A Delay-Insensitive Bus-Invert Code and Hardware Support for Robust Asynchronous Global Communication*

Melinda Y. Agyekum Steven M. Nowick
Department of Computer Science
Columbia University
New York, NY, 10027
Email: {melinda, nowick}@cs.columbia.edu

Abstract. *A new class of delay-insensitive (DI) codes, called DI Bus-Invert, is introduced for timing-robust global asynchronous communication. This work builds loosely on an earlier synchronous bus-invert approach for low power by Stan and Burleson, but with significant modifications to ensure that delay-insensitivity is guaranteed. The goal is to minimize the average number of wire transitions per communication (a metric for dynamic power), while maintaining good coding efficiency. Basic implementations of the key supporting hardware blocks (encoder, completion detector, decoder) for the DI bus-invert codes are also presented. Each design was synthesized using the UC Berkeley ABC tool and technology mapped to a 90nm industrial standard cell library. When compared to the most coding-efficient systematic DI code (i.e. Berger) over a range of field sizes from 2 to 14 bits, the DI bus-invert codes had 24.6 to 42.9% fewer wire transitions per transaction, while providing comparable coding efficiency. In comparison to the most coding-efficient non-systematic DI code (i.e. m-of-n), the DI bus-invert code had similar coding efficiency and number of wire transitions per transaction, but with significantly lower hardware overhead.*

1 Introduction

As digital systems grow in complexity, the challenges of design reuse, scalability, power and reliability continue to grow at a rapid pace [15]. These parameters are expected to become major unsolved bottlenecks in less than a decade. A major focus of recent strategies for organizing such systems is the use of networks-on-chip (NoCs), which support the orthogonalized development of computation blocks (e.g., cores) and the communication fabric.

One promising direction is to use asynchronous global communication, to provide flexibility in system integration, as well as dynamic power which adapts on demand to the current traffic. Such systems can be entirely asynchronous, or a hybrid of synchronous computation blocks interconnected by asynchronous channels, thus forming a globally-asynchronous locally-synchronous (GALS) system [20]. Delay-insensitive codes [24] are especially promising for a timing-robust communication methodology, since they gracefully tolerate arbitrary skew in the arrival of individual bits.

The contribution of this paper is a new class of delay-insensitive codes, called *DI Bus-Invert*. The goal is to minimize the average number of wire transitions per transaction (a metric for dynamic power), while maintaining good coding efficiency (number of bits per wire). An additional goal is to maintain manageable hardware overheads. Two simpler new delay-insensitive codes, called *Hybrid* and *Berger Bus-Invert*, are first introduced. Each code is constructed using a distinct strategy, which are then combined together to form the final proposed DI Bus-Invert code.

To our knowledge, the DI Bus-Invert code is the first approach to migrate the core Stan and Burleson bus-invert technique to delay-insensitive communication. As a first cut, the proposed approach uses a *4-phase* (i.e. *return-to-zero (RZ)*) communication protocol, which has been widely used in both research [4, 22] and in recent commercial asynchronous designs such as by Philips Semiconductors [10], Fulcrum Microsystems [11], Silistix [3] and Achronix [21]. This work can also be a springboard for developing a bus-invert method for a *2-phase* (or *non-return-to-zero (NRZ)*) protocol [9, 12] in the future. (These terms will be introduced in detail in Section 2.)

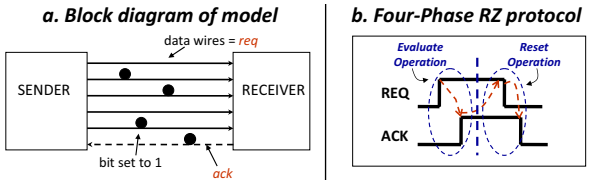

Figure 1. Asynchronous Communication

The earlier synchronous bus-invert approach by Stan and Burleson [18] selectively inverts datawords for optimal power, but it violates delay-insensitivity properties, and therefore cannot be used for the proposed application. A straightforward extension of the Stan/Burleson code to support delay-insensitivity can be easily obtained, as illustrated with the proposed "Berger Bus-Invert" code. However, this solution is shown to be inadequate, with significant degradation in coding efficiency (1-2 extra bits) and mixed results on average number of wire transitions per transaction (from moderate degradation of 2.0 to 14.3% to small improvements of 4.1-15.1%). In contrast, the final proposed "DI Bus-Invert" code uses a more sophisticated approach to creating the appended check field, resulting in roughly comparable coding efficiency to the most optimal DI code (i.e. Berger) yet with significant reduction in the average number of wire transitions (24.6 to 42.9%).

The new approach makes several fundamental modifications over the Stan/Burleson approach: (i) an additional field is appended to ensure delay-insensitivity; (ii) a return-to-zero protocol is used, where all wires are reset to 0 between transactions; (iii) as a result, each dataword has a unique encoding, where in the Stan and Burleson approach there are 2 possible encodings per dataword, depending on the encoding of the previous transmitted codeword; (iv) the new field is directly optimized for a cost metric of reduced wire transitions per transaction, by exploiting the partial order properties of DI codes.

2 Background and Related Work

Asynchronous Communication. The paper assumes a point-to-point asynchronous communication channel [4, 24] between a sender and a receiver.

(a) Asynchronous Communication Channels. Fig. 1(a) gives an example of point-to-point asynchronous communication. Abstractly, the sender provides a request output signal (*REQ*) to the receiver; the receiver in turn provides an acknowledgment input signal (*ACK*) to the sender. If the sender passes actual data to the receiver (rather than providing simple control synchronization), the *REQ* is typically replaced by the encoded data, as shown in the figure. The *ACK* indicates data has been received by the receiver and new data can be sent [24].

(b) Four-Phase Communication Protocol. The most widely-used protocol is *four-phase* or *return-to-zero (RZ)* [4, 24]. As illustrated in Fig. 1(b), the protocol has two operations: (1)

*This work was supported by NSF Award No. CCF-0811504 and by a 2008-2009 Intel Foundation PhD Fellowship.

evaluate and (2) reset. During the evaluate operation, the sender first indicates the start of an event by issuing a rising *REQ+* to the receiver. Once the data has been received, the receiver asserts an *ACK+*. The reset operation then begins: the sender de-asserts the *REQ-* and in turn, the receiver de-asserts its *ACK-* which is the final event of both the reset stage and the four-phase transaction.

