Using Genetic Evolutionary Software Application Testing to Verify a DSP SoC

Adriel Cheng Cheng-Chew Lim
School of Electrical and Electronic Engineering
The University of Adelaide
Adelaide, SA, Australia 5005
{acheng,cclim}@eleceng.adelaide.edu.au

Yihe Sun Hu He Zhixiong Zhou Ting Lei
Institute of Microelectronics
Tsinghua University
Beijing, China 100084
{sunyh,hehu,zhouzx02,leit}@tsinghua.edu.cn

Abstract

A digital signal processor (DSP) system-on-chip (SoC) can be designed using a variety of architectures and techniques. This often presents different verification challenges compared to conventional SoC or processor designs. Verification of such designs should take into account the goals and applications of the DSP, and how they are eventually used. This paper proposes an application based verification methodology and demonstrates this technique on a real-life DSP SoC design. Our technique employs a library of specially devised application functions as test building blocks, followed by a genetic evolutionary test generator to compose these application functions into effective test programs.

1. Introduction

Design verification of system-on-chips (SoCs) is expensive and time consuming. It accounts for up to 70% of resources in a typical design project [1]. Designing digital signal processor (DSP) SoCs create further verification complexities given the range of applications and end products DSPs are employed within.

In order to test a DSP design more effectively, consideration must be given to how the DSP will be used and their intended applications. The eventual real-life usages of a DSP determine the particular design features and functions that are needed in the DSP. It is these design functionalities and their complexities that must be verified in-depth. Therefore, any effective verification strategy must incorporate extensive testing with application functions.

To demonstrate this, the software application level verification methodology (SALVEM) [7,8] is employed to test the Tsinghua University Application Specific DSP (THUASDSP2004) [9]. The SALVEM technique was successfully used on other SoC previously [7,8]. The aim of this paper is to describe the application of SALVEM on a real world DSP SoC; thus demonstrate its feasibility and usefulness for DSP testing. Furthermore, for verification of the DSP, SALVEM is enhanced by an automated test generator that uses genetic evolutionary methods to create tests.

The THUASDSP2004 DSP is an ideal candidate for SALVEM. It was designed specifically for multimedia applications and contains common DSP function blocks such as high performance mathematical and fast data transfer units, along with other specialized modules. These DSP architectural features are to be tested by SALVEM to enhance the design and verification quality of the DSP SoC.

The reminder of this paper is as follows. Section 2 summarizes related work in design verification. Section 3 describes the DSP SoC design. The SALVEM verification approach and test generator are outlined in sections 4 and 5 respectively. Section 6 provides experimental results before the paper is concluded.

2. Related Work

Various solutions have been previously proposed to tackle the design verification problem [2,3,4,5]. Many of these solutions involve an automated test generator based on some form of pseudo random selection scheme to create test cases. The advantage with this approach is that many test cases can be created quickly with minimal effort. However, testing may not be optimized or effective. Due to the inherently random nature of such tests, similar functions may be repeatedly tested or other important test functions overlooked. Ideally, the tests generated should cover as much of the previously untested and critical design functions.

Furthermore, hardware DSP designs require special verification considerations. Some DSP design and modeling environments like Matlab or Simulink do not describe the true hardware design implementation that will be eventually tape-out. They focus on high level DSP algorithmic validation, but the actual hardware design is not tested directly [10]. Another DSP verification solution from Coware [6] allows for hardware design testing. However, to use Coware, designs described in Matlab must be synthesized to an equivalent hardware description using the AccelChip synthesis tool. These DSP test solutions are not suitable for all designs. Our THUASDSP2004 SoC contains different architectural features from conventional DSPs, and does not use the AccelChip synthesis flow.

0-7695-3110-5/08 $25.00 © 2008 IEEE
DOI 10.1109/DELTA.2008.31

Our approach is to employ the SALVEM technique with an inbuilt genetic evolutionary test generator. The aim of SALVEM is to create tests based on the application use cases of the SoC. Hence, important functionalities critical to the real-life operations of the SoC are guaranteed to be tested and verified.

