Coverage-Directed Management and Optimization of Random Functional Verification

Amir Hekmatpour, James Coulter

IBM Microelectronics, Research Triangle Park, North Carolina

Abstract

This paper describes a functional verification methodology based on a system developed at the IBM Microelectronics Embedded PowerPC Design Center, in order to improve the coverage and convergence of random test generators in general and model-based random test generators in particular. It outlines specific tasks and methods devised for qualifying the test generators at various stages of the functional verification process to ensure the integrity of generated tests. It describes methods for calibrating the test generation process to improve functional coverage. In addition, it outlines a strategy for improved management and control of the test generation for faster convergence across corner cases, complex scenarios, and deep interdependencies. The described methodology and its associated verification platform are deployed at the IBM Embedded PowerPC Design Center in Research Triangle Park, North Carolina and has been used in the verification of 4XX and 4XXFPU family of PowerPC Processors.

1.   Introduction

Test generators have become an important part of functional verification. In the advent of the ever-increasing complexity of designs, decreasing design cycles, and cost constrained projects resulting in increased burden on verification engineers, processor design teams are becoming increasingly dependent on automatic test generators. Test generators are very useful in the early stages of logic debugging, but also, if used efficiently could and should find more complex and corner case bugs that, if undiscovered, would make a design a failure. Random test generators are usually very successful in quickly covering shallow instruction sequences with simple interdependencies. Given enough time, such a test generator will cover 75%-80% of a design space. To address the remaining 20%-25% of design space (i.e.. corner cases and complex scenarios) Domain Specific Test Generators (TG) have been developed (e.g. FPGen, Piparazzi, XGen). Such domain specific test generators require deep knowledge (i.e. specific and detailed) about the domain structure and behavior. The resulting generator is highly customized (coverage optimized) for the specific domain, but is usually cumbersome to use and slow to generate tests. A general purpose random test generator on the other hand is easier to setup and faster in test generation but hardly approaches corner cases and complex scenarios, unless the generation focus is explicitly optimized through the associated models and procedures.

2.   Model-based Systems

IBM's GenesysPro and its predecessors (Genesys, RTPG), as well as its derivatives FPGen and Piparazzi can be classified as Model-based Random Test Generators (MBTG) [1]. Therefore, in order to develop a better understanding of the inner workings of such a class of test generators, we should take a closer look at the basic characteristics and behavior of Model-Based Systems and Model-Based reasoning.

A model-based system operates on a model of the structure and behavior of a device or the function that a system is designed to simulate [2]. Observed behavior (what the device is actually doing) is compared with predicted behavior (what the device should do). The differences between observed behavior and predicted behavior are identified as discrepancies, indicating potential defects. The inference component of such a model-based system (e.g. its model-based reasoning engine) is then initiated to diagnose the nature and location of any defects.

A model-based system is usually comprised of several independent components (i.e. models, methods, inference) [3]. Any result generated is based on and influenced by all relevant models and methods. Changes in the inference will impact the quality of the output results for the same set of models and changes in any of the models will impact the result of such a system even if the inference and generation methods remain the same. Ensuring the integrity and quality of solutions generated is an important ongoing activity in development and maintenance of such systems. Providing feedback and guidance to users of such a system on proper utilization and adjustments required to existing methods and procedures (i.e. what adjustments users have to make to the models or methods they have already developed) is another important ongoing activity in such an environment [4].

