Copyright © 2005 IEEE. Reprinted from
IEEE International High Level Design Validation and Test Workshop
(10th : 2005 : Napa Valley, California)

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Adelaide's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

A Software Test Program Generator for Verifying System-on-Chips

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

Atanas Parashkevov[+]
[+]Freescale Semiconductor Australia
2 Second Avenue, Mawson Lakes
Adelaide, SA, Australia 5095
{Adriel.Cheng,Atanas.Parashkevov}@freescale.com

Abstract—**Design verification is crucial for successful systems-on-chips (SoCs). However, validating and proving the correctness of SoCs is often a bottleneck in the design project. This paper presents a novel technique to test the SoC at the system level using software application based programs. Our Software Application Level Verification Methodology (SALVEM) employs test programs composed of dynamic sequences of software code segments. The SALVEM system implements a test generator to create these software test programs automatically. Experiments were conducted applying SALVEM tests to the Altera Nios SoC. A feedback verification flow is also feasible in our SALVEM system. SALVEM test runs are analyzed to direct the test generator toward important SoC scenarios.**

I. INTRODUCTION

System-on-chips (SoCs) are used in many different devices such as consumer electronics and military applications. Advances in semiconductor fabrication and integrated circuit (IC) design techniques have enabled systems traditionally implemented on a printed circuit board, to be embedded completely on a single chip. A SoC integrates many peripherals and processor cores into a standalone system, packing many more transistors and functions into the IC. The increased complexity in a SoC poses many difficulties for design verification.

In a typical IC design project, verification and validation may require as much as 70% of the project lifecycle [1]. Advances in IC manufacturing and design technology have not been matched by similar progress in verification and test. Different techniques must be employed to reduce the verification bottleneck and ensure SoC correctness prior to fabrication. In particular, functional tests must be efficiently generated to test and certify the wide range of behaviors on a SoC.

This paper discusses a technique to generate system tests for verification of SoCs. The test generation technique is encapsulated by our Software Application Level Verification Methodology (SALVEM) previously introduced in [2]. SALVEM employs software application tests for design verification of SoCs during the pre-silicon phase. Software application testing has been employed for co-verification and co-simulation [3,4]. However, these techniques are restrictive. The size of a software application test may be too large requiring fast hardware emulation platforms to ensure they can be executed.

The SALVEM test generator addresses these limitations by extracting and composing segments of software applications known as 'snippets' into software tests. The generated software tests are smaller in size, do not require external hardware testers but still validate the SoC as typical applications would.

The SALVEM concept originated from a number of successful verification projects at Freescale Semiconductor Australia. However, the method employed was ad-hoc and application tests were created manually. The contribution of this paper is an automated software test generator and a feedback verification approach for SALVEM.

The reminder of the paper is as follows. Section II discusses related work in test generation. Section III summarizes the general SALVEM approach. Software test snippets are introduced in Section IV. Section V examines the test generator that composes software test programs using snippets. Experiments demonstrating SALVEM's test generation capabilities are outlined in section VI. Section VII concludes the paper and proposes future research directions.

II. RELATED WORK

The SALVEM verification paradigm is similar to co-verification [3] and co-simulation [4]. These techniques apply stimulus derived from complete software applications to exercise the hardware design. However, co-verification operates later in the design cycle when the hardware design is more complete. Co-verification is often used for confidence-testing to *bring up* the fabricated IC, whilst co-simulation is more suited for software application development and debug.

0-7803-9571-9/05/$20.00 ©2005 IEEE.

SALVEM is solely driven by verification of the hardware design. The software test programs are generated for simulation at the register-transfer-level (RTL) to verify application behaviors and detect hardware design bugs. This alleviates any need for expensive emulation platforms or additional co-simulation software prototypes.

The SALVEM software test generation approach is similar to assembler instructions test generation techniques in [5,7,8,9,10]. Many sophisticated test generator platforms have previously created tests targeting microprocessor cores and memory modules only. In [5,6], the *AVPGEN* test generator generates assembler test programs using constraint solving and symbolic techniques. Similarly, Shen and Abraham [8] developed the *VERTIS* test generator to produce assembler instruction sequences for functional verification and manufacturing testing. IBM has also employed their *Genesys-Pro* generator to verify the POWER5 processor [9].

Corno el at [10,11] developed their *μGP* test generator to create many variants of assembler test programs to stress the Intel i8051, DLX/pII and LEON P1754 microprocessors. Their initial technique was to randomly select assembler instructions from the relevant instruction set architecture library and compose them into test programs. Each test program would contain different sequences of instructions, and each instruction would hold different operand values. The test programs invoke different processor units such as the ALU, cache, pipeline engine, etc to exercise different operations that interact with each other.

