2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

# A Novel Processor Verification Methodology based on UVM

Abhineet Bhojak, Freescale Semiconductor India Pvt. Ltd, Noida , India
(abhineetmnnit26@gmail.com)

Tejbal Prasad , Freescale Semiconductor India Pvt. Ltd, Noida , India

(Tejbal@freescale.com)

Stephan Herrmann, Freescale Semiconductor , Munich , Germany

(Stephan.Herrmann@freescale.com)

***Abstract*—Constantly growing complexity of processor designs increases verification complexity. The Functional space of a processor is very large due to possible permutations and combinations of various instructions and their operands as well as their sequence, so to build stimuli generators to cover all the testable features of processor has been a challenging problem for verification engineers. These results in significant time spent in verification to achieve the goals that are targeted by a verification plan, which is a bottleneck for overall time to market. Different approaches are in use in the semiconductor industry to tackle this problem. Random test pattern generation (RTPG), Test Plan Automation (TPA), Formal property checking, Pure directed testing and some other third party flows are some of the examples.**

**In this paper we propose using the Universal Verification Methodology (UVM) along with advanced System Verilog capabilities as an efficient solution for processor verification. This approach is purely based on Coverage driven Constrained Random Verification and uses the standard methodologies and languages which are well know to a verification engineer without having to depend on a third party flow or a different language. In the paper we have also compared our approach with other existing solutions.**

***Keywords— Processor Verification; Universal Verification Methodology (UVM); System Verilog; Coverage Driven Verification; Constrained Random Testing; Random Test Pattern Generator(RTPG); TestPrograms; Finite state Machine (FSM) Instruction Set Architecture(ISA); System on Chip (SOC); Advanced High Performance Bus (AHB); Read after write (RAW); Write after write(WAW); Write after read(WAR)***

## I. INTRODUCTION

Processor verification has always been a challenging problem to verification engineers. Each processor has its specific Instruction set architecture, addressing schemes, pipeline depth, execution strategies etc. To grow the customer confidence in the design there is a huge functional space that needs to be covered by the verification plan which needs to be created in accordance with the specification. Once the verification plan is complete the next step is to choose the technology to implement it. There are a number of approaches one can take. On abstract level to verify a processor, as a stimulus one needs to generate a program which consist of different instructions to be loaded in instruction memory, the data which these instructions process and some specific register configuration of Processor. Combination of different instructions generate interesting scenarios which can only be targeted with random program generation as it will take significant amount of development cycle time to target these with directed test cases. On the other hand with pure random instruction testing there is a limitation that it may take variable amount of time to converge to required coverage goals and also there are several specific scenarios like infinite loop generation etc. which needs to be taken care of. Another key issue with pure random approach is that in actual use cases a processor program will have a co-relation among the instructions so that they implement a data-processing-flow and these cases lead to several hazards and data coherency issues which may be hard to hit so some intelligence in program generation is required here.

This paper proposes processor verification flow where we have applied Constrained-random verification (CRV) to solve processor verification problem and exploited special utilities of UVM and system verilog language without depending on any third party platform or language. We have implemented a unique “stimuli generation” flow which is well designed to cover the processor specific scenarios such as to cover complex

instruction sets, multiple pipeline-stages, in-order or out-of-order execution strategies, data hazards, branches etc. We generate meaningful programs which have a interlinking among the instructions combined with the specific peripheral register programming sequence and correlated data which would be processed by the processor.

A layered testbench architecture is evolved to have a higher reuse of sequences which reduces initial bring up time and hierarchical constraints are used to generate consequential programs in which instructions are interlinked rather than being pure random. Most of the processor designs are verified against Reference models which are zero time executables in nature so debug ability becomes a bottleneck in finding design defects and hence governing the length of verification cycle. The proposed verification testbench is such that apart from the exhaustive verification & easier debug, it incorporates easy to use hooks to directly port the High level Use case.

Margins, column widths, line spacing, and type styles are built-in; examples of the type styles are provided throughout this document and are identified in italic type, within parentheses, following the example. Some components, such as multi-leveled equations, graphics, and tables are not prescribed, although the various table text styles are provided. The formatter will need to create these components, incorporating the applicable criteria that follow.

