Bus Encoding to Prevent Crosstalk Delay

Bret Victor and Kurt Keutzer
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{bret, keutzer}@eecs.berkeley.edu

Abstract— **The propagation delay across long on-chip buses is increasingly becoming a limiting factor in high-speed designs. Crosstalk between adjacent wires on the bus may create a significant portion of this delay. Placing a shield wire between each signal wire alleviates the crosstalk problem but doubles the area used by the bus, an unacceptable consequence when the bus is routed using scarce top-level metal resources. Instead, we propose to employ data encoding to eliminate crosstalk delay within a bus. This paper presents a rigorous analysis of the theory behind "self-shielding codes", and gives the fundamental theoretical limits on the performance of codes with and without memory. Specifically, we find that a 32-bit bus can be encoded with 40 wires using a code with memory or 46 wires with a memoryless code, in comparison to the 63 wires required with simple shielding.**

1. Introduction

As device geometries shrink, chip sizes increase, and clock speeds get faster, interconnect delay is becoming increasingly significant. In particular, the propagation delay through long cross-chip buses is already proving to be a limiting factor in the speed of some designs, and this trend will only get worse. It has been shown that the delay through a long bus is strongly a function of the coupling capacitance between the wires. Especially detrimental to the delay is the Miller-like effect when adjacent wires simultaneously transition in opposite directions. When the cross-coupling capacitance is comparable to or exceeds the loading capacitance on the wires, the delay of such a transition may be twice or more that of a wire transitioning next to a steady signal. We call this delay penalty the "crosstalk delay".

In some high-speed designs where crosstalk delay would have limited the clock speed, the technique of shielding was used. This involves putting a grounded wire between every signal wire on the bus. Although this certainly is effective in preventing crosstalk within the bus, it has the effect of doubling the wiring area. Cross-chip buses often must be routed in higher metal layers, which are scaled more slowly than the rest of the geometry in order to prevent an unacceptable increase in resistance. Thus, routing resources are scarce at these levels, and it can be difficult to justify doubling the bus width.

However, if we abstract the concept of shielding and just look at the signals on the wires of a shielded bus, we can think of it as a very simple bus encoding. Two wires are used for every data bit. A data bit of "0" is encoded as a "00" signal on the wires, and a "1" is encoded as "10". The purpose of this "encoding" is to prevent adjacent wires from transitioning in opposite directions, and this particular encoding achieves that goal by forcing every other wire to a steady value. But the question arises: *Are there other possible encodings that can achieve the same goal, but with fewer wires*? Such encodings may require extra logic or memory elements, but as the speed of logic goes up and the relative area consumed by logic goes down, such a tradeoff seems increasingly valid.

Indeed, such encodings exist. We will refer to them as "self-shielding" or "crosstalk-immune" codes. In this paper, we will approach this subject from a rigorous theoretical standpoint. Rather than giving ad-hoc designs and heuristic methods, or prematurely attempting to design for efficient implementation, we will instead develop the theory behind crosstalk-immune coding, describe the fundamental capabilities and limitations that the theory implies, and give methods for generating optimal sets of codewords. Such an analysis is always necessary before good codes can be designed and implemented.

2. Background

We can model the chain of communication as shown in Figure 1. Adopting some terminology from coding theory, we say that the data words to be encoded are represented by *symbols*. The mapping between symbols and actual data words is an implementation step and will not be discussed here. The values placed on the channel by the encoder are called *codewords*, and the mapping between symbols and codewords is called a *codebook*. If the codebook changes with time, then the encoding is said to have *memory*.

Specific to crosstalk-immune coding is the notion of which codewords can follow which. The fundamental rule is

Figure 1: Model of Communication Chain

codeword at time 1:	0010	0000	0100	0100
	↓	↓	↓	↓
codeword at time 2:	0110	1111	0001	0010
	valid	*valid*	*valid*	**invalid**

Figure 2: Examples of valid and invalid transitions

that, given a particular value currently on the channel, the next value cannot cause any adjacent wires to transition in opposite directions. We say that a codeword is *connected* to another codeword if it is valid to transition from one to the other. Figure 2 presents some examples of valid and invalid transitions. In order to import some terminology from graph theory, we can form a graph with the codewords as vertices and the connections as edges. This graph is undirected because the connection relation is symmetric. We can then say that the *neighbor set* of a codeword is the set of codewords that it is connected to, and its *degree* is the size of this set. Note that it is valid for a codeword to transition to itself, and thus every codeword has itself as a neighbor.

