Reusable Processor Verification Methodology Based on UVM

Mustafa Khairallah, Boost Valley for Engineering Services, Cairo, Egypt
(*mustafa.khairallah@boostvalley.com*)

Maged Ghoneima, Ain Shams University, Cairo, Egypt (*m_ghoneima@ieee.org*)

***Abstract*— Processor verification has always been a challenging problem for verification engineers. The ever-growing complexity of processor designs increases the verification complexity, and the gap between a verification plan and the available technologies to implement it. This increases the time spent on the implementation, and leads to a questionable quality of the verification process. One of the approaches that can be utilized is formal property checking, but it requires a white-box methodology and some mathematical skills to analyze logical relations in the design, so it is more convenient for block-level verification. The other approach is dynamic simulation; the main challenge in this approach is test-case generation, which is done either through pure directed testing that is not reasonable for complex designs due to the effort required to develop it, or constrained-random coverage-driven testing. The problem is that the functional space of a processor is too wide that the test-case generation mechanism needs to be well designed to cover all the testable features in the shortest possible time. This makes basic coverage-driven random testing not a good approach, unless combined with complex machine learning techniques. Another approach is to use directed testing to cover all the explicit features in the test plan, while using random testing in a later stage of the project to reveal hidden bugs and functionalities. A new practice emerged that minimizes the effort needed to develop functional directed test-cases, which uses a separate, random stimuli generator that generates test programs for the required processor. Based on this approach, IBM developed the Test Plan Automation (TPA) methodology that depends on formulating a scenario-description language to describe test templates to be used in micro-architectural verification process.**

In this paper, we propose using the Universal Verification Methodology (UVM), combined with advanced System Verilog properties, as an efficient solution for implementing a test-template-based testbench. Moreover, a new testing methodology is proposed for processor verification and can be easily reused for different architectures or micro-architectures by extending the scope and quality of the ideas previously utilized in this scope. We define a UVM transaction that is used by the testbench as a test program. It is modeled as an empty test program with some randomized control fields. The stimulus generator (UVM sequence) picks a predefined scenario, and accordingly sets the required control fields of the transaction. Then it uses the information in the control fields to fill in the test program according to a predefined template for each scenario. We also present a case study demonstrating our proposed methodology on the open source Wishbone Bus Z80 Processor available through the opencores community. Finally, some results are shown, such as the exponential decrease in the time needed to add a new scenario for the test suite, and the easy control of test generation through feedback from the testbench, for example, coverage.

Keywords*— *Processor Verification; UVM; Coverage Driven Verification; Constrained Random Testing; Test Generation; Test Program

I. INTRODUCTION

During the past decade, the time spent by digital design teams on functional verification rose to 60% or more of the total project time. Even developers of smaller chips and FPGAs are having problems with the past verification approaches. It has become more difficult to get our goal of verification using conventional verification techniques [1]. Due to the huge functional space of a processor, processor verification is considered to be one of the most challenging problems facing verification engineers. The functional verification of a processor is the process of raising the confidence level in the compliance of a processor design to its specifications. The verification process begins with creating a verification plan that defines what properties and functionalities need to be verified, what are the methods and approaches that will be used in processor testing and what is the expected behavior of the design. In other words, verification planning is the process of studying and analyzing the design specifications with the goal of quantifying the verification problem and specifying its solution. It is the process of defining functional coverage models and functional specifications of the verification

* The work presented in this paper was sponsored by Boost Valley for Engineering Services.

environment [2].

