On-Chip Constrained Random Stimuli Generation for Post-Silicon Validation Using Compact Masks

Xiaobing Shi and Nicola Nicolici
Department of Electrical and Computer Engineering
McMaster University, Hamilton, ON, Canada
Email: shix6@mcmaster.ca, nicola@ece.mcmaster.ca

***Abstract*—During post-silicon validation a large number of constrained random stimuli are applied to expose the subtle design errors that have escaped to the silicon prototypes. In this paper we present a new method to design constrained random stimuli generators, which are programmable and can be placed on-chip to generate extensive random, yet functionally-compliant, sequences for real-time/in-system validation. The basic idea is to translate the constraints for constrained-random variables into binary cubes, whose specified values are used as masks to correct random sequences. To reduce the volume of data needed to be placed on-chip, the cubes are efficiently encoded and expanded in real-time. Experimental results confirm the effectiveness of this new method when compared against the prior work on the topic.**

I. Introduction

Ensuring the quality of integrated circuits is critical throughout the implementation cycle. Pre-silicon verification is commonly employed to ensure the consistency between the design and its specification. Before tape-out what can be measured is limited by the simulation time and accuracy, and designs are released for manufacturing when the confidence level is deemed sufficient. Manufacturing test is focused on screening for physical defects in each fabricated device; considering that its reference is the design implementation, manufacturing test is not concerned with finding and identifying subtle design errors (or bugs) that have escaped to silicon prototypes. Thus the verification tasks employed during the pre-silicon phase continue on these early silicon prototypes, a term commonly referred to as post-silicon validation (PSV).

A. Background and Related Work

Both simulation-based verification and formal verification are used to detect and fix implementation errors before committing to fabrication of a silicon prototype. The simulation-based methods find design errors based on a large set of test (or use) cases. Nonetheless, simulation is known to be slow; for example, the study on a commercial microprocessor [7] argues that it may take weeks of simulation of test cases that will take merely seconds to minutes of real-time execution. Therefore, simulation metrics (e.g., code/assertion coverage) are used as stop signs to balance the verification quality against the time to tape-out. Formal verification's inherent limitation in modelling the whole design confines its applicability within focused units on small scales. Furthermore, when accounting for unique electrical states, such as the ones caused by process variations or effects exercised only under certain process-voltage-temperature corners, it becomes more difficult to develop both accurate and scalable pre-silicon verification methods [19]. Consequently, to compensate for this insufficiency of pre-silicon verification methods, PSV, which is performed on silicon prototypes, is a critical step for finding design errors before committing to high-volume manufacturing.

Key challenges in PSV are error detection and reproduction, as well as root causing [12]. It is necessary to generate both proper stimuli and to record sufficient failing data for error analysis. Considering the unique constraints on controllability and observability during PSV, many approaches have been explored to bridge the gap between pre-silicon to post-silicon validation [20]. Also, reusing design-for-test (DFT) structures, including wrapper registers [1] around the design under validation (DUV) or scan chains [28] across the DUV, has been explored. Nonetheless, scan chains are insufficient because the scan dumps do not provide a history of events of interest that lead to the corrupted state [21]. Thus, some PSV-specific structures, such as on-chip trigger units [16] or hardware assertion-checkers [8], are employed to reduce the latency from error excitation to its detection. They are commonly used together with trace memories [4], [15] or footprint recorders [22], which can track a subset of relevant signals over a window that lead to the failure detection.

A large volume of *random, yet functionally-compliant, sequences* are needed for exposing the design errors, which have escaped to the silicon prototypes [2], [19]–[21], [24]. During pre-silicon verification, the constrained random number generator embedded in the simulator generates stimuli that satisfy user-defined constraints [25]. Considering that transmitting the constrained-random stimuli from simulation environments to the silicon prototype is obviously impractical due to bandwidth limitations, one has to consider how to generate a large volume of randomized functional sequences in real-time. Using instruction-level templates [2], [24], the in-system constrained-random stimuli generation can produce instruction sequences similar to the ones during simulation. Although such types of methods are useful for microprocessor-centric designs, they are limited for high-speed peripherals and hardware accelerators (e.g., video or graphics). Therefore, for logic blocks and data channels not easily accessible and not controlled by programmable embedded microprocessors, on-chip constrained-random stimuli generator (CRSG) structures can be employed to generate at-speed functionally-compliant stimuli for the silicon prototypes. This type of stimuli is subjected to constraints, consistent with the specification and format of data packets fed to the DUV in an application environment. While there have been custom implementations of such CRSG structures (e.g., [32]), a systematic way of designing them is an active area of research [21].

978-1-4799-4722-5/14/$31.00 ©2014 IEEE

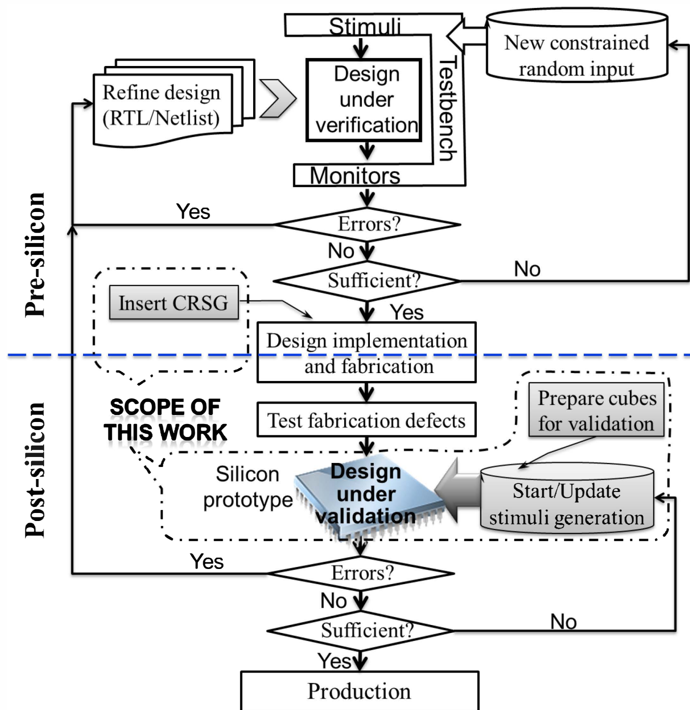


Fig. 1. The scope of this work during the implementation cycle.