(c) Delay-Insensitive Codes. When asynchronous communication is used, as shown in Fig. 1(a), data must be suitably encoded so that the receiver can identify when a packet has been received. Delay-insensitive (DI) codes [2, 24] (i.e. *unordered codes*) are used for this purpose [6].[1] In an asynchronous system, these codes have an inherent timing-robustness, where data can arrive in any order and at any time, and with arbitrary skew between its bits. Their key property is that no valid codeword is "covered" by another valid codeword. More specifically, a codeword y is covered by codeword x if the 1 bits of y are a subset of the 1 bits of x. A pair of codewords is unordered if x does not cover y and y does not cover x.

There is a direct relationship between this covering property and delay-insensitivity: assuming that the communication channel is reset to all-0 (i.e. spacer) between transmissions, then the receiver can unambiguously identify the arrival of a new valid codeword. In particular, as individual 1 bits arrive (i.e. rising signal transitions), it is never the case that another valid codeword will be seen transiently during the transmission, since the 1-bit pattern of each codeword is not covered (i.e. not a subset) by any other. Formal definitions of covered and unordered are as follows [7]:

Definition 1 (*Covering Relation) A codeword* $x = x_0x_1...x_n$ **covers** *another codeword* $y = y_0y_1...y_n$ *if and only if, for each bit position* i*, if* $y_i = 1$ *then* x_i=1. *In this case,* y *is* **covered** *by* x*, or* $y \leq x$.

Definition 2 (*Unordered Relation) A* **codeword pair** $\{x, y\}$ *is* **unordered** *if* $x \not\leq y$ *and* $y \not\leq x$. *Two* **sets of codewords**, X *and* Y*, are* **unordered** *if and only if, for each* $x \in X$ *and* $y \in Y$, $x \not\leq y$ *and* $y \not\leq x$.

Example. Given three codewords, x =001, y = 100, z = 011, the only pairwise covering relation is $x \leq z$. The pair x and y form an unordered pair since $x \not\leq y$ and $y \not\leq x$. Likewise, y and z form an unordered pair.

There are two main classes of delay-insensitive codes: *systematic* and *non-systematic*. A *systematic code* [24] contain two fields: (1) a data (or information) field which contains the original data bits, and (2) a DI field. For asynchronous communication, the DI field provides extra bits to guarantee that the entire code is delay-insensitive. Potential benefits of systematic codes over non-systematic codes are: (i) *ease of data extraction,* where no hardware decoders are necessary (unlike in non-systematic codes [4]) since the original data appears directly in the codeword; and (ii) *generally more coding efficient (i.e., has fewer # of bits per wire)*, since the DI field is typically logarithmic in the size of the data field; as a by-product, the smaller code length often results in improved *transition power* (# of wire-flips per transaction).

Fig. 2 illustrates the method for constructing a common type of systematic code, called a Berger code [5]. The first step, partitioning, takes a given set of datawords and groups them into "weight classes," where all datawords with the same total number of 1 bits are grouped into a distinct class. The next step, shown in Fig. 2(b), is to assign a unique DI field encoding to each weight class. For Berger codes, each weight class is assigned exactly one DI encoding which is the binary count of the 0's within the dataword. The length of the dataword field is $\lceil log_2(k) \rceil$, where k is the number of weight classes. As an example, in Fig. 2(b), a dataword in weight

[1] Although the terms DI and unordered are used interchangeably, DI refers to dynamic behavior during transmission in an asynchronous system, while "unordered" implies a static property of the codes (i.e. they provide error detection but not correction) in their use in synchronous systems.

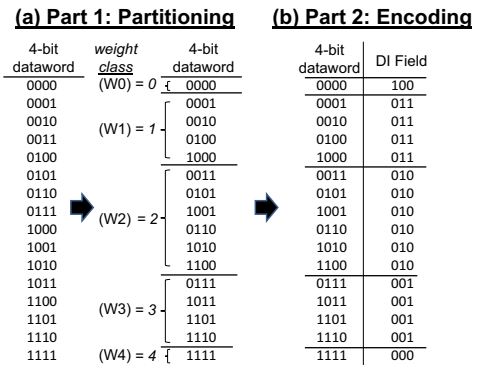

Figure 2. Berger Code Construction

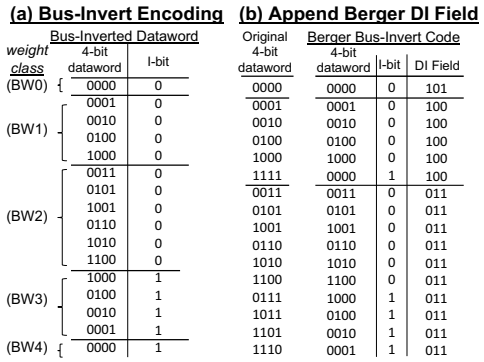

Figure 3. Berger Bus-Invert Construction

class 3 (i.e., 0111) has DI field 001, representing the one 0 in the dataword.

In contrast, in *non-systematic codes* [4, 24], data cannot be directly extracted, since there are no separate data and DI fields; instead, a single field is used, which captures the data values and ensures delay-insensitivity. Common examples include *dual-rail* (i.e., 1-of-2), *1-of-4* and the general class of m-of-n codes [24, 4]. To transmit information using an m-of-n RZ code, exactly m of n wires will transition twice (once high during evaluate, once low during reset).

Related Work. In the asynchronous domain, there have been a number of techniques for timing-robust global bus encoding. Both Smith [17] and Varshavsky [23] develop the theory for systematic delay-insensitive codes, which includes the legal requirements for constructing unordered codes. Delay-insensitive codes have been used in many recent asynchronous systems [3, 4, 11, 21, 16, 22], though most of these codes are simple and with high overhead, e.g. dual-rail and 1-of-4. Additional research has been pursued on developing error-correcting delay-insensitive codes [1], timing-dependent codes (i.e. non-DI) [14], and completion detector hardware to identify the receipt of valid DI codewords [2, 8]. However, none of the above approaches addresses transition power reduction by re-encoding using a bus-invert technique.