The *μGP* ensures the generated instruction sequence is always legal, accounting for various (non-) conditional branch instructions. However, these test programs are not suitable for SoCs as they overlook the other on-chip system peripherals. Assembler test programs are myopically focused on exercising and verifying the processor core exclusively.

SALVEM takes on a similar method to *μGP* but generates software tests at a higher programming level to exercise system-wide SoC functions. Software test programs composed of code segments (snippets) invoke specific operations and collectively exercise many device functions throughout the SoC. Like assembler instructions and operand values, different sequences of snippets and snippet parameters will test the SoC processor and peripherals differently.

However, snippets must be carefully developed to invoke a range of SoC operations. Snippets must allow for parameterization to initiate different device functions and integrate seamlessly into snippet sequences. In contrast, an assembler test generator can use the available instructions from the processor instruction set architecture.

Various techniques to compose assembler instructions have been proposed and implemented. The intuitive approach is to bias-randomize instruction sequences and operands. Other techniques allow user influences and adopt more involved algorithmic processes during test generation. In [5,6], users influence generated test programs toward interesting test behaviors by specifying test generator templates. The templates are described in a constraint-like language. Tests generated from these templates are represented as symbolic instruction graphs and then converted into executable test programs.

Genetic and evolutionary algorithms were also proposed in [10,11] as a means to continually generate high coverage tests. Their aim is to attain the optimal *minimum test size and maximum coverage* test suite. Specifically, $(\mu+\lambda)$ techniques were applied to generate effective tests and attain full coverage. Crossover and mutation operators are used to select the best set of test programs generated.

The SALVEM test generator currently randomizes snippet sequences and parameter values. Snippet biasing facilitates external test generator influences. Users override default randomization and specify biases to direct the test generator toward particular test scenarios.

III. VERIFICATION SYSTEM

The complete SALVEM verification system is summarized in Fig. 1. Initially, the target SoC is analyzed to identify common use-case applications. This forms the basis for generating the types of software test programs simulated. Software code segments are extracted from these use-cases and modularized into software callable functions forming a library of snippets.

The test generator automatically creates test programs that call snippet functions from the library. The snippet functions use low-level device drivers to invoke and test various SoC operations. The Nios compile tool-chain [12] builds the software test program and drivers into an executable binary. Other external data files loaded into memories and streamed by input-output (IO) peripherals are also generated directly by the test generator.

During test generation, a Verilog testbench file configures the SoC hardware environment. The testbench initializes the SoC external pin settings, boot-loads the SoC, and initiates and monitors software test execution. It communicates with other SoC design modules directly to apply external stimulus from the generated data files. Synopsys VCS is used to simulate the SoC according to the test program and various testbench commands. Upon test program termination, the testbench collates SoC test results and coverage data.

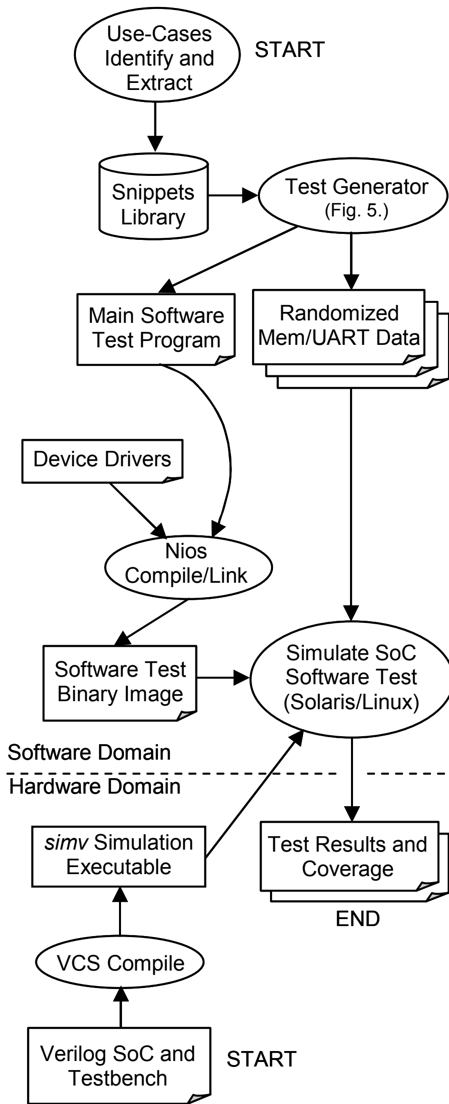

Figure 1. The SALVEM System

IV. SNIPPETS

Snippets initiate and exercise specific operations on a SoC. Each SoC device is assigned a set of snippets. Snippets test specific functionalities on their SoC device but invoke operations on other on-chip peripherals as well. This enables extensive individual device testing and verifies a range of system-wide transactions.