3. Unpruned Code with Memory

All of the codes we discuss will transmit one full data word on each clock cycle. If the data word is b bits wide and all data words are allowed, then at all times there must be at least 2^b symbols that can be expressed, and thus at least 2^b codewords in the codebook. If our code has memory, however, these do not have to be the *same* codewords at all times. The codebook could be any subset of the neighbor set of the codeword currently on the channel. That is, for every codeword, a mapping is defined between the symbol set and some subset of the codeword's neighbor set. This mapping is known by both the encoder and decoder, and thus when the channel transitions to a particular neighbor, it represents a particular symbol and information is transmitted.

For our first code, the "unpruned code with memory", we will make no restrictions on which values are allowed to be in the codebook. That is, we will assume that any possible n-bit value could be a codeword and thus could be on the channel. The maximum number of expressible symbols is limited by the least connected value, or the codeword with the smallest degree. The first step, then, is to derive a formula for calculating the degree of a codeword. We can then find and prove which codeword has the minimum degree for a given width n. This degree will represent the maximum performance available from this type of code.

Definition: A *class 1 codeword* is a codeword with alternating 0 and 1 bits. For example, 01010 and 10101 are 5-bit class 1 codewords.

Definition: d_n is the degree of an n-bit class 1 codeword.

n	d_n	$\log_2(d_n)$
1	2	1.00
2	3	1.58
3	5	2.32
4	8	3.00
5	13	3.70
6	21	4.39
7	34	5.09
8	55	5.78
9	89	6.48

Table 1: Degrees of Some Class 1 Codewords

Theorem 1: *d_n are Fibonacci numbers. Specifically:*

$$d_n = F_{n-2} \tag{1}$$

where F_n is the classical Fibonacci sequence {1,1,2,3,5,8,13,…}.

Proof: Consider, without loss of generality, a class 1 codeword of n bits that begins with a 0 bit. This first bit can either stay or rise. If it stays, the second bit is free to fall or stay, and thus d_{n-1} transitions can be realized. If the first bit rises, the second bit is forced to stay, because it cannot fall next to a rising bit. The third bit is then free to rise or stay, and d_{n-2} transitions can be realized. The total number of possible transitions is the sum of the two cases:

$$d_n = d_{n-1} + d_{n-2} \tag{2}$$

This is the same recurrence relation obeyed by the Fibonacci sequence. In order to show that d_n are in fact Fibonacci numbers, we need to establish two initial conditions. A one-bit class 1 codeword is "0". It can transition to two codewords: "0" and "1". A two-bit class 1 codeword is "01". It can transition to "00", "01", or "11", but *not* "10". We see that $d_1 = 2$ and $d_2 = 3$. These are in fact Fibonacci numbers, F_3 and F_4 respectively. Therefore, $d_n = F_{n-2}$. ❑

Corollary:

$$d_n = \frac{1}{\phi + 1/\phi}\left(\phi^{n+2} - (-\phi)^{-(n+2)}\right), \quad \phi = \frac{1+\sqrt{5}}{2} \tag{3}$$

$$d_n = \begin{cases} \dfrac{2}{\phi + 1/\phi}\cosh\big((n+2)\ln(\phi)\big), & \text{odd } n \\ \dfrac{2}{\phi + 1/\phi}\sinh\big((n+2)\ln(\phi)\big), & \text{even } n \end{cases} \tag{4}$$

Proof: These expressions can be derived by solving the difference equation (2) and applying the initial conditions in the above proof. ❑

Some values of d_n are given in Table 1. Now that we have an expression for the degree of a class 1 codeword, we will proceed to derive the degree of any arbitrary codeword.

Definition: An *independent boundary* in a codeword occurs between two adjacent bits of the same value. A *dependent boundary* in a codeword occurs between two adjacent bits of different values. For example, the codeword 0011 has three boundaries, and they are independent, dependent, and independent respectively.