B. Motivation and Contribution

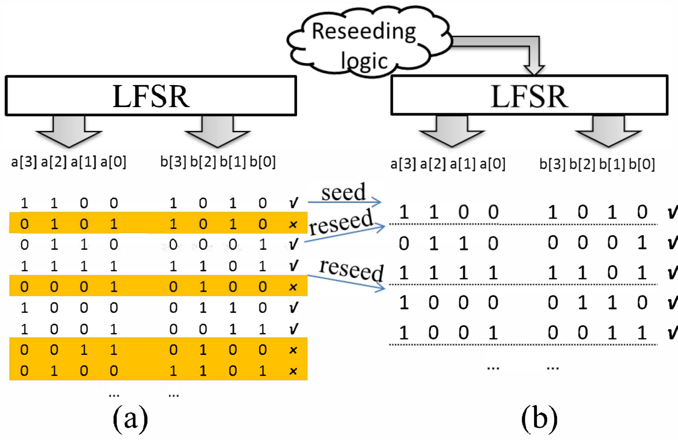
The scope of this work is highlighted in Fig. 1. Pre-silicon verification is concerned with identifying and fixing design errors and therefore the register transfer-level (RTL) description is iteratively refined. Once the confidence level is sufficiently high, the implemented design is sent for fabrication and, after screening for manufacturing defects, the design is validated on a system platform. It is common that in this phase subtle design errors (which affect every single fabricated device) are uncovered. In order to exercise as many use-cases as possible, randomized, yet functionally-compliant, sequences are applied to the DUV. The main advantage of post-silicon application of such randomized/functional tests is the huge volume of clock cycles; considering that in a few seconds to minutes of real-time execution more stimuli are applied to the DUV than during the entire pre-silicon phase, a few hours (or possibly days) of validation can uncover (or help increase the confidence of the lack of) design errors. In order to facilitate controllable experiments of such magnitude, in this work we propose a methodology for the design-time development and insertion of CRSG blocks and a flow for iterative run-time configuration of these blocks (in Fig. 1 the corresponding boxes are in shades of gray and the steps will be elaborated in the following sections). A key advantage over relying solely on stimuli from the native environment, is the ability to control experiments and bias the constrained-random sequences as the validation progresses in a user-programmable manner, as during the pre-silicon phase. It should also be noted that it is unlikely that one would have golden responses and therefore the response checking is done in a similar manner as in pre-silicon, i.e., verifying whether properties/assertions are violated. The contributions from this communication are not concerned with response checking and therefore the interested reader is referred to known methods for implementing assertions into hardware (e.g., [8]).

Some well-understood logic blocks can be employed at the core of CRSGs. The k-bit maximum-length Linear Feedback Shift Register (LFSR) generates $2^k - 1$ patterns if the characteristic polynomial is primitive and irredundant [5]. The use LFSRs for compressed deterministic test has been introduced in [17] and this concept of reseeding LFSRs has been refined and widely adopted in practice during the subsequent decade [6], [23], [31]. Also, many variants of LFSRs, e.g., de Bruijn counter, weighted pattern generator, phase shifter and cellular automaton [29], have been proposed to control the pseudo-random stimuli distribution. Furthermore, there are known methods to alter pseudo-random sequences for manufacturing test (e.g., [9], [26]). Nevertheless, none the above-mentioned methods have been tuned to force all the pseudo-random stimuli to the unique functional constraints defined in pre-silicon verification environments.

The unique requirements of PSV environments motivate our work. Based on a deterministic set of faults, the goal of manufacturing test is to obtain high fault coverage with a few test patterns in short time. It is sufficient if any LFSR output is altered to target a random-pattern resistant fault, which has not been detected up that point. However, PSV is aimed at generating a large volume of valid (functionally-compliant) random stimuli to reveal unforeseen design errors that have escaped to the silicon prototype. Therefore, the duration of PSV experiments on silicon prototypes may last minutes to hours (or even days), unlike testing for fabrication defects in each circuit instance, which is on the order of seconds to tens of seconds. Due to these fundamental differences, all the stimuli applied to the design under validation must satisfy the functional constraints. Besides, the PSV experiments might need to change as the validation process progresses, which mandates in-system programmability of new constraints that have to be satisfied by the randomized stimuli.

A method presented in [13], [14] has tackled the challenge to generate functionally-constrained pseudo-random sequences by removing the noncompliant stimuli by reseeding LFSRs. Consider the case of generating stimuli containing two 4-bit signals a and b, the valid stimuli are constrained as follows: $a \geq b$. As shown in Fig. 2(a), the unconstrained LFSR generates a sequence of random stimuli, among which only some stimuli are valid. Hence the reseeding logic is added to control the state of LFSR as shown in Fig. 2(b). Before the LFSR generates an invalid stimulus, the pre-computed seed would be loaded into the LFSR, hence skipping the invalid subsequence. The preparation for the seeds requires solving system sate equations. The solvability and the frequency of reseeding depend on the LFSR configuration and constraints.

Motivated by the need to reduce the amount of data that is stored on-chip for programmable CRSGs, our contribution from this paper is a new CRSG (and its design method) comprising a hardware random generator and correction logic. It eliminates the need for solving system equations and continuously reseeding LFSR with a set of seeds, as done in prior works [13], [14]. Rather, the CRSG performs real-time on-chip correction for each invalid stimulus at the output of the random generator. The user-defined constraints are translated into a set of equivalent cubes. Fig. 2(c) shows a



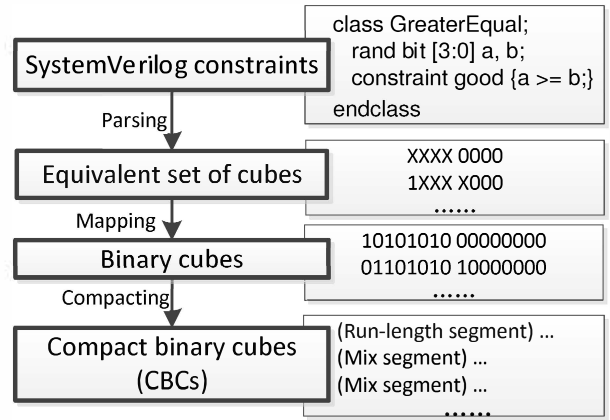
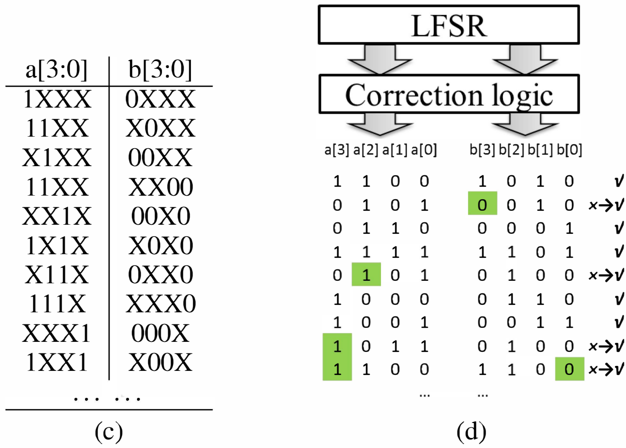


Fig. 3. The flow in the pre-processing phase.



a[3:0]	b[3:0]
1XXX	0XXX
11XX	X0XX
X1XX	00XX
11XX	XX00
XX1X	00X0
1X1X	X0X0
X11X	0XX0
111X	XXX0
XXX1	000X
1XX1	X00X
...	...

