On-Chip Codeword Generation to Cope With Crosstalk

Kedar Karmarkar, *Student Member, IEEE,* and Spyros Tragoudas, *Member, IEEE*

***Abstract*—Capacitive and inductive coupling between bus lines results in crosstalk induced delays. Many bus encoding techniques have been proposed to improve the performance. Existing implementation techniques and mapping algorithms in the literature only apply the specific encoding. This paper presents the first generalized framework for a stall-free on-chip codeword generation strategy that is scalable and easy to automate. It is applicable to the coupling aware encoding techniques that allow recursive codeword generation. The proposed implementation strategy iteratively generates codewords without explicitly enumerating them. Codeword mapping relies on graph-based representation that is unique to the given encoding technique. The codewords are calculated on-chip using basic function blocks, such as adders and multiplexers. Three encoding techniques were implemented using the proposed strategy. Experimental results show significant reduction in the area overhead and power dissipation over the existing method that uses random logic to implement the codec.**

***Index Terms*—Codeword generation, crosstalk, encoding.**

I. Introduction

EVERY BUS line exhibits capacitive and inductive coupling with its neighboring lines. Opposing transitions on coupled bus lines causes slowed down transitions resulting in reduced performance. Over the years many techniques have been proposed to tackle this problem, such as repeater insertion [14], use of shield lines, and bus encoding [1]–[3], [5], [10], [12], [13], [15]–[17], [19], [20].

The worst case crosstalk induced delay is observed when tightly coupled bus lines undergo simultaneous opposing transitions. The simplest solution to this problem would be to insert shield lines in between each pair of bus lines. Encoding techniques such as those presented in [15] duplicate each data line such that there is always at least one transition in the same direction as that on the line under consideration. Both of these solutions essentially double the routing area of the bus, which may not always be acceptable. Another alternative is to use crosstalk avoidance codes. Many crosstalk avoidance codes have been proposed in past that rely on adding redundancy to eliminate crosstalk induced delays. Some of the techniques that are most relevant to this paper have been discussed in the subsequent section.

Lin [9] studies, in detail, the bus invert coding and its effectiveness in reducing coupling and energy consumption on a bus line. It has been observed that the bus invert coding is most effective when the bus is divided into smaller groups, which implies increased redundancy. Such method reduces average number of couplings and/or switching activity thereby reducing the energy consumption. However, such method would not be effective against crosstalk induced delay as the capture mechanism at the receiver end still has to wait for the worst case delay transmission to settle.

The introduction of static delay between adjacent bus line has been discussed in [7]. By delaying the lines that exhibit worst case transition, the technique reduces the Miller like effect observed in the mutual coupling capacitance. This type of transmission scheme is effective in reducing the energy consumption but less effective in reducing the crosstalk induced delay.

The bottom line of all of these encoding techniques is that they eliminate simultaneous opposing transitions on coupled lines by means of adding redundancy. Encoding techniques, such as those presented in [5], [20], and [19], achieve this goal by prohibiting certain bit patterns from appearing in the transmitted codeword. Many of these encoding techniques enumerate valid codewords and map them to the data words. Such mapping becomes impractical for wider buses. For wider buses, an on-chip memory based implementation is impractical as well because it suffers from the same scalability issues arising from enumerative mapping. Nonenumerative encoding techniques, such as those presented in [4], [6], and [18], rely on mathematical modeling that applies only to the code proposed in them. The proposed paper presents a generalized framework to generate codewords in a nonenumerative and scalable manner.

Hardware overhead is measured in terms of the number of redundant bits. However, it is observed that as the number of lines in a bus increases, the size of combinational logic in the encoder/decoder increases exponentially. This also gives rise to optimization issues. A real-time on-chip encoder

Manuscript received February 17, 2013; revised July 21, 2013; accepted August 30, 2013. Date of current version January 16, 2014. This work was supported in part by the NSF I/UCRC for Embedded Systems at SIUC and in part by the National Science Foundation under Grant 0856039. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This paper was recommended by Associate Editor Y. Shin.

K. Karmarkar is with Intel Corporation, Portland, OR, USA and also with the Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL 62901 USA (e-mail: kedar@siu.edu).

S. Tragoudas is with the Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL 62901 USA (e-mail: spyros@heera.engr.siu.edu).

Preliminary version of this work was published in Design, Automation & Test in Europe 2010.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2284017

implementation method is needed to make the encoding scheme scalable for wide buses. The term real-time is used to describe nonenumerative codeword generation. The method is referred to as real-time because the codewords are never explicitly enumerated and stored in memory.

The goal of the proposed method is to provide with a generalized strategy for scalable codeword generation for encoding techniques that allow recursive codeword generation. Most of the coupling aware encoding techniques in the literature fall in this category. In deep submicrometer, the performance of interconnects is critical. As on-chip interconnects are placed closer and closer together, the coupling induced delays become more and more important. As the use of tightly coupled and wider buses become more common, a scalable codec implementations strategy becomes increasingly important. This paper provides details of the application of the proposed framework for effective real-time codeword generation to existing encoding techniques, such as [5], [20], and [19]. Due to the recursive nature of codes discussed in this paper the Fibonacci sequence is a common occurrence in all three. Even though the proposed strategy utilizes Fibonacci numbers for convenience of explanation, it is not limited to encoding techniques based strictly on Fibonacci sequence. As an example, the correlation graph of the weight limited version of the code proposed in [19] has been explained in Section III-A.

This paper is organized as follows. Section II-A describes related work, provides rational. Section II-B provides some definitions and describes properties an encoding scheme must possess to be implementable using proposed strategy. The details of the steps involved in the proposed implementation strategy are provided in Section III-A. Section III-B provides details of scalable real-time implementation based on the correlation graph. Section IV presents experimental results. Three performance metrics, namely, hardware overhead, execution time for design automation tools, and power, are used to evaluate the effectiveness of the proposed method. Section V concludes the paper.

II. Preliminaries

Codewords designed to eliminate crosstalk show a highly structured nature. Codewords with common most significant bit patterns can be classified into various sets. Cardinality of each set can be mathematically determined. During every time frame, the codeword is calculated using the data word using a graph-based recursive procedure. The process is formally introduced in Algorithm 1. Real-time on-chip implementation of encoder and decoder can be achieved by backtracking such a procedure so that the codewords can be formed and mapped in a fast and effective manner without explicitly enumerating them. The pipelined architecture of encoder/decoder results in an implementation that avoids stalling of a microprocessor due to codec delay. The procedure described in subsequent sections can be extended to implement many different kinds of encoding schemes. Implementation of three such codes is described in this paper.

A. Related work

One widely recognized way to avoid crosstalk induced delays is to avoid all opposing transitions on every pair of lines in a bus. This problem was studied in [20] with detail. Two types of codes are discussed: 1) memory-based and 2) memoryless. Memory-based codes rely on codewords from two consecutive time frames while memoryless codes rely on the codeword transmitted during single time frame. The following focus on published work on memoryless codes, the latter is discussed in this paper.

Consider a graph where every codeword is a node and every edge represents a transition that does not involve any two neighboring lines switching in opposite direction. Forming a code book for such memoryless code involves finding the largest clique [20]. It is mathematically proved that the largest clique is formed by all of the neighbors of a codeword that is in the form of alternating 1's and 0's (...101010...) [20]. The codebook is implemented as random logic. This code is referred to as opposing transitions elimination encoding (OTEE) in the following discussion.

Although the technique proposed in [20] is based on graphs, there are three major differences between the graphs described in [20] and those used in the proposed method. First, the nodes in the graph specified in [20] consist of individual codewords while those in the proposed method consist of sets of codewords. Second, the edges of the graph in [20] have no directionality while those in the proposed method do. Finally, graph in [20] is always complete while that used in the proposed method is never a complete graph. In addition, the edges in the graph in [20] have same properties while those in the proposed method have different weights. Each level of the correlation graph used in this paper corresponds to each pipe-lined stage of the encoder/decoder.

The code proposed in [19] avoids all sorts of simultaneous transitions on neighboring bus lines, assisting and opposing, and therefore, avoids speedup or slowdown. [19] is designed as a transition code where every rising or falling edge is represented by a 1 while a stable logic value is represented by a 0. As an example, a codeword 101 indicates that in a three bit bus, lines on extreme ends are undergoing transitions while the middle one is stable. As a result, avoiding any codeword with consecutive 1's eliminates any simultaneous transitions on neighboring bus lines. Although code proposed in [19] relies on codewords from two consecutive time frames, it can be considered as a memoryless code because a simple exclusive-or array can be used to translate the transitions to 1's and 0's. This way for an n bit bus, the decoder logic has only n inputs and not $2n$ as in memory based codes. This code is referred to as no adjacent transitions (NAT) in the following discussion.

It is possible to have comparatively more efficient code that allows some of the opposing transitions if they are compensated by an assisting transition. The code proposed in [5] achieves this goal by eliminating any vector that contains pattern $b,\bar{b},b$ on neighboring bus lines during any given time frame. In order to maintain a simpler code book and encoder logic, authors rely on bus partitioning and group complement bits [5]. This, in turn, results in significant overhead. As discussed in subsequent sections, it is possible to extend the proposed method to implement such code without partitions and group complement such that total routing area for the

bus is significantly reduced. This kind of encoding eliminates crosstalk induced slowdown while maintaining transition speedup. In the following discussion, this code is referred to as slowdown elimination encoding (SEE).

