Limitations of VLSI Implementation of Delay-Insensitive Codes[*]

Venkatesh Akella
ECE Department
University of California
Davis, CA 95616
Phone: 916-752-9810
Fax: 916-752-8428
akella@ece.ucdavis.edu

Nitin H. Vaidya
Computer Science Dept.
Texas A&M University
College Station, TX 77843-3112
Phone: 409-845-0512
Fax: 409-847-8578
vaidya@cs.tamu.edu

Robert Redinbo
ECE Department
University of California
Davis, CA 95616
Phone: 916-752-3087
Fax: 916-752-8428
redinbo@ece.ucdavis.edu

Technical Report 95-050
December 1995

Department of Computer Science
Texas A&M University

Abstract

Implementation of *delay-insensitive* (DI) or *unordered* codes is the subject of this report. We present two different architectures for decoding systematic DI codes: (a) enumeration-based decoder, and (b) comparison-based decoder. We argue that enumeration-based decoders are often *impractical* for many realistic codes. Comparison-based decoders that detect arrival of a code word by comparing the received checkbits with checkbits evaluated using the received data are *practical* but suffer from the following limitation. If the decoder is to be implemented using **asynchronous logic**, i.e., if the gate and wire delays are arbitrary (unbounded but finite), then it is impossible to design a comparison-based decoder for any code that is more efficient than a dual-rail code. In other words, the encoded word must contain at least twice as many bits as the data. The report shows that comparison-based decoders for codes that have the requisite level of redundancy can be implemented using asynchronous logic. The report also shows that, by relaxing the delay assumptions, it is *possible* to implement decoders for delay-insensitive codes that are more efficient than dual-rail codes.

[*]Research reported is supported in part by National Science Foundation Grants MIP-930868 (at UC-Davis) and MIP-9423735 (at Texas A&M University).

1 Introduction

In the past, significant effort has been spent in designing efficient codes for detection and correction of unidirectional and asymmetric errors [2]. Application of such codes to asynchronous buses has also been explored [3, 14, 5, 4, 17]. An asynchronous bus consists of wires whose transmission delays are unpredictable. The problem of detecting the arrival of information on such a bus has been shown to be equivalent to the problem of designing *unordered* or *all unidirectional error detecting* (AUED) codes [17] – such codes are also useful for unidirectional and asymmetric error control. Some codes for correcting different types of errors and skews on asynchronous buses have also been proposed (e.g., [3]). However, the past work has not explored the issues in VLSI implementations of decoders for the proposed codes. While we focus on *asynchronous communication* as the application of *unordered* [6] or delay-insensitive codes, the results of this report have implications for all applications of such codes.

This report deals with design of *asynchronous* decoders for codes used for communication on asynchronous buses. Past work on decoders [2] implicitly assumes synchronous hardware implementation of the decoder. As noted above, unordered codes have been proposed for two types of problems: (i) detection of arrival of data on the asynchronous bus, (ii) detection and correction of various types of errors on the asynchronous bus. The problem of designing asynchronous decoders for the type (ii) codes is strictly harder than that for type (i) codes. As a first step, this report focuses on design of *asynchronous* decoders that can detect when the transmitted information has been received, in the *absence* of any errors. As shown here, even this simple problem is hard to solve (in fact, impossible under certain conditions). This implies that, implementation of *asynchronous* decoders for unidirectional error correcting codes is likely to be very hard.

The codes that are useful for detecting arrival of data on an asynchronous bus are said to be *unordered* [6, 3, 5] or *delay-insensitive* [17]. Mathematically, one can formalize *unordered* or *delay-insensitive* (DI) codes as follows. Consider a binary code C. A code word $u \in C$ is said to be contained in a code word $v \in C$, if v has a 1 in each position where u has a 1. This is denoted as $u \subseteq v$. A code C is said to be *unordered* or *delay-insensitive* (DI) when no code word is contained in another code word. When an unordered code is used, arrival of a code word can be unambiguously recognized by the receiver, in presence of arbitrary delays in the wires. It is easy to see that one-hot and dual-rail (double-rail) codes enjoy this property [17]. Verhoeff [17], Varshavsky [16] and Blaum [2] discuss examples of other DI codes, e.g., Sperner codes and Berger codes among others and describe their mathematical properties.

VLSI implementation of decoders for systematic unordered (or DI) codes is the subject of this report. We first describe a communication protocol called the *four-phase protocol* [10] for the exchange of data on an asynchronous bus. Then we define two possible architectures for the decoders. The first is called *enumeration-based decoder* which examines the entire code word and determines if it is valid or not. It basically implements the membership-test using combinational logic. We argue that it is often *impractical* (and almost impossible given the VLSI technological limits) to implement *asynchronous* enumeration-based decoders for many realistic codes. We then present a *comparison-based decoder* which detects the arrival of a code word by recomputing the checkbits (using the received data bits) and comparing (or *matching*) them with the received checkbits. This is a practical approach but it suffers from the drawback of *hazards*, i.e., due to unpredictable gate and wire delays the decoder could signal a *match* even though the code word is *not* yet received. To avoid such erroneous detection of code words, the decoder needs to be delay-insensitive.

In this report, we prove that it is *impossible* to design a *delay-insensitive* comparison-based decoder for any systematic DI code that uses less redundancy than a dual-rail code. In other words, the encoded word must contain at least twice as many bits as the data. The comparison-based decoder architecture is practical, therefore, our impossibility result is of interest.

The report also shows that comparison-based decoders for appropriate codes (that have the requisite level of redundancy) can be implemented under the above assumptions. Finally, we present some practical constraints on circuit delays under which comparison-based decoders could be implemented for codes with smaller redundancy than dual-rail codes. We illustrate this with the implementation of a Berger code [1].