Definition: A *section* of a codeword is one of the pieces that would result if the codeword were split at its independent boundaries. For example, the codeword 10110100 has three sections: 101, 1010, and 0. Notice that each section , if isolated, would be considered a class 1 codeword.

Definition: The *class* of a codeword is equal to the number of sections. This is also the number of independent boundaries plus one.

Definition: $d_{\{n_1, n_2, \ldots, n_c\}}$, where c is the class, denotes the degree of a codeword with sections of width n_1, n_2, etc.

Theorem 2: *The degree of any codeword is equal to*

$$d_{\{n_1,n_2,\ldots,n_c\}} = \prod_{i=1}^{c} d_{n_i} \tag{5}$$

where c *is the codeword class and* n_i *is the number of bits in the* i*th section.*

Proof: Two adjacent sections, by definition, are separated by an independent boundary. The two bits across this boundary are the same value, and thus it is impossible for one to rise and the other fall. Because the fundamental rule on bit transitions cannot be violated across an independent boundary, the set of transitions allowed for one section is not affected by the transitions chosen for other sections. Thus, we can determine the number of transitions for each section independently, and multiply the results from each section to obtain the total number of transitions allowed. By definition, each section is in isolation a class 1 codeword. Thus, the total number of transitions is the product of d_{n_i}, where n_i is the width of each section. ❑

We now know how to calculate the degree of any codeword. In the next theorem, we will find that the codewords with the smallest degree, which determine the performance of our code, are none other than the class 1 codewords.

Theorem 3: *For a given codeword width* n*, the degree of any codeword of class* c *> 1 is greater than* d_n*.*

Proof: We start by proving this for $c = 2$. Using Theorem 2, the proposition can be stated mathematically as:

$$d_{x+y} < d_x\, d_y \tag{6}$$

Restating this in terms of (3) and applying algebraic transformations (as well as using the fact that $\phi^n = \phi^{n-1} + \phi^{n-2}$) we can reduce the inequality to:

$$\left(\phi^x - (-\phi)^{-x}\right)\left(\phi^y - (-\phi)^{-y}\right) > 0 \tag{7}$$

When $x > 0$ and $y > 0$, this inequality is true. Thus, for a given n, a class 2 codeword has a higher degree than a class 1 codeword. The expression can be applied iteratively for higher classes:

$$d_{x+y+z} < d_{x+y}\, d_z < d_x\, d_y\, d_z \tag{8}$$

and so on. Thus any codeword with class $c > 1$ has a higher degree than a class 1 codeword. ❑

We have found that the minimum codebook size occurs when a class 1 codeword is on the channel, and at that point, the number of symbols that can be expressed is given by (3). Given this information, we can now state the maximum performance of a self-shielding code when any possible value is allowed as a codeword on the channel. If n is the channel width, then the maximum number of information bits is $\log_2(d_n)$. This is plotted in Figure 3. Asymptotically, increasing n by 1 multiplies the number of symbols by ϕ, or 1.62. So, adding a physical wire allows for about $\log_2(1.62)$, or 0.69, more bits of information. We see that a 32-bit bus could be implemented with 46 wires. This compares very favorably to a simple shielding scheme which would require 63 wires. We can conceptually consider $(n-b) / b$ to be the inflation in wire usage due to eliminating crosstalk delay. With shielding, this wiring overhead is 97%, whereas with coding, it is only 44%. However, we can do even better than this.

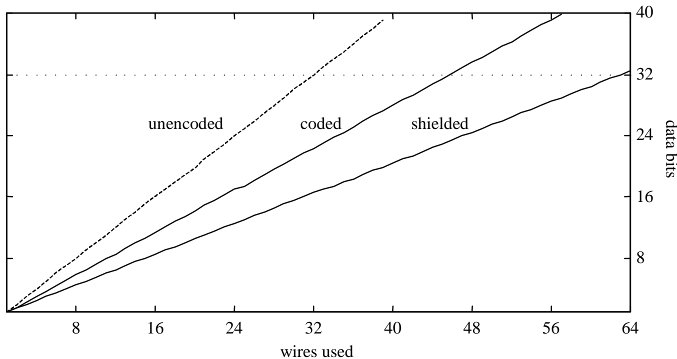


Figure 3: Performance of Unpruned Code

4. Pruned Code with Memory

The previous result was limited by the degree of the class 1 codeword. However, if we ensure that the code never transitions *to* a class 1 codeword, then we know it will never need to transition *from* one. If we throw the class 1 codewords out of the codebook, we are no longer limited by their poor performance, although the degrees of the codewords in the discarded codes' neighbor sets will decrease. Extending this idea leads to the *codebook pruning algorithm*:

Algorithm 1:

```
While there are valid codewords left:
    Find the set of valid codewords with the lowest degree.
    For each codeword W in the set:
        Remove W from the set of valid codewords.
        Decrement the degree of each of W's neighbors.
```

As the algorithm progresses, the limiting degree will increase, hit a maximum, and then decrease as the codebook gets depleted. We choose, of course, the set of codewords that was active when the limiting degree was at its maximum. This algorithm is guaranteed to find the best possible set of codewords because at each step, if anything other than the limiting codewords were removed, the limiting degree could only decrease or stay constant. It could never improve.

To visualize the pruning process, we can make a plot of the limiting degree versus codewords pruned as the algorithm runs. Figure 4 shows these "pruning curves" for a few values of n. The expected shape can be observed. By locating the peak of each curve, we find the maximum performance obtainable by a self-shielding code of the given width.

Unfortunately, pruning as described by the above algorithm is a purely experimental procedure. It is extremely computationally intensive for large n (we could only run the algorithm up to $n = 23$), and worse, it is not amenable to the rigorous mathematical analysis that we seek. However, there is a sub-optimal pruning algorithm that is a fairly close approximation to the optimal one and allows analytic expressions to be derived.

Examination of the optimal pruning process reveals that, especially for small n, the codewords are pruned roughly in class order. That is, most of a given class c is pruned before any codeword in class $c + 1$ is touched. This observation is less true

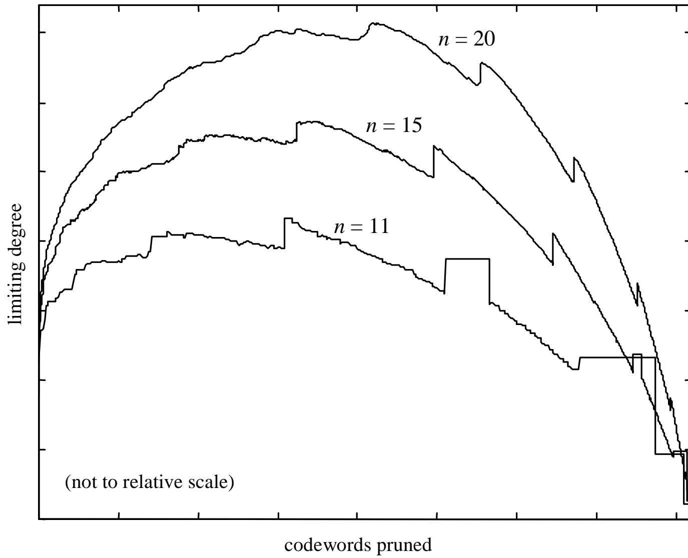


Figure 4: Pruning Curves

for large n (e.g., $n > 16$), but the dependence on class is still strongly visible. In fact, the spikes that are visible in the pruning curves usually occur after an entire class of codewords is fully pruned.

This observation leads to the idea of pruning entire codeword classes at once. The revised pruning algorithm can be written as follows:

Algorithm 2:

For each c from 1 to n:

Remove class c codewords from the set of valid codewords.

Recalculate the degrees of the rest of the codewords.

Again, the limiting degree will rise, hit a maximum, and fall. There is no guarantee of optimality with this algorithm. However, we find that its results match exactly with the optimal for $n < 10$, and the error stays below 10%, or 0.15 bits, for at least $n \leq 23$, which was the highest we could check. Thus, the approximation is fairly good. Furthermore, because it is sub-optimal, the results are always achievable.