The SALVEM approach is similar to the XGEN [5] method from IBM. Both techniques test the overall SoC by initiating system-wide transactions. XGEN identifies low-level system components and interactions to create test cases. SALVEM catalogues the range of SoC applications and breaks them down into ANSI-C snippets of test building blocks to create test programs. XGEN includes a test generator to automatically create their test cases. However, like other random test generators, these tests may not be efficient. Automatically generated tests should be directed by coverage information to enhance overall coverage.

To facilitate this, the SALVEM technique employs genetic algorithms and evolutionary strategies (GA/ES) in its test generator. GA/ES has been applied for instruction program test generators previously, this was described extensively in [4] and the references cited within. However, they focus exclusive on microprocessor testing, whilst our approach here is to verify system-wide functions on a SoC.

3. The THUASDSP2004 DSP

The Tsinghua University Application Specific DSP (THUASDSP2004) SoC [9] consists of a very-long-instruction-word (VLIW) processor, memories, interrupt and memory controllers, and I/O modules like the DMA for transferring large signal data. Figure 1 shows the SoC architecture.

The THUASDSP2004 is designed to be configurable for use in various multimedia applications similar to FPGAs, whilst delivering high performance matching that of an ASIC DSP [9]. To achieve this, the DSP is built upon a clustered VLIW platform whereby different SoC functions can be grouped into different clusters or sub-divided into different function units. Adopting such an approach enables scalability and flexibility. Depending on the intended usages of the DSP, the design can cater for different numbers and configurations of clusters or function units [9].

Signal processing requires intensive numerical calculations and large data storage and transfers. The THUASDSP2004 DSP contains various configurations of specialized arithmetic logic, multiply, address branch, and load/store function units specifically designed for such DSP operations. The THUASDSP2004 DSP also implements a unique register file based on an inter-cluster communication system [9]. In the DSP, a global register file is used to network clusters, and facilitate data transfers between function units. The advantage from such an implementation is that all data transfer delays and conventional bus cycle latencies are eliminated. Using this communication scheme amongst function units speeds up software pipelining and enhances instruction level parallelism [9].

4. SALVEM Verification

Figure 2 summarizes the SALVEM process. (1) Initially, common use-case applications of the SoC are identified. Software code segments are extracted from these use-cases and modularized into software callable functions called *snippets*, and placed in the snippets library. (2) A test generator automatically chooses various sequences of these snippets (and assigns values for their parameters) to create test programs.

Snippets are the test building blocks of SALVEM test programs. They exercise specific operations and verify real-life system behaviors on a SoC. Implementation-wise, snippets are developed in terms of ANSI-C software parameterized routines.