2.1   Model-based Test Generators

GenesysPro and its derivatives can be classified as pseudo-random test-program generators which utilize the combination of randomness and control, thus having the potential to generate a large number of different (and potentially unique) tests. These test generators were primarily developed to minimize the effort of architecture-based test generation and to allow the usual architectural changes and upgrades to be easily modeled within the tool [5][2]. Their Model-Based architecture provides the end-user the ability to incrementally enrich the output of the generator (i.e. content of tests generated) by Testing Knowledge (TK). TK, as an external model attribute management enables the end-user to influence the quality of the generated tests. In this way, corner cases can be assigned a suitable probability of occurring, whereas their chance of appearing randomly would be significantly lower. This incremental enhancement of the model's knowledge-base is especially well suited to the tuning and optimization methodology described in this paper. That is why it is highly recommended that each processor group develop and maintain their own design Testing Knowledge locally and follow the proposed generator qualification and test coverage optimization methods.
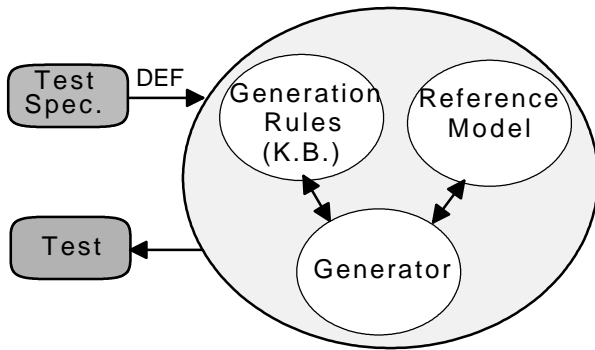

Figure 1: An Overview of a Model-based Test Generator

Model-based test generators such as GenesysPro require continuous calibration of their autonomous components (Generator, Knowledge-base and Reference model) to ensure high quality tests are generated (Figure 1). An ideal model-based test generator is one where the generator is 100% independent of the application domain (i.e.. it has no explicit knowledge of domain structure or behavior) and solely relies on the generation knowledge-base to solve the problem at hand [6][3], perhaps based on a set of constraint satisfaction algorithms [2]. Therefore, each installation of GenesysPro is in fact a custom environment and therefore methods have to be devised for fine-tuning of the components of this model-based reasoning (MBR) system. Since each component could be developed, tested independently and maintained in a different location, the final responsibility for ensuring an optimized and coherent system operation is on the shoulder of the end-user (support & development team) who should be aware of the application domain and specific requirements and restrictions of the design under test. A major characteristic of an MBR system is the interdependency between its components. Any change in one component will impact the behavior of others and the final result [7][1].

It is therefore important to develop a comprehensive strategy for qualifying and calibrating a new release of such a system in order to be able to identify the necessary changes and upgrades in models (generation knowledge-base and reference model) to ensure a behavior within desired boundaries and to redirect the focus of generation according to the application's requirements.

In other words, each time a new version of the generator or the reference model is added to the system or anytime there are major updates to the knowledge-base, the local support team has to qualify the new system and make the necessary adjustments in the knowledge-base to realign the generation behavior to its prerelease condition. The qualification and calibration process is described in more detail in other sections, but the qualification includes generating a suite of tests for the test spec suite and analyzing the new suite against previous ones. Depending on the result of qualification analysis, modifications and updates to the knowledge-base or the test specification suite are then devised to compensate for any system behavior deviations.

3.   Random Verification

At the beginning of the design process, bugs are easy to find. Therefore, there is no point in bombarding the design with a large number of tests. This would only overwhelm the verification team with a huge number of failures. For example coverage analysis of PPC4XX tests have shown that over 65% of all targeted architectural and microarchitectural states are covered in a matter of hours.

Figure 2 shows how coverage of monitors are saturated within a few hours of a coverage database reset (restart). Similarly, unsystematic testing, i.e. running tests without specific properties, is equally inefficient. It induces a non homogenous debugging process, in which some parts of the design arbitrarily receive more attention than others. Testing should be done incrementally to avoid a non-homogenous debugging process which would complicate the verification process. This requires a well defined plan where functions and scenarios to be verified randomly are described in such a way that they could be mapped to a random tool's test spec (e.g. GenesysPro test spec definition file, a.k.a. DEF).
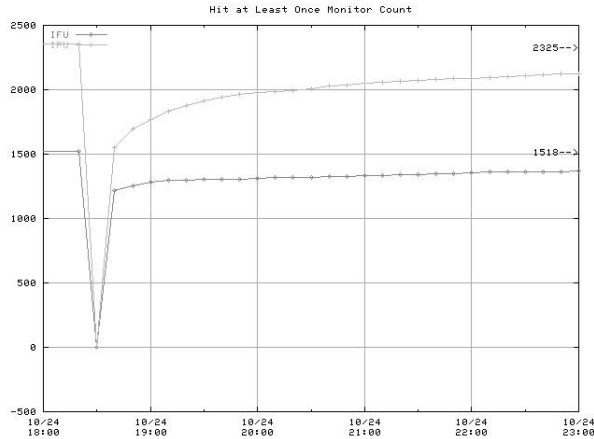
Figure2. Coverage Recovery & Saturation After Reset