The encoding technique proposed by [15] involves duplicating every bus line. This way any line undergoing transition has at least one assisting transition on one of its neighboring lines. This removes slowdown caused by crosstalk. Also the minimum Hamming distance of the resultant code is sufficient to provide certain degree of error detection/correction. It is a scalable method and can easily be implemented on-chip. However, the redundancy is huge and the cost in terms of routing area is prohibitive.

The encoding techniques [5], [19], [20] rely on eliminating certain bit sequences from appearing within the codeword. All three of the encoding techniques eliminate the worst case crosstalk induced delay by avoiding simultaneous opposing transitions on neighboring lines. The worst case delay of a bus line is the same for all three techniques. In other words, the choice of the encoding technique does not change the crosstalk induced delay on the bus. [5] has slightly less redundancy as compared to [20] and [19]; however, as observed in the experimental evaluation section, the codec for encoding technique in [5] exhibits more hardware overhead and power consumption as compared to the codec for encoding techniques proposed in [20] and [19].

With the exception of [15] which duplicates lines, all encoding schemes discussed above enumerate the codewords before mapping them to the data words. Consider a bus connecting two cores on a single chip. Buses as wide as 128 bits are not unimaginable. Searching a space of 2^{128} combinations for valid codewords is extremely computation intensive task if not impossible. Use of additional shield lines to divide a wide bus into smaller more manageable sections may not always be justifiable. This makes the method nonscalable.

Recursive codeword generation procedures have been proposed in [4], [6], and [18] but apply only to the code in [5]. The preliminary work published in parallel in [8] applies the proposed method to the code proposed in [19]. The method in [4], [6], and [18] uses a Fibonacci number system to represent data words. The encoders and decoders are implemented with multibit adders and multiplexers. They rely on a mathematical modeling of the technique in [5], and present a mapping algorithm suitable for the specific code in [5].

The technique proposed in [11] uses a codec based on our work in [8] to generate codewords of an encoding technique. Each codeword is then translated to a valid codeword of another encoding technique suing additional logic. As a result, the technique proposed in [11] is limited to the Fibonacci code family. The proposed technique utilizes the recursion described in Section II-B to generate a graph representation (correlation graph) unique to that encoding technique. Such graph is then used to implement the codec in an automated manner. Due to this, the proposed implementation strategy is not limited to Fibonacci code family.

The key difference between the proposed and the existing work is that the introduced graph-based representation of the codes allows implementation of a wider variety of codes, such as codes with additional restrictions on the maximum weight of the codewords. In that sense, this paper is a framework that generalizes on-chip code generation and is not limited to a specific code as our previous work in [8] or the work in [4], [6], and [18]. The experimental results indicate that the hardware overhead of proposed implementation strategy is of the same order compared to [4], [6], and [18] for the specific code. The results show scalability of the proposed approach when applied to a variety of codes. For the code in [5], the results from [6] are also given. The goal of this paper is to propose a more generalized framework that can be applied to a number of existing codes. Implementation details of three such codes is presented in this paper.

B. Definitions and Terminology

Definition 1: A valid codeword is a bit vector that does not contain bit pattern that causes undesired effects, such as crosstalk induced delays anywhere within it. Such a bit pattern is referred to as prohibited bit pattern.

Definition 2: A k-bit segment of a codeword is a k-bit binary bit string that is a subset of the given codeword

Definition 3: Class is a set of all valid codewords that contain a fixed most significant bit pattern

Example 1: Class *(00, n)* is a set of all n-bit valid codewords that have 00 as most significant bit pattern.

Theorem 1: Any k-bit codeword is valid only if the all code segments with length less than k contained within it are valid codewords themselves. This is a necessary but not sufficient condition.

Proof: Consider an arbitrary k-bit vector, x=(b_k,.., b_2, b_1). If x is a valid codeword, it must not contain a prohibited bit pattern. Consequently, any fragment of x does not contain a prohibited bit pattern and by definition is a valid codeword. Similarly, if x is not a valid codeword for a given code, then x must contain a prohibited bit pattern. As a result any codeword containing x such as (b_k + 1,b_k,.., b_2, b_1, b_0) is not a valid codeword. ■

Example 2: Consider a 5-bit codeword 10100. Under a particular encoding scheme, 10100 is a valid codeword only if two 4-bit segments (namely, 1010 and 0100), three 3-bit segments (namely, 101, 010, and 100), four 2-bit segments (namely, 10, 01, 10, and 00) are valid codewords themselves. This is a necessary but not sufficient condition since a 6-bit codeword 110100, formed by appending a leading 1 to 10100 may not necessarily be a valid codeword.

As a result, to build a set of all valid (k+1)-bit codewords one must work with a set of valid k-bit codewords. A set of (k+1)-bit codewords can be formed by appending a leading 0 or 1 to certain k-bit codewords. Newly formed set of (k+1)-bit codewords can be classified according to the MSB of the codewords. This is a recursive procedure, which is shown in Algorithm 1. A set of n-bit codewords can be formed in n iterations of a while loop starting with set of one-bit codewords. Any code that possesses the property described by above theorem can be implemented using proposed implementation strategy.

Similar codeword generation procedure has been proposed in [12] that depends on explicit codeword enumeration. The

Algorithm 1: Iterative code formation

```
Data: Bus Width (n), Forbidden bit pattern (p)
Result: Set of n-bit codewords
int k = 1; X(k) = {0, 1};
while k < n do
    forall C ϵ X(k) do
        if 0C contains p then
            Discard 0C;
        else
            Add 0C to X(k+1);
        end
        if 1C contains p then
            Discard 1C;
        else
            Add 1C to X(k+1);
        end
        k++;
    end
end
Return X(k);
```

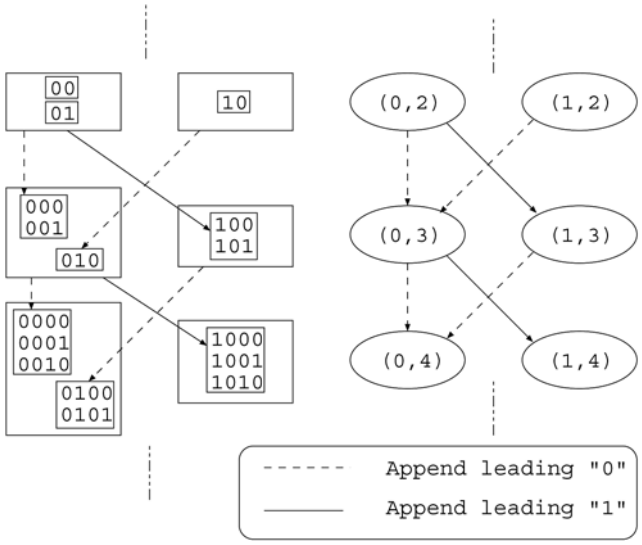

Fig. 1. Unfolded while loop.

proposed implementation strategy does not require explicit enumeration of all the codewords. Algorithm 1 is provided strictly for understanding purpose. The graph representation of the unfolded while loop described in Algorithm 1 is depicted in Fig. 1. Each node in the graph on the right in Fig. 1 represents a class. The nodes of the graph on the left side list the contents of the class. As an example, *(1, 3)* represents a set of all valid 3-bit codewords with MSB equal to 1, namely, {100, 101}. Consider a code that prohibits any consecutive 1's from appearing in the codeword. Any k-bit vector that contains consecutive 1's is not included in the set of valid k-bit codewords X(k). In order to construct a set of valid (k+1)-bit codewords, each member of X(k) must be appended with either a leading 0 or a 1. If the element of X(k) under consideration already has most significant bit as 1, appending another 1 will result in a prohibited bit pattern. Hence, a leading 1 can only be added to the element of X(k) that has 0 as the most significant bit. However, there is no such restriction while appending a 0. As a result if the cardinality of the set of valid k-bit codewords is $|X(k)|$, the cardinality of a set of valid (k+1)-bit codewords is always less than $2|X(k)|$. Due to the recursive nature of the codeword generation process, the cardinality of the set of valid codewords generally progresses in a manner that is similar to the Fibonacci series. However, it is not absolutely necessary that the cardinality of individual sets follow Fibonacci series. As an example, the set sizes for the weight limited NAT code are not members of Fibonacci series as explained in Section III.

Definition 4: Set A is a parent of set B if some elements in set B are formed by adding a leading 1 or 0 to every element in set A. Also set B is a child of set A.

The unfolded while loop discussed above forms a correlation graph where every level corresponds to an intermediate bit width. Every node represents a class. The dotted and solid arrows representing appending 0 and 1 point from parent to child class. Every codeword within a class contributes toward forming one or more codewords in the next level. Being a correlation graph the parent–child relationships are well defined. Union of all classes in the bottom most level of the graph form an ordered set of n-bit codewords. The cardinality of such set $|X(n)|$ determines the number of data bits that can be transmitted using the code. If an n-bit code is capable of transmitting d-bit data, then the inequality $|X(n)| \geq 2^d$ must be satisfied.

Mapping is done in such a way that first codeword in the ordered set is mapped to smallest data (namely, decimal 0), second codeword is mapped to next smallest data (namely, decimal 1) so on and so forth. Assume the codewords in set X(n) are classified according to the most significant bit in two classes, namely, *(0, n)* and *(1, n)*. As a result, while mapping 2^d d-bit data to $|X(n)|$ n-bit codewords, first $|(0, n)|$ data words are mapped to codewords in class *(0, n)* while remaining 2^d-$|(0, n)|$ data words are mapped to first 2^d-$|(0, n)|$ elements of class (1, n). Since $|X(n)| \geq 2^d$, last $|X(n)|$ - 2^d elements of *(1,n)* are unmapped.