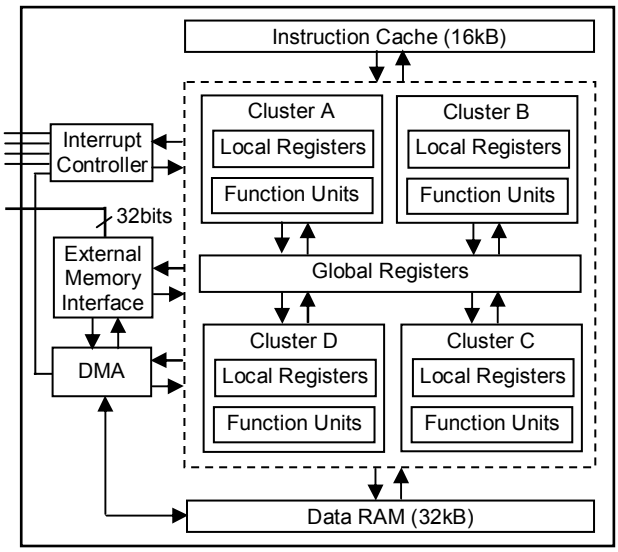

Figure 1. THUASDSP2004 DSP SoC architecture

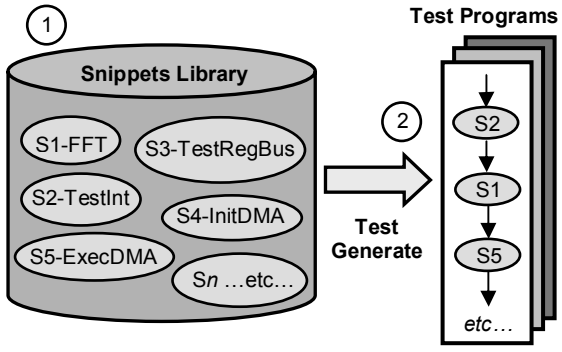

Figure 2. SALVEM process

Snippets initiate SoC operations using device driver application programming interfaces (APIs) to access SoC control and data components such as configuration registers. Snippets also employ parameters. Given different parameter values, various SoC operations are invoked each time the snippet is chosen in a test program. The different sequences of snippets and parameters will facilitate a range of SoC tests. The length of a snippet sequence is constrained by the size of SoC executable program memory that can hold a test program.

4.1. Snippets for DSP SoCs

Snippet test building blocks must be carefully designed keeping in mind the type of SoC under verification. A DSP SoC must perform high intensive arithmetic operations at sufficiently precise fix or floating point level. Data handling mechanisms such as address generation, array handling and data transfers are also important given the large amounts of signal data that must be manipulated efficiently. For example, a DSP often requires many registers to hold temporary or intermediate data during processing. To ensure correctness of these DSP functions, a library of snippets for the SoC was created. Most of these snippets are self-checking, flagging test failure if snippet operations did not perform as expected. We describe the main snippets in the remainder of this section.

The THUASDSP2004 DSP consists of a DMA to handle high throughput transfers of signal data between various SoC on-chip and external memories. Hence, to test the DMA functionality, snippets were created to initialize, execute and check for correctness of DMA transfers. Furthermore, DMA snippets provide parameters to control transfer of different transaction amounts between different memory addresses each time the snippet is selected in a test program.

To initiate SoC operations, snippets rely on low level device drivers to access various SoC configuration registers. For example, reusing the DMA snippets from [7], to initialize a DMA transfer requires the DMA source and destination address registers, and transfer size registers to be configured. The *InitDMA* snippet uses device drivers to initialize these registers and other snippet parameters to test the DMA differently each time. Other DMA snippets also access various on-chip registers to configure, monitor and validate DMA transfers, e.g. *ExecDMA*, *TermDMA* snippets.

In order to test various arithmetic processing capabilities of the SoC, common DSP operations such as discrete Fourier or cosine transforms, and filter functions should be applied. These are the types of applications that use DSP mathematical units and are best suited for testing them. Hence, snippet functions were created for the discrete Fast Fourier Transform (FFT), making use of snippet parameters to vary the type, range, precision and error tolerance of signal data operations carried out. The *FFT* snippets also employ cosine, sine and factorial functions to calculate a range of *n* point FFTs that mimic stress-testing of repetitive and high intensive signal processing operations.

Other specific features of the THUASDSP2004 DSP are also verified by SALVEM snippets. For example, the DSP implements a unique global register file to facilitate inter cluster communication. Each local register in a cluster has a corresponding associate register in the global register file, and vice versa. Whenever data from one cluster is needed by another cluster, the result from one cluster is written to both the local and associate global register. In this way, an external cluster may gather the desired data from the global register file. Using this double associate register writing scheme, communication between clusters is achieved.

