Hardware-Efficient On-Chip Generation of Time-Extensive Constrained-Random Sequences for In-System Validation

Adam B. Kinsman, Ho Fai Ko and Nicola Nicolici
Department of Electrical and Computer Engineering
McMaster University, Hamilton, ON, L8S 4K1, Canada
Email: kinsmaab@mcmaster.ca, henryko@grads.mcmaster.ca, nicola@ece.mcmaster.ca

ABSTRACT
Linear Feedback Shift Registers (LFSRs) have been extensively used for compressed manufacturing test. They have been recently employed as a foundation for porting constrained-random stimuli from a pre-silicon verification environment to in-system validation. This work advances this concept by improving both the hardware efficiency and the duration of in-system validation experiments.

General Terms
Validation, Verification

Keywords
Constrained-Random Sequences, In-System Validation

1. INTRODUCTION
The objective of pre-silicon verification is to validate if the design implementation complies to its specification. Considering the complexity of the state-of-the-art designs, it is not tractable to formally prove the design correctness. Therefore, to increase the confidence of the validation process, pre-silicon verification tools commonly measure which design functionality has been exercised. These collected measurements are subsequently analyzed in order to guide the generation of new stimuli needed to reach the insufficiently exercised corner cases. A key point is that during pre-silicon verification what can be measured is limited by the simulation time and designs are often released to first silicon prototypes when the confidence level is deemed sufficient. Before committing to high-volume production, the validation process continues on the first silicon prototypes. Unlike simulation, these silicon prototypes are fast in operation, nonetheless the limited controllability and observability are hindering to the in-system validation process.

To deal with controllability/observability aspects of in-system validation, the basic infrastructure employed for manufacturing test can be reused for in-system validation. For example, scan chains have been successfully deployed for debugging silicon prototypes [15]. Nevertheless, before applying (and observing) patterns (and responses) through scan it is important to first learn what causes the bugs to manifest themselves and how to excite them. Therefore, recording the traces that caused the failing behaviour without stopping the real-time execution is essential to improve real-time observability and hence understand how to control the failure. Due to the inherent limitations of scan chains to capture data without interruption in real-time environments, embedded logic analysis has been adopted to improve the observability aspects of in-system validation. For this reason, different variants of on-chip trace memories [1, 4, 11, 12] are commonly employed together with event monitors [6] to detect events of interest and acquire traces in-system.

From one perspective it can be considered that the basic concepts for scan chain-based and logic analysis-based debugging have been inspired by the known techniques ordinarily used for manufacturing test and embedded system bring-up respectively. A different direction of work relies on building on the body of knowledge available in the pre-silicon verification domain. During the early phases of pre-silicon verification, when the number of bugs is relatively large, directed tests are iteratively refined to answer "does-it-work"-type of questions. Once the bug detection rate decreases, "are-we-done"-type of questions dominate over the "does-it-work"-type of questions, in which case one has to generate a large volume of functionally-compliant stimuli [8, 10] that attempt to exercise the insufficiently measured corner cases. Constrained-random stimuli combined with assertion checking are the state-of-the-art tools employed in practice during the pre-silicon verification phase. Porting such concepts from pre-silicon [2] to in-system validation has been done successfully for high-performance microprocessors [3]. The key observation is that in a hardware environment the quality of the stimuli is improved by the large number of patterns that can be applied in real-time. While this has been successfully deployed for processor-centric designs, for system-on-chip (SOC) devices, which are known to have application-specific blocks, such as high-speed peripheral controllers and custom hardware accelerators, there are indeed known techniques to implement assertions into hardware [7]: nevertheless, it is not yet adequately understood if and how one can port constrained-random stimuli to in-system validation of random logic blocks. Despite the fact that the fundamental knowledge on how to achieve this task in a systematic manner is not established, it has been demonstrated recently that for complex SOC designs there is indeed value in generating functional sequences from compact constrained-random generators placed in hardware during in-system validation [17].