The testing strategy is one of the major decisions taken in the verification planning phase. Directed testing is well suited for testing single functionalities, but it is hard to hit more complex scenarios using only directed testing. On the other hand, constrained-random verification (CRV) can be very effective in tackling processor verification challenges, such as: complex instruction sets, multiple pipeline-stages, in-order or out-of-order execution strategies, instruction parallelism, multi-precision operations… etc. The most important module of a CRV environment is the test-case generator, which plays a very important role in most of the recent approaches towards developing automated processor verification environments [3]. A test-case generator generates a large set of valid test cases in a pseudo-random way controlled/guided by constrained randomness. The development of such test generators has started to catch attention of functional verification engineers, and researchers since the early 2000s, such as [3, 4, 5, and 6]. Due to the poor features of Hardware Description Languages (HDLs) available back then, Verilog and VHDL, in terms of verification and software, the development of these generators have been categorized as a software problem, tackled either by building them as software applications [6] or by designing new scenario-level languages, such as Test-Template Language in [4]. However, recent efforts have been exerted towards the utilization of System Verilog features as a Hardware Verification Language (HVL) to improve stimulus generation quality such as in [7].

In this paper, we propose a methodology for test generation, and processor verification using the Universal Verification Methodology (UVM) and System Verilog capabilities. We also present a case study demonstrating our proposed methodology on the open source Wishbone Bus Z80 Processor available through the opencores community [8]. Finally, we present other advantages to our proposed methodology, such as the percentage of reuse to add a new scenario for the test suite.

The rest of the paper is organized as follows: In section II some of the related published work is discussed in more details, showing the advantages and disadvantages of each approach. In sections III and IV, the proposed methodology is discussed. Experimental results are illustrated in Section V and the paper is concluded in section VI.

II. Related Work

A lot of efforts have been exerted over the years to face the processor verification challenge. We are going to sum up some of the recent efforts, describing different modern approaches.

Genesys-Pro, IBM’s third-generation test generator for the functional verification of microprocessors, is described in [4]. The major advancements in this generator can be summarized in three points:

A. *Scenario-Level Testing*

The language describing the generated test template “Test-Template Language” has the expressiveness of a programming language.

B. *Genesys-Pro’s framework for modeling processor architectures*

Genesys-Pro provides high-level building blocks specifically suited for modeling processors. However, processor modeling techniques are outside the scope of this paper.

C. *Powerful Generation Engine*

The generator translates the test generation problem into a constraint satisfaction problem (CSP), and uses a generic CSP solver customized for pseudorandom test generation. In other words, each test is based on a set of random variables and a set of constraints that need to be satisfied, instead of developing this test in a directed way.

In [6], the authors discussed the verification flow of TiCore2 processor. They used a verification flow that combined both directed and random testing to achieve the required goal. They also discussed the concept of self-checking test generation, that doesn’t need the presence of reference model in the verification environment.

The generation schemes used in [4,6] have several advantages: (1) They break the generation task into smaller, easier to handle test cases and (2) they allow the generator to use the current state of a processor to control the generation of the following instructions. However, verification plans have evolved to target events in the processor core such as interactions between instructions, which require significant efforts for the creation of test-templates that can generate instructions' dependencies [3].

In 2013, efforts in this direction have evolved to a promising solution by IBM [3]. This solution provided a new method of test generation based on a high level scenario description language that is close to the microarchitecture verification plan, and a microarchitectural model to support this higher level of abstraction. This solution was called Test Plan Automation (TPA). The main drawback in this approach is that the verification engineer had to utilize a non-standard language for test generation, which increases the verification cycle time caused by ramping up engineers to master such a language. Moreover, the proposed language was proprietary of IBM, and designing a whole new language can be a hard task to perform. For example, it took about 4 years for SystemVerilog to replace Verilog as an IEEE Standard.

On the other hand, some approaches, such as [7], proposed object-oriented solutions based on the SystemVerilog language features to tackle the processor verification challenge. In [7], the authors designed a stimulus generato,r in which the stimulus is a program trace that is a collection of one or more instructions. Such a program trace is called a scenario. Each instruction is modeled as an operation that consists of an opcode in addition to some other properties that make it possible to generate the machine code, and at the same time apply some easy to design constraints to these instructions. This approach enabled top-down stimulus planning, while implementing the testing process in a bottom-up manner. The authors claimed that this approach can achieve better verification coverage, and save time by supporting testbench reuse.