## II. RELATED WORK

Different approach for processor verification can be formal property checking, but it requires significant mathematics skill to relate to the scenarios and analyze them. Simulation-based verification thus plays an important role in the functional verification of microprocessor designs. Approaches for instruction level functional verification are mainly concerned with the generation of directed and/or (pseudo-)random test programs. The methods for automatic test program generation include simple random instruction selection, finite state machines (FSM), linear programming, SAT, constraint satisfaction problems (CSP), or graph-based test program generation. Bin [7]and Adir [6] model the test program generation problem as CSP. Their framework, Genesys-Pro, combines architecture specific knowledge and testing knowledge and uses a CSP solver to generate efficient test programs. The test template language of Genesys-Pro is quite complex. It allows for example, biased result constraints. Mishra [8] use graph-based algorithms to generate test programs. While Corno uses a predefined library of instructions, Mishra's work extracts the structure of the pipelined processor directly from the architecture description language and then fed to a symbolic model verifier.

One major drawback of standard RTPG's is that there is a significant learning curve involved to leverage these RTPG's in an industrial environment. Even the minor feature additions require extensive modifications to the RTPG framework, from parsing the template over modifying the RTPG core itself to implement the new features up to maintaining backward compatibility to previous versions. Pure directed checking is not reasonable as it takes a lot of time and a pure random approach may not be able to meet the required coverage goals within the required timelines.

## III. PROPOSED METHODOLOGY

### *A. Overview of Stimuli Generation*

As shown in Fig. 1 stimuli generation consists of generating the program of a random length that consists of atomic instructions which may or may not be interlinked, co-related data for that program and at last some specific peripheral programming sequence related to the program. A combination of these three is called scenario layer and it is created by using object-oriented stimulus generation solution based on the UVM class based library where each one of these can be generated with a sequence under a top virtual sequence. For program generation stimuli generator outputs a series of assembly language instructions, such as :

```
Load address1, R3
Load address2, R4
Add R3, R4, R5
Store R5, address3
```

This is achieved by creating individual instruction with an atomic transaction UVM class. Common aspects to randomize include the opcode, source and destination addresses, jump address, immediate values and values in memory, registers and addressing modes. This is done at bottom atomic layer which also has some intelligence to solve specific problems such as to hit hazard scenarios, avoid infinite loop etc. The Atomic layer transaction class can be extended to create instruction group classes which generate instruction from a given instruction group when randomized and hence have some specific constraints.

In top level scenario layer we decide the size of the program and we can choose a structure of the program with fixed and random instruction to verify a set of instructions. There is a program generator layer which is a UVM sequence class, exist below top level scenario. Based on the information passed from above layer and test case, it calls the atomic layer to generate instructions. All the layers of the approach are explained in more detail below.

**Scenario**: In this top level UVM virtual sequence a random program length is chosen and the skeleton of the program is generated by constraining some locations of the program to be fixed instructions and others to be random instruction chosen from the valid set or a particular instruction group (branching operations, ALU operations). This sequence also incorporates the corresponding peripheral register programming sequence and control the type of input data being randomized for example (negative, positive, all one, max, negative etc.), which may be needed for a particular instruction. The implementation of top level program sequence as shown in Figure 1 is shown with the code of processor program generation sequence in Figure 2.

**Program Generator**: This layer receives the program skeleton information from the above layer and does the decision making on how to randomize each atomic instruction in the program. This layer is an uvm_sequence class it randomizes a processor transaction class which in turn randomizes individual atomic transaction. Processor transaction class is shown in Figure 3.

There are two ways in which it can randomize the instruction. If the instruction is fixed it generates the instruction of the given type. If the instruction is random then it generates instruction from the particular instruction group which is dictated by the uvm_test class corresponding to the particular test case. The test case class uses type_override utility provided by UVM to replace the base transaction atomic class with the specific extended class which may generate instructions of a particular instruction groups or have some specific constraints. This has the advantage that we can test two different instruction group by just changing the base transaction class with override and keeping rest of the things same. It reduced the time by avoiding redundant sequence coding.

**Atomic Transaction**: This layer is the atomic transaction item class, when invoked by the sequence layer it randomizes different instruction fields and packs them as one atomic instruction based on various constraints. This layer also has the intelligence to make the program interlinked in terms of operands used as shown in Figure 4. The aim is mimic the temporal locality in the actual software due to which operands of one instruction are likely to be the operands for the subsequent instructions and hence generating classic data hazards (RAW, WAW, WAR) situations in pipeline. It is implemented in way that operands of one instruction both inputs and outputs can be related to operands of a subsequent instructions.

This has a significant advantage over some random test program generation schemes in which these scenarios may be hard to hit. In this layer certain constraints are also coded such as in jump scenarios it avoids the infinite loop creation. Figure 5 gives the details on how to model the individual instruction as a transaction extended from an uvm_sequence_item class.