3 Berger Bus-Invert Code

The first of two initial delay-insensitive codes is now introduced. Each code offers a distinct strategy which is combined together to form the final DI Bus-Invert code. The Berger Bus-Invert code uses a straightforward extension of the Stan/Burleson method to provide delay-insensitivity. It is a useful starting point for the final method, but will be shown to have significant penalties.

The code is constructed in three simple steps. Step 1 is to partition the set of datawords into weight classes. Step 2 is to encode the datawords using the bus-invert method outlined by Stan and Burleson [18]. In Step 3, a DI field is appended using the classic Berger encoding scheme. As with any other Berger code, the resulting code is unordered.

Step 1: Partitioning. In this first step, the datawords are partitioned into weight classes, where each class contains datawords with the same total number of 1 bits. This step is identical to partitioning in the Berger approach (i.e., Fig. 2(a)).

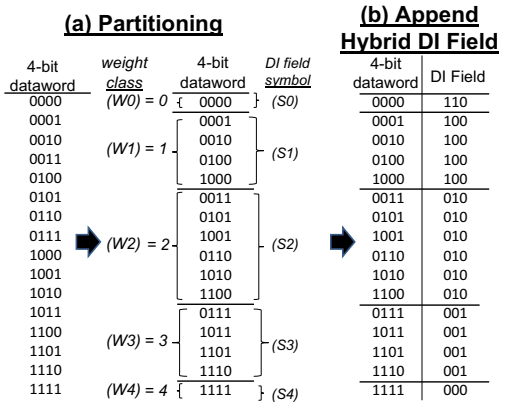

Figure 4. Hybrid Code Construction

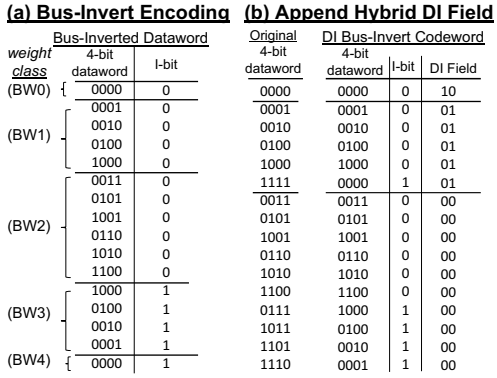

Figure 5. DI Bus-Invert Construction

Step 2: Bus-Invert Encoding. Step 2 is illustrated in Fig. 3(a). The goal of this step is to produce a set of datawords with a reduced number of wire transitions per transaction (i.e., less bits set to 1). In the encoding step, each dataword within a weight class is assigned the same appended field, which is an invert bit (i.e. I-bit). Given an n-bit dataword, the I-bit is assigned as follows:

$$\text{I-bit}(dataword) = \begin{cases} 1 & \text{if } \lceil \text{n/2} \rceil < \text{\# of dataword 1-bits,} \\ 0 & \text{otherwise} \end{cases}$$

Next, the dataword is re-encoded to its complement whenever the I-bit is 1. This method is almost identical to the Stan/Burleson approach; however, a return-to-zero protocol is used where all wires are reset to 0 between data transmissions, hence each dataword has a *unique re-encoding*. In contrast, the Stan/Burleson approach assumes successive data transmissions, with two possible encodings per dataword, depending on the encoding of the previous transmitted codeword.

Step 3: Append Berger DI Field. Step 3 is to append a Berger DI field, as illustrated in Fig. 3(b). This step is identical to the classic Berger encoding approach, but with one exception: re-partitioning is performed on the bus-inverted dataword and I-bit, whereas in Berger, partitioning is performed on the original dataword. Re-partitioning simply classifies the new codewords into weight classes considering both the dataword and I-bit fields, where each weight class has codewords with the same total number of 1-bits. A key feature of a Berger Bus-Inverted code is that it is *quasi-systematic*: since each dataword field is either inverted or not, it can be extracted using extremely simple (almost trivial) decoding hardware.

When comparing the Berger Bus-Invert code (Fig. 3) with the baseline Berger code (Fig. 2) for the 4-bit code example, there is a significant reduction in the number of 1 bits within the dataword and I fields. However, this improvement is undercut by an increase in codeword length (1 extra bit), and an overall increase in average number of wire transitions per transaction when including the check field (6.50 vs. 6.37, see Section 7).

4 Hybrid Code

The second delay-insensitive code is called Hybrid. While the Berger Bus-Invert method reduced wire transitions in the dataword field, the Hybrid method keeps the dataword field intact, but optimally encodes the appended check field by directly exploiting the partial order requirements of DI codes. Fig. 4 illustrates a Hybrid code using a 4-bit dataword field. It is constructed in 2 steps, where the second step optimally targets a DI field with fewest wire transitions.

Step 1: Partitioning. As shown in Fig. 4(a), datawords are partitioned into weight classes, using the same approach as for Berger codes (see Fig. 2(a)).

Step 2: Append Hybrid DI Field. The goal of this step is to assign DI check fields with fewest overall number of 1 bits. First, a relation is formed between weight classes. By observing weight classes, a total order relation exists, where a weight class with more bits set to 1 *covers* a weight class with less bits set to 1. For example, in Fig. 4(a), the total order covering relation for weight classes W0-W4 is:

$$\text{W0} \leq \text{W1} \leq \text{W2} \leq \text{W3} \leq \text{W4} \qquad (1)$$

Algorithm 1: encode-DI-field

```
1  /* input is result of method in Fig. 4a */
     input : {(W_i, S_i)}
2  /* output is set of encoded S_i symbols */
     output: {(S_i, enc_i)}
3  /* N = # of weight classes W_i */
4    N = |{(W_i, Si)}| ;
5  /* n = # of bits needed to encode S_i symbols */
6    n = ⌈log2(N)⌉;
7  /* generate all possible n-bit encodings */
8    Avail-Code-Set = generate-all-binary-encodings(n);
9  /* initialize set of assigned codes which
   follow Eqn. (2) */
10   Assigned-Code-Set= {};
11 /* iterate thru S_i and assign least-power
   unassigned encoding which follows Eqn. (2) */
12   for i = N − 1 downto 0 do
13     Legal-Codes = {x ∈ Avail-Code-Set such that x ⊈ C_k, for each
       C_k ∈ Assigned-Code-Set };
14     enc_i = select encoding in Legal-Codes with fewest 1 bits;
15     Assigned-Code-Set = Assigned-Code-Set ∪{enc_i};
16     Avail-Code-Set = Avail-Code-Set - {enc_i};
17   return {(S_i, enc_i)}
```