Again, the primary motivation behind this algorithm, other than the simplicity of having logarithmically less data points to deal with, was that it can be subjected to mathematical analysis. This is possible through the use of the *class distribution polynomial*. For a given codeword, a polynomial $D(x)$ can be generated where the number of class c codewords in the neighbor set is equal to the coefficient of the x^{c-1} term.[1] Thus, this polynomial describes the class distribution of the neighbor set, and can be used to calculate the effects of pruning various code classes. Specifically:

$$D_{\{n_1,n_2,\ldots,n_c\}}(x) = \frac{1}{x+1}\begin{bmatrix}1 & 1\end{bmatrix}\left(\prod_{i=1}^{c} M_{n_i}(x)\right)\begin{bmatrix}1\\1\end{bmatrix} \quad (8)$$

where

$$M_n(x) = \begin{bmatrix} xP_n & P_n + x^2 P_{n-1} \\ P_n + (x^2+1)P_{n-1} & xP_n + xP_{n-1}\end{bmatrix} \quad (9)$$

$$P_n(x) = \sum_{j=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \binom{n-j-1}{j} x^{2j} = P_{n-1}(x) + x^2 P_{n-2}(x) \quad (10)$$

with $P_0 = 0$ and $P_1 = 1$.

The derivation of these expressions, as well as a discussion of the many interesting results that they imply, is beyond the scope of this paper. We will simply state that it is possible to find which codeword is the limiting one in each class, and the determination of its degree after pruning is a trivial matter of generating the class distribution polynomial and summing the coefficients of the x^a terms for $a \geq c$-1.

With the experimental results from the optimal pruning algorithm, it is fair to say that we have determined the fundamental limits on the performance of self-shielding codes up to $n = 23$. For larger n, we can use the class pruning algorithm in lieu of the optimal one, for an apparently close approximation to the fundamental limit. Notice that, unlike the unpruned code where the code design was left completely arbitrary, these algorithms provide a specific set of codewords to use. The extra performance over the unpruned code comes from restricting ourselves to this set.

Figure 5 plots the maximum performance of the pruned code with memory, using the analytic algorithm. The data from the optimal algorithm would visibly coincide with the plotted line for $n \leq 23$, so it is not plotted separately. We see that a 32 bit bus could be implemented with only 40 wires. In this case, the wiring overhead as defined earlier is only 25%, which compares extremely favorably to 44% with unpruned coding and 97% with simple shielding.

5. Memoryless Code

The previous codes required the encoder and decoder to hold state, because the codebook was dependent on the previous value on the channel. Now, we ask what kind of performance is possible with a memoryless code. Such a code would have a single, unchanging codebook. Thus, every codeword in the book would have to be able to transition to every other codeword. We want to find the largest such codebook.

In graph theory, a *clique* in an undirected graph is

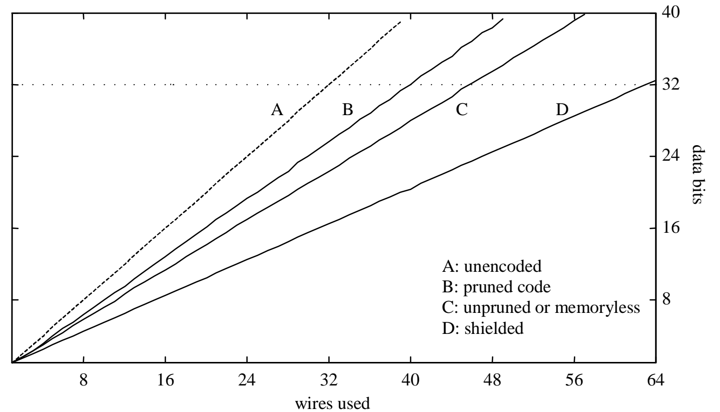


Figure 5: Performance of Code Types

[1] Note that $D_W(1) = d_W$. Thus, this can be seen as a generalization of the previous theorems.

defined as a subgraph where every pair of nodes is connected with an edge. If we represent the valid transitions of the codewords as edges on a graph, then the problem of finding the largest memoryless codebook becomes the problem of finding the largest clique. Interestingly, evaluating this clique problem leads to results identical to Table 1. The size of the largest clique is always d_n.

In the next two theorems, we will show that for a given n, there are two identically-sized largest cliques, each consisting of the entire neighbor set of one of the two class 1 codewords. Theorem 4 will prove that the neighbor set of a class 1 codeword is a clique, and Theorem 5 will prove that there is no larger clique.