Recently [9] has investigated how to adapt pseudo-random sequence generation, commonly used in built-in self-test (BIST) for very large scale integrated (VLSI) circuits, to generate pseudo-random, yet functionally-compliant, sequences that are consistent with the constraints described in a pre-silicon environment. The basic idea from [9] is to map the functionally-compliant sequences onto linear-feedback shift registers (LFSRs). The access to such embedded sequence generators can be facilitated by the emerg-
ing standards for internal instrumentation [13]. Despite presenting the first systematic approach for porting constrained-random stimuli aspects from pre-silicon to in-system validation, the main limitations of the approach from [9] are the excessive area investment when employing large LFSRs, as required for time-extensive in-system validation, as well as intractable runtimes needed when preparing the seeds for large LFSRs during successive in-system experiments. It is the main focus of this paper to address the above shortcomings and enable fast preparation of validation data for in-system experiments, while also significantly reducing the size of the on-chip signal generators.

The scope of this work is illustrated in Figure 1. When the measured coverage is considered sufficient during pre-silicon verification, the first silicon prototypes are built. Regardless whether this prototype is a fully or a partially implemented design, or whether the validation is done in a hardware emulation environment or the target application boards, our objective is to facilitate the rapid development and effective in-system generation of the tests that rely on the constrained-random simulation infrastructure already developed for pre-silicon verification. It is important to note that in this paper we present a methodology for porting only the controllability aspects used in constrained-random verification. Observability hooks for coverage measurement or error detection are considered to be an orthogonal concern to this work and hence any known techniques (e.g., [1, 6, 7]) can be employed. While the design of hardware-efficient on-chip generators is based on the premise that large number of LFSR seeds from which these constrained-random sequences should run for very long times (note, the stimuli count applied in-system is several orders of magnitude larger than during pre-silicon), a key advantage over relying solely on stimuli from the native environment is the ability to control experiments. For example, by accounting for the seeds employed to update the LFSR states, different failing sequences can be reproduced by zooming into particular windows of interest. Such knowledge of the failing sequences can be subsequently used in a pre-silicon environment to further assist with root-causing the underlying problem.

The main contributions of this paper are on designing on-chip programmable signal generators and can be summarized as follows:

- Because generating long functional sequences is a key requirement for in-system validation, one has to determine a large number of LFSR seeds from which these constrained-random sequences can run for long periods of time. Our new approach relies on encoding the pre-silicon verification constraints into compact seed vector bases that can be stored on-chip, as well as automatically determined in-field through rapidly solving linear systems of equations in Galois Fields;

- We present a new hardware-efficient architecture from which \(k\) basis vectors stored on-chip can be expanded into \(2^k\) LFSR seeds, while guaranteeing that the complexity of the hardware circuitry depends only linearly on the size of the LFSR.

2. IMPLEMENTATION FLOW

In order to generate time-extensive constrained-random stimuli for validating a silicon prototype using the efficient architecture that will be detailed in the following section, the set of stimuli constraints that are used during pre-silicon verification has to be transformed using the proposed implementation flow shown in Figure 2. The flow starts by generating stimuli cubes using the constraints from the testbench and information about the on-chip LFSR-based generator. The generated cubes will then be fed to a Galois field (GF2) solver to obtain the set of basis vectors for the chosen generators. These basis vectors will then be loaded onto the on-chip generators in the silicon prototype to generate time-extensive sequence of constrained-random stimuli in real-time. Using on-chip monitors such as hardware assertions, validation coverage of the design can be measured. This process repeats until sufficient coverage of the design is achieved, at which point, the design can be moved to production if no error has been found.

