relatively simple manner. General structure of Genesys-Pro random test generator is given below.

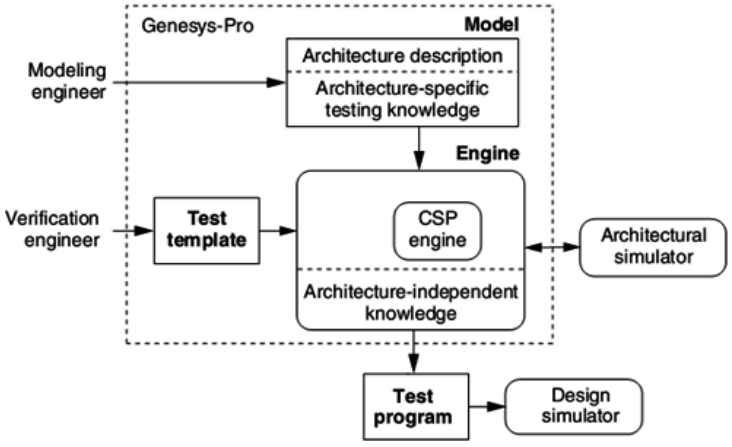

Figure 3 General Structure of Genesys-Pro random test generator [4]

The generator translates the test generation problem into a constraint satisfaction problem (CSP) and uses a generic CSP solver customized for pseudorandom test generation to increase the probability of generation success and to improve test program quality

3.1 The Generation Engine

Genesys is primarily an instruction driven test generator. This means that the whole test generation is based upon the selection of the instructions which will appear in the test. The generation scheme is such that Genesys first selects an

instruction and only then attempts to fulfill the existing biasing requests (i.e. generation of some event). The instruction selection is controlled by the biasing directives. [11]
Beyond the instruction selection, Genesys offers control over the resources such as registers and memory locations which will participate as input and output of the instructions. It is important to note that control is given on two distinct scopes: global and per instruction. In all the biasing directives varying degrees of randomness are allowed. For example, in address selection, the level of biasing may vary from high level specification such as stipulating the degree of alignment, boundary and crossing relative to any standard storage size (i.e. word or page). At the other extreme of low level biasing one may control specific bits of the address. The user has complete control over the probabilities of achieving the events, always, never and any level in between, providing generation of tests ranging from fully deterministic to completely random. [11]

3.2 Test Program Template

The test generator allows you to describe scenarios required for the implementation of a verification test plan. Genesys test program generator has a highly expressive input language (XML Language) for writing such scenarios. The language consists of four types of statements: basic statements, sequencing-control statements, standard programming constructs, and constraint statements. You can combine these statements to compose complex test templates that, in varying detail, describe the test scenarios. Basic statements are nothing but instruction of the DUT processor. In test program template file we can have full control over properties of instruction and operands like data, address etc. We can initialize memory, register, register field etc. we can use hierarchy of higher level by using select, repeat, sequence etc. we can also define variable and macros. The language lets you control the degree of randomness in the generated tests, from completely random to completely direct. In most cases, the template balances both modes, explicitly formulating the essential parts of what is to be tested while leaving the rest unspecified. The generation engine translates these verification scenarios into complete tests. [10]

3.3 Test Program

The main output of the generator is a test-case or test program file which consists of a sequence of instructions, starting from a given initial architectural state, and a section describing the architectural state at the end of the test which forms the expected results. These expected results are used for comparison against the results from running the test on the RTL to determine if a test passes or fails. Figure 4 shows the example of test program template and corresponding test file. As we can observe resources are initialized and instruction and events are generated according to directive used in test-program template file.

Test program template

```
Variable: addr = 0x100
Variable: reg
Bias: Resource-Dependency(GPR) = 30
Bias: Alignment(4) = 50

Instruction: Load R5 ← ?
     Bias: Alignment(16) = 100
Repeat (addr < 0x200)
   Instruction: Store reg → addr
   Select
      Instruction: Add ? ← reg + ?
         Bias: SumZero
    Instruction: Sub ? ← ? - ?
   addr = addr + 0x10
```

Test program

```
Resource Initial Values:
   R6 = 8, R3 = - 25,..., R17 = - 16
   100 = 7, 110 = 25,..., 1F0 = 16

Instructions:
   500:   Load R5 ← FF0
   :
   504:   Store R4 → 100
   508:   Sub R5 ← R6 - R4
   50C:   Store R4 → 110
   510:   Add R6 ← R4 + R3
   :
   57C:   Store R4 → 1F0
   580:   Add R9 ← R4 + R17
```