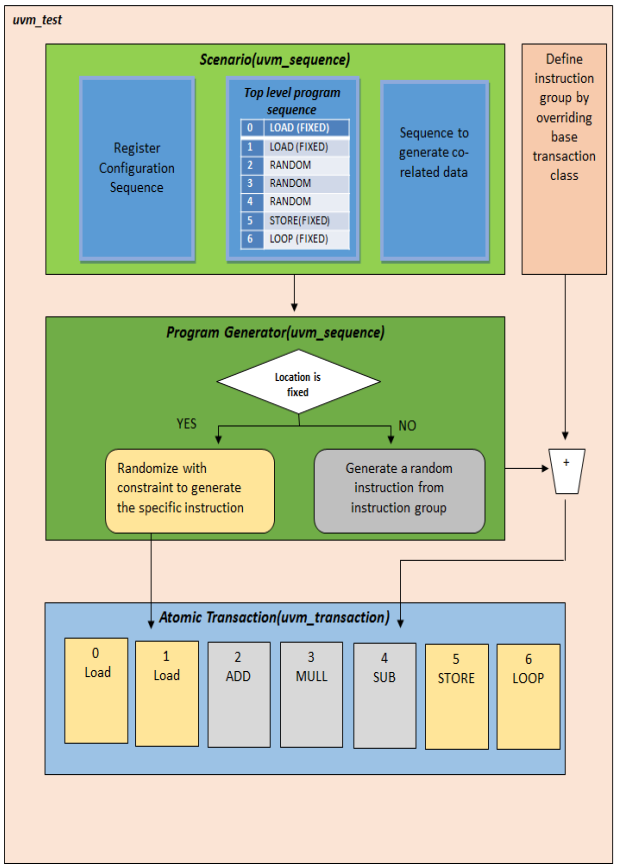


Figure 1 Proposed stimuli generator

```
class processor_program_generation_sequence extends Processor_base_seq;
 rand bit [31:0] program_length;

virtual task body();
 bit fixed_rand_instr_loc_array[64];
 // Set Fixed instruction locations.
 fixed_rand_instr_loc_array[0]                = 1;
 fixed_rand_instr_loc_array[1]                = 1;
 fixed_rand_instr_loc_array[program_length-1] = 1;

 `uvm_do_on_with(processor_user_seq, p_sequencer,
  {
   processor_user_seq.program_length             == program_length ;
   processor_user_seq.seq_lenght                 == program_length ;
   processor_user_seq.seq_array[0]               == MOV        ;
   processor_user_seq.seq_array[1]               == MOV        ;
   processor_user_seq.seq_array[program_length-1] == STORE     ;

   foreach(processor_user_seq.fixed_rand_instr_array[j]){
   (fixed_rand_instr_loc_array[j] == 1)->
    processor_user_seq.fixed_rand_instr_array[j]  ==  FIXED;
   (fixed_rand_instr_loc_array[j] == 0)->
    processor_user_seq.fixed_rand_instr_array[j]  ==  RAND; }
   });

 for( int i = 0 ; i < program_length ; i ++ )
  Regmodel.Processor.IMEM_DATA.write(status,
  processor_user_seq.req.Processor_PROGRAM[i].processor_instruction);
 endtask
endclass :processor_program_generation_sequence
```

Figure 2 Sample code for program generation sequence

```
class processor_transaction  extends uvm_sequence_item;

rand bit [20:0] processor_program_length;
bit      [15:0] processor_imem_base_address;
rand  processorMasterCommandT processor_command_type_seq_array[64] ;
rand  processorInstrFixT      fixed_instr_array[64] ;
processor_transaction_atomic  PROCESSOR_PROGRAM[];

function void  post_randomize();

   for(int j = 0; j < processor_program_length; j++) begin
     if(fixed_instr_array[j] == FIXED) begin
         success = PROCESSOR_PROGRAM[j].randomize() with {
         master_cmd          == processor_command_type_seq_array[j] ;
         instr_fixed         == FIXED;
         program_length      == processor_program_length -1;
         current_instr_index == j ;                    };
     end else begin
         success = PROCESSOR_PROGRAM[j].randomize() with {
         instr_fixed          == RAND;
         program_length       == processor_program_length -1;
         current_instr_index  == j ;                    };
     end

   end
   success = pack_whole_program();
 endfunction : post_randomize

  function pack_whole_program;
    for(int i = 0; i < processor_program_length; i++) begin
      PROCESSOR_PROGRAM[i].current_ip = processor_imem_base_address + 1;
      success = PROCESSOR_PROGRAM[i].generate_instruction_data();
    end
  endfunction : pack_whole_program

endclass : processor_transaction
```