2.1 Stimuli cube generation

To illustrate how to transform the stimuli constraints from the testbench used during pre-silicon verification into stimuli cubes that can be used with a library of on-chip generators, let’s consider a very simple example constraint of \(a \geq b\), where \(a\) and \(b\) are 4-bit inputs to the design to be validated. Under this constraint, the possible number of 8-bit input patterns that can be applied to the design is \(2^{16}\). This set of input patterns can then be transformed

![Figure 2: The proposed implementation flow](image-url)
into stimuli cubes using off-the-shelf logic minimization tools such as [14]. An example of an 8-bit cube from the above constraint would be \(\{a, b\} = \{1XXX, 0XXX\}\), where \(X\) represents don’t care bit. This single cube covers 64 of the 136 possible input patterns specified by the example constraint. For this example, there will be 23 stimuli cubes remaining after this transformation.

2.2 Seeding LFSR-based generators

With the stimuli cubes in hand, the next part of the flow is to encode each stimuli cube as seeds for LFSRs, in the form of a seed vector basis of vectors in GF2 for generating all the seeds which will produce an LFSR output that matches the stimuli cube in the specified positions. The theory of representing LFSR operation through matrix multiplication in GF2 has been well developed in the area of manufacturing test [9] and can be summarized in the statement that for an \(n\)-bit LFSR reseeded (i.e. initialized with a parallel load) with a seed vector \(s\), the state \(q\) of the LFSR \(t\) clock cycles after reseeding is given by \(q_t = L^t s\), where \(q, s\) are \(n\)-bit vectors in GF2, and \(L\) is an \(n \times n\) matrix in GF2 which captures the state transitions of the LFSR (which by nature are linear). Further to this, it is common to map the \(n\) LFSR outputs to the \(m\) circuit inputs \(C\) through an XOR network defined by the \(m \times n\) matrix in GF2 \(P\), yielding \(c_t = Pq_t = PL^t s\).

In light of the well established methods for deriving \(L\) and \(P\) for a given number of circuit inputs \(m\) and a desired LFSR size \(n\), we can treat \(L\) and \(P\) as givens, which reduces the encoding process to merely solving a system of equations in GF2. As an example, consider the cube \(\{1XXX, 0XXX\}\) discussed above where \(m = 8\), and take \(n = 8\) and \(P\) to be the identity matrix \(I\). If we want to find a seed which satisfies this constraint at the time the LFSR is reseeded, we use \(c_0 = 1XXX0XXXT\) in \(c_0 = L^0 s\) (note that \(L^0\) equals \(I\)). This system embodies 8 equations (rows) over 8 variables (columns - the bits of the seed \(s_0...s_7\)), and the 6 rows for this \(c_0 = X\) (rows 1,2,3,5,6,7) can be disregarded because in plain language they say that it doesn’t matter what the value of this bit of the LFSR output is. Thus this constraint is encapsulated by the two equations from row 0 and row 4 of the matrix equation, and any seed (combination of variables \(s_0\) to \(s_7\)) which is linearly consistent with these two equations is guaranteed to produce an LFSR output which satisfies the stimuli cube at the time of reseeding.

If in addition to the above, we want the output of the LFSR to be consistent with the same stimuli cube one clock cycle after reseeding, we can write the equation \(c_1 = L^1 s\) using the same value for \(c_1\) as was used for \(c_0\). The rows of the matrix for which \(c_1 = X\) can be disregarded in the same way as for \(c_0\), yielding two new equations, distinct from the ones for \(c_0\), since \(L\) is now raised to the power 1 instead of 0. Any seed consistent only with these two equations will guarantee nothing about the clock cycle in which the LFSR is reseeded, but will guarantee that in the first clock cycle after reseeding that the LFSR output will be consistent with \(1XXX0XXX\). If we now combine the two sets of two equations each into a single system over the same seed variables and solve again, we will obtain a set of linear dependencies over the seed variables which, if satisfied, will guarantee that the output of the LFSR will be consistent with \(1XXX0XXX\) in both the clock cycle of reseeding and the subsequent one.