When the design reaches moderate stability and the rate of new design bugs (in units & sub-unit) start to decrease, the traditional deterministic tests and regression test suites should be mixed with unconstrained (or lightly constrained) random testing [8]. When the design reaches high stability and the new bugs are hard to find, targeted random-tests with frequent DEF tweaking (as described in this paper) should become the focus of verification effort. Although each solution (bug found) is inherently easy to verify and prove, finding such a solution with a usually unpredictable behavior (signature) among infinite possibilities is difficult and time consuming. Randomness is fundamental in trying to solve such an NP Complete problem. However, one should be aware of the limitations of random testing. Indeed, pure random testing is inefficient since the space domain is enormous and the probability of interesting cases is infinitely small [5]. As such, random test generators have evolved into directed-random and targeted-random generation.

4. Coverage Driven Random Test Generation

The initial random tests should be very simple, small and systematic. For example, it is recommended to start with tests that are aimed at single instructions and limited to the simplest processor mode. Then test complexity can be increased at a pace which matches the design status. Once relative stability has been reached, more focused core level DEFs as well as sub-unit and unit DEFs should be run. Later, as bugs become rare, massive random testing utilizing all DEFs and generating much longer tests should begin. Then, when the hardware is running, it will be used for further testing (i.e.., long tests, huge applications such as Operating Systems). The overall process should be interleaved with comprehensive coverage analysis. Coverage should serve as a feedback mechanism for

additional DEF development as well as an indicator for optimization of the DEF suite, and calibration of the simulation environment and test generators. Coverage analysis of several processors' functional verification, indicates that a targeted coverage-driven random verification based on the methodology described in this paper provides a faster convergence and higher overall coverage compared to the traditional application of a pseudo-random directed TG (TG1), a highly targeted specific domain random TG (TG2) or a combination of both TGs (TG1+TG2), as shown in Figure 3.
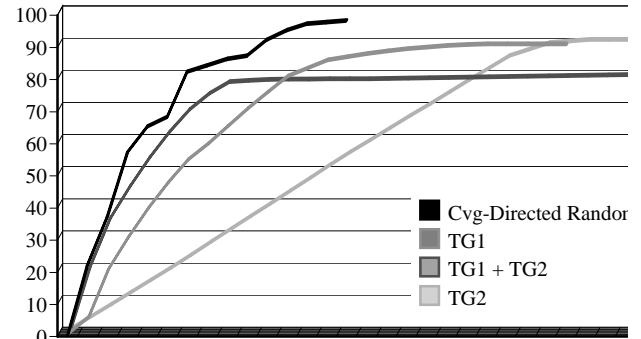

Figure 3. Coverage-Directed vs. Random Test Generation

5. Bug-Driven Random Test Generation Plan

The discovery of a functional design bug is an important event. There are a few lessons to be gained from it. These stem primarily from the two main properties of design bugs (and bugs in general): temporality and locality. These properties point out that, on the one hand, the test which revealed the bug should be retained in order to make sure that the bug will not reoccur (temporality) and, on the other hand, additional tests should be built for adjacent design space areas and functions (locality). For the latter task, the core of the bug should be understood and serve as a basis for additional tests which would also tackle related areas. Each bug found should in fact be used as the seed for a new section of the plan or for further expansion of existing sections. The bug description, as well as its generation and simulation environment attributes should be documented. In other words, not only should tests that have uncovered bugs in the design be preserved, but such a discovery should induce a broader set of actions - including a root-cause analysis and whether this class of bug behavior could apply to other sections of the plan.

At this point, it is important to point out that the legality of a test often expires. In other words, a test that is perfectly legal at some point in time may become invalid after changes have been made to the design [9]. Since changes are often made during the design cycle, the lifetime of a test can be limited. This fact points out that important tests, such as bug finders, might need to be "adjusted".

