Scalable Codeword Generation for Coupled Buses

Kedar Karmarkar
Department of Electrical and Computer Engineering
Southern Illinois University
Carbondale, Illinois 62901–6603
Email: kedar@siu.edu

Spyros Tragoudas
Department of Electrical and Computer Engineering
Southern Illinois University
Carbondale, Illinois 62901–6603
Email: spyros@engr.siu.edu

Abstract—Inductive and capacitive coupling are responsible for slowing down signals. Existing bus encoding techniques tackle the issue by avoiding certain types of transitions. This work proposes a codeword generation method for such techniques that is scalable to very wide buses. Experimentation on a recent encoding technique confirms that the conventional method is limited to 16-bit bus while the proposed method is easily extended beyond 128-bits.

I. INTRODUCTION

Every line in a bus suffers from parasitic coupling with its neighbors that impacts speed of transitions. There are many techniques that can be used to cope with the presence of crosstalk. Repeater insertion which speeds up the transitions on a line however gives rise to complicated layouts and routing issues. Another technique is to use bus encoding [1]–[9] to avoid certain types of transitions on neighboring lines. [3], [7], [9] present theoretical approach to encoding. Temporal shielding [4] will result in reduced throughput.

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 [9] with detail. [9] presents a quick mathematical method to generate codewords of given length. Code book so formed is implemented as random logic. Thus it suffers from the scalability issues. The encoding technique proposed in [8] works by avoiding all sorts of simultaneous transitions on neighboring bus lines, assisting and opposing, thereby avoiding speedup or slowdown. [8] is designed as a transition code where every transition (rising or falling) is represented by a 1 while a steady state is represented by a 0. For example, a codeword 101 indicates that in a three bit bus, edge lines are undergoing transitions while the middle one is in a steady state. As a result, eliminating codewords with consecutive 1’s eliminates all simultaneous transitions on neighboring bus lines. Although the code proposed in [8] relies on codewords from two consecutive time frames, it can be considered as a memoryless code because a simple XOR array can be used to translate the transitions to 1’s and 0’s.

It also is possible to have more efficient code that allows some of the opposing transitions if they are compensated by an assisting transition. The code proposed in [1] achieves this goal by eliminating any data word that contains pattern bbb, br {0,1}, on neighboring bus lines during any given time frame. In order to maintain a simpler code book and encoder logic, the authors rely on bus partitioning and group complement bits which results in significant overhead [1].

The encoding technique proposed by [6] involves duplicating every bus line. In effect this kind of encoding removes slowdown caused by crosstalk and provides certain degree of error detection. It is a scalable method and can easily be implemented on-chip. Redundancy is huge however and the cost in terms of area is prohibitive. All encoding techniques with acceptable redundancy rely on maintaining a code book. Exponential size of the code books make these methods non-scalable using conventional random logic implementation. None of the existing work address the very essential problem of on-chip codeword generation.

Existing methods enumerate the codewords before they can be mapped to the data words. Searching all possible vectors for valid codewords is extremely computationally intensive task. One may argue that a bus can be partitioned into smaller subsets using additional shield lines. However many times these buses are routed on higher metal layers where use of additional shield lines may not always be justifiable [9]. This makes these techniques non-scalable.

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, size of combinational logic in the encoder and decoder increases exponentially. The average number of fan-outs per gate also increases giving rise to optimization issues. For very large buses, on-chip memory based implementation becomes impractical as well.

Focus of this work is to propose an implementation strategy that is easily scaled and automated for very wide buses. Proposed on-chip implementation strategy can be applied to a number of encoding techniques. Proposed implementation strategy has been successfully applied to the encoding techniques proposed in [1], [8], [9]. In this work, implementation of only [8] is discussed due to space limitations.

The proposed method works iteratively. A set of n-bit codewords can be used to derive (n+1)-bit codewords. Such codeword correlation is described in Section II which is essential for scalable on-chip codeword generation. Given a data word, Section III describes how to generate the corresponding codeword bit by bit. On-chip implementation of encoder and decoder can be achieved using this procedure so that the codewords can be formed and mapped in a fast and effective...
manner with out explicitly enumerating them. Section IV gives experimental results that show that proposed strategy is easily scaled for very wide buses. Section V concludes.

II. CODEWORD CORRELATION

A. Overview

Codewords designed to eliminate crosstalk show a highly structured behavior. Codewords with common properties can be classified into various sets. The cardinality of each set can be mathematically determined.

Definition 1: A valid codeword is a bit vector that does not contain a prohibited bit pattern.