Fig. 2. Constrained-random stimuli generation by employing configuration/transformation logic circuitry around an LFSR. In order to force the output from the LFSR in (a) to satisfy the constraint $a \geq b$, the logic in (b) changes the state of LFSR with the new seed whenever an unsatisfied output would be generated. Alternatively, the logic in (d) corrects the LFSR patterns to match the specified bits from at least one cube listed in (c). For example, the original output "0101 1010" in (a) is corrected to "0101 0010" in (d) based on the cube "X1XX 00XX".

typical set of equivalent cubes for the constraint $a \geq b$. For example, the cube "1XXX 0XXX" means the output is valid as long as the most significant bits of a and b are 1 and 0 respectively. The CRSG uses these cubes to mask the invalid stimuli at the output of the LFSR, as conceptually illustrated in Fig. 2(d). The CRSG is in-system programmable and the user can apply different randomized sequences with distinct user-defined constraints by updating the configuration needed for the correction cubes. The basic idea is to let the LFSR run autonomously and the outputs from LFSR are masked by the correction logic. One cube can imply a large number of valid stimuli that satisfy the user-defined functional constraints. Since the original cube masks may still require a large volume of data, they need to be efficiently encoded before being stored on chip in a compact manner; subsequently these compact masks need to be decoded in real-time before being used by the correction circuitry at the output of the LFSR. The method proposed in this paper achieves this goal without the need for a significant investment in on-chip logic or memory resources.

The overview of our method for designing a new type of on-chip CRSG is presented in Section II and Section III provides the specific details of the pre-processing phase. The hardware implementation is given in Section IV. The performance and area cost are evaluated in Section V, followed by the conclusion in Section VI.

II. Method Overview

During the pre-silicon verification phase, a series of test cases can be created automatically using constrained-random tests. The constraints are designed and formalized to SystemVerilog expressions [11], based on the application-specific functionality and verification requirements. The constraint expressions are parsed and handled by the pseudo-random number generator, which then generates valid stimuli. The proposed method reuses the constraints expressions written in SystemVerilog, so as to preserve the validity and effectiveness of pre-silicon verification stimuli.

Our method operates both at design-time and at run-time (or validation-time). At design-time, the configuration of the CRSG hardware should be selected, including, for example, the capacity of the on-chip memory or the size of the LFSR. This step is illustrated by the "Insert CRSG" box in Fig. 1. At run-time, the user is given the freedom to change the configuration of the CRSG, in order to apply functionally-compliant sequences with different (user-programmable) constraints. These constraints for the stimuli are captured in SystemVerilog (i.e., the same language used during the pre-silicon verification) and can be updated iteratively based on the specific debugging needs as the validation process evolves. The constraints are parsed into compact binary cubes (CBCs) as illustrated by the "Prepare cubes for validation" box in Fig. 1 and elaborated in Section III. Then the cubes are loaded into the on-chip memory and activated for stimuli correction thus facilitating continuous functionally-compliant random stimuli generation, which is illustrated in box "Start/Update stimuli generation" box in Fig. 1 and elaborated in Section IV.

The data flow in the pre-processing phase is shown in Fig. 3. The SystemVerilog constraints are parsed into an equivalent set of cubes during the pre-processing phase, which is similar to the cubes shown in Fig. 2(c). The set of cubes covers exactly all the possible valid stimuli which satisfy

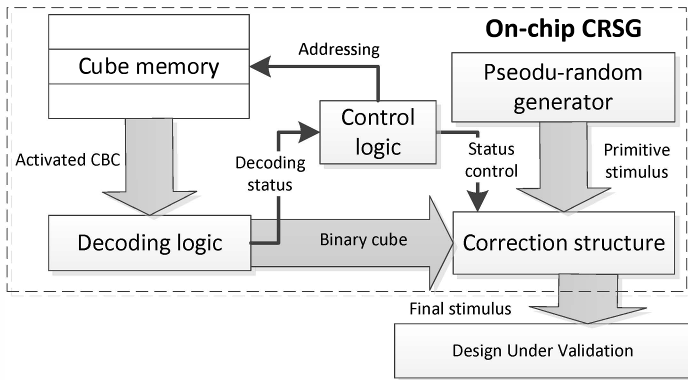


Fig. 4. The top-level architecture for the on-chip CRSG.

the user-specified constraints. Then the cubes are mapped into binary cubes according to the mapping dictionary and then encoded into a certain format of CBCs by the encoding algorithm, for the ease of being stored in the on-chip memory efficiently. The specific details of the mapping and encoding algorithms will be provided in the following section.

The top level architecture for the on-chip hardware is shown in Fig. 4. The control unit controls four main functional parts: an on-chip cube memory, CBCs decoding logic, the pseudo-random generator based on a maximum-length LFSR, and a correction structure comprising multiplexers and register chains. The CBCs are stored into the cube memory preceding on-chip stimuli generation, whose address is issued from the control unit. The pseudo-random generator can generate one primitive stimulus per clock cycle. Meanwhile, one CBC is activated each time, which is first fetched from the cube memory and then decoded into a binary cube. Based on the activated cube, the correction logic simultaneously checks and modifies the primitive stimulus from the pseudo-random generator into a valid stimulus. Hence, the CRSG can continuously generate valid stimuli to the DUV cycle by cycle.

III. Content Preparation for the On-Chip CRSG

A. *Customized SystemVerilog Parser*

We use SystemVerilog for the specification of constraints because it has standardized support for constraint definitions and it is widely used in practice. SystemVerilog offers flexible syntax and libraries to create constrained-random stimuli by user-defined constraints. The syntax support for constraint classes can be broadly classified into constraints on variable values (e.g., arithmetic and logical expressions, or if-else and implication relations), constraints on distributions of patterns, and constraints on the solving order during simulation. Only constraints on variable values are supported by our current methodology for designing and configuring hardware circuity for in-system generation of constrained-random stimuli. In addition, our support for constraint randomization does not include sequential behaviour. Our future work will investigate how to support other features from pre-silicon testbenches, such as controlling the distribution of randomized variables or capturing the sequential relationships between randomized variables, as an extension of the methodology presented in this communication.

Fig. 5(a) shows an example of constraints including logical and conditional expressions, which are encapsulated in a class block. A customized SystemVerilog parser is designed to parse the constraint sources into a set of cubes, which will constrain the patterns in the same way as the source constraints. As shown in Fig. 5(b), each cube in the set covers a subspace of valid stimuli. A logic minimization tool, Espresso [18], is used to reduce the cardinality of the set of cubes, by removing and merging the cubes which are not essential (i.e., the ones that cover uniquely at least one valid pattern). The cardinality of the minimized set of cubes is specific to each constraint and it depends primarily on the distance between valid stimuli within the Boolean space.

Presently our method supports SystemVerilog expressions for constraints only on random variable values. If the constraint expressions can be evaluated to be true or false based on their numerical values, then they can be accepted by the parser to generate content for CRSGs. The method supports arithmetic operators (`+, -, *, /`), shift operators (arithmetic/logic shift, left/right shift), logic operators (`&, |, ^, !, ~`), relational operators (`>, >=, <, <=, ==`), and set membership operator (‘inside’ and its negated form). The implication constraints using if-else statements or the implication operator (`->`) based on randomized variables are supported in constraint expressions. The ‘foreach’ iterative constraints can be supported, so long as each constraint in the unrolled iteration is also supported. The array reduction iterative constraints for randomized variables are processed as a single constraint over all the elements in the array. However, randomizing the dimension of the array is not supported by our method because the number of array elements must be known at design time for the current hardware implementation.