This can be a very time-consuming task. The section on Tweaking DEFs shows how this problem can be overcome using a random test generator such as GenesysPro. Each bug or class of bugs should have a set of corresponding DEFs that generate the test(s) that can reproduce the bugs (if reintroduced) for every major release of the test generator and the design environment

Finding a bug can also serve as a feedback mechanism for the Verification Plan. Had the bug been found by some random testing, it might point out that there is a hole in the Verification Plan. It should have induced tasks, and thus tests, covering the discovered failure. This could also lead to the addition of new tasks for other related potential failures. A targeted random test generator, such as TG1, naturally lends itself to such a bug driven activity in general, and to the locality property, in particular, by generating more tests based on bug attribute generalization.

The bug driven section of the verification plan should also include a table of all previous relevant design bugs found in similar designs and architectures [10]. For example, for PPC4xx processor all bugs and bug escapes in previous relevant processor families such as 40x, and 4x0 should be closely reviewed and analyzed and documented in the errata table (Figure 4a). In addition, bugs and errata reports from other processor families such as 60x, 6x0, 7x0,... should also be reviewed and analyzed, taking into consideration the microarchitectural differences. Any potentially relevant class of bugs identified should be included in this section [11]. Figures 4a, 4b show a sample errata table and its corresponding random DEF table.

#	Title	Description	S	Impact	Work
1	Lwarx Snoop collision hangs processor	If at the same time a lwarx is executed, a reservation killing snoop hits the previous reservation address, the processor will hang.	Closed	Affects any machine that performs snoops to old reservation addresses.	Put a sync before lwarx

Figure 4a. Errata Table

#	Test Plan	DEF id	Biasing	Des	Status
1	IPU 2.5.14	ipu.err1	True & false sharing, fix address range		Completed mm/yy

Figure 4b. Errata DEF Table

6. Random Verification Plan

It is obvious that any verification activity should be preceded by the composition of a verification plan that describes the overall methodology. This includes the description of high level verification goals leading to detailed verification tasks. The plan should be composed from a deep understanding of the architecture and the main properties of the micro-architecture [12][8]. Available verification means (e.g., existing tests, test generators, etc.) should not be taken into account at the early stages of planning, since they could bias or limit the scope of the verification plan. As the design evolves, new tasks are added to cover emerging implementation details, areas of concern, and design features that are not well defined. The plan is therefore a living document reflecting the present state of the design and should be reviewed and updated upon discovery of bugs to ensure the class of bugs found are covered in the plan. In addition, any architectural or microarchitectural design changes should be carefully reviewed and either reflected in the plan or existing sections addressing these changes should be identified.

To facilitate early identification of test generation needs, an outline of the verification plan should be devised as soon as possible. Such a high level outline should be compiled by the architect, design and verification leads. This outline will serve as a template during the plan development. It should include all functions and scenarios (architectural and microarchitectural features and attributes) to be tested in each unit and sub-unit. If any test suite already exists for any of these verification tasks, those should be listed in the corresponding sections. The verification plan will ultimately induce sets of tests that realize all the verification tasks defined. Such a verification plan should remain independent of the tools available for as long as possible. Local test generator development and support teams should come onboard with the plan development process as early as possible in order to compile a list of required features. This list is then checked against the existing features of the generator and those under development to identify the missing ones. In conjunction with the verification schedule and goals, a development plan including the specification and scope of additional generation features should be developed. This generator development plan should be included in the verification dependency list. When an overall outline of the verification plan is available, the task of identifying sections and tasks suitable for random test generation should get started. This activity should be independent of the specific capabilities of any random test generator available at that time.

Many intricate design mechanisms are difficult to fully exercise using standard tests. For example, causing a complex pipeline to reach full capacity depends on timing dependencies among several internal signals and is not usually accessible by regular tests. It is therefore common to directly trigger these mechanisms by artificially causing them to be in a threshold state. Standard tests can then be exercised in such a context. Therefore, a section of the

plan should list checkers, crunches, irritators and other test fixtures necessary in the design's simulation environment that could be manipulated by verification engineers to fine tune the behavior of the generation/simulation.