Definition 2: Class is a set of all valid codewords that contain a predetermined most significant bit pattern.

Classes are labeled as \((s,n)\) where \(s\) is the most significant bit pattern and \(n\) is the number of code bits. For example, Class \((00,5)\) is a set of all 5-bit valid codewords that have 00 as most significant bit pattern.

Theorem 1: Any \((n+1)\)-bit codeword is valid only if the \(n\)-bit code segments contained within it are valid \(n\)-bit codewords themselves.

Proof: Consider a valid \(n\)-bit codeword, \(x=(b_{n-1},b_2,b_1)\). Since \(x\) is valid, by definition, it does not contain the prohibited bit pattern. Consequently two \((n-1)\)-bit vectors contained within \(x\), namely \(x_1=(b_{n-1},b_2)\) and \(x_2=(b_{n-1},...,b_3,b_1)\) do not contain the prohibited bit pattern. Hence, by definition, both \(x_1\) and \(x_2\) are valid codewords. Similarly, if \(y=(b_{n-2},b_2,b_1)\) is not a valid \(n\)-bit codeword, by definition, it must contain the prohibited bit pattern. As a result \((n-1)\)-bit vectors, namely \(y_1=(b_{n-1},b_{n-2},b_2,b_1)\) and \(y_2=(b_{n-2},...,b_3,b_1,b_0)\), must contain the prohibited bit pattern as well. Hence, by definition, \(y_1\) and \(y_2\) are not valid codewords.

In other words, if \(x\) is not a valid codeword for a given code, then neither \(0x\) nor \(1x\) is a valid codeword. This is a necessary but not sufficient condition. As a result, in order 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 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 an iterative procedure. Starting with set of 1-bit codewords, a set of \(n\)-bit codewords is formed in \(n\) iterations as shown in Fig. 1. Each level of the graph in Fig. 1 corresponds to an iteration of the while loop.

Class \(A\) is a parent of Class \(B\) if some elements in Class \(B\) are formed by adding a leading 1 or 0 to every element in Class \(A\). Also Class \(B\) is a child of Class \(A\). As an example, in Fig. 1 Class \((0,2)\) is a parent Class to Classes \((0,3)\) and \((1,3)\) while a child Class of \((0,1)\) and \((1,1)\). Appending a 0 or 1 is depicted by dotted and solid arrows respectively. Arrow points from parent to child Class. Every codeword within a Class contributes towards forming one or more codewords in the next level. Being a structured 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.

Cardinality of such set \(|X(n)| \geq 2^d\) determines the number of data bits \(d\) that can be transmitted using \(n\)-bit codewords.

Definition 3: In an ordered set of codewords, offset of a codeword is defined as the number of codewords occurring before it within that set.

Definition 4: In an ordered set of codewords, offset of a Class is defined as the offset of the first codeword of that Class.

The first codeword in the ordered set is mapped to smallest data (namely decimal 0), the second codeword is mapped to next smallest data (namely decimal 1) etc. Hence the decimal value of the data determines the offset of corresponding codeword and consequently the Class of codeword in the final level. As the classification is done based on the most significant bits, determination of Class also determines the most significant bit \((n^{th}\) bit) of the codeword. Parent Class of the codeword is determined using cardinality of Classes and the parent-child relationships. Determination of parent Class determines the \(n^{th}\) bit of the codeword. Determination of 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. In particular, a topological traversal of the multilevel codeword correlation graph from bottom to top determines the codeword corresponding to a particular data word. The non enumerative nature of the proposed method makes it scalable for wide buses and suitable for automation.

B. Codeword correlation in the \((n,d,t)\)-NAT code

The code proposed in [8] has been designed as a transition code where a \(t\) indicates a transition (either rising or falling) on a bus line while 0 means a stable value (stable 0 or a stable 1). The \((n,d,t)\)-NAT code is an \(n\)-bit code that transmits \(d\)-bit data and has at most \(t\) 1’s in each codeword. Thus more number of 1’s correspond to increased dynamic power consumption. Authors propose to limit power consumption by limiting the maximum number of 1’s (the weight) to value \(t\). For simplicity of explanation, the code without limitations on maximum weight is considered. The maximum weight of the code can be limited by dividing Classes into subclasses based

Fig. 1. Codeword Correlation Graph
on the weights of the codewords. Due to the nature of the code to avoid any consecutive 1’s, a codeword with maximum weight consists of alternate 1’s and 0’s. Thus the maximum possible weight is $[(n/2)]$. In this paper a simpler notation $(n,d)$-NAT is used instead of $(n,d,[(n/2)])$-NAT. Any further increase in the weight will cause more than one consecutive 1’s thereby forming an invalid codeword. In order to avoid transition on neighboring lines, every pattern with more than one 1’s on neighboring lines is discarded.