In turn, to form an unordered code, a symbol (S0, S1, S2, S3, S4) is associated with a given weight class (W0, W1, W2, W3, W4). The symbols, which represent the appended DI fields, will be encoded with a unique binary encoding. The symbol encodings (*enc(Si)*) must observe the following relation:

$$\text{enc(S0)} \not\leq \text{enc(S1)} \not\leq \text{enc(S2)} \not\leq \text{enc(S3)} \not\leq \text{enc(S4)} \qquad (2)$$

Interestingly, Equation 2, proposed by [17, 24], does *not* require a "reverse" covering relation on the appended DI field: *enc(S4)* $\leq$ *enc(S3)* $\leq$ *enc(S2)* $\leq$ *enc(S1)* $\leq$ *enc(S0)*. This latter relation would be sufficient to ensure that the resulting codes are unordered, but is too strong and unnecessary.

The declarative formulation for encoding the DI field begins by assigning the same symbol to each weight class. The length needed to encode the symbol is $\lceil log_2(k) \rceil$ bits, where k is the total number of symbols [17]. All symbols of the desired length can be assigned to a weight class. For example, the possible symbol encodings for the running example are:

$$\{000, 001, 010, 011, 100, 101, 110, 111\}$$

Note that the encodings have a covering relation where an encoding with more bits set to 1 covers and an encoding with fewer bits set to 1 (i.e., 001 $\leq$ 011). Therefore, the final symbol encodings must be selected such that Equation 2 is satisfied.

Algorithm 1 gives the procedure for optimally encoding the DI field. It assumes that partitioning has already been performed (see Fig. 4(a)). The input is the resulting set of pairs,

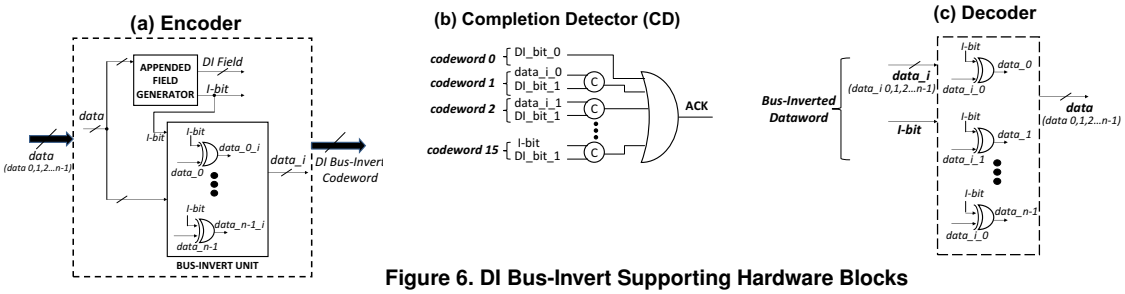

Figure 6. DI Bus-Invert Supporting Hardware Blocks

(W_i, S_i), where each weight class W_i is associated with a symbol S_i. The minimum DI field size, n, is determined such that all symbols can be uniquely encoded, and all candidate encodings are then enumerated ("Avail-Code-Set"). The algorithm then iterates through the symbols, S_i, from the one for the highest weight class (S_{N-1}) to lowest weight class (S_0). For each symbol S_i, it identifies all unassigned candidate codes ("Legal-Codes") that would ensure delay-insensitivity with respect to all previous assignments (in "Assigned-Code-Set"), according to Equation 2, when appended to the datawords in W_i. Of these, it selects the unassigned candidate with fewest number of 1 bits (enc_i), thereby directly targeting the wire transition metric.

The algorithm easily scales to 15-20 bit encodings, but beyond this, has overheads due to explicit candidate codeword enumeration. Since larger field sizes tend to be impractical for DI codes, due to hardware support overhead (see Section 7), it is more common to partition wide asynchronous datapaths into subfields which are separately encoded, hence the approach is sufficient. Improved heuristics and implicit data structures should allow extensions if desired.

When comparing the Hybrid code (Fig. 4) with the baseline Berger code (Fig. 2) for the 4-bit example, the new strategy results in the same check field size (3 bits), but with an improved check field encoding (i.e. fewer 1 bits assigned). This improvement is a direct result of exploiting the partial ordering requirement of DI codes in the code assignment.

Example. Fig. 4(b) illustrates the encoding of the DI field. Symbols $S0$, $S1$, $S2$, $S3$, and $S4$ are assigned to 110, 100, 010, 001, and 000, respectively, such that Equation 2 is preserved.[2] In total, only 5 of the 8 possible encodings are used, and at maximum, only two of three bits will be set high. For symbol $S0$, which corresponds to $W0$ (i.e., the "most covered" weight class in Equation 1), a valid option would be to encode it with 111 which covers all other 2-bit symbol encodings. However, Equation 2 provides a weaker constraint: the encoding of $S0$ must not be covered by the encoding of $S1$ and the encoding of $S2$. Therefore, 110 can safely be assigned to $S0$ since it is not covered by 100, 010, 001, or 000. Hence, to target a low number of wire transitions, the algorithm selects 110, a legal encoding option with fewest 1's.

Theorem 1 *(Hybrid Code Delay-Insensitivity) Every Hybrid code is unordered.*

Proof: Delay-insensitivity is ensured by Equation 2. The equation holds by determining the covering relation between two datawords, and then assigning the DI fields such that if dataword X is covered by dataword Y, then the DI field of X will not be covered by the DI field of Y. Thus, one codeword will not cover another since the unordered relation for a code is preserved under field concatenation. □

5 DI-Bus Invert Code

The two codes presented in the previous sections are used as building blocks for forming the DI Bus-Invert code – the core contribution of this paper. The DI Bus-Invert code is constructed by combining the "Bus-Invert Encoding" step used in the Berger Bus-Invert approach with the "Append DI Field" step used in the Hybrid code approach, in three steps.

Step 1: Partitioning. In this first step, the datawords are partitioned into weight classes, using the same method as in Fig. 2(a) and Fig. 4(a).