7. Test Generator Development Plan

As soon as an overall outline of the verification plan is available, the task of identifying test generator requirements and new features for existing generators should get started. This activity should be independent of the specific random test generator capability at the time. Such random verification requirements should be used to drive the development of the new custom test generators, as well as enhancement of existing test generators. Features not available at this early stage of verification should be well defined and put on the test generator development plan in order to ensure that 9 to 12 months down the road when the early stages of verification have completed, these advanced capabilities are ready for deployment and utilization.

When a complete test generator development plan is compiled, the local TG support and development team should start the development task prioritizing, sizing and scheduling. The local TK development should be coordinated with the necessary generator and reference model development and enhancements, to ensure the timely availability of these required features. The delivery schedule should include a reasonable amount of time for integration of TG components, their testing and calibration.

7.1 Test Generator Qualification

The only way to know if TG1 does what it is supposed to do is to qualify it after every major Release.

Qualification of a test generator is not the same as verifying that it generates a specific set of tests without failing. Nor is it the process of ensuring it can generate tests that it had failed before. Qualification is a qualitative and quantitative assessment process to ensure the current generator has a similar generation signature and generation density as prior versions or as desired at this level of release based on updates and adjustments to the testing knowledge and the generation engine.

A qualification might be to achieve the desired level of generation behavior, consistent with the requirements set forth by the test plans and verification goals. Such requirements may indeed dictate evolutionary generation improvements where the breadth and depth of generation is gradually upgraded to, for example, include additional instructions and architectural attributes in the generation, additional generation biasing, focused data type

randomization, new address ranges, false and true collisions, or expanded CUSP ranges for arithmetic operations.

7.2 Qualification DEF (Test Spec) Suite

TG qualification includes defining (or collecting) a sample set of DEFs which can be used to generate a regression suite after each major release of the TG. In addition to test generator qualification, the simulation environment and external randomizers/irritators/initializors should also be taken into consideration. If tests generated are post-processed, or if the simulator is initialized or external irritators and crunches are used, then these should be taken into consideration in the qualification process. Detailed coverage analysis of the qualification suite in the actual simulation environment is required to identify any necessary changes in the generation process or the simulation environment.

A qualification suite should include a few DEFs for each section of the verification plan in order to ensure that the generator still generates the class of tests required. A qualification suite should also include the types and range of attributes and other biasing necessary to exercise the intended verification task. The qualification should verify that the necessary percentage of tests do indeed generate the combination of biasing, address ranges, page crossings, alignments, architectural resources, etc. expected at the current level of verification effort and the overall suite coverage is consistent with what was recorded in the previous generator qualifications or what is expected based on updates, current stage of verification, new DEFs for new features, etc..

7.3 DEF Tweaking Strategies

Random and pseudo-random test generators usually exhibit a non-uniform coverage signature. If a large finite number of tests are generated for a given DEF, the coverage analysis of these tests reveal that the generator is biased to certain patterns or seems to have a specific generation focus as shown in Figure 5 (a). In order to hit more of the intended domain for a DEF, it may be necessary to make several adjustments in order to rotate (shift) the generation focus as shown in Figure 5 (b). Such adjustments to a class of DEF is what we refer to as DEF tweakings. By DEF tweaking we do not mean changing the DEF or the generator so a specific DEF produces different test cases but rather DEF tweaking involves the following steps:

1. Analyze and compile the coverage signature (coverage trend) of the test suite generated by a DEF
2. Identify the coverage holes in the intended domain of DEF
3. Identify the source & nature of deficiencies (coverage holes)
   - DEF biasings, TK restrictions/assumption & Generator

4. Devise a plan of action to shift the focus of generation
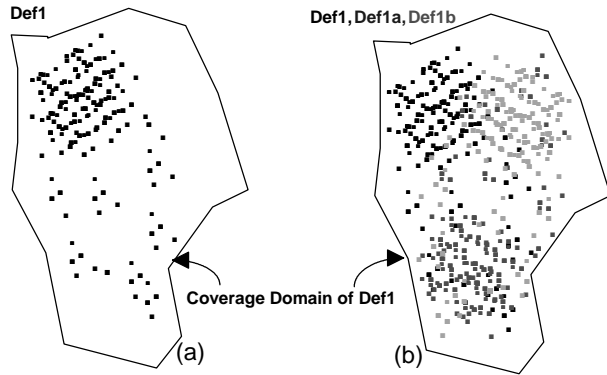5. Apply the changes, generate, collect sample tests and verify