The report is organized as follows. Section 2 discusses our system model. Section 3 discusses the various decoder architectures and their implementation details. Section 4 shows that codes that are not as redundant as dual-rail codes cannot be implemented. Section 5 shows the characteristics of the *encoder* block (logic which recomputes the checkbits) for a delay-insensitive realization of comparison-based decoders. Section 6 shows that decoders for appropriate codes with requisite amount of redundancy can indeed be implemented (a design is presented). Section 7 shows that with some practical delay constraints decoders with smaller redundancy than dual-rail codes could be implemented. Section 8 discusses the implications of the main result of the report and provides directions for future work.

2 System Model

There are two components to our model: (a) the protocol used for communication on an asynchronous bus, and (b) the architecture of the decoder. We will describe the details of the communication protocol in this section and the decoder architectures in Section 3.

2.1 Asynchronous Communication Protocol

Unlike a synchronous system, an asynchronous system does not have a *clock* to validate data. Data communication in an asynchronous system is accomplished by a *handshake* protocol [10]. There are two popular handshake protocols. The *four-phase* (or return-to-zero) protocol and the *two-phase* (or non-return-to-zero) protocol. We will use the *four-phase* handshake protocol in this study. The organization of a system with four-phase handshake protocol is shown in Figure 1.

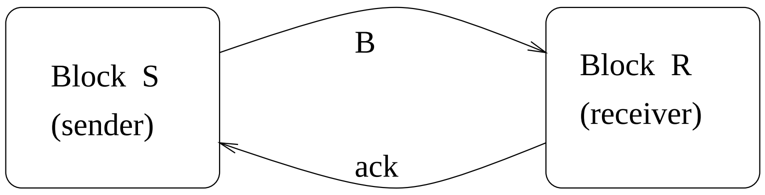


Figure 1: Four-Phase Handshake Protocol

`ack` is the acknowledgment wire and `B` is the asynchronous *bus* (or set of wires) on which encoded data is transmitted by the sender. At the start of the four phase protocol, the initial values are `ack = 0` and `B = (000...0)`. All-0 bus, `B = (000...0)`, is known as the **spacer** [17]. The four-phase protocol has the following four steps (hence the name).

(1) Block S (sender) encodes the data and transmits the code word on the asynchronous bus `B`. As the bus is initially in the *spacer* state, this step causes $0 \rightarrow 1$ transitions on the bus wires corresponding to non-zero bits of the code word. When this $0 \rightarrow 1$ transition arrives at the decoder, we say that the corresponding non-zero (1) bit of the code word has *arrived* at the decoder.

(2) After the code word is received by the receiver block R, it drives the `ack` wire high (or sets to logic level 1). (Note that the non-zero bits on bus `B` may arrive in an *arbitrary order* because of arbitrary delays on the wires.)

(3) Block S waits for `ack` to go high and then *resets* bus `B`, i.e., drives a logic value 0 on all wires of bus `B` (*spacer*).

(4) After an unbounded but finite amount of time, block R detects the spacer, i.e., `B = 000...0`, and in turn drives the `ack` wire low which takes the system back to the initial state, ready for the next transaction.

Basically, in a four-phase protocol, the data bus starts in an all-zero state (also known as the spacer) and transitions to whatever the code word is, and then goes back to an all-zero state. The *ack* wire provides the feedback to the sender so that a new piece of data is *not* sent unless the previous one has been received (or reliably latched) by the receiver. Our model is very simple and does not include the idea of pipelined data communication that was proposed by Blaum and Bruck [4, 5].

3 Decoder Architectures

Assume that the code being used is an (n, k) systematic unordered code. Thus, each code word contains k data bits, and $r = n - k$ checkbits. The sender encodes k bits of data into a code word containing n bits, by appending $r = n - k$ checkbits to the k data bits. The function of the *decoder* at the receiver is to detect when a code word has arrived, so that the receiver can latch the correct data into a register. (The term *decoder* is somewhat of a misnomer, because it only needs to *detect* arrival of a code word. In a systematic code word, the data is available without any further decoding.)

In this section, we present two *generic* architectures for the decoder, which can be used for *any* unordered systematic code.

3.1 Enumeration-based Decoder

An *enumeration-based* decoder implements a membership test to determine if a received word belongs to the code. The decoder looks at the input word and produces a 1 if the received word is a code word. The decoder must be hazard-free, otherwise, it may indicate that a code word has been received when the received word, in fact, is not a code word.

Consider the (4,2) Berger code [1] with 2 data bits (k=2) and 2 checkbits (r=2). The four code words in the (4,2) Berger code are:

d1	**d0**	**c1**	**c0**
0	0	1	0
1	1	0	0
0	1	0	1
1	0	0	1

where (d1,d0) are the data bits and (c1,c0) are the checkbits. A direct two-level AND/OR (sum-of-products) implementation of the decoder that produces a 1 on receiving a code word and a 0 otherwise would result in glitches (hazards) at the output of the decoder due to unpredictable order of the arrival of the bits and the distribution of delays in the gates and wires inside the decoder [7, 15]. This is not acceptable because we expect the decoder output to go to 1 **only if** we receive a code word.