Figure 4 Test Program template and corresponding test [4]

3.4 The Reference Simulator

An architecture specific reference simulator is connected to Genesys so that after an instruction is generated, it is executed by the simulator. This allows the generator to take the current state of the simulator into account for the generation of the next instruction. In case the simulator state after an executed instruction is undesirable, then the last instruction is rejected and the state of the simulator is reverted to the state it was in prior to the rejected instruction. These mechanisms allow complex dependencies in a test, such as conditional branching depending .on the result of a floating-point operation, while making sure that the test does not cause architecturally undefined behavior. This means that Genesys will not create illegal tests. The reference simulator also allows Genesys to include the reference architectural state at the end of the test, which is used to check against the results of the test case running on the DUT.

4. COVERAGE DRIVEN TEST-CASE GENERATION

Test-Coverage is a measure of the completeness of a set of tests, applied to either a design or a model of the design. Each measurable action is called a coverage task, and together these form a coverage model. This may be quite simple such as measuring which lines of code have been executed or how many state bits have been toggled. [2] The selection of a coverage model is a pragmatic choice depending on the resources available for test generation and simulation. The usefulness of coverage depends on how well the chosen coverage model reflects the aims of the testing. State and transition coverage are very useful measures because of their close correspondence to the behavior of a design.

We say that one coverage model is stronger than or implies another coverage model if every test suite which achieves 100% coverage of one model also achieves 100% coverage of the other. In this hierarchy of coverage models it is clear for example that path coverage is stronger than transition coverage which is in turn stronger than state coverage. However coverage models may also be incomparable, in the sense that neither one is stronger than the other. The coverage efficiency of two test suites may also be compared. The most useful measure is probably the amount of simulation resources required, for example the total length of all the test cases.

4.1 Traditional CDG Approach

Figure 5 illustrates the verification process of traditional coverage directed test-case generation. A test plan is translated by the verification team to a set of directives to generate a template test-case file for the random test generator. Based on these directives and embedded domain knowledge, the test generator produces many test-cases. The design under test (DUT) is then simulated using the generated test-cases, and its behavior is monitored to make sure that it meets its specification. In addition, coverage tools are used to detect the occurrence of coverage tasks during simulation. Analysis of the reports provided by the coverage tools allows the verification team to modify the directives to the test generator to overcome weaknesses in the implementation of the test plan. This process is repeated until the exit criteria in the test plan are met.

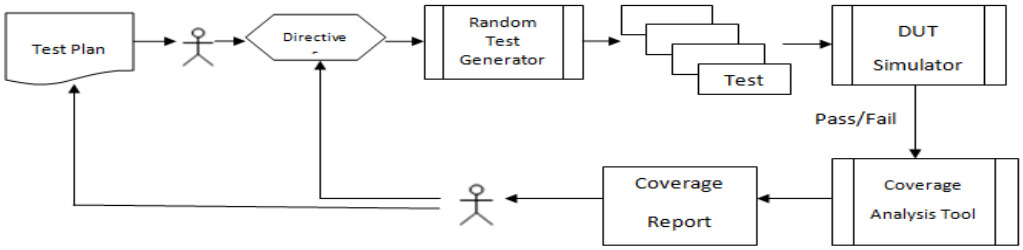

Figure 5 Verification Process of traditional Coverage Directed Test-Case Generation [5]

4.2 Script Based CDG Generation Approach

The use of script test generators can dramatically reduce the amount of manual labor required to implement the test plan. The script parse holes from coverage report analysis it and modifies template test-case program file to generate more directed test-cases which hits the coverage goals even more faster and better way. This can be done by making some random parameter to coverage dependent script directed parameter. The use of script also reduces redundancy in test-cases by decreasing the randomness in the test-case generator. Redundancy reduction reduces the number of test-cases for achieving certain coverage goals hence dramatically reduces the simulation time. Since simulation time is reduced hence verification performance will increase. Even so, the manual work needed for analyzing the test plan and translating them to directives for the test generator, can constitute a bottleneck in the verification process. Therefore, considerable effort is spent on finding ways to automate this procedure, and close the loop of coverage analysis and test generation. This automated feedback from coverage analysis to test generation, known as Coverage Directed test Generation (CDG), can reduce the manual work in the verification process and increase its efficiency.