The snippets employed to test this specialized register bus system should invoke repetitive data transfers and resolve numerous register address selections. To this end, the *TestRegBus* snippet we developed is based on a token ring transfer operation (Figure 3). Initially, data from any arbitrary cluster is written to both its local and global registers. The data is then consumed by another cluster before it is passed back onto the global register *bus* and transferred to other clusters; until finally, the data is checked at the termination cluster. Figure 3 shows the first three data transfers. The non-circle enclosed numbers show the global register file read/writes, while the circle enclosed numbers indicate the sequence of inter-cluster transfers.

The parameters of the *TestRegBus* snippet are the type and size of data to be transferred, the type and number of clusters involved in the transfers, and the transfer start and termination points.

The *TestInt* interrupt snippet tests the DSP's interrupt handling and priority mechanism. The parameters to this snippet specify which interrupts are enabled and their priorities. Each time the snippet is called into a test program, different interrupts and priorities will be chosen to test the interrupt unit differently.

5. Genetic Evolutionary Test Generator

During test generation, the SALVEM test generator selects a sequence of snippets and snippet parameters to form a test program. Depending on the sequence of snippets chosen, a variety of sequential and concurrent operations on the SoC will be executed. For example, if DMA and FFT snippets are chosen one after another, Fourier transforms of signal data can be concurrently tested whilst data is being shifted between memories.

To create effective and efficient tests, SALVEM employs genetic algorithms and evolutionary strategies (GA/ES) [11] to select the snippet sequence and parameters. In GA/ES test generation, tests are created

similar to the way individual life organisms are evolved during an evolutionary process.

The GA/ES SALVEM test generation process is summarized in Figure 4. In the beginning, test individuals are created to fill an initial population of μ number of tests. These initial tests may contain any number of snippets and relies on the evolution process to vary the test individuals further.

The GA/ES process then iterates, using the μ population of tests to create λ number of new test children via variation. In variation, the sequence of snippets and snippets themselves are intermixed and mutated to create different tests. Next, fitness evaluation of these new test individuals is conducted. Fitness evaluation quantifies the SoC coverage attained by the tests. Test selection is then carried out to retain the best coverage yielding tests for the next cycle of the GA/ES process.

Based on coverage fitness, only the best tests from both the parent population (μ) and newly created children population (λ) are retained. This new selection of tests makes up the new population for the next evolution. Therefore, throughout evolutionary test generation, only high coverage yielding tests are used to create further new tests. Such a selection scheme is called (μ+λ) selection, it ensures the quality of tests do not degrade, but improve over time.

The evolutionary cycle then repeats – varying, evaluating fitness of new tests, and retaining only the best tests – until the termination condition is met. For termination, test generation ends when there has been no coverage improvement of the test suite for the last x consecutive generations; typically x is between 5 to 10.

Representation and variation of individuals are important characteristics of a GA/ES process. We discuss these characteristics in more detail. In Figure 5, the test suite represents the populations of μ and λ individuals, and is simply the set of tests generated by SALVEM. An individual test is equivalent to a chromosome and snippets act as genes making up the chromosome as per genetic coding. The snippets library is the set of available snippets genome.

An important phase in GA/ES test generation is variation. In variation, a test is selected from the parent population, then variation operators are applied to create new tests. The variation operators are as follows.

The snippet addition operator randomly chooses a snippet from the snippet library and inserts this new snippet into the test program at a random point in its snippet sequence. The aim is to add new snippets that can invoke interesting combinations of SoC functionalities with existing snippets to attain higher coverage.

The subtraction operator removes a snippet from the test individual. The intention is to eliminate redundant snippets that do not provide additional coverage; thus enabling the test to run more efficiently and free up snippet slots so more useful snippets can be added into the sequence later on. The snippet mutation operator alters a test individual by modifying the characteristics of a random snippet. Specifically, the mutation operator modifies the parameters of the snippet.