The decimal value of the data determines the class of corresponding codeword in the final level. For example, if the decimal value of D of the data is less than $|(0, n)|$, it must be mapped to a codeword in class (0, n). Consequently, the most significant bit of the codeword corresponding to D must be 0. Similarly if D is greater than or equal to $|(0, n)|$, the most significant bit of the corresponding codeword must be 1.

The next step determines the parent class for the codeword. This is precisely the class of (n-1)-bit codewords from which the n-bit class originates The parent class is based on the cardinality of classes and the parent–child relationships. The parent class determines the n-1*st* bit of the codeword. Traversal of the correlation graph from bottom to top determines the codeword corresponding to a particular data word. The procedure is formally presented in Algorithm 2. Each iteration of Algorithm 2 has two steps. The first step calculates the k*th* code bit based on the cardinality of the classes at the k*th* level. The second step scales the input data based on the structure

Algorithm 2: Codeword generation

```
Data: d-bit Data word, Correlation graph
Result: n-bit codeword mapped to given data word
int k = n;
while k > 0 do
    Determine class of codeword at kth level base on:
    1. Decimal value of data
    2. Correlation graph
    if MSB of the class is 0 then
        C(k) = 0;
    else
        C(k) = 1;
    end
    Scale data according to class cardinality.
    k=k-1;
end
Return n-bit codeword = C(n), C(n-1) ... C(2), C(1);
```

of the correlation graph to direct the encoder to the correct class in the (k-1)*th* level.

The determination of the class of a codeword and the parent classes is based solely on the cardinality of the classes involved and not the contents of these classes. Since the cardinality of these classes can easily be calculated mathematically, it is not required to explicitly enumerate all the codewords. The nonenumerative nature of the method makes it more scalable for wide buses as well as more suitable for automation.

III. Proposed Implementation Strategy

The sequence of steps followed in the proposed technique is depicted in Fig. 2. The Process begins with making an initial attempt to identify the correlation between codewords of different sizes by classifying them into various classes according to the most significant bit. This section describes how the codec generation process can be automated.

A. Generation of Correlation Graph

Consider an initial classification of a set X(1) consisting of 1-bit code into two classes, namely, *(0, 1)* and *(1, 1)*. In absence of any encoding technique, the set of 2-bit codewords X(2) is formed by appending a leading 0 and leading 1 to both of the aforementioned classes. The resultant section of graph has four nodes and four edges as shown in Fig. 3(a). During the first iteration of the graph generation flow depicted in Fig. 2, a small software module identifies and lists any invalid codewords that are present in the newly formed 2-bit code set.

As an example consider the code in [19] that forbids consecutive 1's in a codeword. The codeword 11 present in class *(1, 2)* is invalid under this encoding technique and is flagged by the software tool. The codeword 11 is formed by appending a leading 1 to a member of class *(1, 1)*. As a result the solid edge connecting class *(1, 1)* and class *(1, 2)* in Fig. 3(a) must be removed. The resultant graph is shown in Fig. 3(b).

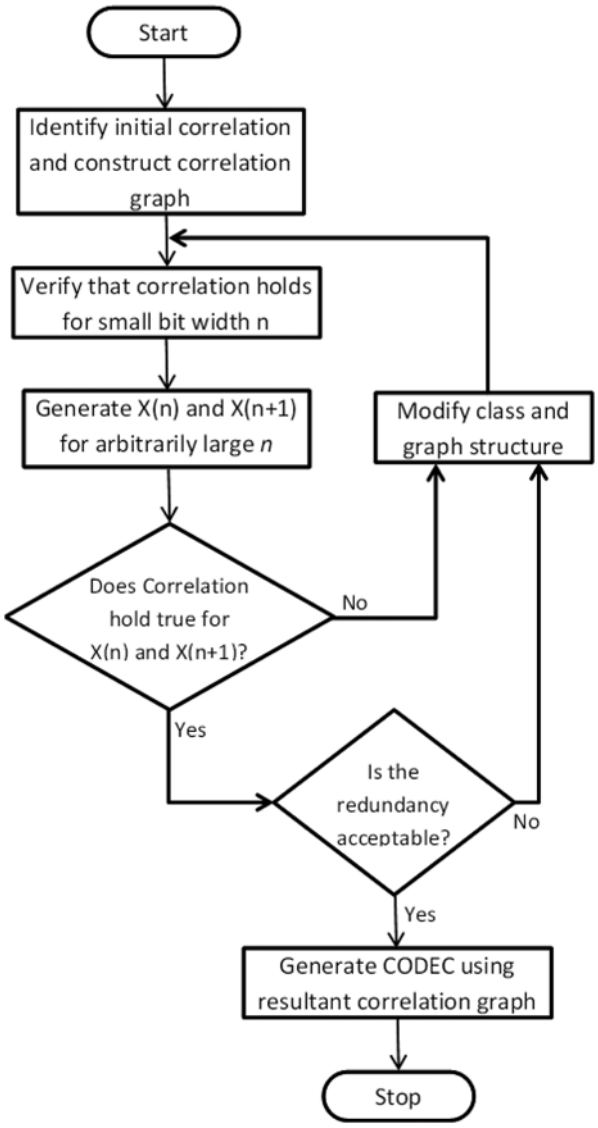

Fig. 2. Steps of proposed implementation strategy.

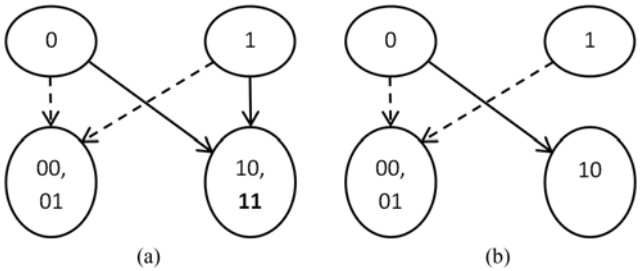

Fig. 3. (n,d,⌈n/2⌉)-NAT correlation graph (step 1).

Next step is to verify whether the relationship between set X(2) and X(3) is the same as that between X(1) and X(2). From the previous step, class *(0, 2)* has two codewords, namely, 00 and 01, while *(1, 2)* has only one codeword, namely, 10. Appending a leading 0 and 1 to each member of X(2) results in six codewords 000, 001, 010, 100, 101, 110 as shown in Fig. 4(a). The only invalid codeword 110 is formed by appending leading 1 to a member of class *(*1, 2).

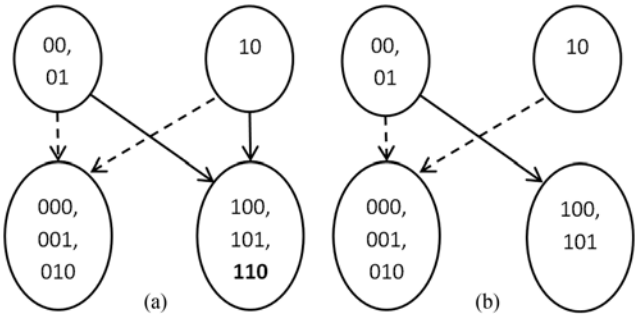

Fig. 4. (n,d,⌈n/2⌉)-NAT correlation graph (step 2).

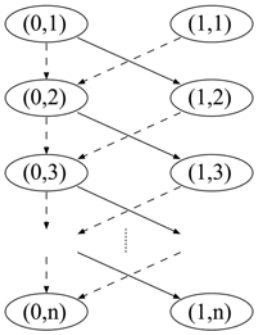

Fig. 5. (n,d,⌈n/2⌉)-NAT correlation graph.

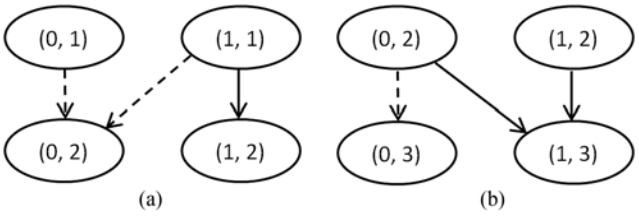

Fig. 6. OTEE correlation graph (Step 1).

Removing the solid edge between classes *(1, 1)* and *(1, 2)* in Fig. 4(a) results in the graph depicted in Fig. 4(b). As relationship between X(2) and X(3) is the same as that between X(1) and X(2), a codeword correlation is established.

An induction like argument is used to validate the correlation graph. If the correlation holds true for smaller bus width, set of valid codewords, X(n), is formed for arbitrarily large n. X(n) and the correlation graph are used to generate X(n+1). The correlation between X(n) and X(n+1) is said to hold true if all elements of X(n+1) are valid (n+1) bit codewords. If the correlation between X(n) and X(n+1) holds true and if the redundancy is acceptable, HDL descriptions of the codec is generated based on the resultant correlation graph. The resultant correlation graph for the NAT code in [19] is depicted in Fig. 5.

For some codes such as OTEE, the relationship between X(1) and X(2) is not the same as that between X(2) and X(3) as seen in Fig. 6. In such a scenario, the procedure described above is repeated till a recursive pattern is observed as seen in Fig. 7. The sections of the correlation graph must fit one on top of the other like identical building blocks.