Snippets are implemented as separate ANSI-C software functions. Snippets are created by breaking down SoC applications into modular operations and embedding them into callable parameterized functions.

Our concept behind snippets is based on the observation that many SoC operations are initiated by reading and writing to device configuration registers. At the highest software operating level, software programs must access these registers to exercise particular behaviors. Hence, our snippet functions contain low-level application programming interface (API) device driver calls, operating system routines, and other software code to implement the correct sequence of register accesses. Fig. 2 shows the embedded software and hardware implementation levels of a SoC. Snippets operate at a higher level initiating hardware functions using device drivers and subsequently via register assesses to bridge the various abstraction layers.

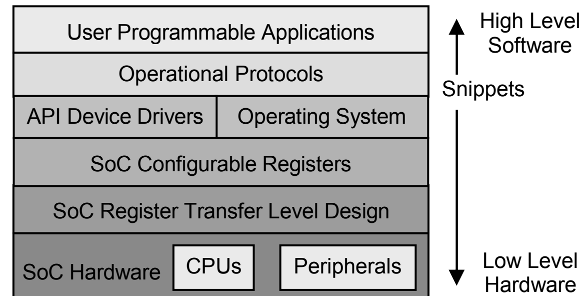

Figure 2. SoC Software-Hardware Interface

Fig. 3 illustrates the internal control flow of a DMA initialization and execution snippets sequence. Each oval-shaped node corresponds to DMA configuration register(s) accesses that initialize DMA operations, initiate DMA transfer, and monitor data transactions. The register accesses are facilitated by device driver calls, and are shown in italics in each node. This enables our high-level snippets to be reused for other SoCs with similar devices. Only device specific drivers for the target SoC needs to be implemented.

Table I shows a basic set of snippets for a typical SoC with a processor, memories and IO peripherals. Each snippet executes sequential or concurrent SoC operations between different devices. The InitDMA snippet configures different data size transfers between memories and IO devices. ExecDMA transfers the data in either blocking or interrupt mode. TermDMA and CheckDMA stops and checks for successful data transfers respectively. Similarly, snippets Tx|Rx|RxTxUart initiates serial transfers. Other administrative snippets supervise the overall test execution.

To compose a SALVEM test program, snippets are combined into random sequences. Fig. 4 shows an abstract graph of a typical sequence of snippets. The sequence is intermixed with snippets that invoke different operations from various on-chip devices. Different snippet sequences are created by the test generator and transformed into executable test programs.

Besides randomizing snippet sequences, snippets provide parameterized variables in its callable function header. Different parameter values will be passed into snippet functions from different sequences. This enables snippets to vary the internal device operations it executes. Parameters control sequential or concurrent operations, enable self-checking, employ prioritized processes, etc. Table II shows the parameters for an InitDMA snippet to configure different types of DMA transfers.