One of the promising approaches that can experience great advancements in the near future in the field of processor verification is formal verification. Many efforts are being exerted towards the adoption of formal verification due to its potential to minimize the time required for verification, such as in [9, 10]. However, these efforts target verifying only part of the processor, i.e. block level verification, because formal verification needs advanced mathematical skills, and are hard to model for large designs.

III. PROPOSED METHODOLOGY

The main goal of our methodology is to improve the stimuli generation mechanism for processor verification, in other words, narrow the gap between the verification plan and its implementation. This is achieved by building an object-oriented stimulus generation solution based on the UVM base class library. The test generator, a UVM Sequence, either chooses or is provided with a scenario, described below, to perform. Each scenario consists of a group of operations, each defined by a group of random properties as it will be described below. Finally, the generator fills one or more test templates to map the scenario being tested into an actual program. The main benefit of this methodology is that it enables the test writer to develop tests based on high level scenarios, while keeping the microarchitectural claims and the instruction set details hidden from him. Moreover, a scenario is built in a generic way that easily enables its integration with other scenarios. Consequently, each scenario can be considered as a test template that can be combined with other templates to build complex scenarios. This solution is built in a layered way, keeping the microarchitectural claims at the lower level, and letting the test writer use a simpler scenario level interface to build different tests. The layers of the methodology are described in Figure 1.

The layered structure shown in Figure 1 consists of:

A. Sequence

The high level interface of the stimulus generator. It combines one or more scenarios to generate the final test-case.

B. Scenario

A scenario contains a group of operations along with constraints on the interactions between them. Scenarios help us simplify the tests generated, as scenarios can be classified according to what they are testing. For example, if we are testing ALU operations, we can fix the source and destination registers. Another example is that if we are testing a Read-after-Write hazard, we can fix the other processor parameters.

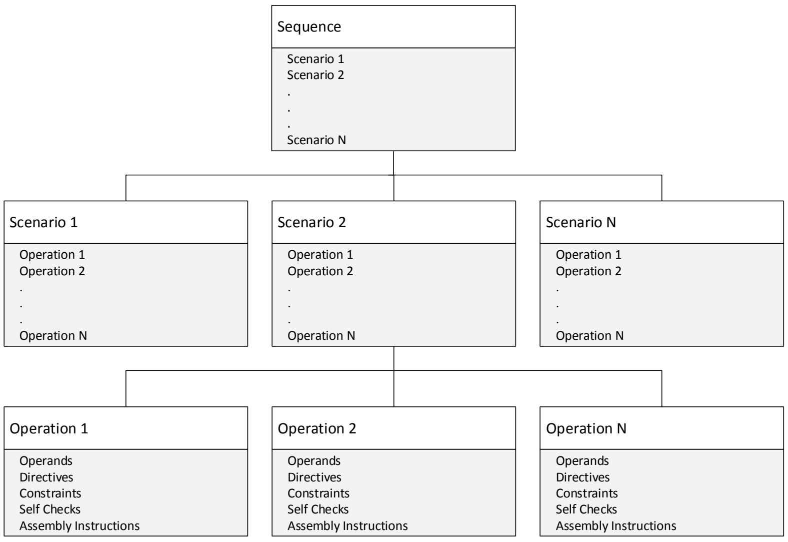

Figure 1 Layers of the proposed methodology

C. *Operation*

An operation is an actual microarchitectural operation that the processor can perform. It contains operands, randomization directives and constraints. It also contains self-checks for the expected behavior of the operation, in addition, it contains the actual assembly instructions needed to perform this operation. A simple example: Load operation has an addressing mode directive, source, destination and data operands, and its check is that after execution (value (source) = value (destination)).