However, we can take advantage of the four-phase protocol to implement the circuit in a hazard-free manner as follows. The protocol states that after a spacer, each bit (data and checkbits) can make a 0 to 1 transition in any possible order, but once they reach the code word they stop changing, i.e., do not change till the next spacer is sent (which marks the beginning of a new transaction). So, the wires undergo the following sequence:

$$SPACER \implies CODEWORD \implies SPACER \implies CODEWORD \ \ldots$$

This protocol, and the fact that the code is unordered, can be used to optimize the decoder function (or the underlying Karnaugh-map) as $f = c1 + d1d0 + d0c0 + d1c0$. Basically, the decoder implementation will contain one AND gate for *each* code word with the inputs of the AND gate being the bits which are 1 in the code word. Outputs of these AND gates will be sent to an OR gate, whose output will be the function f. Function f has the following characteristics:

(1) It is *positive* and *unate* in all the variables: A function $g(x_1, x_2, \ldots, x_n)$ is said to be unate with respect to a variable x_i if only x_i or its complement $\overline{x_i}$ appears in g but not both [8]. A function g is said to be *positive* with respect to a variable x_i if only the literal x_i appears in g.

(2) f is hazard-free for all the allowable transitions under the four-phase protocol that is being used in this discussion. This is because, our implementation of f consists of only AND and OR gates, and the output of an AND gate remains at zero till all its inputs are 1 – all inputs of an AND gate become 1 *only* when the corresponding code word is received.

(3) It can be shown that the *minimal* sum-of-products expression is *unique* [8] which means that all the prime-implicants are *essential*. So, f is the minimal hazard-free sum-of-products (SOP) realization of the enumeration-based decoder for (4,2) Berger code.

Therefore, f cannot be minimized any further in terms of a two-level logic implementation. However, the function can be optimized to minimize the number of literals and fan-in at the expense of the number of levels of logic.

3.2 Complexity of Enumeration-based Decoders

What happens to the complexity (size, fan-in, etc.) of the enumeration-based decoder as the number of the data bits increases? Consider (36,31) Berger code which is close to a typical word in a computer. The number of code words in this code is 2^{31}. In the enumeration-based implementation, one AND gate is required for each code word which means 2×10^9 (2 billion) AND gates and one OR gate with a fan-in of 2×10^9 are required. The fan-in of a typical AND gate in this implementation would be 19 (on an average). In addition, as noted in the previous section, the function is *unate* which means that all its prime implicants are essential so it cannot be minimized any further in a sum-of-products realization. A multi-level logic implementation is also impractical because the number of product terms (AND gates) is too large for any computer-aided design tool to handle and even if there were such a tool the number of levels of logic and the number of literals (wires) would be would be too large for any practical VLSI implementation.

3.3 Comparison-based Decoder Architecture

Figure 2 illustrates the comparison-based decoder architecture. Input to the decoder are the n bits received on the asynchronous bus. The input begins as all-0 *spacer* (i.e., all n bits are 0). When the sender sends encoded data, the received n bits eventually become identical to the transmitted code word. The *Present* output of the decoder is initially 0 (when the input is all-0). The output should remain 0 until a code word has been received on the asynchronous bus. When the code word is detected, the output should become equal to 1. The Present output of the decoder can be used to latch the correct data into a register. *It is, therefore, critically important that the Present output of the decoder should not become equal to 1 before the correct data is received on the bus.* (We will later show that this condition is impossible to satisfy if $r < k$.)

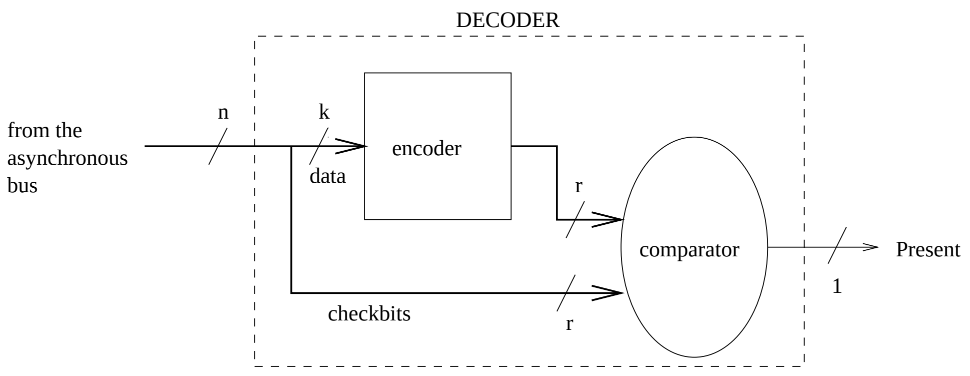


Figure 2: Comparison-based Decoder Architecture

The architecture of the decoder is simple and quite general. The *decoder* is implemented by means of an *encoder* and a *comparator*. The encoder receives the k data bits from the asynchronous bus, and computes the checkbits for the received data bits. The *comparator* compares these computed checkbits with the checkbits received on the asynchronous bus.

When the two match, arrival of the code word is detected (more importantly, arrival of the correct data is detected). This architecture is useful for all systematic codes.

In the previous section we noted that an enumeration-based decoder for (36,31) Berger code is impractical. So, it is interesting to see if a comparison-based decoder could be implemented for a (36,31) Berger code. A comparison-based decoder lends itself to a divide-and-conquer algorithm. The architecture of a circuit to compute the checkbits of (36,31) Berger code is shown in Figure 3.