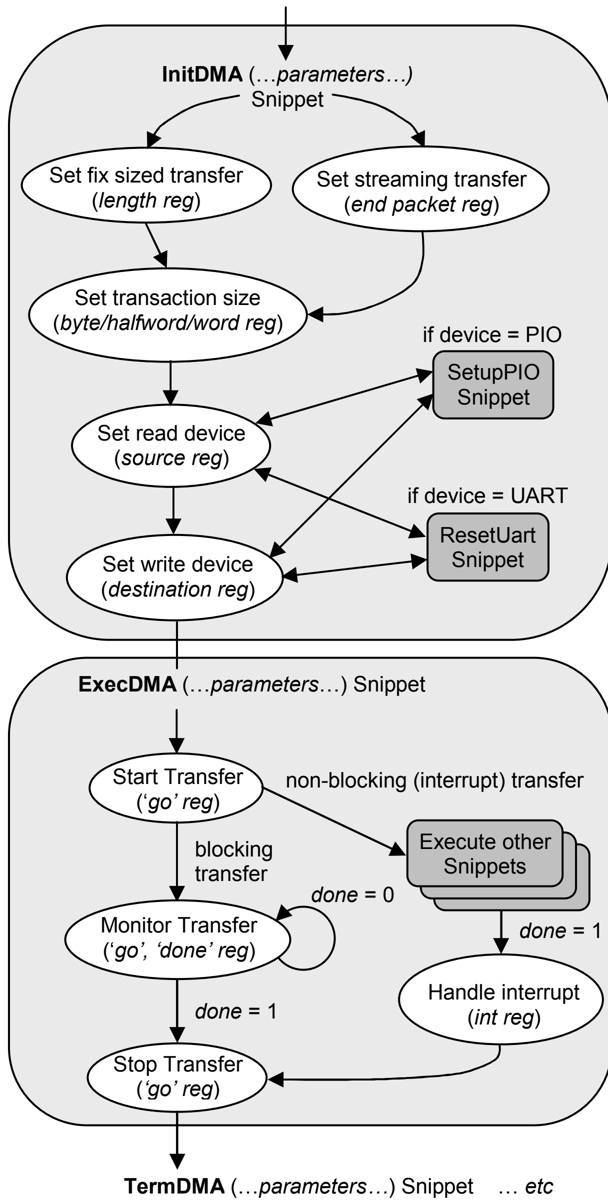

Figure 3. Init-ExecDMA Snippet Control Flow

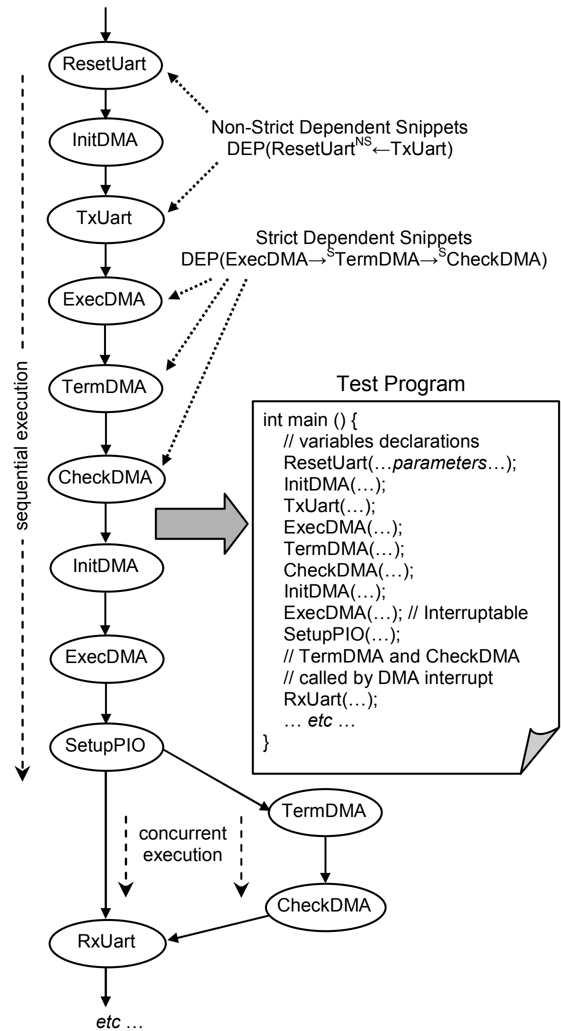

Figure 4. Example Snippet Sequence

TABLE I. EXAMPLE SNIPPET LIBRARY

Snippets	Purpose
InitDMA(…)	Initializes DMA Transfers
ExecDMA(…)	Starts and Monitors DMA Transfers
TermDMA(…)	Stops DMA Transfers
CheckDMA(…)	Checks for Successful Transfers
SetupPIO(…)	Initializes Parallel IO (PIO) Pins for Access
WritePIO(…)	Access PIO Pins for Parallel Transfer
ResetUart(…)	Initializes UART for Serial Data Transfer
TxUart(…)	Transmit UART Serial Data
RxUart(…)	Receives UART Serial Data
RxTxUart(…)	Receives and Transmit UART Serial Data

TABLE II. INITDMA SNIPPET PARAMETERS

DMA Snippet Function Header : InitDMA(id, src, dest, len, eop, t_size, int, src_inc, dest_inc)	
Parameters	**Parameter Options & Values**
Transfer Process ID (id)	32bit Integer Value
Source Read Address (src)	Memory Address ranges or Peripheral Port Addresses
Destination Write Address (dest)	Memory Address ranges or Peripheral Port Addresses
Transfer Length (len)	32bit Integer Value
End-of-Packet (eop)	8bit Character Value
Transaction Size (t_size)	Byte (8bit), Halfword (16bit), Word (32bit)
Interrupt (int)	Boolean Value
Source Address Increment (src_inc)	Boolean Value
Destination Address Increment (dest_inc)	Boolean Value