Step 2: Bus-Invert Encoding. This step is identical to that of the Berger Bus-Invert method. An example of bus-invert encoding is shown in Fig. 5(a). The goal is to reduce the number of wire transitions in the dataword field.

Step 3: Append Hybrid DI Field. The final step is identical to the Hybrid method where an appended field is added for delay-insensitivity, as shown in Fig. 5(b). The goal is to append a DI field which targets both coding efficiency and wire transitions.

Figs. 3 and 5 illustrate the key differences between the Berger Bus-Invert and DI Bus-Invert codes. The DI field for the Berger Bus-Invert code is length 3, while that of the DI Bus-Invert code is length 2. The length of the Berger Bus-Invert depends on the number of zeros in the dataword field and I-bit, while the length of the DI Bus-Invert code depends on the number of weight classes in the code, which is always less than or equal to the number of zeros in the dataword and I-bit fields. Second, the DI Bus-Invert method generally appends smaller-weighted DI fields. For instance, for the DI Bus-Invert code, at maximum, a 1-hot DI field (10 or 01) is appended, and the all-0's DI field is allocated to the weight class with the largest amount of elements (i.e., weight class 2), while the Berger Bus-Invert code has significantly more 1 bits in this field. Finally, the DI Bus-Invert code provides significant reduction in the number of wire transitions by simultaneously targeting both the dataword and check fields.

Theorem 2 *(DI Bus-Invert Code Delay-Insensitivity) Every DI Bus-Invert code is unordered.*

Proof: As in Theorem 1, a DI Bus-Invert code is delay-insensitive since it is constructed by ensuring Equation 2.

6 Hardware Support

There are three key supporting hardware blocks for the DI Bus-Invert code: an encoder, a completion detector (CD), and a decoder. It is assumed that the DI Bus-Invert code on the channel must be converted to single-rail bundled data at the receiver, and that the sender converts from single-rail bundled data. Details of control and latching of data at these interfaces are not provided, and are beyond the scope of the current work, but a variety of pipeline methodologies can be used.

(a) Encoder. An encoder design is shown in Fig. 6(a), consisting of an *appended field generator* and *bus-invert unit*. The appended field generator takes in a single-rail unencoded dataword and produces both appended fields, invert bit (I-bit) and DI field. It uses combinational logic to implement a simple look-up table. The second part is the bus-invert unit. The inputs to the unit are the original dataword and invert bit. The output is either the true or complemented dataword.

(b) Completion Detector (CD). A CD design is shown in Fig. 6(b). This particular design is for a 4-bit dataword, as implied by the 16 codewords. A C-element is allocated for each codeword and is used to detect when a particular codeword has arrived; therefore, exactly one C-element is asserted high per transaction.[3] Two optimizations are performed on the CD. The first is to eliminate literals (i.e., inputs to a C-element); the second is to eliminate a C-element when possible.

[2] Note that the alternative assignments of 101 or 011 for $S0$ are also possible.

[3] A C-element is a standard storage element, whose output is 0 (1) when all inputs are 0 (1), and which otherwise holds its value.