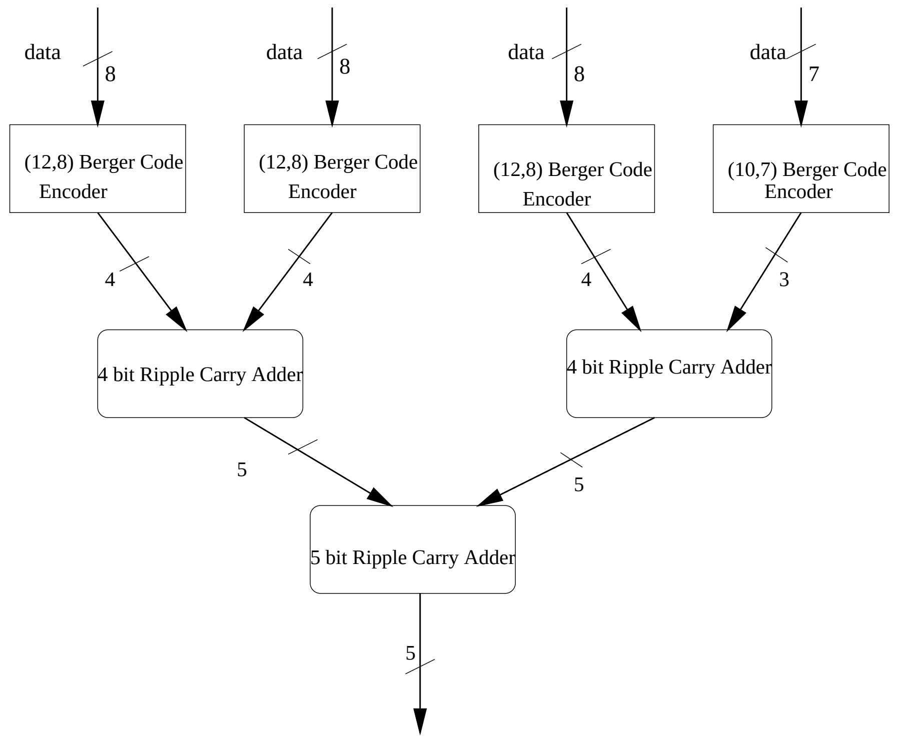

data
8
data
8
data
8
data
7
(12,8) Berger Code Encoder
(12,8) Berger Code Encoder
(12,8) Berger Code Encoder
(10,7) Berger Code Encoder
4
4
4
3
4 bit Ripple Carry Adder
4 bit Ripple Carry Adder
5
5
5 bit Ripple Carry Adder
5

Figure 3: An Implementation of the (36,31) Berger Code Encoder

Using misII logic synthesis tools [1] we found that (36,31) Berger code can be implemented with 8 AND, 167 NAND, 39 OR, 5 XOR, 17 XNOR, 93 NOR, and 79 inverters. All the gates except the inverters were restricted to 2 inputs. Note that there are several possible implementations of Berger codes. The implementation shown in Figure 3 is reasonably efficient and was optimized for a gate-level implementation using multi-level logic. If no constraints are placed on the gate and wire delays, these implementations could have glitches at the output. The issues in a hazard-free realization of comparison-based decoders is the main

[1]Multilevel logic synthesis and optimization tools from University of California, Berkeley

subject of the report and is discussed in the next section in detail.

4 Conditions for DI Decoder Implementation

In this section we derive a necessary condition for a delay-insensitive VLSI implementation of comparison-based decoders. First we state our assumptions and then present the main result of the report and its proof.

4.1 Assumptions

The discussion in this section makes the following two assumptions:

A.1 The bus is asynchronous in that the delay on each wire is arbitrary (but finite). The delays on any pair of wires are independent.

A.2 The *encoder* and the *comparator* in the decoder are implemented using gates and wires with arbitrary (but finite) delays. Thus, delays in producing each output of the encoder are arbitrary and independent.

(In practice, it is sometimes possible to assume some order relationship or bounds on delays, as discussed later.)

4.2 Main Result

Theorem 1 *Given assumptions A.1 and A.2, it is impossible to implement a comparison-based decoder for a systematic delay-insensitive code if $r < k$. That is, a comparison-based decoder cannot be implemented if the level of redundancy is less than a dual-rail code.*

Proof: The theorem states an impossibility result. We present a proof by constructing a scenario wherein the decoder will not work properly unless $r \geq k$.

Recall that, in the four phase protocol, the input to the decoder begins with all-0, that is, all n input bits are 0. Let $C(D)$ denote the r checkbits corresponding to k-bit data D. Also, let $C_i(D)$ denote the i-th checkbit corresponding to data D, $0 \leq i \leq r-1$. Thus, initially, the output of the encoder will be $C(00\cdots00)$. This implies that, initially, the two r-bit inputs to the comparator must be $C(00\cdots00)$ and $00\cdots00$. Recall that the code being used is a delay-insensitive (unordered) code, therefore, $C(00\cdots00)$ cannot be identical to $00\cdots00$ (r bits). Thus, the initial value of *Present* output of the comparator will be 0 (indicating a mismatch of its inputs). Figure 4 illustrates the initial state.

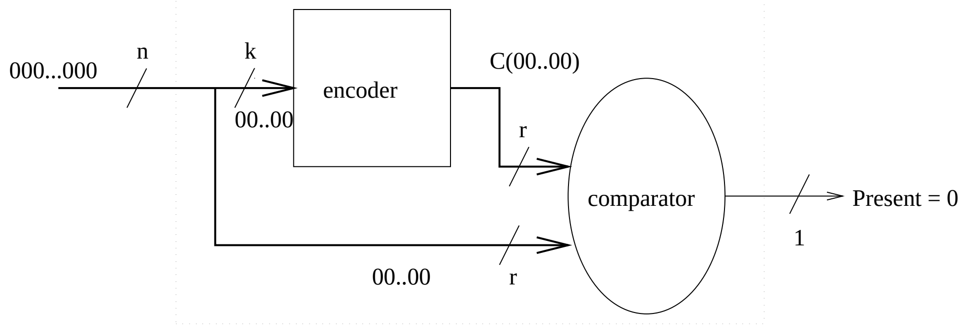


Figure 4: Initial configuration

Now, let the code word transmitted by the sender on the asynchronous bus be $d_{k-1}d_{k-2}\cdots d_1d_0\ b_{r-1}b_{r-2}\cdots b_1b_0$, where $d_{k-1}d_{k-2}\cdots d_1d_0 = 11\cdots1$ and $b_{r-1}b_{r-2}\cdots b_1b_0 = C(11\cdots1)$. Thus, all the data bits transmitted by the sender are 1. To prove the impossibility result stated in the theorem, it is sufficient to construct one scenario where the stated result is true. We now build one such scenario.