Snippet sequences and parameters are intermixed as much as possible to verify the SoC more comprehensively. However, snippet tests must be composed according to test generator and SoC architectural rules. Various constraints are required to ensure illegal conflicts or erroneous conditions do not occur unintentionally. Constraints such as CON((DMA_Read_Addr = UART)→(DMA_trans_size = byte)) ensure only byte sized transactions are executed to read from the UART 8-bit port. Similarly, constraint CON((DMA_non_blocking = 1) && (DMA_int_en = 1)) ensures an interrupt is always invoked for non-blocking DMA transfers. Our test generator creates software test programs that adhere to a list of such constraints. However, to evaluate error handling and verify illegal SoC conditions, the test generator can be configured to ignore these constraints.

Dependency rules ensure only legal snippet sequences are generated. SALVEM defines two types of dependencies. A strict dependency requires a dependent snippet to be executed immediately before (or after) the target snippet. A non-strict dependency implies the dependent snippet can be generated amongst other snippets before (or after) the target snippet. Table III defines notations to describe these dependencies.

TABLE III. DEPENDENCY DEFINITIONS

Strict Dependency	
targ_snip→^{S}dep_snip	Dependent snippet required immediately after target snippet
dep_snip^{S}←targ_snip	Dependent snippet required immediately before target snippet
Non-Strict Dependency	
targ_snip→^{NS}dep_snip	Dependent snippet required after target snippet
dep_snip^{NS}←targ_snip	Dependent snippet required before target snippet

For example, in DMA blocking mode, DEP(ExecDMA→^{S}TermDMA→^{S}CheckDMA) require DMA termination and check snippets to execute immediately after a DMA transfer snippet. This example shows dependencies between snippets can also be cascaded.

Similarly, DEP(ResetUart^{NS}←TxUart|RxUart|RxTxUart) imply the UART can be initialized in between other snippets before it is used to transmit or receive data. A comprehensive set of dependencies governs the types of snippet sequences composed by our test generator. Fig. 4 showed examples of strict and non-strict dependencies.

Random snippets are useful building blocks for SoC software test programs because they verify application functionalities. In contrast, a randomized sequence of assembler instructions will not invoke proper device operations. Randomized instructions are unlikely to access device configuration registers in the correct sequence using suitable values. Snippets implement the correct register accesses in terms of software C code. Snippets are compiled and linked into proper sequences of assembler instructions to invoke device operations like DMA or UART transfers.

V. SALVEM TEST GENERATOR

SALVEM employs a test generator to create many snippets software tests automatically. The test generator pseudo algorithm is summarized in Fig. 5. A snippet is randomly selected from the snippet library. Snippet dependency rules are checked to ensure the selected snippet comply, otherwise a new snippet is selected. At least one complying snippet in the snippet library is guaranteed for selection. Next, the snippet object is created. Snippet parameter values are randomly chosen according to user selection biases. Chosen values are checked against parameter constraints. If any constraint is violated, parameters are re-selected. The snippet sequence and main test program is then updated with the new snippet. The test generation routine is called recursively until the desired snippet test length is attained.

```
snippet_library s_lib = {InitDMA, ExecDMA,  … etc …};

Gen_Snippets_Test (constraint cons, dependency deps,
                   bias biases, snippet_history s_hist) {

    snippet_type s_type = Select_Snippet(s_lib, biases);
    while (Illegal(s_type, deps, s_hist))
        s_type = Select_Snippet(s_lib, biases);

    // Instantiate snippet object and parameterize
    snippet snip = s_type();
    snip.Parameterize(biases);
    while (Illegal(snip, cons))
        snip.Parameterize(biases);

    num_snippet++;
    Add_to_Snippet_Sequence(snip);
    Add_to_Snippet_History(snip, s_hist);
    Update_SoC_State(snip);
    if (num_snippet < snippet_test_length)
        Gen_Snippets_Test(cons, deps, biases, s_hist);
}
```

Figure 5. Test Generator Pseudo Algorithm

During test generation, the history of snippet sequence and the state of the SoC is maintained. This enables the test generator to identify which devices are in use at any stage. Dependency rules can be checked, and snippets are chosen based on available SoC resources released by previous snippets. For example, a DMA snippet cannot initiate streaming UART to memory transfers unless previous UART snippets complete and release the UART device.

The test generator and snippets are implemented in an object-oriented manner. Whenever a new snippet is selected into the test sequence, a snippet object is instantiated by the test generator. The snippet object is self-contained, and executes internal methods to select parameters and self-checks against constraint rules. The snippet object also creates the function interface for the main test code to call.

