2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Coverage Acceleration and Testcase Pruning using Smart Stimuli Generator in SOC Verification

Rahul Laxkar, Naveen Srivastava, Sekhar Dangudubiyyam
SAMSUNG SEMICONDUCTOR INDIA RESEARCH
Bangalore, 560048

***Abstract*- Verification planning involves identifying the features of the DUT that need to be verified, prioritizing those features, measuring progress, and adjusting the allocation of verification resources so that verification closure can be reached on the required timescale. One of the major challenges in SoC verification is to achieve comprehensive coverage while minimizing number of testcases. The complexity of SoC design along with the limited amount of verification closure time often leads to inability to achieve the complete coverage by testing all the possible combination of stimuli. Consequently, some corner cases being left unverified may exhibit abnormal behaviour in real-time application.**

This paper will share an approach to tackle the aforementioned challenges in SoC verification. The proposed approach involves the utilization of a smart stimuli generator that can effectively prune the testcases and accelerate the coverage closure. The proposed approach generates efficient testcases that cover the maximum number of scenarios based on specified coverage target and eliminate redundant testcases. The smart stimuli generator optimizes the test coverage by generating stimuli that are likely to reveal bugs in the design. The performance of the smart stimuli generator has been assessed on group of benchmark designs and the results demonstrate the significant improvements in coverage and overall reduction in the volume of testcases. The proposed approach provides a promising solution for accelerating SoC Verification and reducing the overall verification efforts.

I. INTRODUCTION

In the field of verification, achieving comprehensive functional coverage is of paramount importance to ensure the reliability and quality of SoC. Traditionally, the process of reaching 100% coverage involves iterative testcase development, regression testing and subsequent analysis. However, this approach can be time-consuming resource-intensive, and may not always yield optimal results due to design complexity and limited turnaround time.

To address these challenges, the concept of smart stimuli generation has emerged as a powerful approach to boost the functional coverage process. The smart stimuli generator approach offers a systematic and efficient method to achieve 100% coverage for covergroups or coverpoints using strategically designed testcases. It enables a streamlined verification process while uncovering hidden design issues and reducing the time-to-market.

One of the key advantages of the smart stimuli generator approach is its ability to achieve comprehensive coverage with a single simulation. By iteratively generating datapaths, it ensures that all critical functional requirements or features are exercised, leaving no gaps in coverage. This approach minimizes the need for the manual intervention, post simulation coverage mixing and testcase modification, thereby streamlining the verification process.

Furthermore, the smart stimuli generator approach enables engineers to extend coverage scenarios by incorporating additional scenarios in the verification plan. This flexibility expands the coverage of the design, allowing for the identification of previously undetected bugs and improving overall design.

II. Conventional Approach and Concept of Smart Stimuli

A. *Conventional Approach and Limitations*

It is essential to understand the conventional approach employed for coverage prior to dive into the proposed smart stimuli approach. Consider the cover point for the write transaction of AXI protocol in Fig. 1

```
1  covergroup axi_write_traffic@(posedge coverage_sample_e);
2
3  option.per_instance = 1;
4
5      BURST_LEN:    coverpoint AWLEN   {bins awlen_txs[] = {[0:15]};}
6      BURST_SIZE:   coverpoint AWSIZE  {bins awsize_txs[] = {1,2,3,4};}
7      CACHEBILITY: coverpoint AWCAHCE {bins awcache_txs[] = {[0:7]};}
8      WRITE_ID:     coverpoint AWID    {bins awid_txs[] = {[1:15]};}
9
10 endgroup
```

Figure 1. Cover point for the write transaction of AXI protocol

The above cover group is defined to capture the characteristics of AXI write transaction like AWLEN used to cover burst length, AWSIZE used to cover Burst width etc. The goal is to create test case that exercise all of the bins in the cover point and during simulation, the simulator generates coverage data based on the stimuli provided by the test case. The results of the simulation are analyzed to determine if the coverage goal has been met, and if not, then the stimuli may be updated with additional constraints or seed values to target the untouched bins, and this process continues until the coverage goal is met. Fig.2 shows the flow diagram of an ideal approach to achieve the coverage goals.