Theorem 4: *The entire neighbor set of a codeword is a clique if and only if the codeword is class 1.*

Proof: *("if" case):* A class 1 codeword has only dependent boundaries, so the pair of bits across any boundary is either 01 or 10. Consider a boundary between a 01 pair. All neighbors of the codeword will have either a 00, 01, or 11 across that boundary. Notice that all three possibilities can transition to one another. Therefore, every neighbor can transition to every other neighbor without violating the bit transition rule across that boundary. Consider now a boundary between a 10 pair. All neighbors have either 00, 10, or 11 across this boundary, and again, these three pairs can all transition to each other. This argument holds for every boundary in the codeword. The bit transition rule cannot be violated across any boundary when any neighbor transitions to any other neighbor, so the neighbor set of a class 1 codeword is a clique

("only if" case): A codeword in class $c > 1$ has, by definition, at least one independent boundary. Consider the pair of bits across an independent boundary, either 00 or 11. There are neighbors of this codeword with 00, 01, 10, and 11 bit pairs across this boundary. However, the neighbors with 01 across the boundary cannot transition to the neighbors with 10 across the boundary. Thus, the neighbor set of a codeword in class $c > 1$ is not a clique. ❑

Definition: A clique is said to be *prime* if there is no codeword that can be added to the set with the set remaining a clique. (It does not imply that it is the largest clique; it simply means that it has no room to grow.)

Theorem 5: *There is no clique larger than d_n.*

Proof: (*sketch*) First, we will enumerate all possible prime cliques. It can be shown that every bit boundary in a prime clique is either "01-type" or "10-type". To say a boundary is 01-type implies that in the clique, there are codewords with 00, 01, and 11 across that boundary, but no codewords with 10 across the boundary. Similarly, across a 10-type boundary, codewords in the clique may have 00, 10, and 11, but not 01. Since each of the n-1 boundaries in a prime clique can be one of two types, there are 2^{n-1} prime cliques. It can be shown that the number of codewords in a given prime clique can be calculated using the following algorithm:

Algorithm 3:

$x = y = 1$

For each boundary, from 1 to n - 1:

If the boundary is 01-type, $y = x + y$

If the boundary is 10-type, $x = x + y$

Prime clique size = $x + y$

We wish to construct a clique to maximize $x + y$ at the end of the algorithm. It can be seen that the optimal decision at each step of the algorithm, in order to maximize the running total of x and y, is to place a 01-type when $x < y$ and a 10-type when $x > y$. The choice of the first pair type is arbitrary, because at that point, $x = y$. Thereafter, the optimal choice alternates between the two types, and thus the largest clique consists of alternating 01-type, 10-type boundaries. We see that a class 1 codeword is a member of this clique (with the polarity of the class 1 codeword determined by the choice of the initial boundary). By Theorem 4, the size of this largest clique is d_n. ❑

We have now determined the maximum performance of a memoryless self-shielding code: $\log_2(d_n)$ bits per wire. Notice that this code, like the pruned code with memory, comes with a specific set of codewords to use. The sets found by the pruning algorithms provide additional performance beyond d_n, whereas the set found in this section (the neighbor set of a class 1 codeword) gives no additional performance, but instead provides a code property that may considerably ease implementation.

Because the maximum performance of a memoryless code is the same as that of an unpruned code with memory, the results can be viewed in Figure 5. Again, adding a physical wire allows for 0.69 more information bits, and a 32-bit bus would require 46 wires. But the encoder and decoder could be purely combinational, and there would be a single, fixed codebook.

6. Implementation Issues

Although the primary thrust of this paper is theoretical, we will now take a brief look at some issues related to the