Dependency and constraint rule specifications are also implemented as objects. Similarly, the test generator

maintains the SoC state by instantiating SoC devices as objects. Each on-chip device object checks and updates its internal set of resources ensuring illegal conflicts amongst snippets are avoided.

The SALVEM test generator allows external user influences to direct verification toward certain SoC devices and important test scenarios. Test generator biases enhance the generation likelihood of certain snippet sequences and parameter values. A verification test run can be analyzed to identify uncovered or insufficient testing of specific SoC functions. Biasing is employed to generate new tests to target these functional corner cases. Using coverage test information and biasing, a feedback verification flow is feasible. Section VI demonstrates the use of biasing in our SALVEM feedback flow.

SALVEM implements two types of biasing, weight and range biases. A weight bias assigns relative probabilities to a particular snippet or parameter choice. The choice with the higher weight value is more likely to be selected. A weight of w_x for choice 'x' implies a selection probability of

$$P_x = \frac{w_x}{\sum_{i=1}^{n} w_i}$$, where n is the total number of choices.

For example, weights can be assigned to particular snippets to influence the snippets chosen in the test sequence. Similarly, weights can be used for manipulating the likelihood of byte, halfword, or word transaction sizes in DMA transfers.

Weight biases are useful for constrained discrete choices. Other test generator choices such as UART end-of-packet characters or DMA transfer lengths span a larger range; and each value cannot be specified with a weight. Instead, a range bias is used to specify a sub-range of values. For example, a range bias can specify DMA transfer lengths towards the maximum memory block or segment sizes. The test generator will select values from these bias sub-ranges more likely than other values.

VI. EXPERIMENTS AND RESULTS

The SALVEM system was implemented and applied on the Altera Nios SoC [12]. The Nios SoC is an ASIC synthesizable and FPGA compliable design in RTL Verilog. The SoC can be configured with different devices to implement a variety of applications. Our SoC is configured with common IO and memory devices typical of many SoCs today. The SoC consists primarily of a Nios 32bit 5-stage pipeline processor, RS-232 UART, DMA, PIO, on-chip ROM and RAM, and external SRAM and Flash. The Nios Avalon bus links these devices into a system. Snippets were developed for these devices to verify data transfer, IO communicating, and other SoC functions.

The SoC is simulated using Synopsys VCS to execute snippet test programs and collect coverage data. The SALVEM system, SoC and VCS simulation environment are integrated into an automated flow to generate and batch-run large test suites.

In our experiments, a software test program contains approximately 125 snippets. Previous test suite executions indicate 125 snippets is an ideal test size for debugging test failures and monitoring test simulation. Each test is loaded and executed from on-chip memories. On average, using a 3Ghz CPU and 1Gbyte RAM Linux platform, a typical test generation and execution required 0.30 and 642 CPU seconds respectively. When collecting coverage information however, execution times increased to 1221, 1621 and 7605 seconds for line, toggle and conditional coverage. Without coverage measurement, a large and comprehensive test suite can be created and run in an efficient time frame.

During a verification phase, the SALVEM system was used to generate and execute 25 tests on the Nios SoC. Table IV shows the initial and analyzed (final) coverage statistics. The analyzed coverage is considered the true coverage result. It is obtained by examining coverage data against the SoC design for dead code and other untestable error conditions. For example, unused timer peripherals and several redundant Nios CPU arithmetic units cannot be tested and are considered dead code. Re-configurable features in the Nios SoC imply not all design blocks or functions can be used in the system. Furthermore, some error conditions cannot be tested because the Nios SoC cannot recover from certain illegal operations. For example, the SoC enters a deadlock state when executing DMA transfers between UARTs using data unit sizes larger than the UART ports.

Besides accounting for dead code and error conditions, testing and coverage of the Nios SoC can be improved by employing feedback verification in SALVEM (Fig. 6). During test generation and execution, coverage and test statistics are collected to characterize the type of test programs generated; and identify what SoC devices were insufficiently exercised. Using this information, biasing can be used to direct the test generator to create tests for previously untested functions and improve coverage.