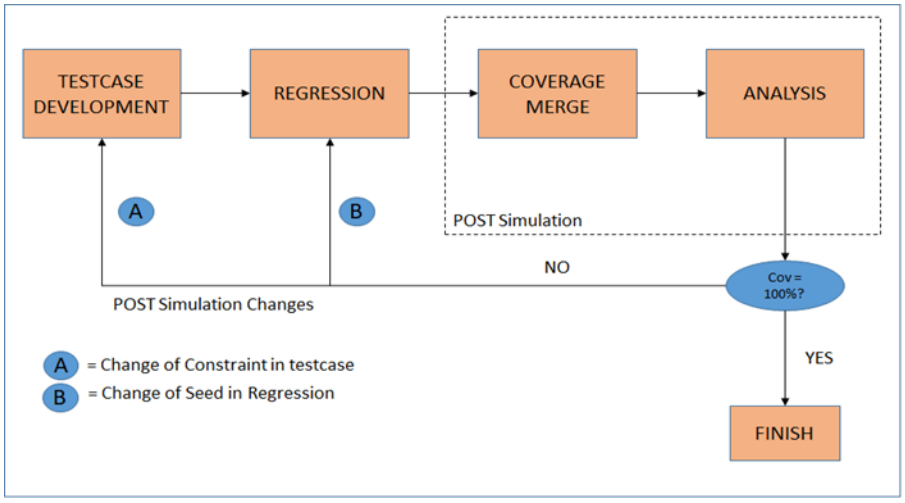


Fig.2: Conventional Approach of Functional Coverage

The above figure is defining the conventional approach to achieve complete functional coverage. It's needed post simulation efforts to analyze the coverage results and further improvements in the constraints to achieve 100% coverage.

However, there are some limitation and challenges of the conventional approach:
1. Time-consuming: Creating Testcases, analyzing results and modifying seed values or constraints can be a time-consuming process, especially for a complex design.

2. Repetitive: This process requires multiple iterations, as it involves modification in the Testcases until the coverage goal is achieved.
3. Limited coverage: The conventional approach may not provide complete coverage for all possible scenarios and may not detect all bugs.
4. Increased complexity: When dealing with complex designs and tight deadlines, this approach can become even more challenging for a verification engineer

B. Concept of Smart Stimuli

Now comes onto the solution part. The Smart Stimuli generator proposes a novel approach to address the challenges associated with functional coverage as discussed earlier. This proposed method leverages a single testcase with smart stimuli generator to eliminate the need for multiple testcases. The stimuli generator is capable of generating new datapath based on the inputs, constraints and coverage information from the previous datapath. The implementation of this approach is illustrated in Fig.3.

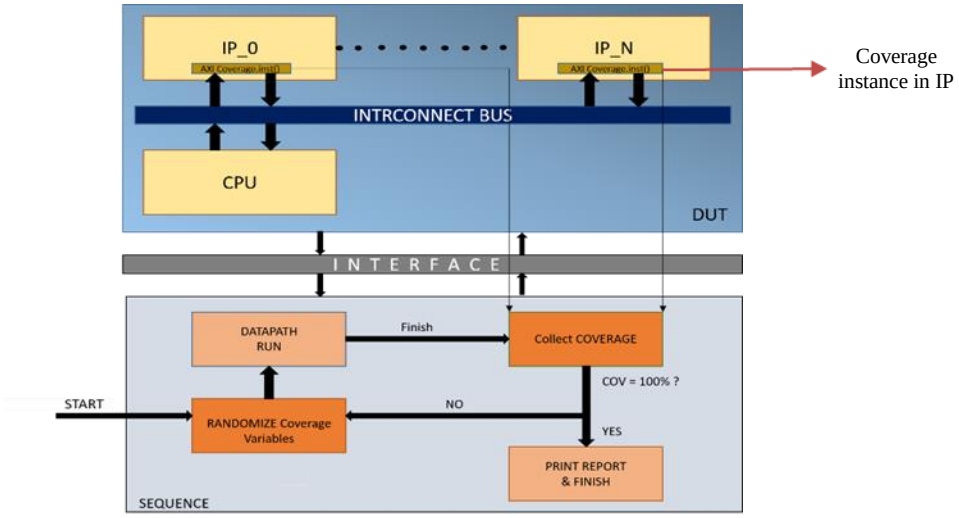


Fig.3: Sequence Implementation

Below is the overview of how Smart Stimuli Generator operates:

1. ***Randomize Coverage Variable***: The initial step of smart stimuli generator involves analyzing the functional coverage requirements of the design such as coverage bin information and identifying the variables that need to hit those bins, those variables will be called as **coverage variables**. The user will serve this data as an input to Randomize Coverage Variable block, which initiates the first phase(i.e., execution of Datapath) of simulation. Following this, the variables are re-randomized for the second phase based on the results obtained in the first phase. For example, if the coverage result of the first phase is determined to be 20%, then this block will generate a new unique value for the coverage variables. This newly generated value is utilized in the second phase of simulation. The uniqueness of this block lies in its ability to generate distinct values for the variables each time it is invoked.

2. ***Datapath Run***: The smart stimuli generator uses a functional datapath that actively exercises the defined coverage bins. The user will be responsible to provide the functional datapath to target the cover bins. This block operates by obtaining inputs from the Randomize Coverage Variable block, specifically the coverage variable that are assigned to target those bins. Its primary purpose is to run the functional datapath with the coverage variable configuration.

3. ***Collect Coverage***: The Collect Coverage block serves the purpose of gathering the coverage data by utilizing the **get_coverage()** task. This task can be called in a hierarchical manner to obtain coverage information for a specific covergroup or cover bin like below:

Grp_Coverage = top.dut….IP0.covergroup_inst.get_coverage();

Or

Bin_Coverage = top.dut….IP0.covergroup_inst.bin_inst.get_coverage();

The task returns the coverage data specifically for that covergroup or cover bin.

4. ***Feedback Loop***: The smart stimuli generator employs a feedback loop that remains active throughout the simulation. This loop continuously evaluates the coverage results obtained from the Collect Coverage block and determines whether the coverage goal has been achieved or not. Based on this evaluation, it directs the execution either back to the first step i.e., Randomize Coverage Variable for further coverage exploration, or proceeds to the Report block for final reporting and analysis.

5. ***Report***: The Report block uses to generate the final results and produce a report in the form of report.xls file that summarize the overall simulation. This block generates a log that provides comprehensive coverage details, including the number of iteration and the coverage percentage for the specified cover bin or covergroup. Additionally, the Report.xls files present a graphical representation of the number of iteration versus the coverage percentage, offering a visual overview of the coverage progress.

Here are the advantages of Smart Stimuli generator:

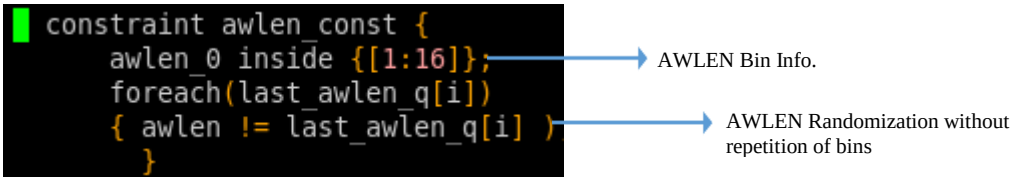
- Auto repetitive process of running datapath with new constraints until 100% coverage achieved in a single simulation
- Eliminate the need for post-simulation coverage merging, results analysis and testcase modification
- Enables addition of more scenarios in verification plan, expanding coverage of design
- Save time and efforts in writing multiple Testcases

This approach can significantly accelerate the SoC verification, particularly when dealing with critical protocols such as Q-channel, LPI and Power Map Testcases.
The conventional approach to functional coverage typically begins after completing the initial verification or 100% testcases pass rate. However, the proposed approach revolutionizes this process by enabling the functional coverage in the early stages of verification as it requires only the bins information and the datapath to achieve the coverage goal. Consequently, it helps in detecting the early bugs in the design.