Consider the scenario where all the non-zero bits in $b_{r-1}b_{r-2}\cdots b_1b_0$ arrive at the decoder before any non-zero bits in $d_{k-1}d_{k-2}\cdots d_1d_0$ arrive at the decoder. Thus, now one input to the comparator is $b_{r-1}b_{r-2}\cdots b_1b_0$ and the other input of the comparator is still $C(00\cdots00)$.

Now, the non-zero data bits start arriving at the decoder. As the data input to the encoder (within the decoder) changes, its output will change from initial value $C(0\cdots0)$ to the final value $C(d_{k-1}d_{k-2}\cdots d_1d_0) = C(11\cdots1) = b_{r-1}b_{r-2}\cdots b_1b_0$. Output of the encoder is an input to the comparator.

A "false match" is said to occur at the comparator if the two r-bit inputs of the comparator are identical **but** the data bits received by the decoder are **not** identical to the data bits transmitted by the sender. A *false match* will result in the receiver accepting incorrect data.

In the scenario under consideration, to avoid a false match at the comparator, we must guarantee that the output of the encoder will not become identical to $b_{r-1}b_{r-2}\cdots b_1b_0$ until all the 1 (non-zero) bits of data $d_{k-1}d_{k-2}\cdots d_1d_0$ have arrived. We now show that a false match can occur if $r < k$.

The input to the *encoder* changes from initial value $00\cdots 0$ (k bits) to final value $d_{k-1}d_{k-2}\cdots d_1d_0 = 11\cdots 11$. Therefore, as shown in Figure 5, the k-bit encoder input can potentially follow the chain $D_0 = 00\cdots 0$, D_1, $\cdots$, D_{k-1},$D_k = 11\cdots 1$. That is, encoder input may change from initial value D_0 to D_1, then D_2, and so on, finally to $D_k = 11\cdots 1$. While the received data bits are changing, the checkbits received from the sender remain steady at $b_{r-1}b_{r-2}\cdots b_1b_0$.

$$
\begin{aligned}
D_0 &= 0000\cdots 0000\\
D_1 &= 0000\cdots 0001\\
D_2 &= 0000\cdots 0011\\
D_3 &= 0000\cdots 0111\\
&\vdots\\
&\vdots\\
D_{k-3} &= 0001\cdots 1111\\
D_{k-2} &= 0011\cdots 1111\\
D_{k-1} &= 0111\cdots 1111\\
D_k \quad &= 1111\cdots 1111
\end{aligned}
$$

Figure 5: A chain from $00\cdots 0$ to $11\cdots 1$

Let the initial output of the encoder $C(D_0) = C(00\cdots 0)$ be denoted as $a_{r-1}a_{r-2}\cdots a_1a_0$. The final output of the encoder will be $C(D_k) = C(11\cdots 11) = b_{r-1}b_{r-2}\cdots b_1b_0$. Thus, output of the encoder changes from $a_{r-1}a_{r-2}\cdots a_1a_0$ to $b_{r-1}b_{r-2}\cdots b_1b_0$ as its input changes from $D_0 = 00\cdots 0$ to $D_k = 11\cdots 1$. (Note that, in our notation, $C_i(D_0) = a_i$ and $C_i(D_k) = b_i$.)

Claim: To avoid a false match, there must exist an i, $0 \leq i \leq r-1$, such that

1. $C_i(D_j) = C_i(D_0)$, $0 \leq j \leq k-1$, and

2. $C_i(D_k) = \overline{C_i(D_0)}$.

This claim implies that as the data input of the encoder changes from D_0 to D_{k-1}, at least one checkbit computed by the encoder, say i-th, remains constant. This checkbit is complemented only when the data input changes to D_k.

Proof of the claim: The proof is by contradiction. Assume that the claim is false. This implies that, for all i $(0 \leq i \leq r-1)$, there exists j_i $(0 \leq j_i \leq k-1)$ such that $C_i(D_k) = C_i(D_{j_i})$.

Assume that the encoder input has become equal to D_{j_i}, and the i-th checkbit computed by the encoder has become equal to $C_i(D_{j_i}) = C_i(D_k) = b_i$. As the encoder is asynchronous (by assumption A.2), it is possible that its i-th output bit does not change (for a long time) even after the encoder input has changed from D_{j_i}. In this manner, i-th output of the encoder, for all i, becomes equal to $C_i(D_{j_i})$ (or $C_i(D_k)$) and stays there, <u>before</u> the data input of the encoder becomes equal to D_k (recall that $j_i \leq k-1$, $\forall i$). Thus, when the encoder input is equal to D_m where $m = \max(j_i)$, the encoder output will be equal to $C(D_k)$, although the data bits received by the receiver are not D_k – this would cause a *false match* at the comparator. Thus, the above claim is proved. □

The above proof of the claim considers the case when D_k is the data being transmitted by the sender. The above proof can be repeated for each data D_l $(l > 0)$ in the chain to conclude that, to avoid a false match, for each l $(1 \leq l \leq k)$, there must exist $i(l)$, $0 \leq i(l) \leq r-1$, such that

(condition B.1) $C_{i(l)}(D_j) = C_{i(l)}(D_0)$, $0 \leq j \leq l-1$, and

(condition B.2) $C_{i(l)}(D_l) = \overline{C_{i(l)}(D_0)}$.

If the above conditions are not satisfied, a false match can occur. It is obvious that, the above conditions cannot be satisfied, unless $r \geq k$. This concludes the proof of Theorem 1.