Figure 5. DEF Tweaking Impact on Coverage

The generation environment could be programmed to randomly collect N tests for each active DEF during a specific period of generation. Comprehensive architectural and microarchitectural coverage analysis is then applied to each collected test suite and its time-stamped coverage profile is stored. Appropriate adjustments based on the steps described above are then applied, a new set of tests are generated, and its new coverage profile is compiled. When the total coverage of all generated sets approach 100% (Figure 5 (b)), the generated DEF suites are archived, to be reapplied to the next design release, as described in Figure 8.

7.4   Dynamic Processing of DEFs

One method for timely tweaking of DEFs is updating them based on external feedback at certain times; for example, once a DEF has generated N tests, or when certain coverage metrics are reached. The external criterion activates a utility which modifies DEFs automatically as shown in Figure 6.
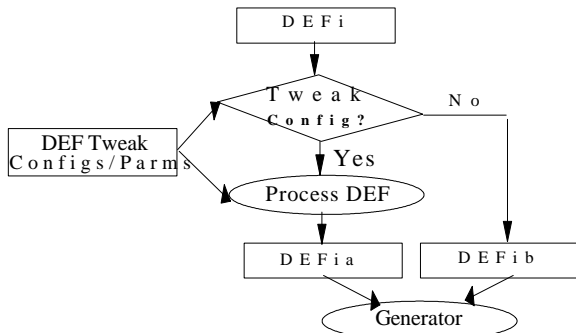

Figure 6. Dynamic Test Spec (DEF) Tweaking

7.5 Number of Instructions in a Test Does Matter

Some of the more complex tests and special scenarios require many instructions to prepare the conditions (setup the prerequisites) before the actual scenario under test can be exercised. A close analysis of prior bugs and bug-escapes also indicate that deep and multi-layer scenarios are required for some classes of bugs.

Therefore, it is imperative that the random verification includes a plan for incrementally increasing, on a regular basis, the length of tests generated. At the beginning of verification effort where there are plenty of easy-to-find bugs, a large number and variety of short (shallow) tests should be generated/simulated. When the design has matured and the rate of new bugs found has dropped or saturated, the number of instructions per test should be increased and additional complex DEFs should be added to the DEF suite. This cycle can be repeated for each new model release where coverage databases are reset and all bugs found in the previous step have been fixed.
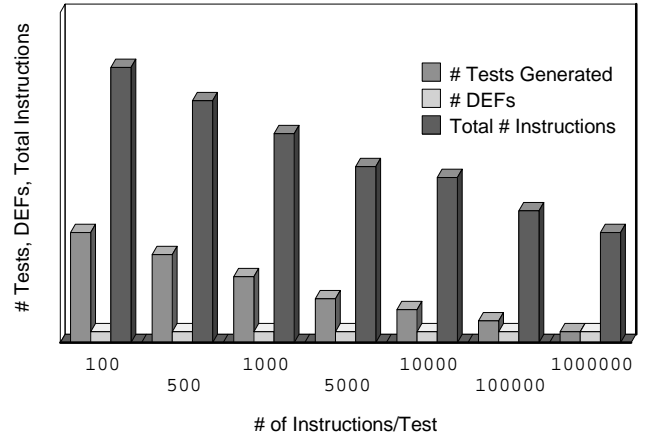

Figure 7. Generation Throughput vs. Test Length

As we increase the number of instructions per test, test generation time per instruction will increase, the number of generation retries (intermediate fails) will also increase (at a much higher rate than the increase in the number of instructions) and more test generations will fail to complete due to memory limitations, generation time limit, or biasing conflicts. Therefore, the number of generated tests/day will decrease and the total number of instructions/day will also decrease as shown in Figure 7. But, it is imperative to generate and simulate longer and longer tests as time passes. The majority of complex bugs are found in complex scenarios, which require long tests to initialize critical resources and to setup the interdependent states. Initializing caches, architectural resources, machine register states, and address translation schemes does alleviate the initialization, but does not stress the behavior of the system under test.