III. Implementation and Practical Consideration

The successful implementation of the smart stimuli generator approach involves several key consideration and steps. Let's explore these aspects based on the following points:

1. ***Dynamic Constraint Generation***: To leverage the smart stimuli generator, it is essential to provide coverage bins information. This information guides the generator in generating the constraints of the coverage variable for each iteration. The generator uses the techniques such as constrained-random testing, directed testing or a combination of both to generate diverse and comprehensive values for coverage variables that target on specified coverage bins. By defining the coverage bins, the generator can prioritize and generate the specific constraints for comprehensive coverage. Below constraint used by the stimuli generator to randomize the variable with the knowledge of coverage bins and the foreach loop has been used to avoid repetition of same values.



2. ***Coverage Goal***: Defining the coverage goal is an important step in implementing the smart stimuli generator approach. The coverage goal specifies the desired coverage level that needs to be achieved for the given design or functional requirements. The generator will continuously run the datapath until it accomplishes the coverage goal, ensuring through verification.

3. ***Datapath Information***: Providing accurate and complete datapath information is crucial for the smart stimuli generator. This includes specifying the relevant signals, registers and components that need to be exercised during the verification process to target the bins. This generator will provide the constraints for the datapath and ensuring the appropriate coverage is achieved.

IV. Impact on the Verification Process

The smart stimuli generator has a significant impact on the verification process. It improves the overall efficiency and effectiveness of verification efforts. Below are the major impacts of the approach:

1. ***Reduction in Turnaround Time****:* Unlike traditional approaches that require multiple simulation and subsequent analysis, the smart stimuli generator approach achieves coverage goal with a single simulation. This reduction in turnaround time accelerates the verification process, allowing for faster design cycles and quicker time-to-market.

2. ***Enhanced Functional Coverage****:* The smart stimuli generator approach ensures thorough verification by achieving 100% coverage for covergroups or coverpoint. It systematically generates testcases that exercise all the critical functional requirements or features, leaving no gaps in coverage. This comprehensive coverage minimizes the risk of undetected bugs and improves overall design reliability.

3. ***Proactive Bug Hunting****:* The smart stimuli generator approach promotes a proactive bug hunting methodology. By continuously running the datapath until the coverage goals are achieved. This proactive approach ensures the potential issues are addressed early, preventing them from causing more significant problems downstream in the development process.

4. ***Streamlined Verification flow****:* The approach streamlines the verification flow by eliminating the need for post-simulation coverage mixing, results analysis and extensive testcase modification. This streamlining reduces manual intervention and improving overall productivity

5. ***Emulation Scope:*** Another significant impact of the smart stimuli generator approach is its compatibility with emulation platform. The approach can be effectively implemented on emulation environments, particularly when there is a synthesizable functional coverage available. By leveraging the power of emulation, the turnaround time for achieving complete coverage can be further improved.

V. CASE STUDIES AND RESULTS

A. *Case Study*

1. *Implementation on SoC*

The smart stimuli generator has been tested on a set of DMAs in SoC, shown if Fig.4.