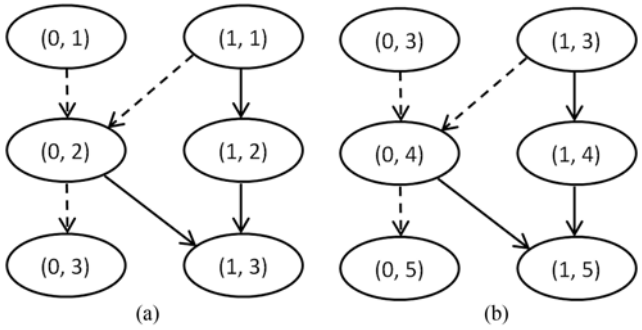

Fig. 7. OTEE correlation graph (Step 2).

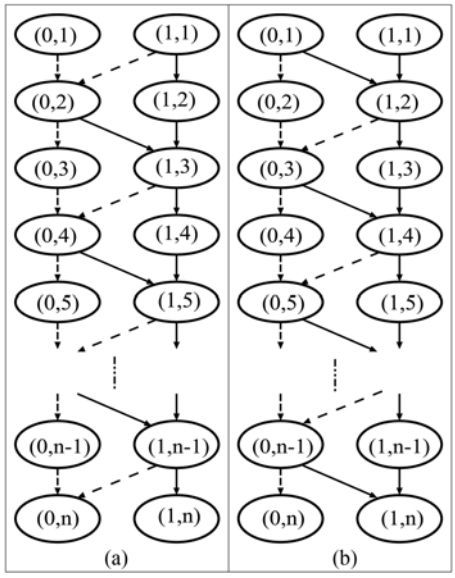

Fig. 8. OTEE correlation graph (2-cliques).

There are three scenarios in which it becomes necessary to change the initial classification of codewords during the run time of the proposed algorithm. They are as follows.

First scenario occurs when the correlation between X(n) and X(n+1) does not hold true for reasonably large n and X(n+1) contains invalid codewords. This implies that the correlation established in the aforementioned procedure is incomplete.

The second scenario occurs when a recursive pattern in the correlation graph is not observed after testing a reasonably large number of levels in the graph. As discussed before, the correlation for NAT is established by checking three levels of the graph while the correlation is established for OTEE after checking five levels of the graph. For more complicated code the number could be higher. The two cliques [20] for the OTEE code are depicted in Fig. 8.

The third scenario occurs when the redundancy of the resultant code in not acceptable. It is worth noting that the algorithm described in this section only removes edges in the graph that result in invalid codewords. Such removal may potentially remove a few valid codewords. The algorithm does not check for missing valid codewords. This is acceptable as long as the redundancy of the resultant code is acceptable. Consider for example that d-bit data is to be transmitted over

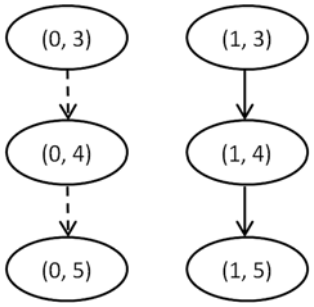

Fig. 9. SEE correlation graph (unacceptable redundancy).

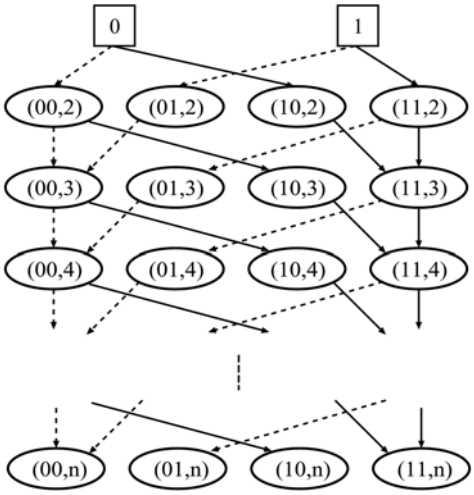

Fig. 10. SEE correlation graph.

the bus and according to the encoding technique under consideration, n-bit bus is required and (n-d)-bits are redundant. As a result, the inequality, $X(n) \geq 2^d$ must be satisfied. If X(n) produced using the correlation graph is less than 2^d, then additional redundancy would be required. In such case, the redundancy is deemed unacceptable and the correlation graph is modified further until the aforementioned inequality is satisfied.

As an example consider the correlation graph for the SEE code that prohibits patterns 101 and 010. If the initial classification based on the most significant bit is adopted the resultant graph is depicted in Fig. 9. The set X(n) will always have four codewords. Consequently, the redundancy is deemed unacceptable. The correlation graph is modified by splitting the classes based on two most significant bits and repeating the correlation graph formation procedure described above.

Using four classes, namely, *(00, n)*, *(01, n)*, *(10, n)*, and *(11, n)*, for SEE code, the resultant correlation graph is depicted in Fig. 10. The algorithm is formally presented in Algorithm 3.

It also possible to introduce additional constraints to the code using additional classes and switches in the software tool. As an example, the encoding technique proposed in [19] limits the switching power of the bus by limiting the weight of the codewords. Consider a NAT code with weight limit of two. The codewords are classified using weight as well as the most significant bit. The classes are represented as *(b, n, w)* where b is the most significant bit pattern, n is the width of the codewords and w is the weight of all the codewords in the class. For example, class *(0, 3, 1)* will contain all valid 3-bit codewords with most significant bit 0 and weight equal to 1. It is worth noting that classes *(1, n, 0)* are always empty sets as any codeword with most significant bit 1 has minimum weight of 1. Following similar logic, the classes represented by shaded nodes in Fig. 11 are empty sets.

Algorithm 3: Correlation graph generation

```
Result: Codeword Correlation Graph
Initialize correlation graph;
while Redundancy is not acceptable do
    while Correlation not established do
        forall Invalid codewords ϵ X(k) do
            Remove edge resulting in invalid codewords;
        end
        Add another level to graph and validate
        correlation;
        if codeword correlation not established then
            Modify the graph by adding more classes;
        else
            if Redundancy is acceptable then
                Break all while loops;
            else
                Modify the graph by adding more classes;
            end
        end
    end
end
Generate codec HDL description based on Correlation
graph;
```

Appending a leading 1 to a codeword results in increase in the weight of the new codeword by one as compared to the original one. As an example, appending 1 to a 3-bit codeword 001 with weight one will result in a new codeword 1001 with weight two. Appending a leading 0 does not change the weight of a codeword. The software module that verifies the validity of the codewords may flag a codeword as invalid either because it contains a forbidden bit pattern or because its weight exceeds the desired limit. Regardless of the reason, the edge in the graph that results in the invalid codeword is removed. As a result, any class *(b, n, 2)* has only dotted outgoing edge (implying only leading 0 can be appended) in the graph.

Applying Algorithm 3 to the NAT code with maximum weight of two results in the correlation graph depicted in Fig. 11. It is worth noting that due to the additional constraint of maximum weight, the cardinality of sets of codewords does not follow the Fibonacci sequence. The weight limited NAT code is one of the examples of non-Fibonacci encoding techniques that can be implemented using the proposed implementation strategy.

B. Scalable Implementation

The codewords are calculated bit by bit by traversing the correlation graph from bottom to top. If n-bit code is used to transmit d-bit data, the ordered set X(n) formed by union of all classes on the n*th* level of the correlation graph are

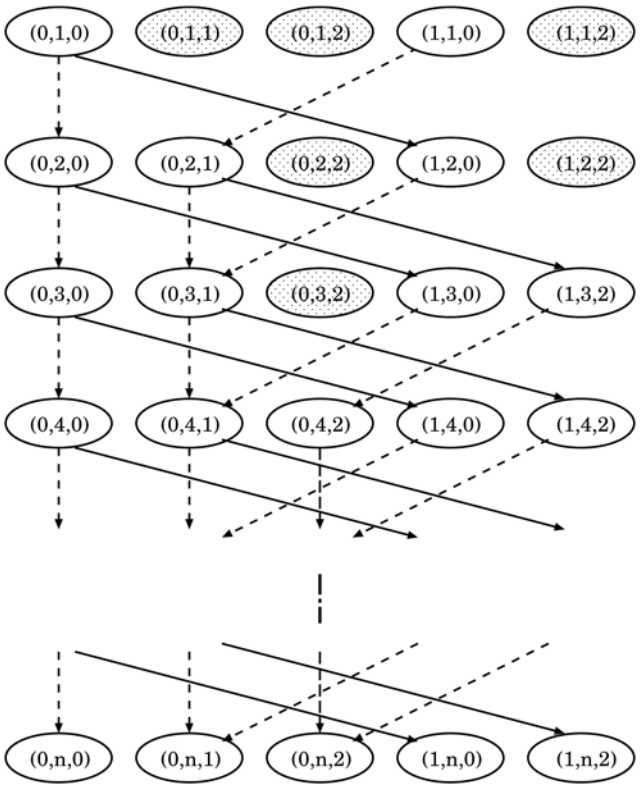

Fig. 11. (n,d,2)-NAT correlation graph.