Figure 3 Sample code for processor transaction class

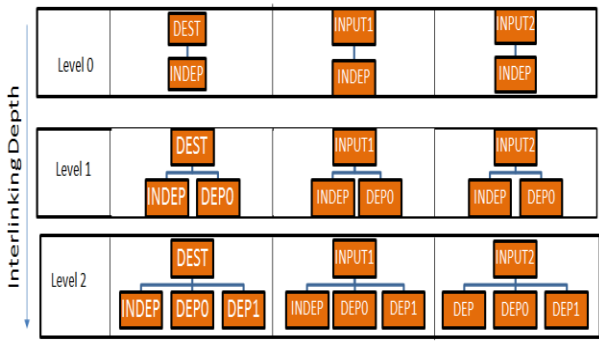


Figure 4 Random interlinking of instruction operands

```
class processor_transaction_atomic extends uvm_sequence_item;

// represent instruction groups
rand    processorMasterCommandT    master_cmd;
//  represents individual instructions
rand    processorCommandT          processor_command_type;

// Variables for construction the final instruction
bit      [31:0]   processor_instruction;
rand bit [7:0]    processor_input_operand_1;
rand bit [7:0]    processor_input_operand_2;
rand bit [7:0]    processor_destination_address;

constraint processor_master_command_decoding {
  (master_cmd) inside
      { /*define the valid instruction groups here */};
  (master_cmd == MASTER_ADD)  ->
      processor_command_type == PROCESSOR_CMD_S_ADD;
  (master_cmd == SCALAR_COMMANDS) -> {processor_command_type inside
    {PROCESSOR_CMD_S_MOV ,PROCESSOR_CMD_S_ADD };}}

constraint processor_operand interlinking {
 (dest_op_relation) inside { INDEP, DEP0, DEP1 }
 (dest_op_relation == INDEP) -> {};
 (dest_op_relation == DEP0)  ->
   {processor_destination_address == last_operand_type};
 (dest_op_relation == DEP1 ) ->
   {processor_destination_address == 2nd_last_operand_type}};

function generate_instruction_data();
   if(processor_command_type ==PROCESSOR_CMD_S_ADD)
   begin
    processor_instruction[23:16]   = processor_destination_address;
    processor_instruction[15:8]    = processor_input_operand_1;
    processor_instruction[7:0]     = processor_input_operand_2;
   end
endfunction : generate_instruction_data

endclass : processor_transaction_atomic
```

Figure 5 Sample code for processor transaction atomic class

### *B. Jump and Loop instruction verification*

While generating sequences of assembly instruction with the random approach loops and jump commands need special attention. These commands have certain associated flags, iteration counts, direction of the jump, amount or location of the jump as their attributes. The set-up, iteration and termination must be well defined otherwise there can be scenarios in which program ends up in an infinite loop. A backward jump without any terminating condition in between is a simple example of infinite loop. With nested loops and different jump instruction there can be several such scenarios.

The key to avoid infinite loop generation problem lies with the direction and amount of jump which can be controlled differently with different kind of addressing modes for example absolute, relative etc. We take an example of relative addressing scheme for jump. As described in Figure 6 we can set the skeleton of the program in top level scenario layer such that it has a loop setup (move a value to loop counter) command at the start followed by a combination of some random ALU instructions, which set different flags and jump commands.

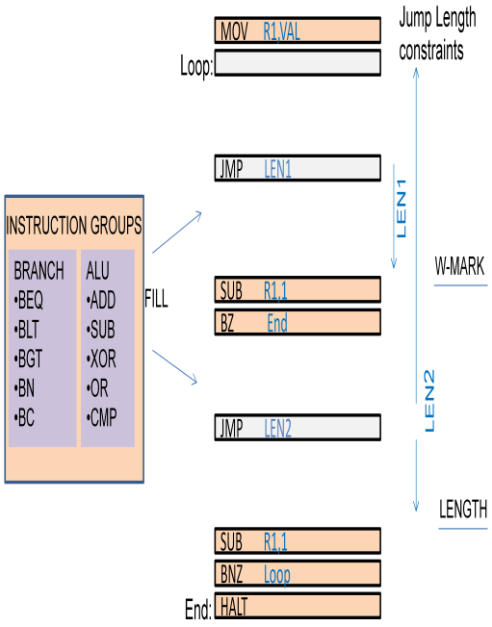


Figure 6 Proposed flow application in jump command verification

These jump commands are only allowed to have a maximum forward jump up to a watermark level such that program always crosses this watermark which is nothing but a decrement of loop counter and acts as a terminating condition, no backward jump allowed in this range. Similarly the jump commands after the watermark level are only allowed to forward jump till the end or if they jump backward then they must jump before the watermark. These jump length constraint can be coded in atomic transaction layer for jump specific command group class which will be selected by the test case for jump scenario verification.