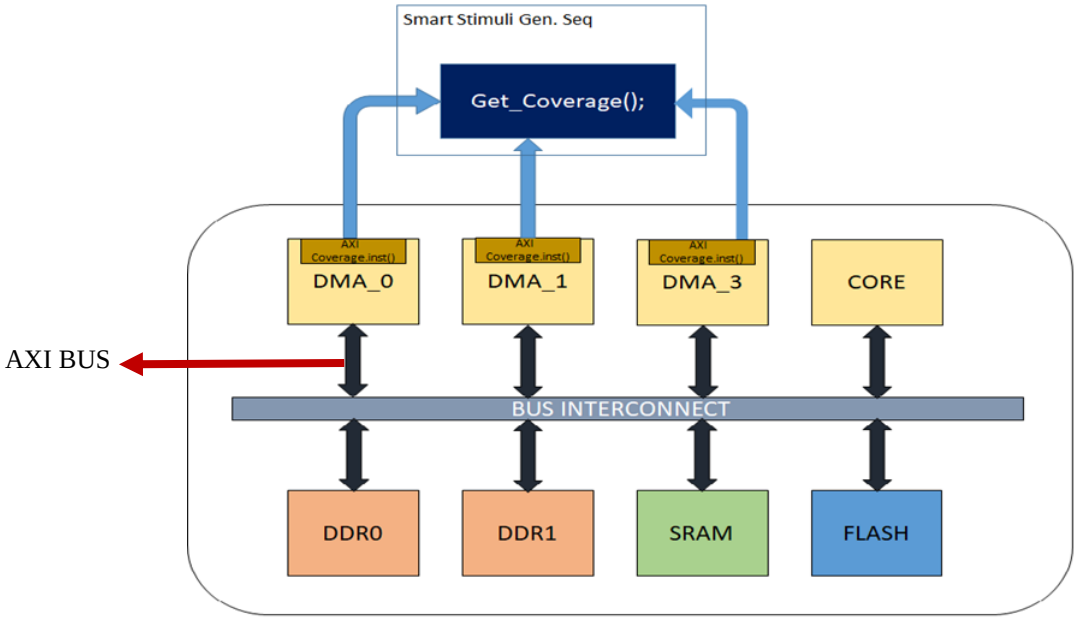


Fig.4: Smart Stimuli Gen Application on SoC

The above figure shows the application of smart stimuli on SoC. The smart stimuli generate various constraints for AWLEN and AWSIZE for the DMA to create diverse flavors of Write transaction and try to accomplish the coverage goals. Below are the cover points of coverage.

```
205 covergroup axi_write_traffic@(posedge coverage_sample_e);
206
207 option.per_instance = 1;
208
209     C4_AWLEN: coverpoint AWLEN {bins awlen_txs[] = {[0:15]};}
210     C5_AWSIZE:coverpoint AWSIZE {bins awsize_txs[] = {1,2,3,4};}
211
212     AWLEN_X_AWSIZE: cross C4_AWLEN, C5_AWSIZE;
213
214 endgroup
```

The DMA datapath has been written to create a transfer from DMA to DDR0/1 by using the AWLEN & AWSIZE constraints generated by the stimuli generator. The initial datapath runs with the random constraints of AWLEN and AWSIZE. At the end of DMA datapath, the coverage percentage of the given cover group collected by the sequence using **get_coverage();** task and then stimuli generator creates new constraints of AWLEN & AWSIZE for the next iteration of DMA transfer. The smart stimuli generator never repeats the same constraints which is already covered. Fig.5 show the flow of smart stimuli generator.

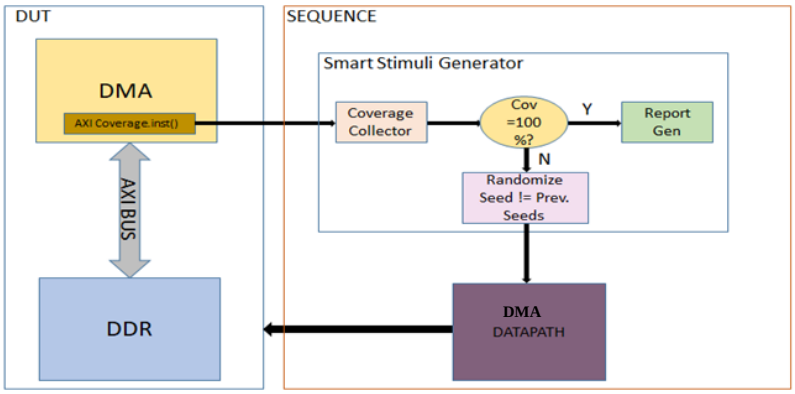


Fig.5: Smart Stimuli Generator Flow

Fig.6 shows the coverage increment with the number of iterations. In this way, we can achieve the 100% coverage for any covergroup or coverpoint in a single simulation by using smart stimuli generator