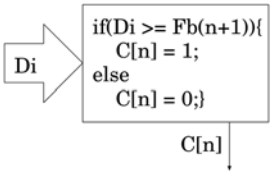

Fig. 12. OTEE code bit generation.

mapped to the 2^d data words. Consider for example the OTEE correlation graph depicted in Fig. 8(a). The first data word, namely, decimal 0 is mapped to the first member of class *(0, n)*, the second data word, namely, decimal 1 is mapped to the second member of the class *(0, n)* and so on. As a result, the decimal value of the data word determines the most significant bit of the corresponding codeword. If the decimal value of the data word is less than $|(0, n)|$, the codeword belongs to the class *(0, n)* and consequently the most significant bit of the codeword is 0. The next bit of the codeword is calculated by following the appropriate edge in the correlation graph to the parent class for the codeword. This is achieved using if-else structure in the HDL that is implemented using multiplexers. $|(0, n)|$=Fb(n+1). If d-bit input data is D_i then the n*th* code bit C(n) is generated as shown in Fig. 12.

The comparison between input data *Di* and the comparison threshold t is achieved using multibit adders and the two's complement of t. If t inside the if-statement is a k-bit number, then the sum of t and the two's complement of t is 2^k. If the two's complement of t is added to *Di* using a k-bit adder, the carry-out of the adder is set if *Di* is greater than or equal to

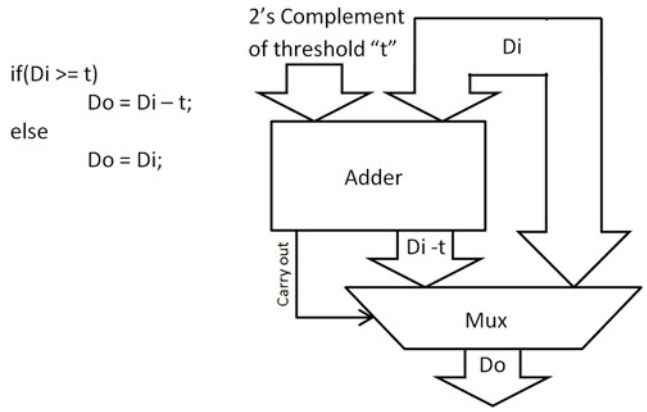

Fig. 13. Encoder block implementation.

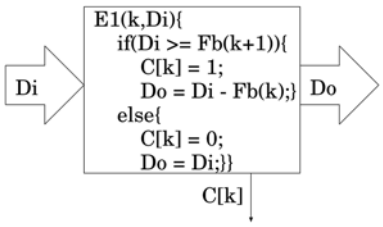

Fig. 14. OTEE encoder block E1.

the t. The data scaling discussed in the rest of this subsection ensures that the input data *Di* is less than $2^k + 1$. Consider as an example that the input data is to be compared to threshold of decimal value 5. The two's complement of 5 is decimal value 3. When *Di* is added to 3, the sum is greater than 7 if *Di* is greater than or equal to 5. As a result the carry-out of the adder acts as a comparator output. Using advanced adders such as carry select adders or carry look-ahead adders, comparison can be achieved significantly fast. The structure of the HDL modules is depicted in Fig. 13.

The $|X(n)|$ codewords available in the n*th* level originate from the $|X(n-1)|$ codewords in the n-1*st* level. Hence, while migrating from the n*th* level to the n-1*st* level on the correlation graph, number [$|X(n)|$-$|X(n-1)|$=Fb(n)] must be subtracted from the input data at the n*th* level. As a result, while migrating from n*th* level to n-1*st* level of correlation graph in Fig. 8(A), the number Fb(n) must be subtracted form the input data to the n*th* level.

In general, if D_i is the input data to the k*th* level then D_o the data input for k-1*st* level and C(k) the k*th* code bit are obtained by function E1(k,D_i) depicted in Fig. 14.

The parent–child relationship between k*th* level and k-1*st* level of the graph is different than the one between the k-1*st* and the k-2*nd* level. As a result the function generate the k-1*st* bit [say E2(k-1, D_i)] is different than function E1(k, Di) discussed above. Using similar logic, E2(j, D_i) for j=k-1 can be determined as shown in Fig. 15.

Example 3: The encoder implementation using E1(k, D_i) and E2(j, D_i) as building blocks is shown in Fig. 16. Data input for the example shown in Fig. 16 is 6 (i.e., 110). The first block performs operation E1(4, 6). Since 6≥Fb (5), C4=1 and D_o=6-Fb (4)=3. As a result, the second block performs E2(3, 3).

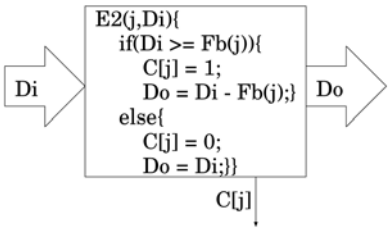

Fig. 15. OTEE encoder block E2.

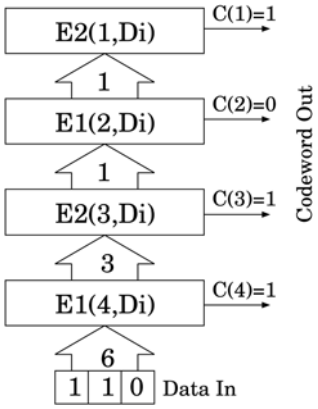

Fig. 16. OTEE encoder.

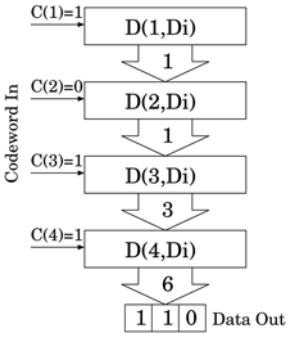

Fig. 17. OTEE decoder.

Since 3≥Fb (3), C3=1 and D_o=3-Fb (3)=1. The third block performs E1(2, 1). Since 1≱Fb(3), C2=0 and D_o=1. Finally, the fourth block performs E2(1, 1). Since 1≥Fb (1), C1=1 and D_o=1-Fb(1)=0. Irrespective of the input data, D_o from the last block is always 0. The path traversed on the correlation graph that corresponds to input data 110 is shown in bold.

A closer observation of functions E1 & E2 reveals that the encoder essentially breaks down the input data into sum of Fb(k). According to the functionality of the encoder, if Fb(k) is subtracted from the input data, the k*th* bit is set. The decoder must recover the original data by adding Fb(k) for every k*th* code bit that is set. The decoder simply adds up the Fb(k) corresponding to every bit that is set. Binary equivalent of the original data DR is given by: $\sum_{i=1}^{n}(C(i) * Fb(i))$ where C(i) is the i*th* code bit and Fb(i) is the i*th* Fibonacci number.

The breakdown of 3-bit data into Fb(4), Fb(3), Fb(2), Fb(1) and the resultant mapping is shown in Table I.

TABLE I
MAPPING 4-BIT OTEE CODE TO 3-BIT DATA

Data	Data Word	Σ (k^{th}bit*Fb(k))	Code Word
0	000	0*3+0*2+0*1+0*1	0000
1	001	0*3+0*2+0*1+1*1	0001
2	010	0*3+1*2+0*1+0*1	0100
3	011	0*3+1*2+0*1+1*1	0101
4	100	0*3+1*2+1*1+1*1	0111
5	101	1*3+1*2+0*1+0*1	1100
6	110	1*3+1*2+0*1+1*1	1101
7	111	1*3+1*2+1*1+1*1	1111

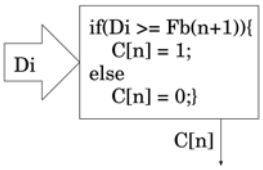

Fig. 18. OTEE decoder block.

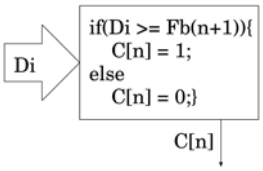

Fig. 19. NAT code bit generation.

The block diagram of the decoder using D(k,D_i) function blocks is given in Fig. 17. Function D(k,D_i) is given in Fig. 18.

Following similar logic, real-time implementation of the NAT and SEE codes is achieved as discussed in the reminder of this section. The first *|(0, n)|* data words are mapped to all of the elements in the class *(0, n)*. The remaining data words are mapped to the first (2^d-*|(0, n)|*) elements of *(1, n)*. The remaining elements of *(1, n)* are unmapped. For (4,3,2)-NAT code mapping shown in Table II (*|(1, n)|*+*|(0, n)|*)=2^d; hence, there are no unmapped codewords.

Encoder essentially traverses the correlation graph in Fig. 5 from bottom to top. It is observed that $|X(k)|$=Fb(k+2), *|(0, k)|*=Fb(k+1), and *|(1, k)|*=Fb(k). Starting with the last iteration, comparison of decimal value of the input data (D_i) with *|(0, k)|* determines whether the codeword belongs to class *(1, n)* or *(0, n)*. The resultant functionality is depicted in Fig. 19.

Since $|X(n)|$ - $|X(n-1)|$=Fb(n), while migrating from n*th* level to n-1*st*, Fb(n) must be subtracted from the data input to the n*th* level. The parent–child relationship between every pair of levels is identical. The resultant function block E(k, D_i) is shown in Fig. 20.

The (4,3,2)-NAT encoder is shown in Fig. 21

Example 4: In Fig. 21, the input data is 6 (110). Since 6≥Fb(5), output D_o of E(4,6) is $6 - 5 = 1$ and C(4) is 1. Next comparison is done with Fb(4)=3. Since 1≱Fb(4), D_o is 1 and C(3) is 0. This determines the class of the codeword in the second last level. At the end of the process, codeword 1001 is formed. As a result, bus lines at extreme ends undergo transition while middle lines are stable.