In pre-silicon verification environments, the constraint expressions can be written as class blocks or as in-line constraints and they can be enabled/disabled seamlessly in a testbench; in hardware they can also be enabled/disabled via in-system/online reprogramming of the CRSG, however it does involve human intervention because the new content for the CRSG needs to be regenerated and uploaded into the on-chip memory. Both the content and the size of equivalent set are independent of the hardware architecture. For a cube including q ‘X’s, which

```
typedef enum {ADD, SUB, SHIFT_L,SHIFT_AR,SHIFT_LR} op_type;
class StimuliForALU;
  rand op_type opcode;   rand bit[7:0] opr1, opr2;
  constraint opr_range {
    (opcode==SHIFT_L || opcode==SHIFT_AR || opcode==SHIFT_LR)
      -> opr2 inside {[0:7]};
  }
endclass
```

(a) User-defined constraints written in SystemVerilog

opcode[2:0]	opr1[7:0]	opr2[7:0]
00X	XXXXXXXX	XXXXXXXX
01X	XXXXXXXX	00000XXX
100	XXXXXXXX	00000XXX

(b) The equivalent set of cubes contains three cube strings after minimization

Fig. 5. The SystemVerilog constraints are parsed into the equivalent set of cubes.

TABLE I. THE DICTIONARY FOR MAPPING CUBE STRINGS INTO BINARY CUBES.

Character in a cube string	2-bit mapped binary code
'0'	00
'1'	01
'X'	10

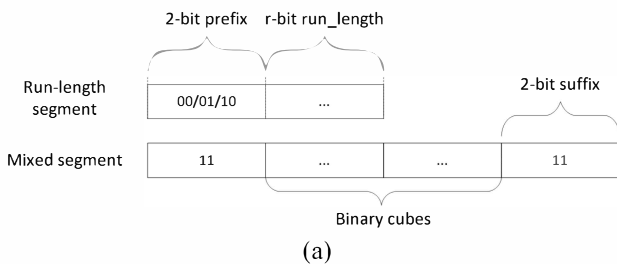
denotes q free bits, the number of valid stimuli implied by the cube is 2^q. For example, the 19-bit cube 01X XXXXXXXX 00000XXX from Fig. 5(b) covers a total of 4096 valid stimuli. The complete set of cubes implies all the possible valid stimuli that satisfy the user-defined constraints.

B. Mapping and encoding algorithms

The cubes parsed from the SystemVerilog constraints are mapped into binary cubes based on a mapping dictionary and further encoded into a more elaborated CBC format, which does not only take advantage of the available embedded memory space with higher efficiency, but it also simplifies the correction logic.

A dictionary encoding algorithm is used to encode the cube strings into binary cubes. As shown in TABLE I, the only three valid characters in cubes, i.e. '0', '1' and 'X' occupy three 2-bit binary code points, leaving the code point 11 reserved for compaction purposes. The code point for 'X' prefixed with 1, which is different from the other two characters, can simplify the logic of the correction structure, which will be elaborated in the following section. For instance, the 19-character long cube string "01X XXXXXXXX 00000XXX" is mapped into a 19-code long binary cube in 38 bits "000110 1010101010101010 0000000000101010".

Some cubes might include consecutive-'X', consecutive-'0' and consecutive-'1' sequences. The cause of this can be explained by practical requirements of verification, e.g., if a variable is not constrained, all the q bits in the variable are filled with consecutive 'X's in the cube. Likewise, the consecutive-'0' can be used for resetting a variable under some user-defined conditions. Based on this observation, our algorithm combines prefix encoding with run-length encoding in order to compact the binary cubes into CBCs. It first partitions a cube into pieces of segments. There are two types of segments: the run-length segment and the mixed segment, as illustrated in Fig. 6(a). If the count in a consecutive sequence goes beyond a threshold, it is partitioned as a run-length segment, with a 2-bit binary code prefix denoting the consecutive character. Otherwise the sequence between two run-length segments is partitioned as a mixed segment, which is filled with the original binary codes and edged by a 2-bit prefix and a 2-bit suffix equal to the 11 binary code. The threshold depends on the length of run_length field, so as to make a consecutive sequence compacted shorter by being partitioned as a run-length segment rather than as a mixed segment. The converted CBCs from Fig. 5(b) are shown in Fig. 6(b), in which case the threshold is set to 2.



(a)

Cube string	CBC (r=6)
00X XXXXXXXX XXXXXXXX	11000011,10010001
01X XXXXXXXX 00000XXX	11000111,10001001, 00000101,10000011
100 XXXXXXXX 00000XXX	1101000011,10001000, 00000101,10000011

(b)

Fig. 6. The format and examples for the run-length segment and the mixed segment.

The compaction rate varies with the regularity of constraints and the format setting. Provided the length of run_length field is r bits, a binary cube can be compacted into a CBC down to the size of $1/2^{r+1}$ of the original binary cube. On the other hand, for the worst case when the binary cube is a single mixed segment, the overhead is a constant nibble, i.e. a 2-bit prefix and a 2-bit suffix.

IV. ON-CHIP CRSG ARCHITECTURE AND FUNCTIONALLY-COMPLIANT STIMULI APPLICATION

The proposed CRSG consists of the control unit, the on-chip cube memory, the decoding logic, the pseudo-random generator and the correction structure. It supports the workflow from storing and decoding CBCs, primitive random stimuli generation and correction, to the final stimuli output. Both the throughput and memory logic can be flexibly adapted to the validation environment.

A. Cube memory

The on-chip cube memory stores the CBCs. When the control unit activates a CBC, its initial address is sent to the cube memory. The fetched CBC is loaded into a buffer for decoding. Considering the issue of word alignment for the memory, each CBC starts with a new address, so that no in-word offset information is needed for activating a new CBC.

If the stimuli for the DUV were transmitted directly from a host or an on-board memory, i.e., without compaction and on-chip buffering, then each stimulus would be used once and then discarded. Such mechanism requires new stimuli frequently and its main limitation is the need for very high-bandwidth interfaces. By contrast, by employing an on-chip embedded memory for buffering CBCs, the proposed CRSG architecture receives compact cubes, which are then expanded on-the-fly to correct LFSR patterns during the stimuli application. Each CBC can remain activated for an arbitrary number of cycles to constrain the pseudo-random generator to generate a user-controlled amount of valid stimuli. Because both the number and the size of CBCs are much smaller than the expanded stimuli, it alleviates the need for high-bandwidth interfaces. The cube memory can be implemented either as a FIFO addressed by implicit increment, or a dual-port RAM. In the case the volume of CBCs is very large, the capacity of the embedded memory can be lowered by buffering only a subset

of CBCs, which will be used during a limited time window for stimuli application; the subsequent subset of CBCs can be uploaded concurrently with the application of the stimuli expanded from the current subset of CBCs. Finally, it should be noted that the capacity of the cube memory is not influenced by the circuit size, since it is determined only by the number and dimension of cubes, which are influenced by the type of constraints and the size of randomized packets.