To construct a multilevel code correlation graph, consider a set of valid 2-bit codewords $X(2)=\{00,01,10\}$. In order to form valid 3-bit codewords, every element in $X(2)$ must be augmented with either a leading 0 or a 1. Adding a 0 as an MSB does not result in invalid codewords. While adding a 1 however one must check the MSB of the element in $X(2)$. For example appending 1 to 10 will form an invalid pattern 110.

To simplify the procedure further set $X(2)$ is divided into two Classes, namely $(0,2)$ and $(1,2)$. As a result, $X(2) = (0,2) \cup (1,2)$ and $(0,2) \cap (1,2) = \emptyset$. It is clear that a leading 0 can be added to all the elements of set $X(2)$ while a leading 1 can only be added to all elements of Class $(0, 2)$. This is a recursive procedure. Results of first four iterations are presented in Table I. Thus Class $(0,3)$ is a child of $(0,2)$ and $(1,2)$ both, where as $(1,3)$ is a child of $(0,2)$ only. The resulting codeword correlation graph is shown in Fig. 2.

### Table I

| Code Bits $(a)$ | $(0,n)$ | $(1,n)$ | $|0,n|$ | $|1,n|$ | Data Bits $(d)$ | Redundancy $(e)$ |
|----------------|---------|---------|--------|--------|----------------|-----------------|
| 1              | 0       | 1       | 1      | 1      | 2              | 1               |
| 2              | 00, 01  | 10      | 2      | 1      | 3              | 1               |
| 3              | 000, 001, 010, 100, 101 | 3      | 2      | 5      | 2              | 1               |
| 4              | 0000, 0001, 0010, 0100, 0101 | 5      | 3      | 8      | 3              | 1               |

### III. On-chip Implementation of $(n,d)$-NAT Encoder

First $|0,n|$ data words are mapped to all of the elements in the Class $(0,n)$. Remaining data words are mapped to first $(2^d-|0,n|)$ elements of $(1,n)$. Remaining elements of $(1,n)$ are unmapped. For $(4,3)$-NAT code mapping shown in Table II $(|1,n|+|0,n|)=2^d$ hence there are no unmapped codewords.

Encoder essentially traverses the codeword correlation graph depicted in Fig. 2 from bottom to top. It is observed that $X(k)=Fb(k+2)$, $|0,k|=Fb(k+1)$ and $|1,k|=Fb(k)$. Here $Fb(k)$ is the $k^{th}$ Fibonacci number. Starting with the last iteration, comparison of decimal value of the input data $(D_i)$ with $|0,n|$ determines whether the code word belongs to Class $(1,n)$ or $(0,n)$. Hence the $n^{th}$ code bit $C(n)$ is 1 if $D_i \geq Fb(n+1)$, 0 otherwise.

Since parent and child Classes may have different offsets, input data scaling may be required while migrating from a child Class to a parent Class. As seen in Fig. 3, if the codeword belongs to Class $(0,k)$, scaling of input data is not required since both child Class and the first parent Class, namely $(0,k-1)$ have same offset (i.e. zero). Class $(1,k)$ however, has an offset of $Fb(k+1)$ while that of its parent, namely $(0,k-1)$, is zero. As a result in order to direct the encoder to proper parent Class, the input data $D_i$ must be scaled by subtracting $Fb(k+1)$ from it. In other words, if code word belongs to $(1,k)$ then $D_i=D_i-Fb(k+1)$. The functionality of the $k^{th}$ encoder block with input data $D_i$ labeled $E(k, D_i)$ is given in Fig. 4. Block
E(1,D_i) can be eliminated since C(1) is always equal to the input D_i to that block and D_o is always zero.

\[
\text{IF}(D_i > Fb(k+1)) \text{ THEN } \\
\{ C(k)=1; D_o=D_i-Fb(k+1); \} \\
\text{Else} \\
\{ C(k)=0; D_o=D_i; \}
\]

Fig. 4. Functionality of k^{th} Encoder Block

The (4,3)-NAT encoder is shown in Fig. 5. In the example in Fig. 5, input data is 6 (110). Since 6 is greater than 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 is not greater than or equal to 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, code word 1001 is formed. As a result, bus lines at extreme ends undergo transition while middle lines are stable.