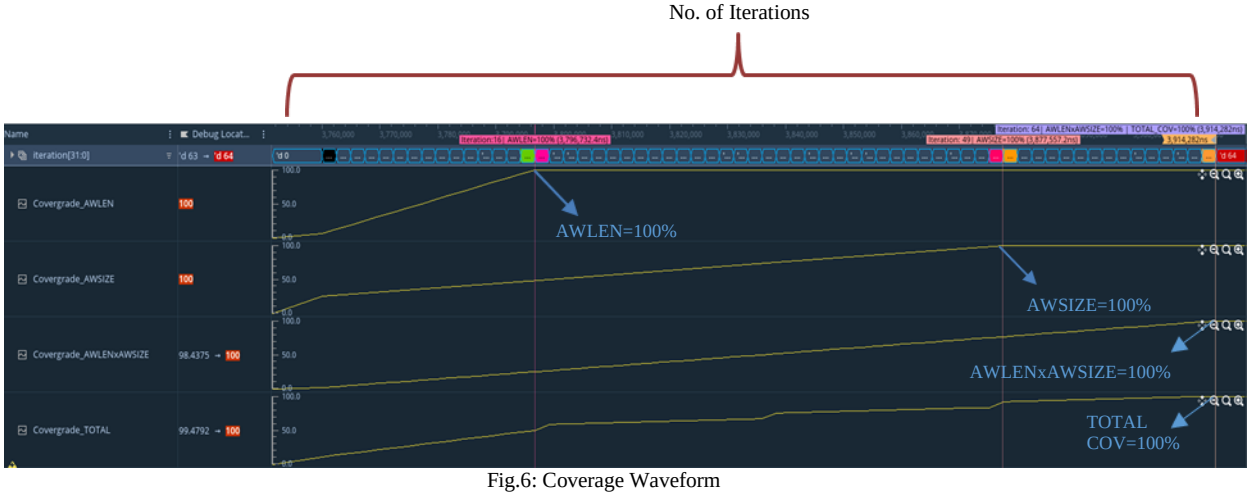


Fig.6: Coverage Waveform

2. Bug Detected

During our experiment utilizing the smart stimuli generator approach to cover the functional coverage of SRAM addresses, an important bug was detected. The coverage bins for the SRAM address were defined as follows:

```
A : coverpoint (A) iff( CSN == 1'b0 ){
    bins addr_0 = {[0:1]};
    bins addr_1 = {[2:3]};
    bins addr_2 = {[4:7]};
    bins addr_3 = {[8:15]};
    bins addr_4 = {[16:31]};
    bins addr_5 = {[32:63]};
    bins addr_6 = {[64:127]};
    bins addr_7 = {[128:199]};
    bins addr_8 = {[256:511]};
    bins addr_9 = {[512:1023]};
    bins addr_10 = {[1024:2047]};
    bins addr_11 = {[2048:4095]};
    bins addr_12 = {[4096:8191]};
    bins addr_13 = {[8192:16383]};
    }
```

During one of the datapaths, the generator produced the boundary address for the mem_wrapper, which resulted in a failure. This issue revealed a problem in the SRAM design, specifically with the handling of boundary accesses. If we had followed the conventional approach then this crucial bug might have been missed.

The detection of this bug highlights the effectivenessof the smart stimuli generator approach in uncovering design flaws that could otherwise remain undtected.

B. Results

The smart stimuli generate a Report.csv after achieving the coverage goal which runs with a script to generate Report.xls at the end of simulation. This Report.xls comprises the coverage report of all cover points along with the corresponding number of iteration required to achieve complete coverage. Fig.7(a) illustrate the Report generation flow and Fig.7(b) Coverage Graph in Repot.xls. Fig.8 shows results in log.