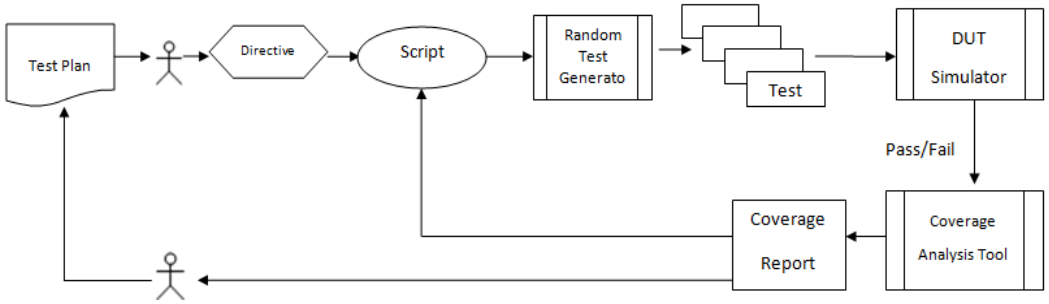

Figure 6 Verification Process of Script Based Coverage Directed Test-Case Generation

4.3 Script Functionality

Figure 7 illustrates the basic functionality of script. Script takes template test-case file as a input file modifies it according to holes parameter from functional coverage report and generate many test-case file which are more prone to hit coverage goals. This script performs basic function of pattern matching and string manipulation. Pattern matching is used for parsing holes from functional coverage report and string manipulation is used to manipulate template test-case file according to holes, so that it can generate more directed test-cases. These pattern matching and string manipulation functions can easily performed using PERL (Practical Extraction and Report Language), so we used Perl for this purpose.

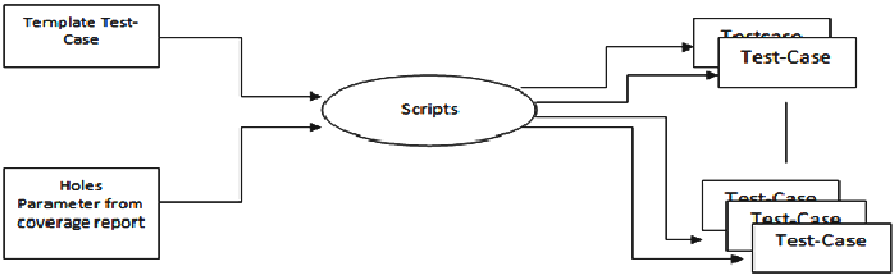

Figure 7 Script Functionality

5. RESULT AND DISCUSSION

Traditional CDG approach and Script based CDG approach is applied to verify different DSI (Data Storage Interrupt) and ISI (Instruction Storage Interrupt) scenarios of IBM Power 7 Processor. The ability to increase the coverage rate for the uncovered tasks of traditional coverage directed test-case generation approach and script based coverage directed test-case generation approach is monitored. The benefit of the script based CDG approach over traditional CDG is also evident from the results shown in Figure 8. The script based CDG approach achieves more coverage goals than the traditional CDG approach using significantly less no of test-cases hence in less simulation time hence we can conclude that verification performance is quite high in script based CDG approach. It happens because it decreases redundancy in test-case generation by reducing randomness in test-case generation.

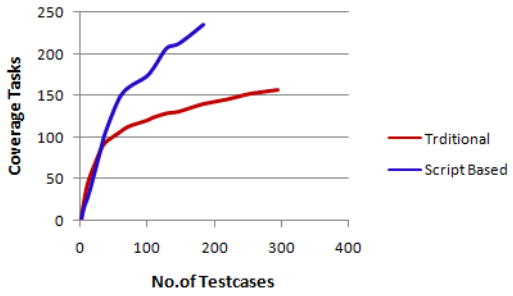

Figure 8 Comparison of Traditional and Script Based CDG

6. CONCLUSION

In this thesis script based test-case generation approach is introduced. This script reduces randomness and generates more directed test-cases. During the course of this paper it is shown that how script based CDG approach reduces randomness in test-case generation. Since randomness is reduced hence more directed test-case is generated hence redundancy is reduced. Reduction in redundancy causes reduction in number of test-cases for achieving coverage goals and hence simulation time will be less. Since simulation time to achieve coverage goals decreases hence verification performance increases.

Acknowledgment

I would like to acknowledge Dr. N.S. Rajput from IIT (BHU), Varanasi India and Mr. Madhusudan Kadiyala from IBM India Pvt. Ltd. Bangalore, India for their valuable suggestion and support.