8.   Test Generator Calibration

The first step in test generator calibration is developing a qualification suite. Such a suite is used to qualify that the

output behavior of a test generator is consistent with what is expected and to identify new generation trends and biases. A qualification suite consisting of *N* DEFs is run on each new version of the generator and *P* tests are randomly selected from a pool of at least *10*P* generated tests comprising a new qualification suite. This suite should then be analyzed and compared with the previous suite to identify major deviations and deficiencies. The qualification analysis includes functional coverage (architectural & micro-architectural), addressing range/type/scope, data and resource dependency types, data and address sharing types and frequency, page/segment/cacheline alignments and crossings, exceptions and interrupts and instruction type distribution.

Based on the differences, deficiencies and deviations identified in the qualification analysis step, the corresponding testing knowledge-base and generator behavioral adjustments, modifications, and upgrades are identified and implemented. The majority of such calibration adjustments and upgrades should be in the locally-supported and maintained testing knowledge (TK).

In addition to the above adjustments and upgrades, certain DEF coding requirements might be necessary to either compensate for internal system biases or in order to take better advantage of or properly trigger generation and validation methods. For example, certain adjustments to DEFs may be necessary to compensate for any behavior deviation which could not be compensated for in the knowledge-base. Such DEF coding changes (or restrictions) should be fully documented and propagated to all involved in DEF development/maintenance (e.g. Verification engineers, verification leads, test coders,...).

9. Coverage Driven Verification Management

Early in the verification process, a schedule of verification resource allocation and utilization, as well as Generator release and DEF suite development and optimization, should be defined as a component of test plan.

A combination of the functional coverage results and the number of simulated cycles or instructions could be used as intermediate checkpoints. At the end of each checkpoint, a new qualified and calibrated generator is released, the new DEF suite is tweaked according to calibration feedback and the new generator biasing.

The table in Figure 8 shows a sample generator release and DEF suite tweaking schedule for a functional verification effort with 10 Billion total simulation cycles (or simulated instructions) allocated. Each pre-RIT checkpoint is allocated one Billion cycles. At the end of

CHK4, with no new major generator release, all valid versions of previous DEFs are simulated for a total of 2 Billion cycles. There should be at least 1 Billion clean cycles before a RIT is committed. Each project can define their own checkpoint table in order to ensure enough test generation optimization and DEF suite tweaking is planned for the best utilization of allocated simulation resources.

CHK1	CHK2	CHK3	CHK4	RIT1	CHK5	CHK6	RIT2
1 B	1 B	1 B	1 B	2 B	1 B	1 B	2 B
$TG1._1$	$TG1._2$	$TG1._3$	$TG1._4$	$TG1._4$	$TG1._5$	$TG1._6$	$TG1._6$
DEF1	DEF2	DEF3	DEF4	DEF1-4	DEF5	DEF6	DEF1-6

Figure 8. TG Release & DEF Tweaking Schedule

For each checkpoint (i.e. CHK1, CHK2, ...), the minimum number of allocated simulation cycles (or simulated instructions) should be simulated with the new major release of the qualified and optimized test generator and the release's corresponding optimized new DEF suite.

Progress of verification based on new and additional coverage should be taken into consideration to identify intermediate checkpoints within each planned stage. If the coverage is saturating quickly at a level close to the previous checkpoint's coverage level (as shown in Figure 2), additional minor DEF tweaking and/or new DEFs are required (sub checkpoints CHK1a, CHK1b, ...) to improve the number of new design states verified by new tests. As the design matures, intermediate generator calibration and DEF suite tweaking might be required to optimize the utilization of the allocated resources.

In order to manage the test generation/verification checkpoints, it is necessary for the verification team and test generator team to work closely. A Coverage Engineer (A Verification Engineer with the specific role of generating, analyzing coverage data and monitoring the progress of functional verification) would serve as the liaison between the Verification Engineers and the test generator development and support team. Close monitoring of coverage data (both microarchitectural and architectural) is necessary to identify functions and units whose coverage has saturated and to devise plans for either:

- Retiring their corresponding monitors, assertions and data collection tasks (in case the saturation point is satisfactory for the current level of design) or

- Analyze the corresponding set of DEFs to identify whether any DEF needs to be updated (biasing, parms, ...) or the testing knowledge requires calibration in order to shift the focus of generation for the DEF set.