Snippet recombination produces new offspring test programs by selecting two existing test parents and combining their snippet sequences. First, parents are selected using tournament selection. Next, the recombination process selects random *crossover* points from each parent (Figure 6). The first child test program is produced by linking the snippet sequence of the first parent up to its crossover snippet with the snippet sequence of the second parent from its corresponding crossover snippet onwards. The second child test program is produced similarly in an inverse manner.

Addition, subtraction, and mutation ensure new tests are created to explore new areas of the SoC test space, whilst recombination preserves snippet sequences and parameters that maintain high coverage throughout future generations. In this way, the SALVEM test suite continues to verify new SoC functions, whilst being guided by tests from previous evolutions that attained high coverage.

6. Experiments and Results

The SALVEM GA/ES test generator was applied to the THUASDSP2004 SoC to determine the feasibility of applying such a verification technique for DSP designs. SALVEM conducts simulation and coverage measurement of test programs using Synopsys VCS on a register transfer level (RTL) Verilog description of the SoC design. All experiments were run on a 2.2 GHz AMD CPU and 4GB RAM Linux box.

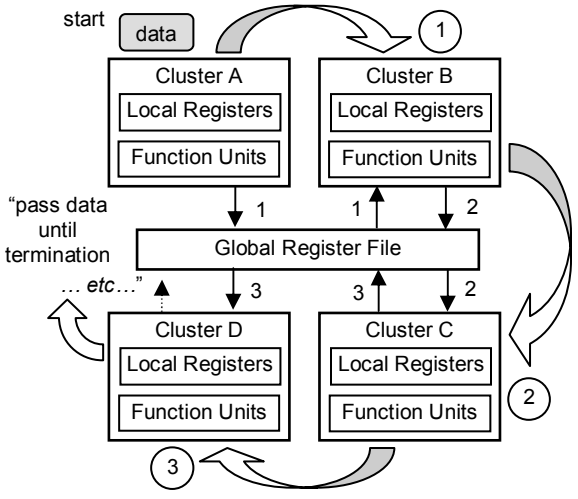

Figure 3. *TestRegBus* snippet operation

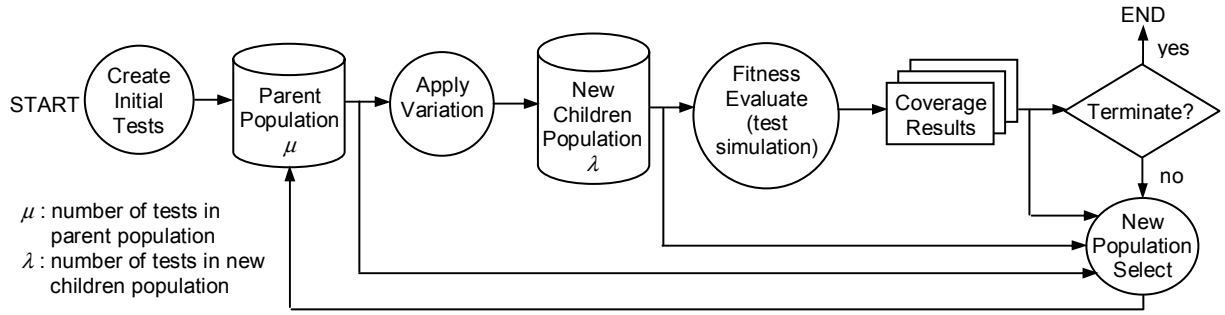

Figure 4. SALVEM GA/ES test generator flow

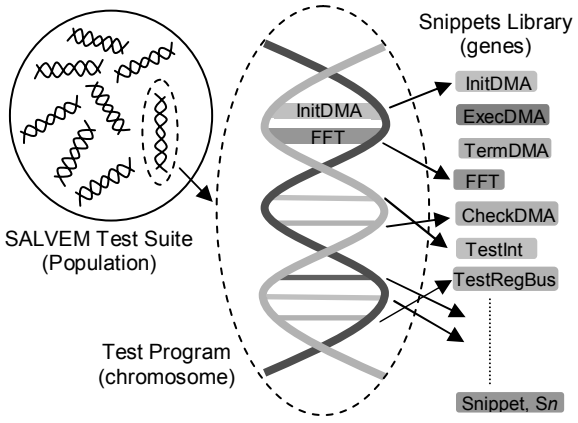

Figure 5. SALVEM GA/ES test representation

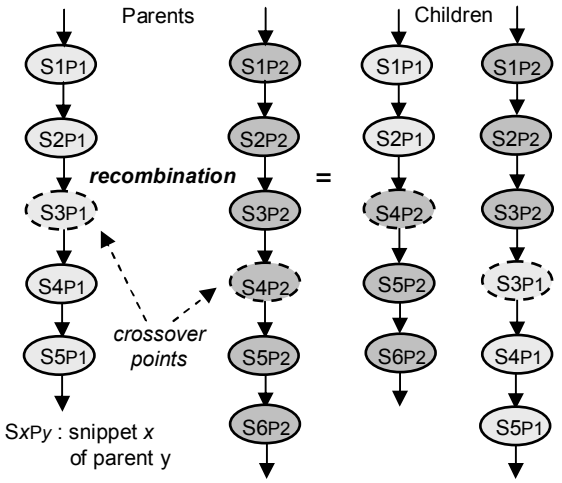

Figure 6. Recombination variation operator

For preliminary experimentation purposes, the test generator used a basic DSP snippets library that includes the snippets in Section 4. The test generator was configured with μ and λ population sizes of 10 and 20 respectively. The test generator is also responsible for creating the random stream of input data that will be processed by the SoC. Three separate test generation runs were conducted targeting line, toggle and conditional coverage as the fitness evaluator. The duration of each test run was about 50 evolutions. In total, 707, 751, and 785 tests were generated respectively for the line, toggle and conditional coverage test runs.