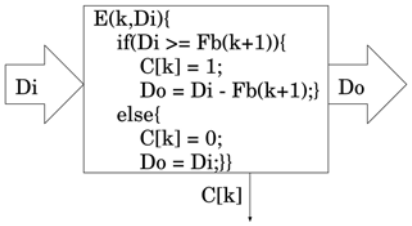

Fig. 20. NAT encoder block.

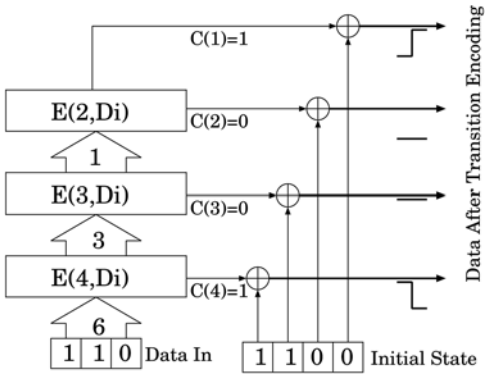

Fig. 21. NAT encoder implementation.

TABLE II
MAPPING 4-BIT NAT CODE TO 3-BIT DATA

Data	Data Word	$\Sigma(k^{th}bit * Fb(k+1))$	Code Word
0	000	0*5+0*3+0*2+0*1	0000
1	001	0*5+0*3+0*2+1*1	0001
2	010	0*5+0*3+1*2+0*1	0010
3	011	0*5+1*3+0*2+0*1	0100
4	100	0*5+1*3+0*2+1*1	0101
5	101	1*5+0*3+0*2+0*1	1000
6	110	1*5+0*3+0*2+1*1	1001
7	111	1*5+0*3+1*2+0*1	1010

The NAT encoder breaks down the input data into sum of Fb(k+1) values. If Fb(k+1) is present in the sum, the k*th* bit is set. Decoder simply adds the Fb(k+1) values corresponding to every code bit that is set. The original data *Dr* from an n-bit codeword can be given by:

$$\sum_{i=1}^{n}(C(i) * Fb(i+1))$$

where C(i) is the i*th* code bit and Fb(i+1) is the i+1*st* Fibonacci number.

The 3-bit data broken-up as a sum of k*th*bit*Fb(k) and the corresponding mapping is shown in Table II. As shown in Fig. 23 array of exclusive or gates translate the transitions on the bus lines to codewords. According to Table II the data '2' is mapped to code 0010, data '4' is mapped to 0101 while data '7' is mapped to 1010. It is worth noting that the actual bits transmitted over the bus depend upon the initial state. As an example, consider data '2', '4' and '7' transmitted over the bus with initial state 0000 during consecutive time frames t_0, t_1 and t_2. The resultant bus states are enumerated in Table III. Absence of consecutive 1's in

TABLE III
NAT TRANSMISSION WITH INITIAL STATE 0000

Time Frame	Data to be Transmitted	Mapped transition codeword	Initial State	Next State
t_0	010	0010	0000	0010
t_1	100	0101	0010	0111
t_2	111	1010	0111	1101

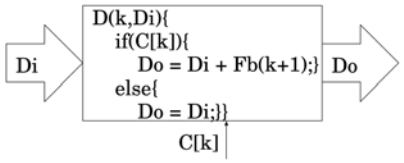

Fig. 22. NAT decoder block.

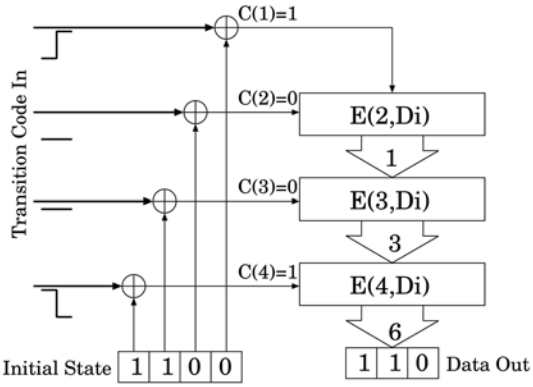

Fig. 23. NAT decoder implementation.

the codewords ensures absence of adjacent transitions thereby avoiding crosstalk induced delay. Code bits are individually fed to each of the functional blocks. As long as proper code bits are fed to proper blocks, the order in which the addition takes place is irrelevant. Functionality of block D(k, D_i) is depicted in Fig. 22.

The length of the codeword is determined by the number of function blocks. The number of function blocks (E and D) is proportional to the number of code bits. An n-bit encoder has n (or n-1 function blocks, depending upon the type of the code). As an example, the 4-bit encoder/decoder circuits in Figs. 16, 17, and 28 have four blocks. A 5-bit encoder/decoder will have five blocks. Each function block consists of multibit adders and multiplexers. An n-bit encoder/decoder will require n-bit adders and multiplexer. The gate count provided in experimental results does not reflect the number of function blocks but the actual number of gates that are used to build the adders and multiplexers. For very large codes, throughput of the encoder and decoder can be improved by using pipelined architecture. The encoder implementation as a systolic machine is shown in Fig. 24. Pipelined architecture also improves the energy consumption per transmitted data word.

The k*th* level of the correlation depicted in Fig. 10 consists of four classes, namely, *(00, k), (01, k), (10, k), and (11, k)* with cardinalities Fb(k), Fb(k-1), Fb(k-1), and Fb(k), respectively. Consequently, the cardinality of X(k) is 2Fb(k+1). The first Fb(k) data words are mapped to class *(00, k)* next Fb(k-1)

TABLE IV
CLASSES OF SEE

Code Bits (n)	*(00,n)*	*(01,n)*	*(10,n)*	*(11,n)*	$\|X(n)\|$	Data Bits (d)	Redundancy (r)
2	00	01	10	11	4	2	0
3	000, 001,	011	100	110, 111	6	2	1
4	0000, 0001, 0011	0110, 0111	1000, 1001	1100, 1110, 1111	10	3	1

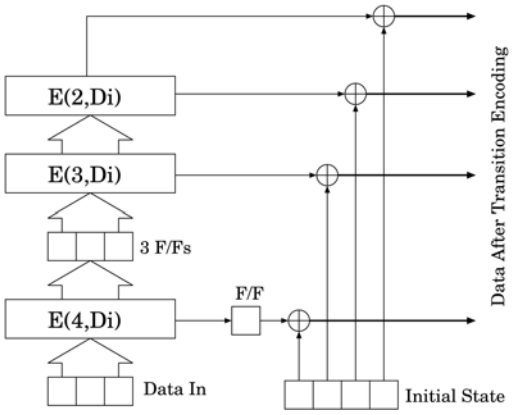

Fig. 24. NAT encoder (pipelined).

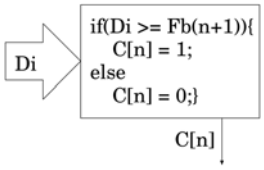

Fig. 25. SEE code bit generation.

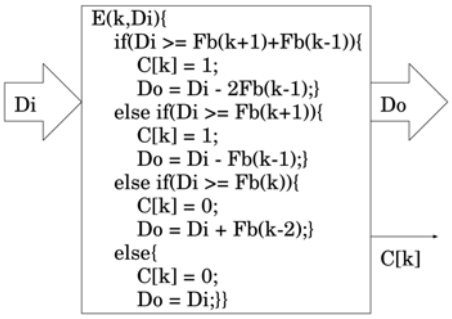

Fig. 26. SEE encoder block.

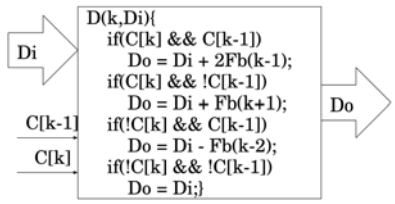

Fig. 27. SEE decoder block.

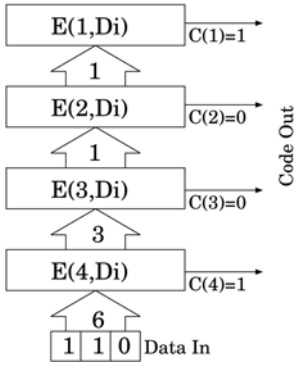

Fig. 28. SEE encoder.

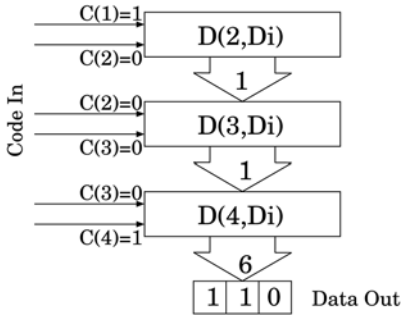

Fig. 29. SEE decoder.

data words are assigned to class *(01, k)*, and, so on. As a result, the most significant bit of the codeword corresponding to first Fb(k+1) data words is 0, the most significant bit is 1 for the remaining. The n*th* code bit is generated as shown in Fig. 25.

A comparison with Fb(k+1) does not determine the class of the codeword. Next a comparison is made with Fb(k) and [Fb(k+1)+Fb(k-1)]. If the codeword belongs to *(00, k)*, no adjustment of input data is required. If the codeword belongs to *(01, k)*, input data must be scaled by adding Fb(k-2). For *(10, k)*, Fb(k-1) must be subtracted while for *(11, k)*, 2Fb(k-1) must be subtracted. The functionality of the encoder block E(k, D_i) is depicted in Fig. 26.