Fig. 7 shows the data flow for activating and updating CBCs based on the cube memory built with dual-port RAM. One port is used for fetching the activated CBC via r_addr and r_data. The other port including w_data and w_addr is reserved for updating a new CBC when it is ready from the control unit. The bitstream transmission via a low-bandwidth interface takes less pin resources, while the control unit reconstructs the CBC and sends word by word to the cube memory. The recently sent CBC is updated and after being activated and used, the CBC can be set to be outdated and can be overwritten by a new CBC. The addressing control unit issues the CBC address for update and activation independently. The control unit keeps track of the activated address and the update address where the used CBC can be overwritten by a new CBC transmitted from the host.

B. Decoding logic

The decoding logic consists of multiple byte-wise decoders to support parallel decoding and a $2n$-bit buffer to store the decoded n-code binary cube, as shown in Fig. 7. It supports to decode p binary codes (each binary code has 2 bits) per clock cycle, where p denotes the degree of parallelism. Each combinational byte-wise decoder determines the segment type by the 2-bit prefix (if the type is not inherited from the previous byte). The byte is interpreted into 2 to 4 codes as a mixed segment or 2 to 2^r codes as a run-length segment. In each clock cycle, the first p codes from parallel decoders are shifted into the binary codes buffer, leaving the remainders for the following cycles. Thus decoding a CBC of n codes requires $\lceil n/p \rceil$ cycles. For example, a 168-code binary cube (as used in our experiments detailed in the next section) is decoded from the CBC format in 21 cycles if p is 8, or 11 cycles if p is 16. The parallelism facilitates rapid continuous cube switching.

C. Pseudo-random generator and correction structure

The pseudo-random generator consists of a k-bit maximum-length LFSR and a k-to-m phase shifter ($k \leq m \leq n$), as shown in Fig. 7. The period of the LFSR is $2^k - 1$. The phase shifter is combinational XOR gate logic, which expands each k-bit output from the LFSR to m-bit primitive stimulus.

The correction structure consists of a $2n$-bit shadow register and m bitwise multiplexors, as shown in Fig. 7. Each two bits in the shadow register are paired with a bitwise multiplexor. The shadow register pipelines n decoded binary codes from the decoding logic, which avoids stalling stimulus correction during cube switching. A virtual $2m$-bit window is created and rolled in the $2n$-bit shadow register, which indicates the m activated binary codes for the current cycle. Each multiplexor decodes a 2-bit binary code in the virtual window and arbitrates whether to output the corresponding bit from the pseudo-random generator or to correct it to a constant

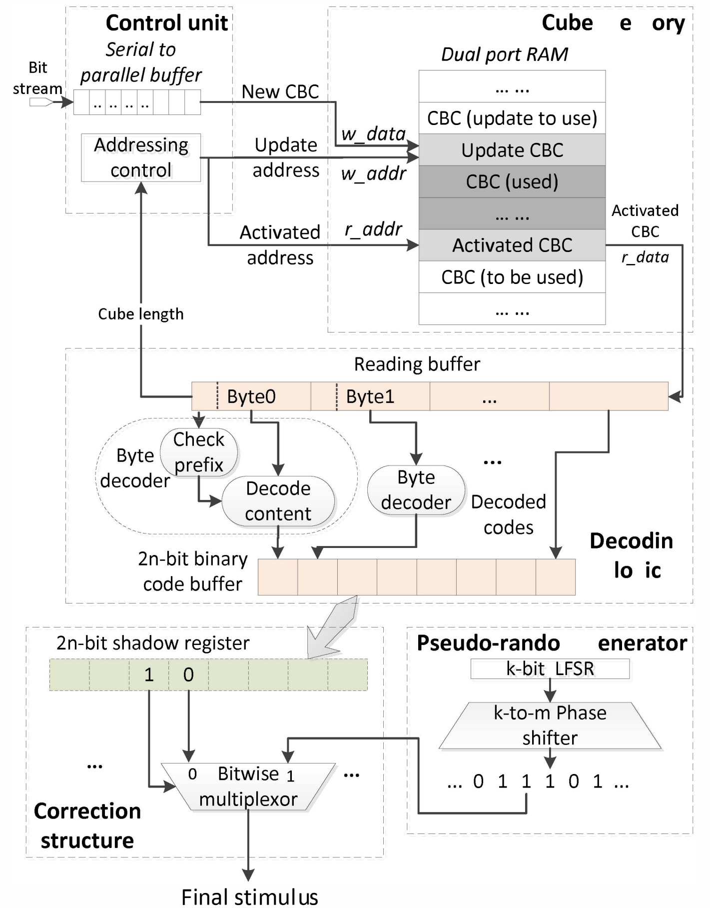


Fig. 7. The dataflow from the primitive stimulus in the pseudo-random generator to the final stimulus corrected by the correction structure which gets the activated binary cube from the cube memory and the decoding logic.

0 or 1. Based on the encoding dictionary, the left bit in the binary code can directly serve as the selection signal and the right bit is the constant output when it is corrected.

The distribution of the stimuli generated by the CRSG relies on the uniform distribution associated with LFSRs based on primitive characteristic polynomials, however it also dependent on the position of 0s, 1s and Xs in each cube; in addition, the distribution is also biased by the state of the LFSR when a particular CBC is activated. Taking a 4-bit LFSR as an example, if the LFSR has two adjacent states '1100' and '1001', i.e., the LFSR shifts left by one position and it feeds '1' at the rightmost position, an unconstrained CRSG (i.e., the activated cube is 'XXXX') will output the two binary values as two consecutive stimuli; hence the distribution of samples at the output of the LFSR will contain each of the samples '1100' and '1001' exactly once. If the activated cube is 'XX10', the two generated stimuli are '1110' and '1010', thus each of them will again count once in the distribution. However, if the activated cube is 'X1X0', both stimuli from the output of the LFSR will be corrected to '1100' and hence this particular sample will be accounted for twice in the distribution. The impact of the correction logic on the sample distribution is experimentally assessed in section V.

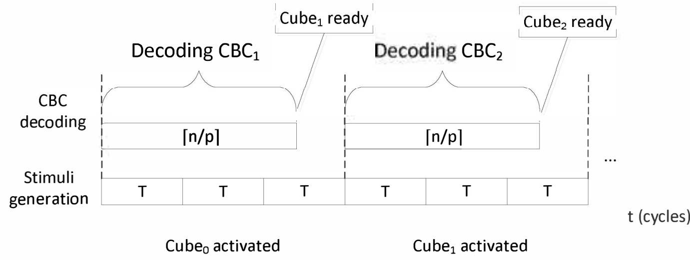


Fig. 8. Timeline for CBCs decoding and switching during stimuli generation.

D. The control unit

The control unit keeps tracks of addresses for the cube memory, as shown in Fig. 7. It uses the cube memory as a circular queue. Both the activated address and the update address move down to the following CBC position until the end of the memory, or reset to the initial address. In order to compute the address for the next activated CBC, the control unit receives the length of currently activated CBC for the decoding logic, which is added to the current activated address. The activated CBC is fetched from the cube memory and decoded into a binary cube by the byte decoders. Then it is copied to the shadow register in one cycle, based on which the multiplexors arbitrate each output bit between the pseudo-random bit and the lower bit in the mask code.