References

[1] Aihong Yao Jian Wu Zhijun Zhang: "Functional Coverage Driven Verification for TAU-MVBC", in fifth International Conference on Internet Computing for Science and Engineering, Page No- 89-92.

[2] Yingpan, Wu Lixin Yu, Wei Zhuang Jianyong Wang: "A Coverage-Driven Constraint Random- Based Functional Verification Method of Pipeline Unit" ICIS '09 Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science Pages 1049-1054.

[3] Amer Samarah, Ali Habibi, Sofiene Tahar, and Nawwaf Kharma: "Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm" in 2006 IEEE International High Level Design and Test Workshop.

[4] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv: "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification" Mar-Apr 2004, appered in Design & Test of Computers, IEEE, VOL2, Pages 84-93.

[5] S. Fine, A. Ziv.: "Coverage Directed Test Generation for Functional Verification using Bayesian Networks" Proceedings of the June 2003, Design Automation Conference Pages 286 – 291.

[6] S. Hao, F. Yuzhuo.: "Priority Directed Test Generation on for Functional Verification Using Neural Networks", Proceedings of the January 2005 Asia and South Pacific Design Automation Conference Vol. 2, Pages 1052 – 1055.

[7] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal , Gerard Mas, Ralph Smeets: "A Study in Coverage-Driven Test Generation ".DAC '99 Proceedings of the 36th annual ACM/IEEE Design Automation Conference Pages 970-975.

[8] M. Bose, S. Jongshin, E.M. Rudnick, and T. Dukes, M. Abadir: "A Genetic Approach to Automatic Bias Generation for Biased Random Instruction Generation", Proceedings of the 2001 Congress on Evolutionary Computation May 2001, pages 442 – 448.

[9] Andrew Piziali: "Functional Verification Coverage Measurement and Analysis". Kluwer Academic Publishers New York, Boston, Dordrecht, London, Moscow. ebook ISBN: 1-4020-8026-3, published in 2004

[10] P. Faye, E. Cerny, P. Pownall: "Improved Design Verification by Random Simulation Guided by Genetic Algorithms", Proceedings of the ICDA/APChDL, IFIP World Comp. Congress, 2000, Pages 456 – 466.

[11] IBM Power7 instruction Set Architecture (ISA) version 2.05. Available online at website https://www.power.org/resources/reading/PowerISA_V2.05.pdf

AUTHORS

Jitendra Kumar Sao was born in Raigarh, Chhatisgarh, India. He has received his Bachelor of Engineering degree in Electronics & Telecommunication Engineering from Chhatisgarh Swami Vevikanand Technical University, Bhilai in 2009. He has completed his Master of Technology from Department of Electronics Engineering, Indian Institute of Technology (Banaras Hindu University), Varanasi (U.P.) India. He worked as lecturer and Asst. Prof. in different Engineering colleges of chhatisgarh. Presently he is working as lecturer in Electronics and Telecommunication department of Kirodimal Institute of Technology, Raigarh. His areas of interest are in the field of Digital Techniques, Digital Design, Neural Network, Fault-tolerant Systems, and Microprocessor.

Amarish Dubey was born in Kanpur, (U.P.) India. He received his Bachelor degree in Electronics and Communication Engineering from U.P. Technical University, INDIA. He has completed his Master of Technology from Department of Electronics Engineering, Indian Institute of Technology (Banaras Hindu University), Varanasi (U.P.) India. He worked as an Asst. Prof. in different Engineering colleges of U.P. Technical University. Presently he is working as an Asst. Professor in Department of Electronics and Communication Engineering, Rama Institute of Engineering and Technology, Kanpur (U.P.) India. His areas of interest are Digital Design, Fault Tolerant system, Leakage and Power reduction in VLSI Design, Radiation Effects in Semiconductor Devices, etc.

Atul Kumar Dewangan was born in Raigarh, Chhatisgarh, India. He has received his Bachelor of Engineering degree in Electrical Engineering from Guru Ghasi Das University, Bilaspur, Chhattisgarh, India in 2006. He has completed his Master of Technology from Department of Electronics and Telecommunication Engineering, Bhilai Institute of Technology (Chhattisgarh Swami Vevikanand Technical University), Bhilai (C.G.) India. He worked as lecturer and Asst. Prof. in different Engineering colleges of Chhattisgarh. Presently he is working as Lecturer in Electrical Engineering Department of Kirodimal Institute of Technology, Raigarh. His areas of interest are in the field of Power System, Power Electronics, Instrumentation and Control, Electrical Circuit and Network.