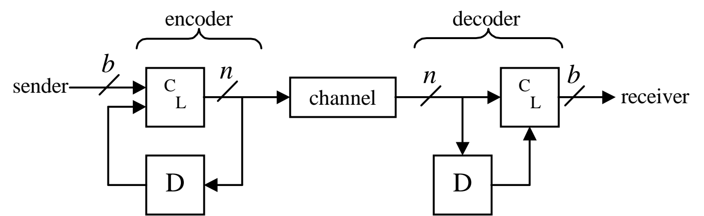


Figure 6a: Unpipelined Circuit Model for Code with Memory

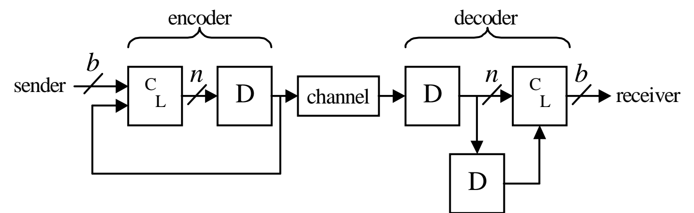


Figure 6b: Pipelined Circuit Model for Code with Memory

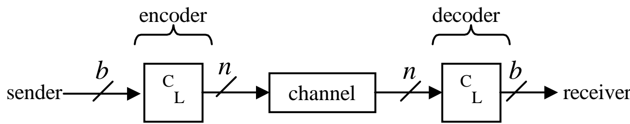


Figure 7a: Unpipelined Circuit Model for Memoryless Code

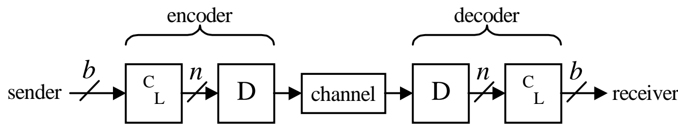
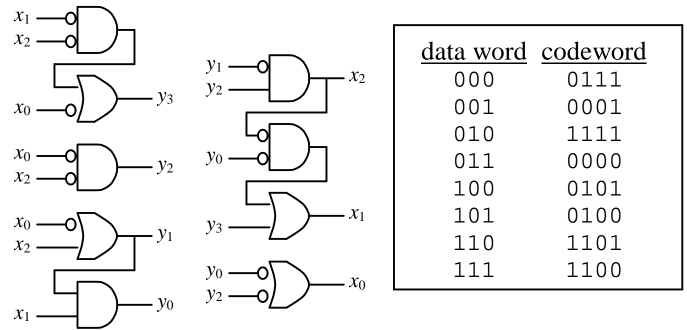


Figure 7b: Pipelined Circuit Model for Memoryless Code



data word	codeword
000	0111
001	0001
010	1111
011	0000
100	0101
101	0100
110	1101
111	1100

Figure 9: Example 4-Bit to 3-Wire Coder

logical and physical implementation of self-shielding codes.

Encoder and Decoder Circuit Models

Figure 6a shows a block diagram of an encoder and decoder that can implement a self-shielding code with memory. It is easy to recognize the encoder as a simple finite state machine (a Mealy machine), and the decoder is a function only of the current and immediately previous input, with no feedback at all. However, with this architecture, a combinational path exists from the input of the encoder to the output of the decoder, which adds the logic delay to the delay of the channel. A pipelined circuit model, shown in Figure 6b, gives the data almost the full clock cycle to travel across the channel, in exchange for two extra clock cycles of latency. The encoder is now a Moore machine, and the decoder uses two memory elements.

Figures 7a and 7b show unpipelined and pipelined circuit models respectively for a memoryless coder. Because a memoryless code depends only on the current input, these models are almost trivially simple.

Partial Coding

The results in the previous sections give the theoretical maximum performance for a code of a given width. However, it may be infeasible to design a circuit to encode 32 or more bits of data at once. In such a case, the bus can be broken into sub-buses of smaller width which could be encoded individually onto sub-channels. Each sub-channel would then have to be shielded from its neighbor with a dedicated ground wire. But it should be noted that, in practice, such a wire might be needed anyway simply as a current return path.

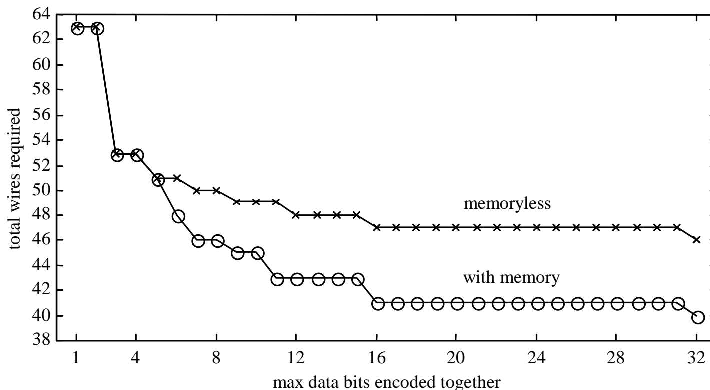


Figure 8: Wires Required with Partial Coding

Figure 8 shows the number of wires required, including shield wires, for encoding a 32-bit bus when the individual sub-buses are no wider than a given number of bits. Note that the number of wires required drops off sharply even when the sub-buses are small. For example, a very simple 3-bit to 4-wire memoryless code requires a channel width of only 53 wires. A code with memory using one 4-bit and four 7-bit sub-buses requires only 46 wires.