An operation consists of 5 different groups of data:

1) *Operands:* Such as source, destination, data, address …etc.
2) *Directives*: Such as addressing modes.
3) *Constraints:* To ensure that the test-cases are within the operation range of the processor and meet the coverage requirements in the verification plan.
4) *Checks:* A self-checking test can be used to detect the bugs in both the RTL DUT, and the reference model. It is a test scenario with predictable results that can be easily calculated either inside the stimulus generator itself, or inside the UVM scoreboard. The main advantage of this type is that we can start our verification process before having a fully functional reference model. Complex scenarios can be divided into groups of simpler scenarios, using initialization and advanced checking mechanisms provided by the UVM Register Layer. Moreover, the same generated scenarios and sequences, in additon to more complex ones, can be utilized when we have a reference model. The self checks will not be used.
5) *Assembly Instructions:*The actual binary instructions.

IV. Test Template Design

As mentioned earlier, the implementation of our methodology is done in a bottom up manner: (1) the functional space of the processor to be tested is divided into families of possible execution scenarios, namely scenario families. Each of these families contains a part of the processor's functionality. (2) For each family a group test templates is designed. (3) The operations required to execute each scenario, and their relevant properties are defined. (4) Finally, the test generator (sequence) is defined and used to generate scenarios within one or more scenario families.

An object-oriented design is strongly convenient for such a layered structure. A System Verilog class object contains properties, constraints and methods that operate on the properties. In our proposed work, we modeled the 'Sequence' as a UVM Sequence and the 'Scenario' as a UVM Sequence Item, inheriting from the UVM Base Class Library. The 'Operation' is a virtual layer inside the scenario, defined by the properties of the UVM sequence item. A property is typically either a field of the UVM Sequence Item or a piece of supporting information that is needed to generate this sequence item. A scenario family runs a group of similar scenarios. Each scenario family is a UVM Sequence Item class; it has basically the following properties:

A. Scenario Name:

The scenario name is an enumerated-type variable that takes a value from a predefined set of scenarios. It is the main property that directs randomization during test generation.

B. Memory Map

The processor being tested is unaware of the Transaction Level Modeling (TLM) nature of the UVM verification environment. It expects to deal with a memory address space, which is usually a multi-gigabyte address space. However, during a typical test, probably the processor will only access no more than a thousand memory locations. Therefore using System Verilog associative arrays enables modeling a very large address space while allocating only the used elements of this space inside the memory map that emulates a real Random Access Memory [11].

C. Input and Output Data.

D. Register Name:

The internal register name that the scenario will use is decided using this property from the internal registers of the processor.

E. Addressing Mode:

This property decides and directs the instruction into one of the processor's available addressing modes.

F. Operation Name:

A group of enumerated-type variables that specify which operations are to be performed during the execution of the scenario.

The previous properties are not the only properties defining a scenario. Each family of scenarios needs its own properties. For example, a Jump instruction needs a jump address property, which includes the value that should either be loaded into or added to the program counter.

After properties are defined, constraints are defined to describe the relationships between different properties. For example: a shift scenario should execute shift and rotate operations, data addresses should be outside the space of the program inside the memory map …etc. In addition to that, constraint can be used to insert errors inside test cases in an easy way as they can be enabled or disabled during runtime.

The UVM Sequence Item class has a group of methods that represent utilities that deals with the objects instantiated from this class. Some of these methods are just basic utilities used to manipulate the object, such as new, copy, compare … etc. Other methods are used during test generation. We call this type of methods Test Templates. An example of a test template is shown in Figure 2. As shown in Figure 2, a test template is a function that takes a starting address as an input and returns the address where the template ends "end address". It fills the memory map with actual assembly instructions.

At the lower level, the operations are translated into binary instructions. These binary instructions are structured according to the instruction set architecture (ISA) of the processor being tested. Consequently, this part of the methodology is the major part that needs to be changed when testing processors with different instruction sets. An example on how to model these instructions is shown in Figure 3. In fact, this part of the methodology can be made even easier if we integrated a higher level compiler in the verification environment. However, this approach is not safe as the compiler itself may include bugs. Finally, the instructions shown in Figure 3 are placed inside this memory map, which are sent to the UVM Driver to communicate with the DUT using a Bus Functional Model that varies according to the bus protocol used by the processor under test.