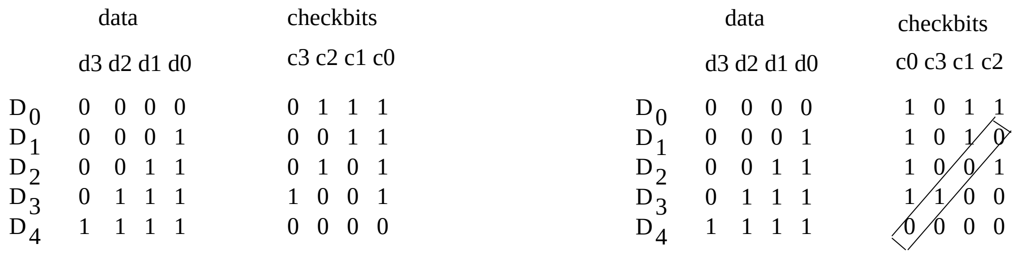
□

5 Characteristics of the Encoder Function

We now present some properties that the *encoder* function should satisfy for the existence of a delay-insensitive comparison-based decoder implementation for a systematic unordered code.

5.1 Diagonal Property

Figure 6(a) presents an example of a chain of data and the corresponding checkbits that satisfy conditions B.1 and B.2 listed in the proof for Theorem 1. For this example, using the notation used in conditions B.1 and B.2, we have: (i) for $l = 1$, $i(1) = 2$, (ii) for $l = 2$, $i(2) = 1$, (iii) for $l = 3$, $i(3) = 3$, and (iv) for $l = 4$, $i(4) = 0$. Observe that, as shown in



	data				checkbits			
	d3	d2	d1	d0	c3	c2	c1	c0
D_0	0	0	0	0	0	1	1	1
D_1	0	0	0	1	0	0	1	1
D_2	0	0	1	1	0	1	0	1
D_3	0	1	1	1	1	0	0	1
D_4	1	1	1	1	0	0	0	0

(a)

	data				checkbits			
	d3	d2	d1	d0	c0	c3	c1	c2
D_0	0	0	0	0	1	0	1	1
D_1	0	0	0	1	1	0	1	0
D_2	0	0	1	1	1	0	0	1
D_3	0	1	1	1	1	1	0	0
D_4	1	1	1	1	0	0	0	0

(b) permuted checkbits

Figure 6: Example: The *diagonal* property

Figure 6(b), the checkbit positions can be permuted such that the permuted checkbits are complemented at a diagonal position, as marked by the diagonal box in Figure 6(b). For the decoder to be implemented, for each chain of data, there must exist a permutation of the checkbits such that the checkbits are complemented at diagonal positions as illustrated above. We call this property the "diagonal property". Thus, for a DI decoder to be implemented, the systematic code must satisfy the *diagonal property.* In fact, the *diagonal* property is sufficient to implement the decoder. It is interesting to note that, the dual-rail code satisfies the *diagonal* property.

5.2 Initial Condition

As the number of checkbits must be at least k, we now focus on codes with exactly k checkbits. (As codes with $r > k$ are not of practical interest, generalization of the next result to $r > k$ is omitted here.) The theorem below uses some notation developed in the proof of Theorem 1.

Theorem 2 *Given a systematic unordered code with $k = r$, for a comparison-based decoder to be implemented under assumptions A.1 and A.2, a necessary condition is that $C(00\cdots0) = 11\cdots1$. That is, checkbits corresponding to all-0 data must be all-1.*

Proof: This proof uses some notation presented in the proof of Theorem 1. We assume that $r = k$ for the code under consideration.

Let the code word transmitted by the sender on the asynchronous bus be $d_{k-1}d_{k-2}\cdots d_1d_0\ b_{r-1}b_{r-2}\cdots b_1b_0$, where $d_{k-1}d_{k-2}\cdots d_1d_0 = 11\cdots11$ and $b_{r-1}b_{r-2}\cdots b_1b_0 = C(11\cdots11)$. Thus, all the data bits transmitted by the sender are 1. Let the data received by the encoder follow the chain D_0 through D_k, as defined in the proof of Theorem 1. Unlike the proof of Theorem 1, in this proof, we do **not** assume that all non-zero checkbits arrive before the data bits.

Without loss of generality, assume that the checkbits are named such that c_0 is the first to be complemented, followed by c_1, c_2, etc., in that order, as illustrated in Figure 6(b). This assumption implies that, $i(l) = l - 1$ for $1 \leq l \leq k$ (using the notation in conditions B.1 and B.2). More specifically, for $1 \leq l \leq k$,

(condition P.1) $C_{l-1}(D_j) = C_{l-1}(D_0)$, $1 \leq j \leq l-1$, and

(condition P.2) $C_{l-1}(D_l) = \overline{C_{l-1}(D_0)}$.

The proof of Theorem 2 is by contradiction. Thus, we assume that at least one checkbit in $C(D_0) = C(00\cdots0)$ is 0. Now let l denote the **largest** integer, such that $C_{l-1}(D_0) = 0$. By the above conditions, it follows that, $C_{l-1}(D_l) = 1$. Now assume that the data transmitted by the sender is D_l (l is not necessarily equal to k). The data input to the decoder could potentially follow the chain $D_0, D_1, \cdots, D_l$.

Assume that the receiver receives the most significant $(r-l)$ checkbits, before any other checkbits or data bits are received. More specifically, the lower r-bit input to the comparator is now assumed to be $b_{r-1}b_{r-2}\cdots b_l\ 0\ 0\cdots0$. Additionally, in the scenario under consideration, the least significant l checkbits transmitted by the sender are assumed to encounter a large delay on the asynchronous bus (larger than all the data bits and other checkbits). – Therefore, those l checkbits will remain 0 at the receiver during the scenario under consideration here.