Dataword Field Size	DI Code Type	DI Code	Total # of Rails	Coding Effiency		# of Wire Transitions / Transaction	
				Actual	DI Bus-Invert Improvement	Actual	DI Bus-Invert Improvement
2	Systematic	*DI Bus-Invert*	4.00	0.50	-	2.00	-
		Berger Bus-Invert	5.00	0.40	25.0%	4.00	50.0%
		Hybrid	4.00	0.50	0.0%	3.50	42.9%
		Berger	4.00	0.50	0.0%	3.50	42.9%
	Non-Systematic	*1-of-4*	4.00	0.50	0.0%	2.00	0.0%
		1-of-5	5.00	0.40	25.0%	2.00	0.0%
		1-of-6	6.00	0.33	50.0%	2.00	0.0%
3	Systematic	*DI Bus-Invert*	6.00	0.50	-	3.75	-
		Berger Bus-Invert	7.00	0.43	16.7%	5.50	31.8%
		Hybrid	5.00	0.60	-16.7%	5.00	25.0%
		Berger	5.00	0.60	-16.7%	5.00	25.0%
	Non-Systematic	*2-of-5*	5.00	0.60	-16.7%	4.00	6.3%
		2-of-6	6.00	0.50	0.0%	4.00	6.3%
		2-of-7	7.00	0.43	16.7%	4.00	6.3%
4	Systematic	*DI Bus-Invert*	7.00	0.57	-	3.88	-
		Berger Bus-Invert	8.00	0.50	14.3%	6.50	40.4%
		Hybrid	7.00	0.57	0.0%	6.00	35.4%
		Berger	7.00	0.57	0.0%	6.37	39.2%
	Non-Systematic	*3-of-6*	6.00	0.67	-14.3%	6.00	35.4%
		2-of-7	7.00	0.57	0.0%	4.00	3.1%
		2-of-8	8.00	0.50	14.3%	4.00	3.1%
5	Systematic	*DI Bus-Invert*	8.00	0.63	-	5.56	-
		Berger Bus-Invert	9.00	0.56	12.5%	7.19	22.6%
		Hybrid	8.00	0.63	0.0%	7.31	24.0%
		Berger	8.00	0.63	0.0%	7.63	27.1%
	Non-Systematic	*3-of-7*	7.00	0.71	-12.5%	6.00	7.3%
		3-of-8	8.00	0.63	0.0%	6.00	7.3%
		2-of-9	9.00	0.56	12.5%	4.00	-39.0%
6	Systematic	*DI Bus-Invert*	9.00	0.67	-	5.75	-
		Berger Bus-Invert	10.00	0.60	11.1%	7.75	25.8%
		Hybrid	9.00	0.67	0.0%	8.65	33.5%
		Berger	9.00	0.67	0.0%	8.81	34.7%
	Non-Systematic	*4-of-8*	8.00	0.75	-11.1%	8.00	28.1%
		3-of-9	9.00	0.67	0.0%	6.00	4.2%
		3-of-10	10.00	0.60	11.1%	6.00	4.2%
7	Systematic	*DI Bus-Invert*	11.00	0.64	-	7.28	-
		Berger Bus-Invert	12.00	0.58	9.1%	9.38	22.3%
		Hybrid	10.00	0.70	-9.1%	10.00	27.2%
		Berger	10.00	0.70	-9.1%	10.00	27.2%
	Non-Systematic	*4-of-10*	10.00	0.70	-9.1%	8.00	9.0%
		3-of-11	11.00	0.64	0.0%	6.00	-21.3%
		3-of-12	12.00	0.58	9.1%	6.00	-21.3%
8	Systematic	*DI Bus-Invert*	12.00	0.67	-	7.56	-
		Berger Bus-Invert	13.00	0.62	8.3%	10.75	29.7%
		Hybrid	12.00	0.67	0.0%	10.71	29.4%
		Berger	12.00	0.67	0.0%	11.21	32.6%
	Non-Systematic	*4-of-11*	11.00	0.73	-8.3%	8.00	5.5%
		4-of-12	12.00	0.67	0.0%	8.00	5.5%
		3-of-13	13.00	0.62	8.3%	6.00	-26.0%
9	Systematic	*DI Bus-Invert*	13.00	0.69	-	9.09	-
		Berger Bus-Invert	14.00	0.64	7.7%	11.83	23.2%
		Hybrid	13.00	0.69	0.0%	11.99	24.2%
		Berger	13.00	0.69	0.0%	12.42	26.8%
	Non-Systematic	*5-of-12*	12.00	0.75	-7.7%	10.00	9.1%
		4-of-13	13.00	0.69	0.0%	8.00	-13.6%
		4-of-14	14.00	0.64	7.7%	8.00	-13.6%
10	Systematic	*DI Bus-Invert*	14.00	0.71	-	9.41	-
		Berger Bus-Invert	15.00	0.67	7.1%	12.62	25.4%
		Hybrid	14.00	0.71	0.0%	13.24	28.9%
		Berger	14.00	0.71	0.0%	13.63	30.9%
	Non-Systematic	*5-of-13*	13.00	0.77	-7.1%	10.00	5.9%
		5-of-14	14.00	0.71	0.0%	10.00	5.9%
		4-of-15	15.00	0.67	7.1%	8.00	-17.7%
11	Systematic	*DI Bus-Invert*	15.00	0.73	-	10.91	-
		Berger Bus-Invert	16.00	0.69	6.7%	13.59	19.7%
		Hybrid	15.00	0.73	0.0%	14.45	24.5%
		Berger	15.00	0.73	0.0%	14.77	26.1%
	Non-Systematic	*6-of-14*	14.00	0.79	-6.7%	12.00	9.1%
		5-of-15	15.00	0.73	0.0%	10.00	-9.1%
		5-of-16	16.00	0.69	6.7%	10.00	-9.1%
12	Systematic	*DI Bus-Invert*	16.00	0.75	-	11.27	-
		Berger Bus-Invert	17.00	0.71	6.3%	14.31	21.3%
		Hybrid	16.00	0.75	0.0%	15.61	27.8%
		Berger	16.00	0.75	0.0%	15.87	29.0%
	Non-Systematic	*6-of-15*	15.00	0.80	-6.3%	12.00	6.1%
		5-of-16	16.00	0.75	0.0%	10.00	-12.7%
		5-of-17	17.00	0.71	6.3%	10.00	-12.7%
13	Systematic	*DI Bus-Invert*	17.00	0.76	-	12.76	-
		Berger Bus-Invert	18.00	0.72	5.9%	14.84	14.0%
		Hybrid	17.00	0.76	0.0%	16.75	23.8%
		Berger	17.00	0.76	0.0%	16.92	24.6%
	Non-Systematic	*7-of-16*	16.00	0.81	-5.9%	14.00	8.9%
		6-of-17	17.00	0.76	0.0%	12.00	-6.3%
		5-of-18	18.00	0.72	5.9%	10.00	-27.6%
14	Systematic	*DI Bus-Invert*	18.00	0.78	-	13.14	-
		Berger Bus-Invert	19.00	0.74	5.6%	15.25	13.8%
		Hybrid	18.00	0.78	0.0%	17.87	26.5%
		Berger	18.00	0.78	0.0%	17.96	26.8%
	Non-Systematic	*7-of-17*	17.00	0.82	-5.6%	14.00	6.1%
		6-of-18	18.00	0.78	0.0%	12.00	-9.5%
		6-of-19	19.00	0.74	5.6%	12.00	-9.5%

Table 1. Code Comparison

(c) Decoder. A decoder is used by the receiver to obtain the original dataword. A general n-bit dataword decoder is shown in Fig. 6(c). The inputs are a DI bus-inverted dataword (i.e., dataword and invert bit), and the output is the original dataword. A two-input XOR gate is allocated for each dataword bit. When the invert bit is set to 1, the complement of the data bit is obtained, otherwise the original data bit value is obtained.

7 Evaluation

(a) Code Evaluation. Table 1 compares the new DI Bus-Invert code to several other systematic and non-systematic codes. Two metrics are evaluated, *coding efficiency* (i.e., # bits/wire) and a *wire transition metric* (i.e., average # wire transitions/bit/transaction), for dataword field sizes ranging from 2 to 14 bits.[4] Since an asynchronous four-phase protocol is assumed, with all wires initially set to 0, this metric counts every 1 rail twice: a rising transition (evaluation phase) followed by a falling transition (reset phase).

Four systematic codes are considered: the three proposed codes (DI Bus-Invert, Berger Bus-Invert, Hybrid) and a baseline code (Berger, with optimal coding efficiency). Also, three non-systematic m-of-n codes are also considered. The first is the most coding-efficient m-of-n code (i.e. "optimal"), where the smallest n is selected such that an m-of-n code can cover the complete set of 2^k symbols. The second and third are used to provide a wider set of comparisons, slightly relaxing the coding efficiency in an attempt to improve the wire transition metric. They increase the number of wires to $n+1$ and $n+2$, respectively; in each case, m is further reduced if possible, as long as a complete set of 2^k symbols can still be covered by the code. The "Improvement" columns compare the DI Bus-Invert code against all of these other codes.

Systematic Code Comparison. Overall, the new DI Bus-Invert code has significant benefits over the other three systematic codes. In particular, it has substantially better wire transition metric than Berger codes, with roughly comparable coding efficiency. For all field sizes listed, the new DI Bus-Invert codes have between 24.6 to 42.9% fewer wire transitions per transaction, yet maintains the same coding efficiency as Berger (with the exceptions of field sizes 3 and 7). This latter observation is particularly interesting, since Berger codes are theoretically the most coding-efficient systematic DI codes.