2.3 Expansion of all seeds

The method above links each stimuli cube (or sequence of stimuli cubes) to a system of GF2 equations over the LFSR seed variables; and the LFSR output sequence is guaranteed to satisfy that stimuli cube (or sequence of stimuli cubes) when the LFSR is seeded with any seed that is linearly consistent (i.e. valid) with the system of equations. Based on this system of equations, we would like to be able to generate all the valid seeds, as this is the key to generating time-extensive sequences from a compact representation. If we write the aggregated system of equations over the seed \(s\) discussed in the previous section as \(As = b\), the encoding of all the valid seeds is accomplished by making use of the nullspace of \(A\) : \(N(A)\), i.e. the space spanned by all solutions to \(As = 0\). If we have a vector \(s_0||s_0 = b\), then we know that for any vector \(s' \in N(A)\), \(s_0 + s'\) is also a solution since \(A(s_0 + s') = A(s_0) + As' = b + 0 = b\). Gaussian elimination produces the base solution (i.e. origin \(s_0\)) and generation of all valid seeds unfolds by combining \(s_0\) with each \(s'\) in turn; this sequence is generated by iterating through all linear combinations of the basis vectors of \(N(A)\). For a GF2 nullspace of rank = \(k\), the basis will be \(k\) vectors yielding a sequence of \(2^k\) seeds. The \(k + 1\) vectors (basis + origin) on \(n\) bits plus the value of \(k\) itself (the rank) constitute the entire encoding for a given stimulus, and comprise the information which is passed to the on-chip generators in Part 3 of Figure 2. The in-system programmable architecture which expands this encoding in a hardware efficient way is discussed in Section 3, and experiments regarding expanded data vs. encoded data for two case studies are given in Section 4.
3.1 Basis storage / expansion

Figure 3 shows the connection of the Seed Vector Memory where the encoded basis is stored to the Seed Register which contains at any point the currently generated seed in the sequence of all seeds encoded by the basis. The addressing is driven by the Basis counter, which traverses the entire sequence of $[0, 2^k - 1]$ for $k$ basis vectors, through the Addressing Mapping Logic which derives the next system vector to load based on the current position in the entire sequence. The Seed FSM controls moving from one stored system to the next, with the current system residing at the Basis start offset. Finally, a low bandwidth interface is the means by which in-system programmability is provided, i.e. generator specific information (such as choice of LFSR polynomial) and the encoded bases themselves are uploaded via this interface.

The encoded bases are stored sequentially in the Seed Vector Memory, with each system starting with the origin of the expansion (Section 2.3). Next come the $k$ basis vectors comprising the basis for a system of rank $k$. This way, the Seed FSM takes care of setting the Basis start offset to the location of the origin while loading a new system (to look up the origin from the memory) and then sets it to the location of the origin + 1 during expansion.

During operation, the origin is first loaded from the Seed Vector Memory. Then, as the Basis counter iterates through the space of $2^k$ combinations of the basis vectors, at any given time the Seed Register holds the origin plus the sum of basis vectors for which the corresponding bit of the value of the Basis counter is a 1. For example, in a system with 8-bit seeds, and 5 basis vectors (i.e. rank of 5), there are 32 combinations of the basis vectors represented by the values 00000 (the origin vector only) to 11111 (the origin vector plus the sum of all 5 basis vectors). The same end is achieved by the architecture proposed in [9], however, that approach loads all the basis vectors into registers which are selectively masked before summation by the bits which have value 0 of their equivalent to our Basis counter. In this way their method requires the entire matrix of setting the basis vectors into registers which are selectively masked before multiplication to be performed for the generation of each seed.

The problem reduces therefore to producing a numerical sequence which indicates the index of the changing bit during a Gray-code counting sequence. Figure 4 shows the tree based construction of the well known Binary-reflected Gray-code, indicating at each node which bit should change its value. The symmetric nature of the tree allows the code to be built recursively from n bits to n+1 bits by joining the roots of two copies of the n-bit tree by a node of value n+1. The process has been expanded to 4 bits, giving a Gray-code sequence of length 16 across the bottom of the figure.