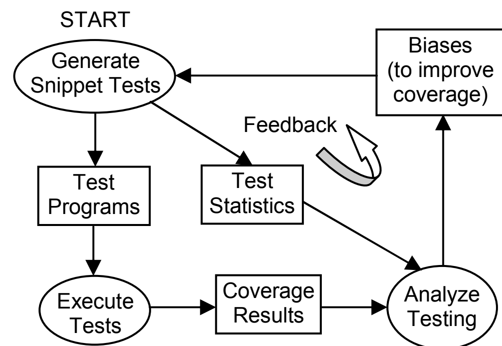

Figure 6. SALVEM Feedback Verification

Fig. 7 and Fig. 8 show a subset of the test generation statistics collated from the test suite in Table IV. The histogram in Fig. 7 identifies which snippets were deficient in the test programs. For example, compared to other

snippets, DMA and WritePIO snippets were not generated and executed as often. The histogram shows more snippets initialized the PIO instead of using it. Additional WritePIO snippets would improve the current PIO coverage of the unbiased test suite.

Despite low DMA snippets and inadequate parameterized test configurations (Fig. 8), the unbiased test suite achieved full line coverage for the DMA device (Table IV). This result is misleading because line coverage satisfaction criteria require statements in the design code be exercised once only. The structure of the DMA hardware design code requires only a few transfer scenarios to exercise each line of the DMA device. Lower DMA toggle and conditional coverage confirms other transfers scenarios were untested and the DMA was insufficiently tested. Similarly, higher PIO and UART line coverage may not imply these devices were sufficiently verified. In general, line coverage is always greater than toggle or conditional coverage. However, line coverage alone is not always reliable and other metrics must be used as well. New coverage methods for SALVEM are also being investigated [2].

Fig. 8 shows the DMA snippet parameter selections. Examining this histogram, the UART, PIO and ROM device was chosen insufficiently as DMA source and destination transfer devices. Subsequently, this resulted in unsatisfactory coverage for these devices (Table IV). The ROM was selected once partly because the ROM can only be chosen by the DMA as a read-only source device. Parameter selections for DMA execution modes also favors non-interrupt (blocking) mode heavily, suggesting insufficient simultaneous SoC operations were executed.

TABLE IV. COVERAGE RESULTS FOR INITIAL UNBIASED TEST SUITE

Device		Full SoC	CPU	Memories	DMA	UART	PIO
Initial Coverage % (Unbiased)	Line	84.55	97.86	67.40	100	95.93	89.57
	Toggle	76.11	83.27	57.43	74.81	70.99	59.74
	Conditional	66.48	66.77	26.36	93.98	72.62	60.00
Final Coverage % (Unbiased)	Line	90.99	97.95	68.85	100	98.15	89.57
	Toggle	78.06	84.98	58.09	77.08	74.05	63.64
	Conditional	66.48	66.77	26.36	93.98	72.62	60.00

TABLE V. COVERAGE RESULTS FOR REVISED BIASED TEST SUITE

Device		Full SoC	CPU	Memories	DMA	UART	PIO
Final Coverage % (Biased)	Line	91.43	98.03	69.03	100	98.15	98.26
	Toggle	80.79	86.39	63.70	82.37	74.43	83.12
	Conditional	69.35	69.72	29.09	94.98	72.62	83.33

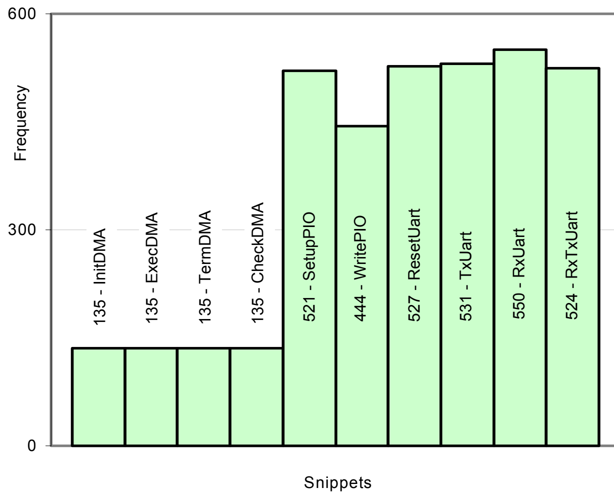

Figure 7. Snippet Statistics

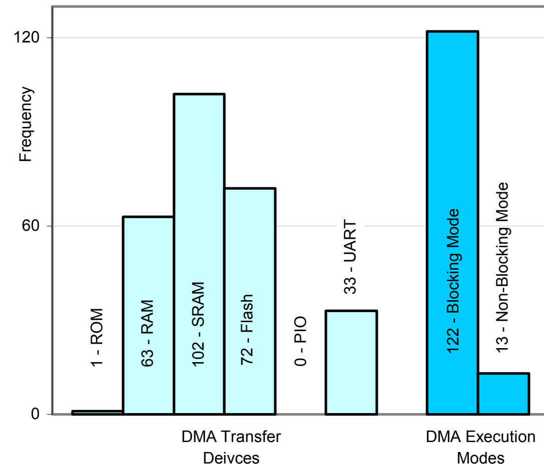

Figure 8. DMA Parameter Statistics

Based on our analysis, a revised test suite is created by the test generator according to biases in Table VI. For the DMA parameters, UART, PIO and ROM devices are selected more often as transfer devices; and we increase the number of DMA interrupt executions. DMA and WritePIO snippets are also weighted with greater selection likelihood. DMA snippets are assigned a greater weight because the DMA exercises other source and destination transfer devices concurrently. The effect of these bias recommendations is an increase in coverage for most of the on-chip devices and overall SoC (Table V).