Design Example

Figure 9 gives gate-level schematic diagrams of a sample encoder and decoder that implement a 3-bit to 4-wire memoryless code. The mapping between data words and codewords is shown as well. Notice that, indeed, the set of codewords used is the neighbor set of a class 1 codeword. Using the partial coding technique described above, an array of ten of these simple coders could be used to implement a crosstalk-immune 32-bit bus with 53 wires. When compared to a 63-wire shielded channel, this amounts to cutting ten wires from the channel for the cost of a handful of gates.

7. Comparison to Other Techniques

In the literature, there are a number of other techniques designed for combating crosstalk. Many of them, such as those described in [1], [2] and [3], employ creative routing strategies in order to minimize crosstalk delay within a datapath or logic block. Our technique. on the other hand, is intended for use with long, straight buses, and thus these routing schemes are not applicable to our domain of interest. [4] and [5] mention some techniques that are more relevant, such as skewing the timing of signals on adjacent wires, interleaving mutually exclusive buses, and precharging the bus. However, skewing requires careful, technology-dependent circuit design and brings up tricky timing issues, whereas our technique is technology-independent and fully synchronous, with the crosstalk immunity “correct by construction.” Interleaving is a useful technique, but it cannot be used with buses that are allowed to transition on any and every clock cycle. Precharging a long bus can incur detrimental power costs, and is usually not an option.

Probably the most common technique is simply using large repeaters to drive the Miller capacitance through brute force [6]. A quantitative comparison between our technique and optimally-sized repeaters is technology- and implementation-

dependent, and will not be given. However, conceptually, using large repeaters is a power-hungry technique, and shielding is an area-hungry technique. Crosstalk-immune bus encoding avoids crosstalk delay with a modest impact on either area or power.

8. Conclusion

In this paper, we have introduced the concept of using data encoding to mitigate crosstalk delay on buses, and we presented a theoretical framework for understanding crosstalk-immune coding. We determined the fundamental limits on performance, in terms of required channel width versus data bits, for codes with and without memory, and found them to be very satisfactory. Future work will include designing codes such that the coding circuitry can be implemented efficiently and exploring hybrid code designs. The latter would involve codes that eliminate crosstalk delay as well as reduce average power consumption, perform error detection or correction, or accomplish some other task that is well suited to bus encoding.

References

[1] A. Vittal and M. Marek-Sadowska, “Crosstalk Reduction for VLSI,” *IEEE Trans. Computer-Aided Design*, vol. 16, no. 3, 1997.

[2] T. Gao and C. L. Liu, “Minimum Crosstalk Channel Routing,” *IEEE Trans. Computer-Aided Design*, vol. 15, no. 5, pp. 465-74, 1996.

[3] T. Xue, E. Kuh, and D. Wang, “Post Global Routing Crosstalk Synthesis,” *IEEE Trans. Computer-Aided Design*, vol. 16, no. 12, pp. 1418-30, 1997.

[4] J. Yim and C. Kyung, “Reducing Cross-Coupling among Interconnect Wires in Deep-Submicron Datapath Design,” *Proceedings. 1999 Design Automation Conference,* pp. 485-90. 1999

[5] K. Hirose and H. Yasuura, “A Bus Delay Reduction Technique Considering Crosstalk,” *Proceedings. Design, Automation and Test in Europe Conference and Exhibition 2000*, pp. 441-5. 2000.

[6] D. Li, A. Pua, P. Srivastava, and U. Ko, “A Repeater Optimization Methodology for Deep Sub-Micron, High-Performance Processors,” *Proceedings. International Conference on Computer Design, VLSI in Computers and Processors*. pp. 726-31. 1997