### *C. Testbench architecture to reduce debug time*

Most of the processor designs being pipelined in nature are verified against reference modes which are functional equivalent of processor but does not take care of the pipelining as they are built in a high level language with abstraction. Processor can also have asymmetric pipeline and different execution strategies such as a mix of in-order and out-of-order execution of instruction. If we only compare the model output with the design output at interface level which may be an AHB bus then to debug the exact point of failure takes significant amount of time. We have implemented a scoreboard infrastructure against the reference models which reduces debug time for the complex scenarios and help easier debug with different levels of checkers as shown in Figure 7.

- Conventional checker**:** This is a conventional checker which takes data from memory interface monitor and compares it with golden output data from model. This checker has to do fewer computations as it checks only the final output and it will help in verification of individual instructions but as the program becomes complex it becomes difficult to debug the actual cause of failure.
- Data Trace checker: To quickly reach to the point of failure we have used Data trace checker which compares the change sequence of each processor register with reference mode with a queue based checker.

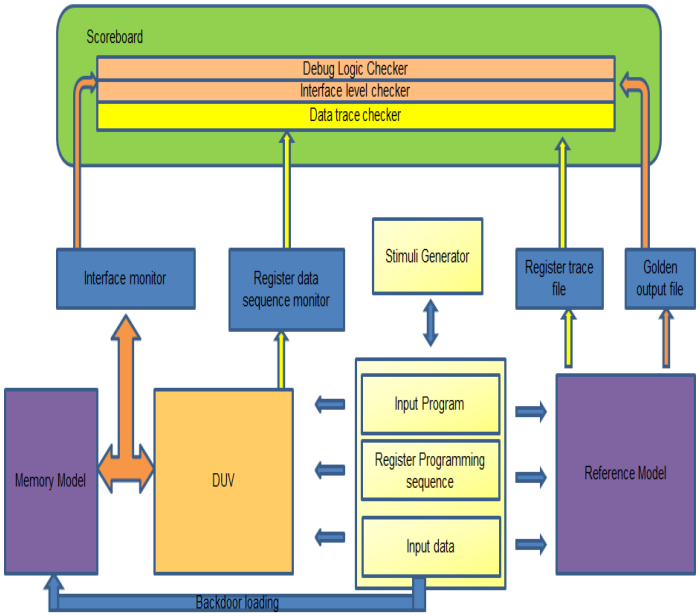


Figure 7 Testbench Architecture

Data trace checker is very useful in verifying out-of-order scenarios because in these scenarios the model register trace changes immediately but the design updates them at later point so having a change based method is useful as compared to per cycle or per instruction checking. For storing all the data traces, golden output as well as inputs, a file based approach is used where data is stored to and read from the files and then filled in queues as having only the dynamic queues might affect the simulation speed due to huge data size and also using files eases the process of debugging.

### *D. Seamless use case poritng from block level to SOC level*

To enable seamless use case porting which are received from software teams as well as for faster scenario replication testbench architecture implements a two way communication between stimuli generator and the input files. So the testbench can either generate the stimuli or consume the inputs provided.

- Stimuli to input: This is a conventional way in which the stimuli generator generates random stimuli and writes it to input files in terms of program file, data file and programming sequence file which is used by reference model and design.
- Input to stimuli: In this mode the stimuli generation is turned off with a parameter. The use-case which is present in input files is read by stimuli generator and used by testbench .

## IV. PERFORMANCE ANALYSIS INFRASTRUCTURE

Performance is one of the most critical aspects of any processor design and hence it is imperative to have a well defined infrastructure in the verification environment upfront with which we can find the performance limiting factors earlier in the design cycle. The design for which we implemented our verification environment

was an image processing core which was to process real time vision data and therefore had stringent performance requirements. Performance was to be analyzed in two ways. First was the bandwidth requirement of the core for accessing memory and other was the pipeline latencies in different image processing operations due to the accelerators present alongside the main pipeline.