10. Conclusions

In response to the increased burden on Verification Engineers due to ever-increasing complexity of designs, decreasing design cycles, and cost constrained projects, processor, ASIC and SOC design teams have resorted to incorporating productivity and quality improvement automation in their functional verification methodology. One such solution has been the incorporation of advance random and pseudo-random test generators.

In order to take full advantage of random test generators, their capability, as well as their limitations and built in biases should be well understood and taken into consideration in the overall verification planning. In addition, a continuous analysis, qualification and calibration of such test generators should be planned for and treated as an important and integral component of the verification methodology. It is also essential to systematically assess the quality of tests being generated and to proactively manipulate and influence the random generation process for optimum coverage and efficient test generation and simulation resource utilization.

Coverage as a measure of functional verification progress has to be fully understood and utilized in order to not only direct the random test generators to the neglected areas of the design space, but also to better manage the limited simulation resources. Functional coverage can be utilized to identify tests not contributing new states and removing these from the regression suite, thus making the CPU cycles available to new test generation and simulation.

We outlined specific tasks and methods which we have devised for qualifying IBM test generators at various stages of the functional verification process, to ensure the integrity of generated tests. We described methods for calibrating the test generation process to improve functional coverage. In addition, we defined a strategy for improved management and control of the test generation focus for faster convergence across corner cases, complex scenarios, and deep interdependencies. We also gave an overview of methods for managing verification resources to reduce number of bug escapes.

This methodology and its associated autonomous web-based verification platform called FoCuS is deployed at the PowerPC Embedded Processors Design Center at IBM Research Triangle Park, North Carolina and has been successfully applied to the verification of 4XX and 4XXFPU family of embedded PowerPC processors.

11. Acknowledgments

12. References

[1] Y. Lichtenstein, Y. Malka and A. Aharon, "Model-Based Test Generation For Processor Design Verification", Innovative Applications of Artificial Intelligence (IAAI), AAAI Press, 1994.

[2] D. Lewin, L. Fournier, M. Levinger, E. Roytman, G. Shurek, "Constraint Satisfaction for Test Program Generation", IEEE 14th Phoenix Conference on Computers and Communications, 1995.

[3] A. Hekmatpour, S. Powell, Paul Chi, "KINFIDA: An Object-Oriented Model-Based Digital Filter Design Assistant," Proc. of IEEE Int. Con. on Acoustics, Speech and Signal Processing , New Mexico, April 1990.

[4] A. Hekmatpour, A. Orailoglu, and P. Chau, "Hierarchical Modeling of the VLSI Design Process," IEEE EXPERT, Special Track on Object-Oriented Programming in AI, April 1991.

[5] Laurent Fournier, Yaron Arbetman, Moshe Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator - Application to the X86 Microprocessor Family," , IBM Haifa Research Lab., DTE99 Proceedings, Munich, Germany, March 1999.

[6] B. C. Williams and P. P. Nayak, "Model-based automation Systems," AI in the New Millennium, AI Mag, Fall 1996.

[7] B. C. Williams and P. P. Nayak, "Model-based execution and planning in the reactive loop," Proc. Of IJCAI-97.

[8] A. Aharon, D. Goodman, et. al., "Test Program Generation for Functional Verification of PowerPC Processors in IBM", 32nd DAC, San Francisco, June 1995, pp. 279-285.

[9] Ken Albin, "Nuts and Bolts of Core and SOC Verification," DAC 2000, March 2000.

[10] J. Monaco, D. Holloway, R. Raina, "Functional Verification Methodology for the PowerPC 604 Microprocessor", 33rd. Design Automation Conference, June 1997, pp. 319-324.

[11] A. Hekmatpour and C. Elkan, "Categorization-Based Diagnostic Problem Solving in the VLSI Design Domain, " Proc. of the Ninth IEEE Conference on Artificial Intelligence Applications (CAIA), Orlando, March 1993.

[12] M. Kantrowitz, L. M. Noack, "I'm Done Simulating; Now What?", 33rd Design Automation Conference, June 1997, pp. 325-330.