When the tree is flattened, as in the bottom of the figure, the sequence of bit changes can be seen. Inspection of the pattern reveals that of the entire sequence, every other element indicates a change to bit zero. Removal of these zero elements, leaves a sequence where every other element is a node of value 1. The recursive construction procedure gives rise to a recursive procedure for extracting the location of the changing bit. The hardware structure which implements this function is a priority encoder, and it is detailed in the middle section of Figure 3. Starting from the least significant bit, when the value is zero (which happens every other element in the counter sequence) a zero is passed. In the remaining cases (i.e. the other input of the bit-zero multiplexer), every other element has value 1, this corresponds to bit 1 of the counter being 0. This pattern continues up to bit n-1 of the counter, thus creating the mapping from the n bits of the counter to the $\log_2(n)$ bits needed to encode a position within the counter.
4. EXPERIMENTAL RESULTS

The proposed methodology (explained in Section 2), together with the efficient architecture (detailed in Section 3), enable the generation of time-extensive constrained-random stimuli on-chip to aid in-system validation of digital circuits. To verify the effectiveness of the proposed technique in terms of area (discussed in Section 4.1) and length of the generated stimuli (analyzed in Section 4.2), it has been applied to generate patterns that conform to the constraints extracted from the third generation of PCI-express (PCIe) transaction layer protocol (TLP) [5], as well as the real-time transport protocol (RTP) payload format for H.264 video codec [16]. It should be noted that in the same way as reported in [9], the goal of these experiments is to show how long runs of constrained-random stimuli can be generated on-chip efficiently in real-time. Thus, the stimuli used for the experiments only contain the header part of the PCIe packets and the header part of the RTP packet for H.264. The payload part of the PCIe packets, as well as the RTP packets for H.264 are not considered in these experiments. This is because for PCIe, the payload can be entirely randomized as long as the header information conforms to the standard. On the other hand, since we are not targeting to generate conformance bitstream for H.264, we focus our experiments to generate the header part of the RTP packets with their corresponding network abstraction layer (NAL) units for providing packetized data for H.264 decoder.

For PCIe, the total number of 128-bit stimuli cubes obtained after the transformation described in Section 2 is 5119. Of the 128 bits, 27 bits are fully specified in all the cubes (i.e. care-bit density of 1), while another five bits have care-bit densities between 0.2 and 0.5. One example of fully specified bits is the 4-bit start-of-packet fixed to 0xF, and the 5-bit header type partially specified for PCIe. For H.264, 335 stimuli cubes of 168-bits are obtained, with 3 of the bits (the 2-bit V and 1-bit F) fully specified, 2 out of 3-bit of Type having care-bit densities over 0.9, and 37 other bits having care-bit density below 0.2. The detailed formats of the patterns for PCIe and H.264 are given in Figures 5 and 6.

4.1 Architecture area

Figure 7 shows the relationship between area and LFSR size both for the reference [9] and the proposed method. The quadratic relationship between LFSR size and area identified within [9] is evident in the figure. Note that results only up to an LFSR size of 28 \((\log_2(\text{LFSR})=4.8)\) are reported in [9], thus the area values for LFSRs of 32, 64 and 128 bits given here for that method have been estimated. In scaling from 20 to 28 LFSR bits, [9] reports area increase from 4035 to 7586, or a factor of 1.88 increase in area for a factor 1.40 increase in LFSR size. We thus use 4035 \((\text{LFSRsize}/20)^{1.8}\) as our estimation, matching [9] for a 20-bit LFSR, and underestimates the quadratic dependence discussed therein.

Regardless of precise values, the key difference between the proposed architecture and [9] is the shift from quadratic to near-linear dependence; indeed Figure 7 shows how LFSR size scaling is much closer to linear. This results from removal of the register array in the column logic of [9], which [9] points out as the dominating factor in the area required. This removal is made possible by the observation from Section 3 that the overall seed can be updated one basis vector at a time. In this way, at only the cost of increased sophistication of the control logic, this method leverages memory access opportunities which are wasted by [9] where the memory sits idle after the load phase at the start of any basis expansion.