Now, assume that the data input to the encoder has changed from D_0 to D_{l-1}, along the chain $D_0, D_1, \cdots, D_{l-1}$. By conditions P.1 and P.2, the most significant $(r-l+1)$ bits of $C(D_{l-1})$ must be equal to $b_{r-1}b_{r-2}\cdots b_l 0$. (Recall that $C_{l-1}(D_0) = C_{l-1}(D_{l-1}) = \overline{C_{l-1}(D_l)} = 0$.) Also, P.1 and P.2 imply that, for each $i \leq l-2$, there exists $m(i) \leq l-1$, such that $C_i(D_{m(i)}) = 0$. Even though the data input of the encoder has changed to D_l, it is possible (by assumption A.2) for the output of the encoder to equal $b_{r-1}b_{r-2}\cdots b_l\ 0\ 0\cdots 0$. This will occur if the most significant $r-l+1$ checkbits produced by the encoder linger on from $C(D_{l-1})$, and each of the i-th least significant $l-1$ checkbits $(0 \leq i \leq l-2)$ produced by the encoder lingers on from $C(D_{m(i)})$. Thus, in this situation, the encoder output and the checkbits received on the bus are both $b_{r-1}b_{r-2}\cdots b_l\ 0\ 0\cdots 0$, while the data bits received on the bus are D_{l-1} (although data bits transmitted are D_l). Thus, the comparator will produce a false match, before the data has arrived. This concludes the proof.

$\square$

6 Delay-Insensitive Decoder For Dual-Rail Code

We now demonstrate that the bound in Theorem 1 is tight, by presenting a delay-insensitive (asynchronous) comparison-based decoder, for the dual-rail code, based on the architecture shown in Figure 2. The dual-rail code is commonly used in asynchronous systems, and for this code $k = r$. Our design is very similar to the implementations found in asynchronous literature [10, 12]. Without loss of generality, let us assume that the number of data bits is 2. Therefore, $r = k = 2$. The table below shows the code words where d_1 and d_0 are data bits and c_1 and c_0 are checkbits.

d_1	d_0	c_1	c_0
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	0

It follows that, $c_1 = \overline{d_1}$ and $c_0 = \overline{d_0}$.

Figure 7 shows the circuit-level implementation of the decoder. The gate marked C denotes a Muller C-element [10, 9]. It is a special latch which has the following behavior. The output of the C element is *high* (or logic 1) when all of its inputs are high and the output is *low* (or logic 0) when all of its input go low; otherwise it *retains* its previous state.

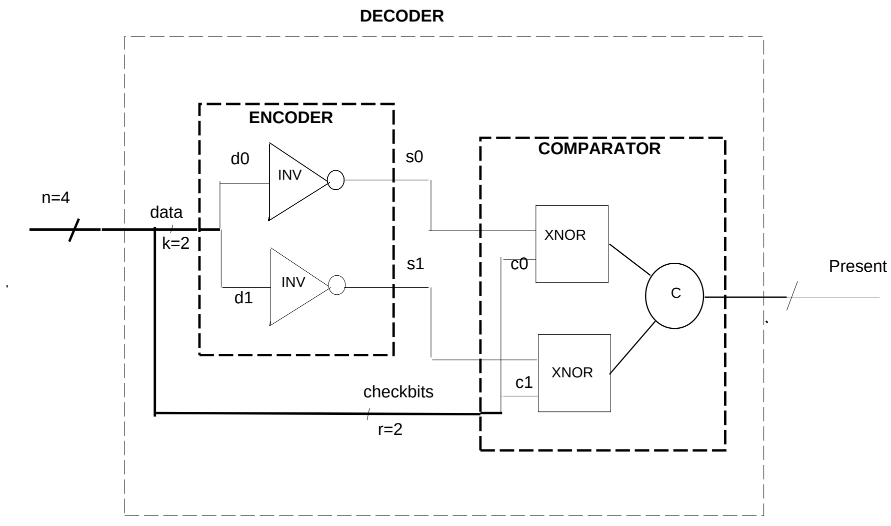


Figure 7: DI Implementation of the Decoder for 2 bit Dual-Rail Code

It may seem that the XNOR gate inside the comparator maybe prone to hazards (glitches) and hence result in a false match. However, a close examination of the possible transitions of the XNOR gate inputs reveals that it is not the case. The following is the informal justification which can be verified by inspecting the transition space of the XNOR function. At the beginning of the data transmission (refer to the details of the four-phase protocol in Section 2.1), the decoder input is the all-0 *spacer*, and the two inputs of each XNOR gate are 1 and 0 which results in an output of 0. After the code word arrives, **one** of the inputs of the XNOR will change resulting in the output changing *monotonically* to 1. Hence, the XNOR gate will not glitch. This implementation of the decoder for the dual-rail code is delay-insensitive. This proves that there exist codes with $k = r$ for which comparison-based decoders can be implemented under assumptions A.1 and A.2 in Section 4.

7 Decoders with Delay Assumptions

Section 4 proves that, if $k > r$, an asynchronous comparison-based decoder cannot be implemented under assumptions A.1 and A.2. In this section, we demonstrate that it may be *possible* to implement such decoders if we are allowed to place realistic restrictions on transmission and circuit delays. Specifically, we present an implementation of a decoder for the (5,3) Berger code that makes some assumptions regarding circuit delays. Note that for this code, $k = 3$ and $r = 2$ – thus, $k > r$.