The proposed encoder and decoder are shown in Figs. 28 and 29, respectively. The functionality of the decoder block is more complicated. Since there are four classes at each level, each decoder block requires two code bits, namely, k and k-1, to recreate the data. In the encoder block, if the codeword belonged to class *(11,k)*, then 2Fb(k-1) is subtracted from the input data; hence, to recover the original data, 2Fb(k-1) must be added. Applying similar logic to classes *(00, k), (01, k), and (10, k)* the functionality of decoder block can be derived as shown in Fig. 27.

The implementation of all three encoding techniques follows the procedure described in Algorithm 2. It is worth noting that the first step in Algorithm 2 that generates the code bits is the same for the three codes. As a result, Figs. 12, 19, and 25 are same. However, the way in which data is scaled while traversing the graph is different for different encoding techniques. As a result the second step of Algorithm 2 results in different functionalities of the final encoder/decoder blocks.

The functionality of the encoder and decoder blocks of the weight limited NAT code are more complex due to the

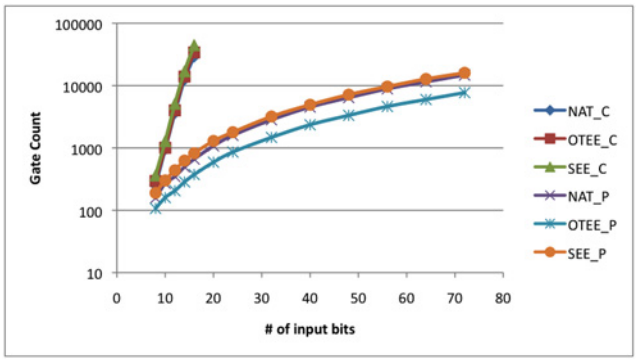

Fig. 30. Hardware requirement.

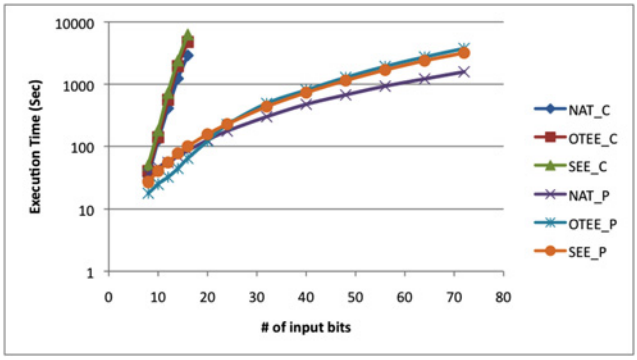

Fig. 31. Execution time.

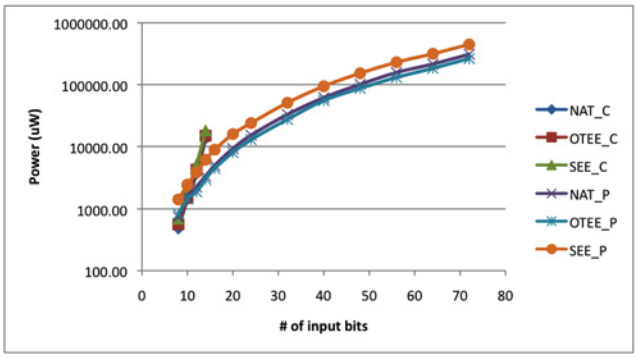

Fig. 32. Power consumption.

complicated nature of the correlation graph. The information regarding the weight of the codeword is lost in the encoding process as a result a small module to calculate the weight of the code is required.

IV. Experimental Results

In order to verify the effectiveness of the proposed implementation method, VHDL codes were written for encoder and decoder as random logic, as well as proposed method. These VHDL codes were then synthesized using Cadence encounter tool using 180nm technology and circuit statistics were analyzed. The HDL codes were written and synthesized in a generic 180nm technology using cadence RTL compiler and Cadence encounter to generate the schematics and layout for the codec. Power consumption of the bus lines is based on post extraction figures provided by the ASU predictive model for 180nm node.

Each function block terminates with a register where the calculated value is stored for the next clock cycle. The

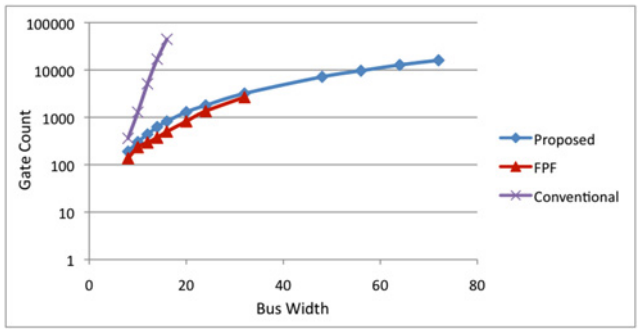

Fig. 33. Hardware requirement of code in [5].

synthesis tool is constrained to achieve the delay of each pipeline stage of the codec to be no more than 1nS. Since the codec produces one codeword each cycle, the average energy consumption is reduced to a reasonably small value for sufficiently large amount of data transfer.

For the encoding techniques that do not provide the scalable implementation, it is assumed that the codec is implemented as a truth table based random logic circuit. Such circuit has d-bit data input and an n-bit code output that is transmitted over the bus. Give the data word as an input, the circuit produces the corresponding codeword as an output. Table VIII describes the hardware overhead of implementation of OTEE for both the methods. The number of functional blocks required is linearly proportional of number of code bits and the functionality of each block is fairly simple. As a result, for wider buses, the proposed method requires much less hardware. A similar trend is observed in synthesis of (n,d,$\lceil n/2 \rceil$)-NAT code as shown in Table IX and that of SEE as seen in Table X. The synthesis tool failed to synthesize the conventional encoder/decoder implementation for buses beyond 16-bits due to memory requirement violation. The experimental results for up to 72-bit bus are provided to emphasize the fact that the proposed method is in fact capable of completing the task of encoder/decoder synthesis as opposed to the random logic implementation, which runs out of memory beyond 16-bit bus.

Fig. 30 compares the three encoding techniques with respect to gate count. The curves labeled NAT_C, OTEE_C and SEE_C represent the gate counts of conventional random logic implementation of the three encoding techniques while the curves labeled NAT_P, OTEE_P and SEE_P represent the gate counts of implementations using the proposed strategy. The same naming convention is used in Figs. 31 and 32.

The conventional method also exhibits a considerable computational overhead and consequently time required for code generation as well as synthesis of the encoder/decoder circuitry is considerably high as observed in Fig. 31. The execution time of the software tool to synthesize a 16-bit conventional encoder is comparable to the time for synthesis of a 72-bit encoder using proposed implementation strategy.

A similar trend is observed in case of power consumption of the encoder/decoder implementation as seen in Fig. 32. The power consumption is estimated using the Cadence RTL compiler. The power estimation for 16-bit and beyond conventional encoders is not shown as the RTL compiler terminated due to memory requirement violation. Table V depicts the

TABLE V
POWER CONSUMPTION WITH AND WITHOUT ENCODING

Encoding Technique	Non-encoded bus Worst case power (uW)	Encoded bus Worst case power (uW)	codec Power Conventional (uW)	codec Power Proposed (uW)	Total Power Conventional (uW)	Total Power Proposed (uW)
OTEE	11534.2	9872.6	14840.78	2962.16	24713.38	12834.76
NAT	11534.2	9872.6	13463.57	3305.12	23336.17	13177.72
SEE	11534.2	9872.6	19037.97	6088.52	28910.57	15961.12

power consumption on a 2500 micrometer bus capable of transmitting 14-bit data. As such, the bus without encoding has 14 lines while the bus with encoding has 20 lines. The bus was modeled using π-model based on [21] using the same technology as the encoder/decoder. Despite having more number of lines, the worst case dynamic power consumption on the bus without encoding is higher since the coupling capacitance exhibits a Miller-like effect. Since bus encoding eliminates simultaneous opposing transitions, the Miller-like effect is not observed in encoded bus. As a result despite having more number of lines, the total power consumption on an Nonencoded bus is slightly less. As observed in Table V. The total power consumption of the proposed technique is considerably less as compared to the conventional one. The main goal of crosstalk avoidance encoding is to improve the performance of the bus. The proposed implementation strategy improves the performance while consuming between 11% to 38% more power as compared to an Nonencoded bus.

The total power consumption of the encoded bus is the sum of power dissipated on the bus and that on the logic. If the total power of encoded bus is restricted to that of the nonencoded bus, the power consumption on the bus must be reduced to accommodate the additional power consumed by the logic circuit. This can be achieved by reducing the coupling capacitance between neighboring lines. This can be done by separating the bus wires over a larger area while keeping the wire width constant.

The inverse relationship between the line separation and coupling capacitance and consequently power, results in drastic increase in area when power budget is tight. Table VII represents the overhead of encoding in terms of redundancy as well as actual routing area. The results show that to bring the power to the level of nonencoded bus, the area increases by about 61% to 67% for some codes. In addition, we observe that the codec is necessary. Due to relatively higher power consumption for the codec for SEE, the routing overhead of the technique is considerably higher compared to NAT and OTEE. The primary goal of crosstalk avoidance encoding is performance enhancement in terms of speed. This can be achieved by either treading off power or routing area. It is worth noting that increasing routing area also improves the delay on the bus.