TABLE VI. RECOMMENDED BIASES

Snippets	Bias	DMA Parameters		Bias
InitDMA	3	Transfer Devices	ROM	2
ExecDMA	3		RAM	1
TermDMA	3		SRAM	1
CheckDMA	3		Flash	1
SetupPIO	1		PIO	3
WritePIO	2		UART	3
ResetUart	1	Execution Modes	Blocking	1
TxUart	1		Interrupt	2
RxUart	1	(*A weight bias greater than 1 indicates greater selection likelihood*)		
RxTxUart	1			

This case study demonstrates SALVEM to be a feasible feedback verification technique. The initial coverage and test feedback iteration provided a small but significant increase in coverage. Additional coverage feedback iterations will enhance verification toward full coverage. In this paper, we have chosen to focus on the DMA and PIO device. However, similar analysis and biasing feedback can be applied to the UART and memory devices. Additional memory specific snippets are also required to improve overall SoC coverage. Furthermore, during SALVEM tests execution, a number of design bugs were uncovered in the Nios SoC. For example, vector port width mismatches and inconsistent interrupt masking or priority behaviors.

VII. CONCLUSION

Verification of SoC designs is a significant bottleneck for many design projects. This paper presented a new approach to tackle SoC verification. Software code fragments (snippets) are extracted from application use-cases to test specific SoC tasks. A test generator automatically composes snippets into software test programs.

The SALVEM technique was applied to the Altera Nios SoC. Experiments demonstrated the verification effectiveness of the SALVEM test programs. SALVEM can be applied as a feedback verification system. A case study was conducted using coverage and test information to manually drive test generation and improve coverage.

Our work so far has established SALVEM as a feasible and promising SoC verification method. In the future, we will explore automated coverage feedback techniques in SALVEM. Alternative snippet test composition techniques will also be investigated.

REFERENCES

[1] Collett International Research, "2002 IC/ASIC Functional Verification Study, " 2002.

[2] A. Cheng, A. Parashkevov, and C.C. Lim, "Coverage measurement for software application level verification using symbolic trajectory evaluation techniques," in *2nd IEEE International Workshop on Electronic Design, Test & Applications (DELTA2004)*. Perth, Australia: IEEE Computer Society, 2004, pp. 237-242.

[3] L. Semeria and A. Chosh, “Methodology for hardware/software co-verification in C/C++,” in *IEEE International High Level Design Validation and Test Workshop (HLDVT’99)*. San Diego, 1999, pp. 67-72.

[4] H. Hubert, “A survey of HW/SW cosimulation techniques and tools,” Internal Report, Vetenskap Och Konst Royal Institute of Technology. Stockholm, Sweden, 1998, 48 p.

[5] A. Chandra, V. Iyengar, R. Jawalekar, M. Mullen, I. Nair, and B. Rosen, "Architectural verification of processors ssing symbolic instruction graphs," in *IEEE International Conference on Computer Design*, 1994, pp. 454-459.

[6] 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, pp. 188-200.

[7] A. Hosseini, D. Mavroidis, and P. Konas, "Code generation and analysis for the functional verification," in *33rd Annual Conference on Design Automation (DAC'96)*. Las Vegas, Neveda, USA, 1996, pp. 305-310.

[8] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," in *International Test Conference*. Washington, DC, USA, 1998, pp. 990-999.

[9] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov, "Industrial experience with test generation languages for processor verification," in *41st Annual Conference on Design Automation (DAC'04)*. San Diego, California, USA, 2004, pp. 36-40.

[10] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Evolutionary test program induction for microprocessor design verification," in *11th Asian Test Symposium*, 2002, pp. 368-373.

[11] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Fully automatic test program generation for microprocessor cores," in *Design, Automation and Test in Europe (DATE2003)*. Munich, Germany, 2003, pp. 1006-1011.

[12] Altera Inc. "Nios Hardware Development Tutorial," ver 1.0, 2003, 54 p.