The control unit also synchronizes the functional parts, so that the architecture supports to generate an n-bit final stimulus (or packet) in a user-specified number of clock cycles (denoted as T). Therefore the stimulus is split into m-bit slices (m is equal to $\lceil n/T \rceil$), except the last slice if the remainder is not zero. As shown in Fig. 8, three packets are generated within $3T$ cycles based on Cube_0. Meanwhile CBC_1 is decoded into Cube_1 and will be activated after the third packet is completely generated. Generally, while an n-bit packet is generated in T cycles, the next CBC is being decoded and will be ready within $\lceil n/p \rceil$ cycles. Then it switches to be active immediately after the previous complete packet is generated. Fig. 8 illustrates the minimum cycle requirement for switching to a new cube, within which $\lceil n/pT \rceil$ packets must be generated based on the same cube.

V. Experimental results

In this section we examine the cost of the proposed CRSG and we assess its effectiveness. The proposed CRSG is compared against the known work on the same topic [14], which tackles exactly the same challenge of designing and applying user-programmable constrained-random sequences in real-time. We should note that the research presented in this paper is focused only on the controllability aspects of post-silicon validation (see Fig. 1). For dealing with the observability aspects, the interested reader is referred to on-line response checking using hardware assertions [8] or event detection using programmable trigger units [16], and real-time trace collection [4], [15].

We analyze the proposed CRSG by varying the length of the LFSR (denoted as k-bit) and the length of the final stimulus (denote as n-bit). Note, the length of the final stimuli should support the maximum size of random packets for the different blocks that are validated using the respective CRSG. For this experiment we assume that all the stimuli bits for the entire packet are applied in a single clock cycle ($T = 1$). The synthesis results based on the k-bit LFSR are shown in Fig. 9. Compared with the reference design [14], whose area is directly influenced by the size of the LFSR, the area cost of the proposed CRSG grows insignificantly with the length of the LFSR. Considering that the period of the LFSR-generated sequence has an exponential dependence on the dimension of the LFSR (assuming the characteristic polynomial of the LFSR is primitive and irredundant), the proposed CRSG architecture can employ large LFSRs to avoid the repetition of pseudo-random stimuli, which are corrected within the CRSG to be functionally-compliant, when very long validation times are needed; for example, even a 50-bit LFSR that works at 1 GHz can operate autonomously for over ten days. Nonetheless, due to the $2n$-bit binary code buffer and the $2n$-bit shadow register (see Fig. 7) and, unlike [14], the proposed CRSG is dependent on the size of the validation stimuli (packets) that are applied to the DUV. The synthesis results of the architecture according to different lengths of stimuli are illustrated in Fig. 10.

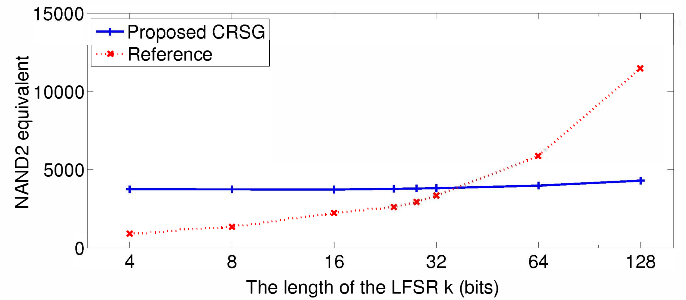


Fig. 9. Hardware cost of proposed CRSG and the design in [14] according to the length of LFSR k (given $T = 1$, $n = 16$ and $p = 8$).

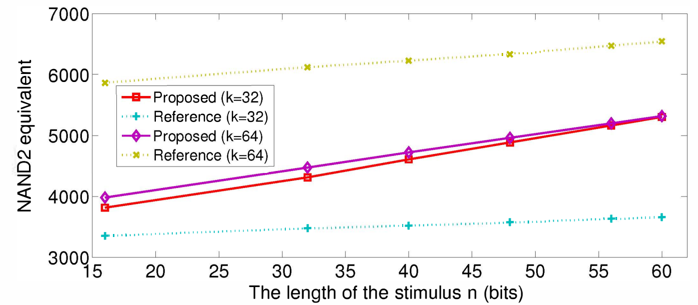


Fig. 10. Hardware cost of proposed CRSG and the design in [14] according to the length of the stimulus n (given $T = 1$ and $p = 8$).

As discussed in Section IV, an important parameter, which influences how fast the masks used for correction of pseudo-random sequences can be switched, is the degree of parallelism p. The synthesis results by varying p are illustrated in Fig. 11 and Fig. 12. As for the previous experiments, it is assumed that the stimuli are applied in a single clock cycle ($T = 1$). As expected, both the area and the critical path delay are affected by p, because each byte decoder must decide the segment

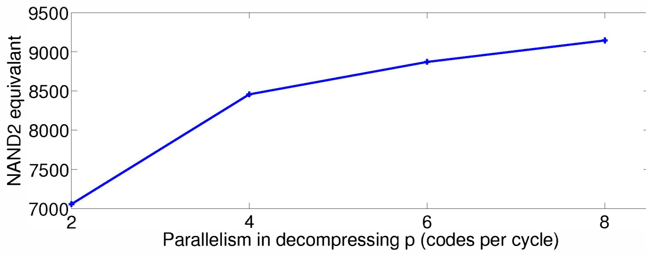


Fig. 11. Hardware cost according to the degree of parallelism in the decoding logic (given $T = 1$, $k = 168$ and $n = 168$).

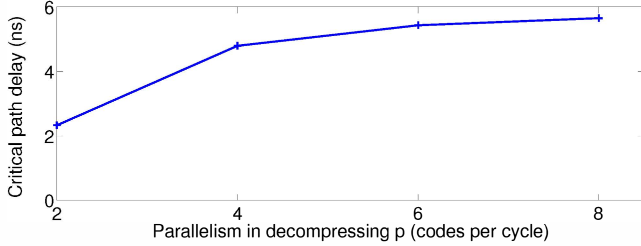
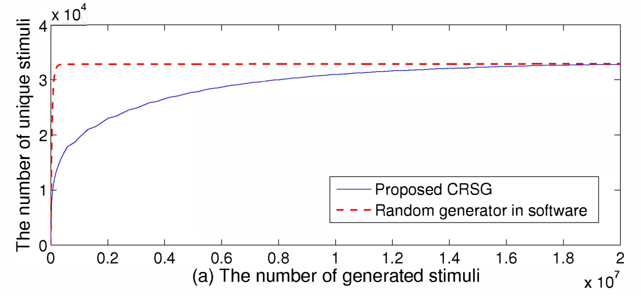


Fig. 12. Critical path delay according to the degree of parallelism in the decoding logic, estimated by static timing analysis with a CMOS 90nm standard cell library (given $T = 1$, $k = 168$ and $n = 168$).