For every k^{th} code bit that is set (1 ≤ k ≤ n), Fb(k+1) is subtracted from the input data and for every k^{th} code bit that is reset, 0 is subtracted from the input data. In other words, the 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. If C(i) is the i^{th} code bit and D_i is the input data to the i^{th} decoder block then recovered data D_r from an n-bit codeword is given by:

\[
D_r = \sum_{i=1}^{n} C(i) \ast Fb(i+1)
\]

As an example, 3-bit data broken-up as a sum of products of the i^{th} bit and Fb(i+1) as well as corresponding mapping is shown in Table II. As shown in Fig. 7 array of exclusive or gates translate the transitions on the bus lines to codewords. 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 shown in Fig. 6.

Length of the codeword is determined by the number of function blocks. For example, to encode 4-bit data, code length of 6-bits is required. Therefore 5 function blocks are needed (E(1,D_i) is unnecessary). For very large codes, throughput of the encoder and decoder can be improved by using pipelined architecture. Encoder implementation as a systolic machine is shown in Fig. 8.

IV. EXPERIMENTAL RESULTS

The proposed method was implemented in VHDL for the (n,d)-NAT code. Also the random logic (conventional approach) was implemented in VHDL. These VHDL codes were
Fig. 8. NAT Encoder (Pipelined)

TABLE III

<table>
<thead>
<tr>
<th>Data bits (d)</th>
<th>Gate Count (Conventional)</th>
<th>Gate Count (Proposed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>253</td>
<td>157</td>
</tr>
<tr>
<td>12</td>
<td>3648</td>
<td>348</td>
</tr>
<tr>
<td>16</td>
<td>29375</td>
<td>602</td>
</tr>
<tr>
<td>24</td>
<td>N.A.</td>
<td>1595</td>
</tr>
<tr>
<td>32</td>
<td>N.A.</td>
<td>2828</td>
</tr>
<tr>
<td>48</td>
<td>N.A.</td>
<td>6424</td>
</tr>
<tr>
<td>64</td>
<td>N.A.</td>
<td>11531</td>
</tr>
<tr>
<td>96</td>
<td>N.A.</td>
<td>26120</td>
</tr>
<tr>
<td>128</td>
<td>N.A.</td>
<td>47124</td>
</tr>
</tbody>
</table>

then synthesized using Cadence Encounter RTL Compiler tool using generic technology and circuit statistics were analyzed.

Table III and Fig. 9 show the hardware overhead of implementation of (n,d)-NAT code for both proposed and conventional method. Table III clearly shows that conventional approach is limited to 16-bit buses as the RTL compiler was unable to synthesize due to memory requirement violation. It also shown that under identical settings the proposed method easily extends beyond 128-bits. Hardware overhead of a 96-bit bus encoder implemented using the proposed strategy is less than that of a 16-bit bus encoder using conventional approach.

Overhead of the proposed method exhibits scalable behavior as opposed to the exponential behavior of conventional method. Conventional method also exhibits a considerable computational overhead and consequently time required for code generation as well as synthesis of the encoder/decoder circuitry. The complexity of the code book in the conventional method results in drastically increased synthesis time as observed in Fig. 11. Fig. 10 lists the HDL file size. Fig. 12 gives power consumption estimated by Encounter RTL compiler for both proposed and conventional method.

Fig. 10, 11 and 12 exhibit similar behavior as Fig. 9. In other words, using proposed method, the hardware overhead, synthesis time and memory requirement of the design automation tool reduce considerably. For simplicity, the adders used in the encoder blocks are synthesized as ripple carry adders. However, in practice it is advisable to use faster adders like carry look-ahead adder. The circuit of the proposed method may also be trivially pipelined to improve the throughput.

V. CONCLUSION

The conventional strategy for encoder implementation, that maintains a code book is not scalable for wide buses. The proposed implementation strategy transforms the encoder into a structural model that uses adders and multiplexers as building blocks. This requires that any codeword can be obtained from a codeword of smaller length. Such correlations are represented by a correlation graph.

It has been shown that the model can be easily scaled to accommodate very wide buses. The proposed strategy improves area requirements by eliminating need to partition the bus using shield lines and by reducing the gate count of encoder as compared to the conventional random logic approach. Other
This paper presents the implementation of only one code due to space limitations. The journal version of this work will present the proposed method in a parameterized manner such that, given a particular code, the generation of the codeword correlation graph and the on-chip implementation can be done in an automated way.

REFERENCES


advantages include reduced memory requirement for the RTL compiler and less computational overhead.