As shown is Fig. 4, these templates can be combined together to build complex scenarios, without the need to design these complex scenarios from the beginning. The main challenge now is how to design these templates in an efficient way. For example, a conditional branch in a program can have a low probability of being executed if the data it depends on are purely random. Such problems are handled using the constraints attached to each scenario family, to ensure the efficiency of the generated test cases. The template connection block in Figure 4 checks where the previous template has ended, and calculates the correct address for the next template to start at.

```
function int jump_template (int start_adr);
  int i;
  i = start_adr;
.
    mem_array[i+x]    = JP;
    mem_array[i+x+1] = jp_address [7:0];
.
  jump_template = (i+LENGTH_OF_TEMPLATE);
endfunction // jump_template
```

Figure 2. An example of a test template

Jump Instruction
```
Instruction[0] = JP;
Instruction[1] = jp_address[7:0];
Instruction[2] = jp_address [15:8];
```

Add Immediate Value Instruction
```
Instruction[0] = ADDsA_N;
Instruction[1] = data;
```

Subtract a Register from the Accumulator Instruction
```
Instruction[0] = {5'b10010, reg8};
```

Figure 3. Examples of binary instructions

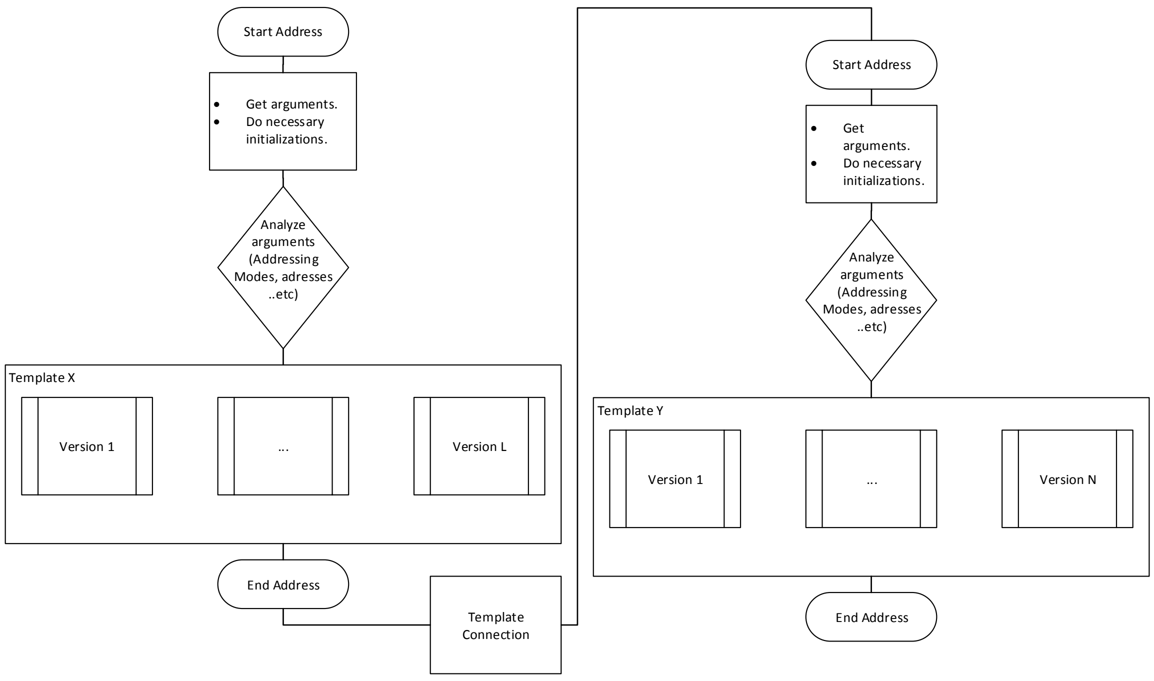
Start Address
Get arguments.
Do necessary initializations.
Analyze arguments (Addressing Modes, adresses ..etc)
Template X
Version 1
...
Version L
End Address
Template Connection
Start Address
Get arguments.
Do necessary initializations.
Analyze arguments (Addressing Modes, adresses ..etc)
Template Y
Version 1
...
Version N
End Address