4.2 Sequence generation

Table 1 shows the results for encoding stimuli cubes for both PCIe and H.264 onto linear basis vectors for varying sizes of LFSR. For each design, each stimuli cube has been encoded onto its own linear basis; the total basis vectors column indicates the total basis vectors across all stimuli cubes.

A key point to notice is that for both designs, as the LFSR size is increased, the number of basis vectors increases by the number of patterns \(\times\) additional LFSR bits. For example, when moving from the 24-bit to the 32-bit LFSR for the H.264 design, the total basis vectors increases from 7040 to 9720, or by the amount 2680 = 335 \((32 - 24)\). This occurs because once the basic linear dependencies which capture the specified bits in each cube are encoded, each additional bit appended to the LFSR brings a complete degree of freedom, encoded through an additional basis vector.

<table>
<thead>
<tr>
<th>Table 1: Seed Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
</tr>
<tr>
<td>PCIe</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>H.264</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The implication of the above is that the nature of the specified bits in the cube (i.e. the number of linear dependencies which they impart through the selected polynomial / XOR network) imposes a lower bound on the required LFSR size. Beyond this lower bound, further LFSR bits double the number of seeds which will expand into a sequence consistent with the stimuli cube. From the converse perspective, given an LFSR of size \( n \), each stimuli cube can be seen as “consuming” degrees of freedom in seed generation in accordance with the number of linear dependencies between its specified bits. The remaining \( k \) degrees of freedom can be encoded as a basis of \( k \) vectors of \( n \) bits, expanded by the architecture into \( 2^k \) seeds.

The above relationship produces a quadratic dependence of the basis size (in bits) on the LFSR size; the number of basis vectors (i.e. degrees of freedom) increases linearly with the LFSR, as does the size of each basis vector. For example, in both designs increasing the LFSR by factors of 2 (i.e. 32 to 64 to 128) increases the total seed data by factors of 4. Alongside the quadratic LFSR size vs. seed data dependence is the exponential dependence of number of seeds (and by proxy sequence generation length) on the LFSR size, as evidenced by the approximate squaring in the number of seeds as the LFSR size goes as before 32-64-128. This exponential scaling is what enables the proposed method to generate time-extensive constrained-random stimuli on-chip. Note that in some cases (i.e. 24-bit LFSR for both designs) the total seeds can even exceed the number of distinct seeds (\( 1e9 > 2^{24} \)). This is analogous to an instance where thousands of 8-bit numbers are generated. The order of magnitude in the sequence changes as values are repeated; there are many more sequences of patterns than distinct patterns.

A final important note regarding seed data is that while 80Mb (i.e. PCIe with 128-bit LFSR) of memory is by current standards considerably large, perhaps even impractical to be placed in-system for the sole purpose of validation, the basis vector memory need not in fact to be so large. Given the in-system programmability for the sole purpose of validation, the basis vector memory need not be so large, perhaps even impractical to be placed in-system (i.e. PCIe with 128-bit LFSR) of memory is by current standards could store an entire basis (\( 300\text{kb/s} \)) of seed data. This modest rate of \( 300\text{kb/s} \) is well within reach of system configuration interfaces such as JTAG.

5. CONCLUSION

The work presented in this paper has shown how long in-system validation sequences can be generated from compact signal generators placed on-chip that need only to be re-initialized infrequently through low-bandwidth interfaces. By using the programmability feature for the on-chip signal generators the validation engineers can run successive in-system experiments. The methodology presented in this paper further advances the body of knowledge for porting of pre-silicon stimuli constraints to in-system validation environments in a cost-effective and systematic manner.

Acknowledgement.

The authors acknowledge the financial support of the University Research Office (URO) from Intel Corporation.

6. REFERENCES