The goal of using bus encoding is to improve the performance using redundant bits. Based on SPICE simulations performed using the π-model on buses from 1500um to 2500um, the delay of the encoded bus is approximately 37% better compared to an unencoded bus as seen in Table VI. It is worth noting that despite having different redundancies, the worst case delay for all three codes is the same as a result the performance improvement for SEE, OTEE, and NAT is the same.

TABLE VI
PERFORMANCE IMPROVEMENT USING ENCODING

Encoding Technique	Performance Improvement		
	1500um	2000um	2500um
OTEE	37.1%	37.07%	37%
NAT	37.1%	37.07%	37%
SEE	37.1%	37.07%	37%

TABLE VII
OVERHEAD WITH RESPECT TO NONENCODED BUS FOR NO EXTRA POWER

Encoding Technique	Redundancy	Area Overhead
OTEE	42.86%	61.7%
NAT	42.86%	67.8%
SEE	42.86%	194%

TABLE VIII
HARDWARE OVERHEAD OF OTEE

Data bits (d)	Gate Count (proposed)	Gate Count (Conventional)
8	107	295
10	160	1007
12	206	4006
14	285	13949
16	374	34084
20	590	-
24	859	-
32	1476	-
48	3334	-
56	4627	-
64	5984	-
72	7716	-

The rest of the section focuses on comparing the proposed method to the one presented in [6]. The comparisons are only meaningful for the code proposed in [5]. The experimental results provided in [6] are interpreted from the chart published in the paper without having first hand data. Fig. 33 plots the results of the proposed method and those interpreted from [6] represented by the curve labeled forbidden pattern free (FPF). The vertical axis represents the gate count while the horizontal axis represents the bus width. Unfortunately, the results provided extend only upto 32-bit buses. Although the gate count is a good indication of hardware overhead, the actual transistor count depends upon the technology and libraries used for implementation. It can be seen that the hardware overhead of proposed strategy is of the same order as compared to that of the technique proposed in [6]. It is seen in Fig. 33 that with increase in bus width, the hardware overhead of [6] appears to increase more rapidly than the proposed method. Therefore, for buses larger than 32-bits, the cost is expected to be higher.

TABLE IX
HARDWARE OVERHEAD OF (N,D,⌈N/2⌉)-NAT ENCODER

Data bits (d)	Gate Count (proposed)	Gate Count (Conventional)
8	157	253
10	270	960
12	348	3648
14	496	12593
16	662	29375
20	1080	-
24	1595	-
32	2828	-
48	6424	-
56	8892	-
64	11531	-
72	14778	-

TABLE X
HARDWARE OVERHEAD OF SEE

Data bits (d)	Gate Count (proposed)	Gate Count (Conventional)
8	190	362
10	301	1285
12	438	5112
14	627	16959
16	822	44473
20	1295	-
24	1790	-
32	3189	-
48	7163	-
56	9641	-
64	12772	-
72	16029	-

V. CONCLUSION

The conventional implementation strategy is based on explicit enumeration of codewords. In case of very wide buses, such codeword enumeration results in exponential hardware overhead. Maintaining a codebook is not a practical approach. Proposed implementation strategy takes advantage of the iterative structure of the encoding technique. The encoder/decoder are described as structural HDL model that uses multibit adders, comparators, and multiplexers as building blocks. This makes the strategy scalable for very wide buses.

The proposed strategy has been applied to a variety of encoding techniques. The properties an encoding technique must possess to be implementable using the proposed strategy are described in this paper. Three of the existing encoding techniques that fit the criteria were implemented using proposed strategy with encouraging outcomes. All three encoding techniques exhibit similar scalable trends in areas such as hardware overhead, power consumption, memory requirements and time complexity.

REFERENCES

[1] M. Anders, N. Rai, R. K. Krishnamurthy, and S. Borkar, "A transition-encoded dynamic bus technique for high-performance interconnects," *IEEE J. Solid-State Circuits*, vol. 38, no. 5, pp. 709–714, May 2003.

[2] R. Ayoub and A. Orailoglu, "A unified transformational approach for reductions in fault vulnerability, power, and crosstalk noise & delay on processor buses," in *Proc. ASP-DAC*, vol. 2. Jan. 2005, pp. 729–734.

[3] K.-C. Cheng and J.-Y. Jou, "Crosstalk-avoidance coding for low-power on-chip bus," in *Proc. 15th IEEE Int. Conf. Electron. Circuits Syst.*, Aug.–Sep. 2008, pp. 1051–1054.

[4] C. Duan, V. H. C. Calle, and S. P. Khatri, "Efficient on-chip crosstalk avoidance CODEC design," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 4, pp. 551–560, Apr. 2009.

[5] C. Duan, A. Tirumala, and S. P. Khatri, "Analysis and avoidance of cross-talk in on-chip buses," in *Proc. Hot Interconnects*, vol. 9. Aug. 2001, pp. 133–138.

[6] C. Duan, C. Zhu, and S. P. Khatri, "Forbidden transition free crosstalk avoidance CODEC design," in *Proc. ACM/IEEE Design Autom. Conf.*, Jun. 2008, pp. 986–991.

[7] M. Ghoneima and Y. Ismail, "Delayed line bus scheme: A low-power bus scheme for coupled on-chip buses," in *Proc. ISLPED*, Aug. 2004, pp. 66–69.

[8] K. Karmarkar and S. Tragoudas, "Scalable codeword generation for coupled buses," in *Proc. Design Autom. Test Eur.*, Mar. 2010, pp. 729–734.

[9] R.-B. Lin, "Inter-wire coupling reduction analysis of bus-invert coding," *IEEE Trans. Circuits Syst. I Reg. Papers*, vol. 55, no. 7, pp. 1911–1920, Aug. 2008.

[10] C.-G. Lyuh and T. Kim, "Low power bus encoding with crosstalk delay elimination," in *Proc. 15th Annu. IEEE Int. ASIC/SOC Conf.*, Sep. 2002, pp. 389–393.

[11] M. Mutyam, "Fibonacci codes for crosstalk avoidance," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 10, pp. 1899–1903, Oct. 2012.

[12] M. Mutyam, "Preventing crosstalk delay using fibonacci representation," in *Proc. 17th Int. Conf. VLSI Design*, 2004, pp. 685–688.

[13] M. Mutyam, "Selective shielding: A crosstalk-free bus encoding technique," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design*, Nov. 2007, pp. 618–621.

[14] D. Pamunuwa and H. Tenhunen, "Repeater insertion to minimise delay in coupled interconnects," in *Proc. Int. Conf. VLSI Design*, Jan. 2001, pp. 513–517.

[15] D. Rossi, C. Metra, A. K. Nieuwland, and A. Katoch, "New ECC for crosstlk impact miimization," *IEEE Design Test Comput.*, vol. 22, no. 4, pp. 340–348, Jul.–Aug. 2005.

[16] D. Rossi, A. K. Nieuwland, A. Katoch, and C. Metra, "Exploiting ECC redundancy to minimize crosstalk impact," *IEEE Design Test Comput.*, vol. 22, no. 1, pp. 59–70, Jan. 2005.

[17] S. Salerno, E. Macii, and M. Poncino, "Crosstalk energy reduction by temporal shielding," in *Proc. ISCAS*, vol. 2. 2004, pp. 49–52.

[18] S. Sinha, R. Kar, and A. K. Bhattacharjee, "Bus encoding technique using forbidden transition free algorithm for cross-talk reduction for on-chip VLSI interconnect," in *Proc. ACE*.

[19] P. Subrahmanya, R. Manimegalai, V. Kamakoti, and M. Mutyam, "A bus encoding technique for power and crosstalk minimization," in *Proc. 17th Int. Conf. VLSI Design*, 2004, pp. 443–448.

[20] B. Victor and K. Keutzer, "Bus encoding to prevent crosstalk delay," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2001, pp. 57–63.

[21] Y. Cho, *Predictive Technology Model* [Online]. Available: http://ptm.asu.edu/

Kedar Karmarkar received the B.E. degree in electronics from the University of Mumbai, Mumbai, India, in 2004, the M.S. and Ph.D. degrees in electrical and computer engineering from Southern Illinois University, Carbondale, IL, USA, in 2008 and 2013, respectively.

He is currently with Intel Corporation, Portland, OR, USA. His research interests include high-speed bus architectures for VLSI circuits, crosstalk elimination, scalability, and automation of bus encoding techniques for error correction and detection.

Spyros Tragoudas (M'87–SM'07) received the Diploma degree in computer engineering from the University of Patras, Patras, Greece, in 1986, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Dallas, Richardson, TX, USA, in 1988 and 1991, respectively.

He has been a Faculty Member with the Computer Science Department, Southern Illinois University (SIUC), Carbondale, IL, USA, and the Electrical and Computer Engineering Department, the University of Arizona, Tuscan, AZ, USA. He is currently a Professor and the Department Chair with the Electrical and Computer Engineering Department, SIUC, and the Director of the National Science Foundation (NSF), Industry University Cooperative Research Center (IUCRC) on Embedded Systems at the SIUC site. His current research interests include VLSI design and test automation and computer networks. He has published over 70 journal papers and over 130 articles in peer-reviewed conference proceedings in these areas.

Dr. Tragoudas is on the Editorial Board of several IEEE journals. He has been on the Program Committee of many International Conferences in the area of VLSI testing, and has served in several NSF panels. He received the ICCD94, ICCD97, and the ISQED01 Outstanding Paper Awards for research in VLSI testing. He has received funding from industry and federal agencies, including the NSF, for research in electronic design and test automation.