Figure 4. Template Design and Connection inside a Sample Scenario

V. EXPERIMENTAL RESULTS: WISHBONE Z80 PROCESSOR CASE STUDY

For the Wishbone Bus Z80 Processor, the tests are classified as shown in Table I. The specifications of the instruction set along with the variable properties, and the expected behavior of each instruction are mentioned in the Z80 user’s manual, pages 75-282 [12].

Table I. Instruction Level Random Tests

	Scenario Family	Description
1	Simple Load/Store Data	• In this scenario the testbench writes and reads from all the available registers and a percentage of the available RAM. • The percentage of available RAM is user defined before runtime. • The memory locations are chosen randomly.
2	Load/Store Data with Different Addressing Modes	• Apply the same previous scenario, but with different addressing modes. • Test exchange, I/O and block instructions as well.
3	ALU Basic Operations	• Test all the 8/16-bit arithmetic and logic, rotate and shift and bit-manipulation. • Load immediate values into registers. • Reset the processor between different test cases.
4	ALU Switching	• Switch between the 4 available ALUs (8/16-bit arithmetic and logic, rotate and shift and bit-manipulation) without resetting the processor in between.
5	ALU Operations with Different Addressing Modes	• Test all the instructions in scenarios #2 & #3 along with randomizing the addressing modes.
6	Interaction between ALU Instructions	• Randomize between ALU operations on fixed data and let the scoreboard calculate the expected value.
7	Jump/Call/Return	• Put some routines in different, yet fixed, places in the memory, jump to their locations and check for their expected values.
8	Jump/Call/Return (Control Hazards)	• Combine scenarios #6 & #7. i.e. put scenarios in #6 at different memories for #7 to work on them.
9	Interrupt	• Repeat the scenarios #5 to #8 with random interrupt requests inserted within the scenarios.
10	Data Hazards	• Data hazards (Read-after-Write, Read-after-Read, Write-after-Read and Write-after-Write) are tested. This can be done after scenarios#1 and #2.
11	Exceptions	• CPU Control Instructions are tested.

The Wishbone Z80 Processor has a two-stage pipeline-microarchitecture, which is tested implicitly within the hazard testing mentioned in Table I. First, the basic tests number 1, 2 and 3 were developed. Test templates, checks and coverage collector were developed for each test. In Table II, the number of templates for each scenario family can be found. It is worth mentioning that the classification in Table I has been slightly modified upon implementation. Scenario families 1, and 2 have been merged together as there was no actual need to separate them. Similarly, scenario families 4, 5, 6, and 10 were performed together in the same sequence. The number of cycles in the table is the simulation time, in terms of the processor clock, required to cover the events specified for each scenario family. The verification environment implemented is a Coverage Driven Environment, so each test is terminated when some events are covered. An event is one possible occurrence of a property from the processor's specification, for example, load from register A to register B, add two values with a carry generated, perform a Read-After-Write operation from Memory …etc. The required events are written inside a coverage collector. The percentage of reuse is the ratio between the reused code lines from older scenarios to the total code lines required to implement a certain scenario.