For comparison, in addition to GA/ES tests, a random test generation process was also conducted. Under this scheme, all test creation decisions are random without any influence from previous coverage or other test information. Snippet sequences and parameter assignments are completely random. The random approach created an equivalent number of tests as GA/ES for each of the line, toggle and conditional coverage test runs. By that stage, coverage levels were already maximized with further improvement unlikely.

The accumulated coverage results and total number of snippets executed from GA/ES and random tests are shown in Table 1. For GA/ES, the results represent the best coverage achieved when the test suite has evolved to an optimized state.

In the random approach, the number of snippets in each test was random, as long as the test size did not exceed SoC memory limits. In contrast, GA/ES tests contain smaller number of snippets, relying on the evolutionary process to cultivate them into larger test individuals over time. This provides more efficient usage and lower test sizes overall.

For all coverage measures, the GA/ES approach attained better results compared to random tests. Despite more snippets, the random approach could not match the coverage from GA/ES. The snippets sequences evolved under the GA/ES method was more effective compared to randomly combining snippets together. Conditional coverage is lowest for both test methods. This is due to the nature of conditional coverage measuring whereby the number of conditional paths to traverse in a design increases exponentially with design size, and is much more difficult to exercise.

For comparison purposes in Table 1, we use number of snippets instead of tests because the test sizes between each GA/ES test process and the random approach differ. The total number of snippets (and tests) created by GA/ES each time varies depending on the variation conducted during the evolutionary process. In variation, various snippets will be added, removed or replaced. Therefore, our tests contain different number of snippets, and one GA/ES test is not equivalent to one random test. If recombination is used, two new tests with new sizes are created each time, and each evolution may end up with more than λ new tests. Note that the average size of a snippet was 71 lines of ANSI-C code or approximately 800 bytes after test compilation for both GA/ES and random tests.