Our infrastructure was based on but not limited to analyzing the performance data which was dumped with the design in form of different counter values which indicated performance factors in different forms. A performance monitor was implemented which collected performance data in each testcase throughout the regression and then with the help of some scripting it converted the data to a more analyzable from and written it into an xls file. The data mined out in this way was represented in from of different bar graphs, pie charts and scatter plots with which it was simple to look at the performance bottlenecks. This entire infrastructure was used in a plug-and-play fashion in the sub-system level verification environment where actual use cases were to be run.

## V. EXPERIMENTAL RESULTS

The Proposed verification flow was tested on a custom image processing core which had a scalar and a vector variant. Scalar unit had scalar & matrix operands, 9 ALUs, 1 Multiplier and ~ 256 GPRs. Vector units had scalar, vector & matrix operands, 4 ALUs, 4 Multipliers and ~100 GPRs. Both units had three-stage pipeline-micro architecture, branch prediction and Out-of-order execution. Apart from this there were dedicated accelerators for implementing image processing kernels which are working in parallel with the pipeline. Accelerators were processing the image data present in Matrix Register and were triggered by writing on some special purpose register. These accelerators added new intricacies to the design by adding several hazard scenarios with the main pipeline.

Total verification time for the image processing CPU and hardware accelerator was 30 man weeks. The verification environment described herein was developed completely from scratch. Approximately 10K functional coverage bins were created. Typically between 12K to 15K random test runs were required to achieve desired functional coverage. 200 odd bugs were found in both design and reference model during verification cycle. The design was signed off with 100% functional & code coverage to get to the confidence. No additional bugs in the CPU or hardware accelerator were found by anyone after IP level verification. Silicon has been evaluated and is considered to be a first pass success.

In addition to verifying the core pipeline and hardware accelerators, with the UVM block level testbench different image processing algorithms were verified. Black level measurement, Dead pixel detection, Debayering, Denoise, High dynamic range, Edge detection using Sobel and Laplacian operators are some of the examples of image enhancement operations that were simulated in the block level testbench.

## VI. FUNCTIONAL COVERAGE

The Coverage model included the functional cover-points to cover different instruction opcodes, operands and special fields. Cross coverage was extensively used to insure that all permutations and combinations of different instruction and operands are covered. Hazard scenarios needed special attention in functional coverage model as well. Transition coverage was instrumented for instructions and operands to model and cover all possible hazard conditions. For example multiplication was a 4 cycle instruction and rests of the instructions were 3 cycle instructions. If same operand is used in the consecutive commands after multiplication that implies there should be a stall in the design. Similar was the case with accelerators as if accelerator was working on some set of registers and those registers are also accessed by main pipeline it lead to hazards.

## VII. Conclusion and Future work

In this paper we proposed a verification methodology based on UVM and System Verilog as an efficient solution for various processors specific verification challenges. This methodology can be easily adapted for different processor architectures without significant changes, one needs to embed the instruction set architecture and this solution can be reused. The solution is purely based on open source methodology UVM and standard language System Verilog. We also briefly discussed other existing solutions for processor verification and compared them for their advantages and disadvantages. In our future work, we plan to test the methodology in the verification of high-end processor designs and its extensions for multi-core processor verification at subsystem level needs to be explored.

## References

[1] C. Spear, "System Verilog for Verification, A Guide to Learning the Testbench Language Features," Springer, 2008G.

[2] S. Rosenberg, M. A. Kathleen, "A Practical Guide to Adopting the Universal Verification Methodology (UVM)", Cadence Design Systems, 2010.

[3] E. Hennenhoefer, and M. Typaldos. "The evolution of processor test generation technology," Obsidian Software Inc., 2008.

[4] J. C. Chen, "Applying CRV to microprocessors," EE Times-India, December 2007.

[5] Seonghun Jeong , Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. "Random Test Program Generation for Reconfigurable Architectures". In *13th International Workshop on Microprocessor Test and Verification (MTV)*. Austin, USA, December 2012.

[6] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon,Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in testprogram generation for functional processor verification. IEEE Des.Test, 21(2):84–93, March 2004.

[7] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using a constraint satisfactionformulation and solution techniques for random test program generation.IBM Syst. J., 41(3):386–402, July 2002.

[8] Prabhat Mishra and Nikil Dutt. Graph-based functional test program generation for pipelined processors. In Proceedings of the conference on Design, automation and test in Europe - Volume 1, DATE'04, pages10182–, Washington, DC, USA, 2004. IEEE Computer Society