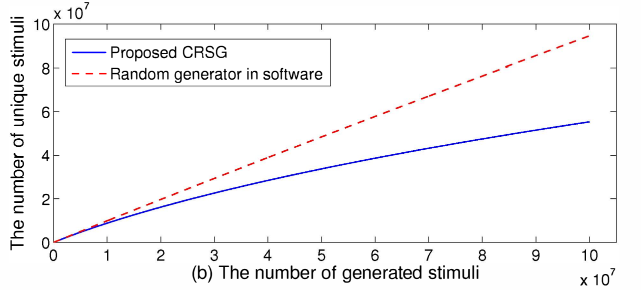


Fig. 13. The relation between the number of generated stimuli and the number of unique stimuli based on the constraint $a \geq b$. The unsigned variables a and b are set to 8 bits in (a) and 16 bits in (b) respectively.

type based on the previously decoded byte or the current byte prefix (as described in subsection IV-B). This parameter p is independent of the complexity of specified constraints, which might influence only the number of mask cubes that are compacted and stored in the on-chip memory. Nonetheless, the higher the degree of parallelism, the faster one can switch between these mask cubes.

Concerning the side-effects of the proposed CRSG architecture on timing, when one considers the whole view of on-chip functional units and interconnection logic, the delay paths in the CRSG are unlikely to dominate the circuit's operating frequency. What the CRSG architecture impacts is the timing delay from original function signal to the port of the DUV, which is now multiplexed between the original signal and the stimulus from CRSG. In the event that CRSG will impact the operating frequency, an optional n-bit pipeline register chain can be inserted between the output of the correction structure and the DUV.

Concerning the quality of the randomized stimuli generated by the CRSG, we first examine their distribution. Fig. 13(a) illustrates the relation between the number of stimuli generated and the number of unique stimuli based on a simple constraint $a \geq b$, in which a and b are unsigned 8-bit variables. If the random values are drawn from the uniform distribution then, until we exhaust the entire valid space (as defined by the constraint), the value on the Y axis should match the value on the X axis; thereafter, the value on the Y axis saturates to the maximum number of unique stimuli. The maximum number of valid 8-bit a and b pairs that satisfy $a \geq b$ is 32,896 and the software-based random generator in a System-Verilog compliant simulator [27] reaches this saturation point after 543,085 patterns; it should be noted that these results have been obtained using the "rand" type for the randomized variables, for which the random values are not drawn from the uniform distribution (the "randc" type needs to be used in order for the software random generator to sample from the uniform distribution). As the number of generated patterns increases, the number of unique patterns generated by the hardware method is approximately half of the ones generated in a software simulator (for the same number of total patterns). Fig. 13(b) shows this trend more clearly, where a and b are set to be 16 bits each. It should be noted that during post-silicon validation it is reasonable to have experiments with a significantly larger number of clock cycles than during pre-silicon verification; therefore, though the valid randomized stimuli are repeated more often in the hardware implementation, as confirmed by this experiment, the extensive number of clock cycles exercised on silicon prototypes (at least four orders of magnitude more than during pre-silicon simulation [10]) are expected to compensate for this repetition of constrained-random stimuli.

Concerning the impact of constraint complexity on the cube memory, we have performed the following experiment using integer linear programming (ILP) constraints because they can intuitively illustrate the numerical relationships between variables (and many arithmetic, relational and even some logic constraints can be converted to ILP forms). TABLE II(a) shows the trend of the size of the cube set when we incrementally impose four linear constraints on two 12-bit variables. If only the first constraint is used, the total number of valid pairs is 11,943,292 and the number of cubes is 5,724. If the second constraint is used (in addition to the first one), the number of cubes becomes 4,482 and so on for the third and the forth constraint where we have 3,398 and 2,120 cubes respectively. It demonstrates for this type of problems that the cube count tends to go down when more constraints are added, mainly

TABLE II. THE CHANGES TO THE SIZE OF THE CUBE SET WHEN ADDING CONSTRAINTS INCREMENTALLY.

(a) Constraints of ILP inequalities on 12-bit variables

Constraints	Valid pairs	Cubes
(1) $1000 \leq x + 2y \leq 8000$	11943292	5724
(2) $y \geq 5x - 6000$, and (1)	5243774	4482
(3) $x \geq 600$, and (1),(2)	3143474	3398
(4) $y \geq 2000$, and (1),(2),(3)	1582674	2120

(b) Constraints of integer non-linear programming inequalities on 8-bit variables

Constraints	Valid pairs	Cubes
(1) $a < c^3 + d < b$	3623738	14755
(2) $a^2 + bc > bcd + d^3$, and (1)	262442	2615
(3) $d \sin a \geq b \cos c$, and (1),(2)	69555	938

because the valid pairs count is reduced. We have also observed a similar trend when non-linear constraints are used, as shown in TABLE II(b).

In order to evaluate the effectiveness in reducing the storage requirements for large validation sequences, we have configured our CRSG to generate stimuli for resembling 168-bit packet heads for H.264 real-time transport protocol (RTP) [30], as well as 160-bit packet heads in the PCI-express (PCIe) 3.0 transaction layer packet (TLP) format [3]. Each field in the packet head must satisfy the requirements specified in the protocol standards, including the format, defined/reserved values and the coordination among fields. The fields that can be randomized are extracted for the design of constraints, thus leaving the non-random CRC field to be attached by CRC computation logic. A series of SystemVerilog constraints are designed to guide cube generation. The constrained bits involved in SystemVerilog constraints for the packet head vary according to the protocol specification. The results of parsing the constraints to cubes (the average number of constrained bits and the total number of cubes) are listed in TABLE III.

Only the cubes in the CBC format are required to be loaded to the cube memory; this requires a quarter to a half of the storage needed for the binary cubes. Compared to [14], which stores basis vectors from which LFSR seeds are expanded on-the-fly, the volume of data that is required by the proposed method is at least an order of magnitude less. This is because the number of basis vectors from [14] needed to satisfy a particular cube can be large and, more importantly, the dimension of each of these vectors is as large as the LFSR size. Hence, the savings of the proposed CRSG are explained by the fact that the storage requirements are not dependent on the LFSR size or the number of LFSR seeds that can expand into sequences that match the constraint provided by each cube. Considering that the total number of constrained random patterns that can be applied to the DUV using only one cube is defined by 2^q, where q is the number of unspecified bits (which can be computed by subtracting specified bits from cube length provided in TABLE III), it is evident that the number of stimuli that can be used for validation can easily meet the objectives of real-time execution that lasts for hours.

We should note also that in the event that the capacity of the on-chip memory is a tight implementation constraint (e.g., approximately 80 Kbytes for PCIe TLP might be excessive for some designs), one can update CBCs on-chip dynamically, as described in Section IV. The main reason why this dynamic

TABLE III. PACKET HEAD GENERATION RESULTS FOR PCIE AND H.264.

Packet format	Cube length (bit)	Constrained bits	Cube count	Binary cubes size (KBytes)	CBCs size (KBytes)
H.264 RTP	168	10	335	14.91	3.9
PCIe TLP	160	33	5119	204.76	81.8

update is feasible is because *any* CBC will decode into a valid mask that will ensure that the pseudo-random patterns at the output of the LFSR will be mapped onto functionally-compliant stimuli. This is a direct consequence of translating the SystemVerilog constraints into cubes, as described in Section III. For example, considering that 5,119 CBCs for PCIe TLP in TABLE III require 81.8 Kbytes, one can store approx 250 CBCs into a 4 Kbyte memory block; in such a memory-constrained environment, the CRSG can iterate through the masks expanded by these 250 CBCs, while a new subset of CBC is loaded through a low-bandwidth serial interface from on-board storage or directly from the host.