In contrast, the two new simpler codes, Hybrid and Berger Bus-Invert, show at best only modest improvements over Berger. The Hybrid code always has identical coding efficiency as the Berger code, with only limited reductions in wire transitions per transaction: up to 5.8% fewer wire transitions per transaction. The Berger Bus-Invert always has worse coding efficiency than Berger, with codewords requiring an extra 1-2 bits for each field size. Results for the wire transition metric with Berger Bus-Invert are mixed. In some cases the Berger code was better (by 2.0-14.3%), and in other cases Berger Bus-Invert was better (by 4.1-15.1%).

In summary, the combination of two strategies used in the DI Bus-Invert code provides much greater benefit than using them in isolation. These strategies fit together well: the bus-invert approach produces a code with fewer weight classes after inversion, thereby allowing the hybrid strategy to use a smaller DI field with a minimal number of 1 bits.

Non-Systematic Code Comparison. The comparisons between DI Bus-Invert and the m-of-n codes show complex tradeoffs for both coding efficiency and reduced wire transitions. However, overall the DI Bus-Invert was competitive.

For coding efficiency, both Berger (theoretically-optimal for systematic unordered codes) and DI Bus-Invert were similar or slightly worse than the best m-of-n code (theoretically-optimal for non-systematic unordered codes of its type): usually either identical codeword length or 1 extra wire needed, resulting in degradation of 5.6 to 11.1%. Conversely, when compared to the most relaxed non-systematic code using $n+2$ wires (i.e. 3rd non-systematic row), Berger and DI Bus-Invert are always more coding-efficient, usually with 1 fewer wire needed, resulting in improvements from 5.6 to 11.1% for most field sizes (5-14 bits, with greater improvements for 2-4 bit field sizes).

For the wire transition metric, when compared to the optimal m-of-n code, listed as the first non-systematic row for each dataword field size, the DI Bus-Invert codes have small to moderate improvements for most field sizes from 3 to 14 bits, ranging from 5.5 to 9.1% (but with better improvements for two cases, 4 and 6 bit fields). When compared to the relaxed non-systematic code with $n+2$ wires, the DI

[4] Larger field sizes can be partitioned and encoded by concatenating several smaller fields.

a. DI Bus-Invert Code Implementations

Dataword Field Size	Encoder			CD			Decoder			Total	
	i/o	area*	delay†	i/o	area*	delay†	i/o	area*	delay†	area*	delay†
2	2/4	35.28	0.05	4/1	33.84	0.12	3/2	29.64	0.04	98.76	0.21
3	3/6	65.63	0.08	6/1	76.17	0.24	4/3	44.46	0.04	186.26	0.36
4	4/7	86.08	0.08	7/1	88.89	0.33	5/4	59.28	0.04	234.25	0.45
5	5/8	133.37	0.11	8/1	170.71	0.40	6/5	74.10	0.04	378.18	0.55
6	6/9	178.54	0.12	9/1	195.40	0.44	7/6	88.92	0.04	462.86	0.60
7	7/11	275.16	0.14	11/1	299.13	0.49	8/7	103.74	0.04	678.03	0.67
8	8/12	431.10	0.16	12/1	433.16	0.54	9/8	118.56	0.04	982.82	0.74
9	9/13	551.72	0.17	13/1	658.30	0.71	10/9	133.38	0.04	1,343.40	0.92
10	10/14	848.73	0.18	14/1	509.38	0.70	11/10	148.20	0.04	1,506.31	0.92
11	11/15	1,094.18	0.19	15/1	915.84	0.93	12/11	163.02	0.04	2,173.04	1.16
12	12/16	1,724.87	0.21	16/1	1,018.00	0.96	13/12	177.84	0.04	2,920.71	1.21
13	13/17	2,164.38	0.22	17/1	1,397.71	1.08	14/13	192.66	0.04	3,754.75	1.34
14	14/18	3,727.00	0.23	18/1	1,415.40	1.14	15/14	207.48	0.04	5,349.88	1.41

** = area reported in μm2 † =delay reported in ns*

b. m-of-n Code Implementations

Dataword Field Size	Total Results / Comparision to DI Bus-Invert			
	area*	area improvement ratio	delay†	delay improvement ratio
2	89.58	0.9	0.17	0.81
3	187.67	1.0	0.32	0.89
4	324.52	1.4	0.50	1.11
5	686.42	1.8	0.59	1.07
6	1,271.16	2.7	0.64	1.07
7	2,850.71	4.2	0.78	1.16
8	5,249.84	5.3	0.79	1.07
9	9,774.60	7.3	0.97	1.05
10	17,718.81	11.8	1.01	1.10
11	31,513.51	14.5	1.12	0.97
12	65,165.26	22.3	1.16	0.96
13	120,132.23	32.0	1.34	1.00
14	245,216.00	45.84	1.63	1.16

Table 2. Hardware Evaluation

Bus-Invert codes range from slight improvement (+4.2%) to greater degradation (-39.0%) for field sizes from 5 to 14 bits.

Two other widely-used non-systematic codes, *dual-rail* and *1-of-4* [4, 11, 24], can also be compared to DI Bus-Invert (not in the table due to space limitations). Both have much worse coding efficiency for most field sizes (4 and higher): using 2 wires per bit with 0.5 efficiency, compared to 0.57-0.78 efficiency for DI Bus-Invert codes. For the wire transition metric, for most field sizes, dual-rail is much worse than DI Bus-Invert (two wire transitions per bit, i.e. one 1 rail out of 2, making a rising/falling transition); and 1-of-4 is slightly worse (two wire transitions per two bits, i.e. one 1 rail out of 4, making a rising/falling transition).

Overall Trends: Summary. For *coding efficiency,* the DI Bus-Invert code is nearly always identical to Berger, which is theoretically optimal for code length of the systematic codes. It typically is only slightly worse than the optimal non-systematic m-of-n code and slightly better than the relaxed non-systematic code using $n + 2$ wires. For the *wire transition metric,* an approximation for dynamic power, DI Bus-Invert is significantly better than Berger (between 24.6 to 42.9%) and slightly better than the optimal m-of-n code (usually 5.5 to 9.1%, but better for two cases). Compared to the relaxed non-systematic code with $n + 2$ wires, it ranges from slight improvement (+4.2%) to greater degradation (-39.0%).