Test generation times between the GA/ES and random approach were similar and negligible compared to test simulation times. Over the entire test run, the average GA/ES test simulation time per snippet was 2.4, 4.4, 5.3 CPU seconds respectively for line, toggle and conditional coverage. Whilst for random approach, it took 2.1, 4.7, and 6.0 CPU seconds. The longer test simulation times for toggle and conditional coverage arises from the greater computing resources needed to monitor state elements and track execution paths. Including other overhead, the GA/ES approach required a total of approximately 5 days for the three test runs compared to 8 days for random tests. This was expected given the larger number of snippets executed under the random approach.

The application of SALVEM GA/ES test generation for the DSP required 3 months of a single engineer's effort. The main effort was the development of snippets. However, once implemented, the snippets can be used by the test generator to automatically create many tests to exercise various scenarios, reducing overall effort if test cases had to be created manually.

Whilst we did not discover any new bugs, the goal was to demonstrate SALVEM GA/ES test generation on a real-world SoC design. By doing so, the design quality and confidence in error-free SoC operation can be considered enhanced. Although full coverage was not attained, our preliminary experiments show that applying the SALVEM GA/ES technique is indeed feasible. By expanding our snippets library with more extensive snippets that perform other DSP filtering, transforms, or mathematical functions, and analyzing for remaining dead code in the design, we are confident full coverage can be achieved with greater efficiency.

7. Conclusions

A verification technique using genetic evolutionary algorithms to create software application test programs was applied to a DSP SoC. The technique involves creating specialized snippets of DSP test building block functions, and composing different sequences of these snippets in an evolutionary manner to create tests. Experiments show the genetic evolutionary approach is feasible, and will enable further research to enhance DSP testing based on our approach.

Table 1. Coverage and snippets executed results

Coverage %	Line	Toggle	Conditional
GA/ES	91.3	86.7	80.2
Random-only	83.0	78.1	70.4
Snippets #	**Line**	**Toggle**	**Conditional**
GA/ES	30,400	28,100	42,300
Random-only	56,800	60,100	62,800

8. Acknowledgements

This research paper was supported by the Australia Endeavour Cheung Kong Award and the Australian Research Council (Grant No. LP0454838).

9. References

[1] Collett International Research, "2005 IC/ASIC Functional Verification Study," 2005.

[2] A. Aharon, A. Bar-David, B. Dorfman, G. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM System Journal*, vol. 30, pp. 527-538, 1991.

[3] A. Chandra, D. Geist, Y. Wolfsthal, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, and R. Armoni, "AVPGEN – A Test Generator for Architecture Verification," in *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*. Vol. 3, 1995.

[4] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Code Generation for Functional Validation of Pipelined Microprocessors," *Journal of Electronic Testing: Theory and Applications*, vol. 20, pp. 269-278, 2004.

[5] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, Guy Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-GEN: A Random Test-Case Generator for Systems and SoCs," in *IEEE International High Level Design Validation and Test Workshop (HLDVT'02)*, 2002.

[6] CoWare SPW DSP Workbench and AccelChip DSP Synthesis, http://www.coware.com/news/press263.htm

[7] A. Cheng, A. Parashkevov, and C.C. Lim, "A Software Test Program Generator for Verifying System-on-Chips," in *10th IEEE International High Level Design Validation and Test Workshop 2005 (HLDVT'05)*. Napa Valley, California, USA: IEEE Computer, 2005, pp. 79-86.

[8] A. Cheng, A. Parashkevov, and C.C. Lim, "Verifying System-on-Chips at the Software Application Level," in *IFIP-WG Very Large Scale Integration System-on-Chip (VLSI-SoC'05)*. Australia: 2005, ISBN 07298-0610-3.

[9] Y. Zhang, H. He, Z. Zhou, X. Yang, and Y. Sun, "A Scalable DSP System for ASIP Design," in *Proceedings of Asian Solid-State Circuit Conference*. 2006.

[10] A. Dauman, "Improved Design Methodology for FPGA-based DSPs," in *Embedded Control Europe*, April 2005.

[11] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd Ed: Springer-Verlag, 1996.