VI. CONCLUSION

In this paper, we have presented a new method for designing constrained-random stimuli generators for post-silicon validation. Unlike previous works [13], [14], which force autonomous random generators to skip the functionally non-compliant stimuli, our method corrects the output of a random generator to meet the user-specified constraints. The hardware cost is comparable to the previous works, while the volume of data that needs to be placed on-chip is reduced. This type of functionally-compliant random generators placed on-chip can be used for user-controlled random validation experiments on the silicon prototypes that might require extensive periods of time (hours to days) without costly storage requirements. The proposed method is applicable to any digital blocks and can leverage the constraints developed during pre-silicon verification.

Acknowledgement: The authors acknowledge the financial support of the University Research Office (URO) from Intel Corporation. They are also grateful for the feedback received from the technical program committee of the IEEE International Test Conference.

REFERENCES

[1] M. Abramovici. In-System Silicon Validation and Debug. *IEEE Design & Test of Computers*, pp. 216-223, 2008.

[2] A. Adir, S. Copty, S. Landa, A. Nahir, G Shurek, A. Ziv, C. Meissner, H. Schumann. A Unified Methodology for Pre-Silicon Verification and Post-Silicon Validation. in *Proc. IEEE/ACM Design, Automation & Test in Europe (DATE)*, pp. 1-6, 2011.

[3] J. Ajanovic. PCI Express 3.0 Overview. In *Hot Chips: A Symposium on High Performance Chips*, 2009.

[4] E. Anis and N. Nicolici. On Using Lossless Compression of Debug Data in Embedded Logic Analysis. In *Proc. IEEE International Test Conference (ITC)*, 2007.

[5] P. H. Bardell, W. H. McAnney, J. Savir. Built-In Test for VLSI: Pseudorandom Techniques. *John Wiley & Sons*, 1987.

[6] C. Barnhart, V. Brunkhorst, F. Distler, O. Farnsworth, B. Keller, and B. Koenemann. OPMISR: The Foundation for Compressed ATPG Vectors. In *Proc. IEEE International Test Conference (ITC)*, pages 748–757, 2001.

[7] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 244-248, 2001.

[8] M. Boule, J.-S. Chenard, X. Zilic. Debug Enhancements in Assertion-Checker Generation. *IET Computers and Digital Techniques*, vol.1, no.6, pp. 669-677, Nov. 2007.

[9] V. Gherman, H.-J. Wunderlich, H. Vranken, F. Hapke, M. Wittke, and M. Garbers. Efficient Pattern Mapping for Deterministic Logic BIST. In *Proc. IEEE International Test Conference (ITC)*, pages 48- 56, 2004.

[10] J. Goodenough and R. Aitken. Post-Silicon is Too Late: Avoiding the $50 Million Paperweight Starts with Validated Designs. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 8-11, 2010.

[11] IEEE Standard 1800. IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language. *International Electrotechnical Commission*, 2009.

[12] J. Keshava, N. Hakim, C. Prudvi. Post-silicon Validation Challenges: How EDA and Academia Can Help. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 3-7, 2010.

[13] A. B. Kinsman, H. F. Ko, N. Nicolici. In-System Constrained-Random Stimuli Generation for Post-Silicon Validation. In *Proc. IEEE International Test Conference (ITC)*, paper 3.3, 2012.

[14] A. B. Kinsman, H. F. Ko, N. Nicolici. Hardware-Efficient On-Chip Generation of Time-Extensive Constrained-Random Sequences for In-System Validation. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, paper 39.6, 2013.

[15] H. F. Ko, A. B. Kinsman, N. Nicolici. Distributed Embedded Logic Analysis for Post-Silicon Validation of SOCs. In *Proc. IEEE International Test Conference (ITC)*, paper 16.3, 2008

[16] H. F. Ko, N. Nicolici. Resource-Efficient Programmable Trigger Units for Post-Silicon Validation. In *Proc. IEEE European Test Symposium (ETS)*, pp. 17-22, 2009.

[17] B. Koenemann. LFSR-coded Test Patterns for Scan Designs. In *Proc. IEEE European Test Conference (ETC)*, pages 237–242, 1991.

[18] P. C. McGeer, J. V. Sanghavi, R. K. Brayton, L. L. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions. *IEEE Trans. VLSI Systems*, vol. 1, no. 4, pp. 432-440, Dec. 1993.

[19] S. Mitra, S. A. Seshia, N. Nicolici. Post-Silicon Validation Opportunities, Challenges and Recent Advances. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 12-17, 2010.

[20] A. Nahir, A. Ziv, R. Galivanche, A. Hu, M. Abramovici, B. Bentley, etc. Bridging Pre-Silicon Verification and Post-Silicon Validation. In *Proc. IEEE/ACM Design Automation Conference (DAC)*, pp. 94-95, 2010.

[21] N. Nicolici. On-Chip Stimuli Generation for Post-Silicon Validation. In *IEEE High Level Design Validation and Test Workshop (HLDVT)*, pp. 108-109, 2012.

[22] S.-B. Park, T. Hong, S. Mitra. Post-Silicon Bug Localization in Processing Using Instruction Footprint Recording and Analysis (IFRA). *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 28, No. 10, pp. 1545-1558, Oct. 2009.

[23] J. Rajski, M. Kassab, N. Mukherjee, N. Tamarapalli, J. Tyszer, and J. Qian. Embedded Deterministic Test for Low-Cost Manufacturing. *IEEE Design & Test of Computers*, 20(5):58–66, Sept 2003.

[24] S. K. Sadasivam, S. Alapati, V. Mallikarjunan. Test Generation Approach for Post-Silicon Validation of High End Microprocessor. In *Euromicro Conf. on Digital System Design*, pp. 830-836, 2012.

[25] C. Spear. SystemVerilog for Verification, 2nd ed. *Springer*, 2008

[26] N. Touba and E. McCluskey. Bit-Fixing in Pseudorandom Sequences for Scan BIST. *IEEE Transactions on CAD*, 20(4):545–555, Apr 2001.

[27] Synopsys, Inc., VCS - Functional Verification Solution. www.synopsys.com/VCS, 2014.

[28] B. Vermeulen, T. Waayers, S. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proc. IEEE International Test Conference (ITC)*, pp. 638-647, 2002.

[29] L.-T. Wang, C.-W. Wu, X. Wen. VLSI Test Principles and Architectures. *Morgan Kaufman*, 2006.

[30] Y.-K. Wang, R. Even, T. Kristensen, Tandberg, R. Jesup. RTP Payload Format for H.264 Video. RFC 6184, 2011.

[31] P. Wohl, J. A. Waicukauski, S. Patel, and M. B. Amin. X-Tolerant Compression and Application of Scan-ATPG Patterns in a BIST Architecture. In *Proc. IEEE International Test Conference (ITC)*, pages 727–736, 2003.

[32] Y. Wu, S. Thomson, D. Mutcher, and E. Hall. Built-In Functional Tests for Silicon Validation and System Integration of Telecom SoC Designs. *IEEE Trans. VLSI Systems*, 19(4):629–637, April 2011.