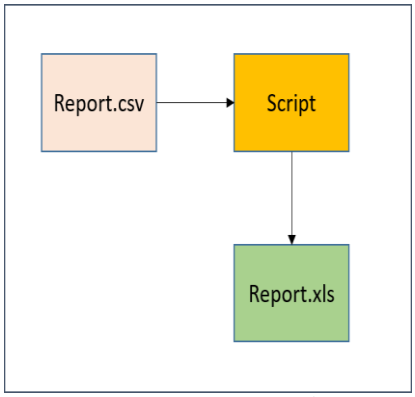


Fig.7(a): Report Generation flow

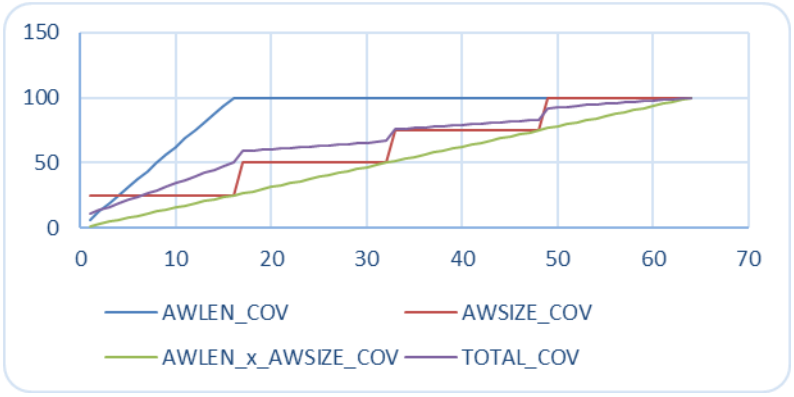


Fig.7(b): Coverage vs Iteration graph in Report.xls

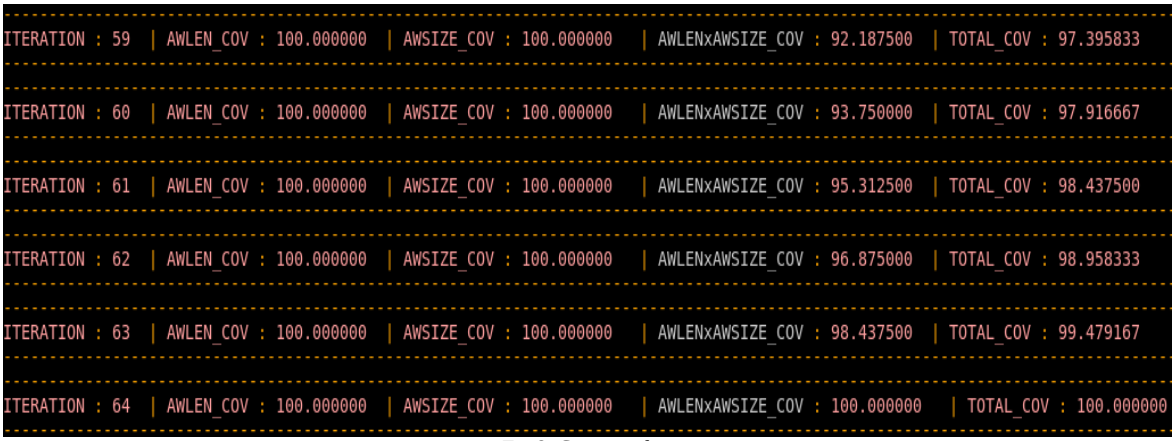


Fig.8: Coverage log

VI. CONCLUSION

The Smart Stimuli Generator approach is a powerful tool for engineer working in the design and verification of complex electronic systems. In addition to its time-saving benefits like simulation time, automatic datapath generation, one simulation for complete coverage, less machine cycle used etc. can accelerate the overall SoC verification process. This approach also improves the overall quality of the design by enabling the coverage expansion through addition of more scenarios, which can identify and address any issue in the design. Below table shows the advantages of the proposed approach over conventional approach.

TABLE I
COMPARISON OF CONVENTIONAL APPROACH AND SMART STIMULI APPROACH

Tasks	Convention approach	Using Smart Stimuli Generator
Testcase volume	36 Blocks x 6 DMAs ➔ 288 Tests	One testcase per block. Overall only 36 tests (~88% Testcases pruned)
Machine time	Average runtime per test – 8hrs 288 x 8 ➔ 2304hrs Max 72 Simulator licenses needed to complete all tests in a day	Average runtime with Smart Stimuli generator is 12hrs. Overall machine time 36 x 12 ➔ 432 hrs Max 18 licenses required to complete all tests in a day

REFERENCES

[1] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.

[2] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017) , vol., no., pp.1-458, 14 Sept. 2020, doi: 10.1109/IEEESTD.2020.9195920