In summary, for these two metrics, the new DI Bus-Invert code has substantial benefits over the optimal systematic code, Berger. It also has complex tradeoffs with the best non-systematic m-of-n based codes, but is a viable competitor. However, the next subsection highlights that DI Bus-Invert codes have a major additional advantage over m-of-n codes, in terms of hardware overhead.

DI Bus-Invert: Optimal Field Length. Table 1 also provides a basis to identify the optimal field size for the DI Bus-Invert code. The trend shows a strong and consistent improvement in coding efficiency as field sizes increase, from 0.50 (2-bit) to 0.78 (14-bit). For the wire transition metric, the goal is to assess the "per-bit" efficiency across different field sizes, that is, the *average # of wire transitions per bit per transaction.* The table does not include these results directly, but they can be derived. The range for this metric is fairly narrow and stable, from 0.94 to 1.12 (except for one outlier, 1.25), with no clear trend as field size increases. Hence, this latter metric is not strongly influenced by field size.

(b) Hardware Evaluation. To fully compare the new codes against its closest competitor, the hardware designs for the DI Bus-Invert and the m-of-n codes are implemented and evaluated. Only the most coding efficient m-of-n code is evaluated, due to the extensive design effort required to generate these results. It is assumed that hardware overheads for the two alternative m-of-n codes will be roughly comparable.

Technology-mapped implementations of the supporting DI Bus-Invert and m-of-n hardware components are synthesized using the UC Berkeley logic synthesis tool *ABC* [19]. Each component is specified in PLA format, then ABC's delay script [13] is applied for multi-level logic optimization and technology mapping using industrial $90nm$ standard cell library. Area is reported in μm^2 and delay is reported in ns. Table 2(a) shows results for the supporting hardware of the DI Bus-Invert code. Hardware overheads for the DI Bus-Invert code are moderate in area and delay. As expected, both metrics increase with larger field sizes. The total area ranges from 98.76 to 5349.88 μm^2, and the total delay ranges from 0.21 to 1.41 ns. Table 2(b) shows implementation results for the most coding efficient m-of-n code. It has significantly greater area overhead than the DI Bus-Invert code: *4.2x to 45.8x worse* for medium to larger field sizes (i.e., sizes 7-14).

8 Conclusions

A novel class of delay-insensitive RZ codes, called DI Bus-Invert, is introduced. This work builds on an earlier synchronous bus-invert approach by Stan and Burleson, and to our knowledge, is the first to provide a bus-invert strategy for delay-insensitive communication. A preliminary evaluation shows a substantial reduction in the average number of wire transitions per transaction over an existing systematic code, and generally close tradeoffs in wire transitions and coding efficiency when compared to the best non-systematic codes. Designs for supporting hardware have been implemented, and are shown to have substantially lower area overhead when compared to the most coding efficient m-of-n codes.

References

[1] M.Y. Agyekum and S.M. Nowick. An error-correcting unordered code and hardware support for robust global communication. In *DATE*, March 2010.

[2] V. Akella, N.H. Vaidya, and G.R. Redinbo. Asynchronous comparison-based decoders for delay-insensitive codes. *IEEE Trans. on Comp.*, 47(7):802–811, 1998.

[3] J. Bainbridge and S. Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.

[4] W. J. Bainbridge, W. B. Toms, D. A. Edwards, and S. B. Furber. Delay-insensitive, point-to-point interconnect using M-of-N codes. In *IEEE Async Symp.*, pages 132–140, May 2003.

[5] J.M. Berger. A note on error detecting codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[6] B. Bose. On unordered codes. *IEEE Trans. on Comp.*, 40(2):125–131, 1991.

[7] B. Bose and T.R.N. Rao. Theory of unidirectional error correcting/detecting codes. *IEEE Trans. on Comp.*, 31(6):521–530, 1982.

[8] M. Cannizzaro, W. Jiang, and S.M. Nowick. Practical completion detection for 2-of-n delay-insensitive codes. In *ICCD*, October 2001.

[9] M. E. Dean, T. E. Williams, and D. L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In *Proc. of UC Santa Cruz Conf. on Advanced Research in VLSI*, pages 55–70, 1991.

[10] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *IEEE Async Symp.*, pages 96–107, April 1998.

[11] A. Lines. Asynchronous interconnect for synchronous SoC design. *IEEE Micro*, 24(1):32–41, January 2004.

[12] P.B. McGee, M.Y. Agyekum, M.A. Mohamed, and S.M. Nowick. A level-encoded transition signaling protocol for high-throughput asynchronous global communication. In *IEEE Async Symp.*, pages 116–127, April 2008.

[13] A. Mishchenko, R.K. Brayton, and S. Jang. Global delay optimization using structural choices. In *IWLS*, pages 1–6, June 2008.

[14] S. Ogg, B. Al-Hashimi, and A. Yakovlev. Asynchronous transient resilient links for NoC. In *CODES*, pages 209–214, October 2008.

[15] J.D. Owens, W.J. Dally, R. Ho, D.N. Jayasimha, S.W. Keckler, and L.-S. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[16] L.A. Plana, S.B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS infrastructure for a massively parallel multiprocessor. *IEEE Design & Test of Comps.*, 24(5):454–463, 2007.

[17] J. Smith. On separable unordered codes. *IEEE Trans. on Comp.*, c-33(8):741–743, August 1984.

[18] M.R. Stan and W.P. Burleson. Bus-invert coding for low power I/O. *IEEE Trans. on VLSI Systems*, 3(1):49–58, 1995.

[19] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eccs.berkeley.edu/ alanmi/abc, 2005.

[20] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Comps.*, 24(5):418–429, 2007.

[21] J. Teifel and R. Manohar. An asynchronous dataflow FPGA architecture. *IEEE Trans. on Comp.*, 53(11):1376–1392, November 2004.

[22] Y. Thonnart, E. Beigne, and P. Vivet. Design and implementation of a GALS adapter for ANoC based architectures. In *IEEE Async Symp.*, pages 13–22, May 2009.

[23] V. I. Varshavsky. *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publishers, USA, 1990.

[24] T. Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.