Table II. Samples of Experimental Results for Our Case Study

Test/Scenario Family (Refer to Table 1)	**Number of Created Templates**	**Number of Test-Cases**	**Number of Cycles (approximated to 1K cycles)**	**Number of Events Covered**	**% of Reuse (Approx.)**
1 and 2	7	2,912	38K	85	0%
3	18	29,192	421K	332	20%
4, 5, 6 and 10	18	245	9.5K	73	70%
7	5	552	5K	14	20%

```
address = txn.alu_instruction_template(address+1,
                                       txn.instruction1,
                                       txn.addr_mode1,
                                       txn.register8i1,
                                       txn.data_i1[15:8]);
address = txn.alu_instruction_template(address+1,
                                       txn.instruction2,
                                       txn.addr_mode2,
                                       txn.register8i2,
                                       txn.data_i2[15:8]);
```

Figure 5. A portion of a scenario for generating interaction and data hazards

The third row of Table II is designed using the scenario shown in Figure 5, which reuse templates from previous tests, along with some constraints to stimulate hazards and dependencies between instructions. There are no new templates required to generate scenario families 4, 5, 6 and 10, as all 18 templates used for these families are actually reused templates from families 1, 2, and 3. This is clear in Table II, as the number of templates in the second and third entries is equal, and the percentage of reuse indicates that a major part of families 4, 5, 6 and 10 is reused. In addition, it allowed us to use a constrained random approach and at the same time achieve fast convergence to the complete coverage of events.

During our testing process several bugs were found. Among them some instructions, especially those with different addressing modes, or in Read-After-Write (RAW) hazards, were found to function incorrectly when issued in a row, and have to be separated by No-Operation instructions. Another major bug was found in branching instructions, where the program counter is not loaded correctly.

VI. Conclusion and Future Work

The ever-growing complexity of processor architectures and microarchitectures create a gap between the verification requirements and the test generation mechanisms available. In this paper, we briefly discussed the latest directions and approaches in processor verification. In addition, we explored the capabilities of UVM to be used efficiently in processor functional verification. We proposed a new testing methodology for processor verification that can be easily reused for different architectures or micro-architectures, which is based entirely on UVM and SystemVerilog. We provided a practical example for the adoption of this methodology in the verification of the Open Source Wishbone Bus Z80 Processor.

In our future work, we plan to extend the methodology to use UVM Registers for presetting the processor into a specific state at the beginning of execution of any scenario or for checking the behavior of more complex algorithms. Moreover, the validity of extending the proposed methodology to other areas, such as multithreading, will be investigated. The adoption of the proposed methodology in the verification of high-end processor designs will be also be explored. Finally, expansion efforts to add integration features with compilers are planned, so that templates can be designed in a high level language such as C-programming language.

References

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

[2] A. Piziali, “Verification Planning to Functional Closure of Processor-Based SoCs,” Incisive Verification Article, Cadence Design Systems, May 2006.

[3] Y. Katz, M. Rimon, A. Ziv, “A Novel Approach for Implementing Microarchitectural Verification Plans in Processor Designs,” Hardware and Software: Verification and Testing, Springer, 2013.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv, “Genesys-Pro: Innovations in test program generation for functional processor verification,” Design & Test of Computers, IEEE , vol.21, no.2, pp.84,93, Mar-Apr 2004.

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

[6] F. Bruno, T. Blackmore, “Verifying the TriCore2 Multithreaded Microprocessor,” DesignCon, Santa Clara, California, U.S.A., 2006, pp. 766-789.

[7] J. C. Chen, “Applying CRV to microprocessors,” EE Times-India, December 2007.

[8] B. Porcella, “Wishbone Z80 Core Specification”, www.opencores.org.

[9] R. Kaivola et al., “Replacing Testing with Formal Verification in Intel ® CoreTM i7 Processor Execution Engine Validation,” Computer Aided Verification, Lecture Notes in Computer Science Volume 5643, 2009, pp 414-429.

[10] E. Zarpas, “A Case Study: Formal Verification of Processor Critical Properties,” International Federation for Information Processing 2005, pp. 406-409.

[11] C. Spear, “System Verilog for Verification, A Guide to Learning the Testbench Language Features,” Springer, 2008.

[12] “Z80 Family CPU User's Manual,” ZiLOG, Inc., 2001.