Figure 8 presents the code words from the (5,3) Berger code in the form of a "lattice". A code word u precedes code word v in this lattice if the *data* bits in code word v cover the data bits in code word u, and the number of non-zero data bits in u and v differs by 1. In Figure 8, most significant 3 bits of each code word are the data bits $d_2d_1d_0$, and the least significant 2 bits are the checkbits c_1c_0. (Note that only the data bits are being compared here, not the whole code word.)

$$\begin{tikzcd}
& 000\ 11 \arrow[dl] \arrow[d] \arrow[dr] & \\
001\ 10 \arrow[d] \arrow[dr] & 010\ 10 \arrow[dl] \arrow[dr] & 100\ 10 \arrow[dl] \arrow[d] \\
011\ 01 \arrow[dr] & 101\ 01 \arrow[d] & 110\ 01 \arrow[dl] \\
& 111\ 00 &
\end{tikzcd}$$

Figure 8: Lattice representation of the (5,3) Berger code

In Figure 8 observe that, if the sender is prevented from sending the code word 111 00 (corresponding to data 111), then all the remaining code words satisfy the *diagonal* property stated previously. In particular, the data received by the decoder must traverse a chain starting from 000 to the actual data value. As 111 is a forbidden data, the transmitted data can only be of weight ≤ 2. The *diagonal* property is apparent from the observation that, the checkbit c_0 is complemented only when a data with weight 1 is received, and checkbit c_1 is complemented only when a data with weight 2 is received. Thus, from the discussion in Section 5 it follows that, the comparison-based decoder will function correctly under assumptions A.1 and A.2, provided the sender if forbidden from transmitting the code word 111 00.

Now, we suggest some assumptions on the circuit delays to allow the sender to transmit code word 111 00 as well. To allow this, we only have to ensure that the output of the encoder (within the decoder in Figure 2) will not become equal to 00 unless the data received by the encoder is 111. This can be guaranteed as follows: (i) Partition the encoder into two circuit blocks $S0$ and $S1$ to independently produce c_0 and c_1, respectively, as a function of the received data. The data bits received from the bus are sent to both $S0$ and $S1$. (ii) Ensure that, when a transition occurs on the bus, the delay in producing the new output value is smaller for $S0$ as compared to $S1$. (i.e., in response to an input data change, the encoder computes the new value of c_0 faster than the new value of c_1.)

The above two conditions guarantee that the encoder will *not* produce output 00, unless the input is 111.[2] This will ensure that a false match cannot occur when the transmitted code word is 111 00. We already know that a false match cannot occur when any other code word is transmitted. Thus, the decoder will function correctly under the circuit delay assumption stated above.

It is worth noting that the circuit delay assumption made above is easy to implement in practice. We conjecture that comparison-based decoders can be implemented for all Berger codes under practical assumptions on circuit delays. Design of decoders for larger Berger codes is a subject of on-going work.

8 Discussion of the Results

The key contribution of the report is the investigation into the VLSI implementation of decoders for *unordered* or *delay-insensitive* (DI) codes. We showed that enumeration-based decoders are impractical for codes of any reasonable size. Comparison-based decoders are feasible but are prone to glitches if the underlying gate and wire delays are arbitrary (unbounded but finite). The key implication of this result is the non-existence of *delay-insensitive*

[2]To see why that is true, note that the encoder can possibly produce output 00 *only if* its data inputs change from 000 to a data of weight 1 (e.g., 001), and then to a data of weight 2 (e.g., 101), and finally to 111. When data changes along this chain, the encoder output can prematurely become 00 only if the c_0 checkbit produced when data input was 001 stays at 0 even when the data has changed to 101 and, in response, c_1 has changed to 0 (from 1). The circuit delay assumption made above, ensures that c_1 cannot change to 0 (from 1) in response to an input change from 001 to 101, before c_0 changes from 0 to 1. Thus, the encoder output cannot prematurely become equal to 00, and a false match cannot occur.

comparison-based decoders for systematic delay-insensitive codes with redundancy less than 50%. In other words, comparison-based decoders for systematic DI codes which are more *efficient* (in terms of number of wires per bit) than dual-rail codes **cannot** be implemented in delay-insensitive manner. However, if one is prepared to make some delay assumptions in the underlying implementation, codes with smaller redundancy could be implemented reliably, i.e., in a hazard-free manner. This was demonstrated in Section 7 by enumerating the conditions under which a decoder for a Berger code could be implemented. This is an interesting result because it brings out a curious relationship between the timing (delay) assumptions in the decoder implementation and the redundancy of the unordered (or delay-insensitive) code. An interesting problem is to determine the lower bounds on the redundancy in the unordered (or DI) codes to implement comparison-based decoders with quasi-delay-insensitive (QDI) assumption [12, 9, 13] and the speed-independent circuit theory [11].

Acknowledgements

We thank Janlung Sung for discussions during the early part of this research.

References

[1] Berger, J. M. A Note on Error Detection Codes for Asymmetric Channels. *Information and Control 4* (1961), 68–73.

[2] Blaum, M., Ed. *Codes for Detecting and Correcting Unidirectional Errors.* IEEE Computer Society, 1993.

[3] Blaum, M., and Bruck, J. Unordered error-correcting codes and their applications. In *Digest of papers: The 22th Int. Symp. Fault-Tolerant Comp.* (July 1992), pp. 486–493.

[4] Blaum, M., and Bruck, J. Coding for Skew-Tolerant Parallel Asynchronous Communications. *IEEE Transactions on Information Theory 39*, 2 (March 1993), 379–388.

[5] Blaum, M., and Bruck, J. Delay-Insensitive Pipelined Communication on Parallel Buses. *IEEE Transactions on Computers 44*, 5 (May 1995), 660–668.

[6] Bose, B. On Unordered Codes. *IEEE Transactions on Computers 40* (February 1991), 125–131.

[7] Eichelberger, E. B. Hazard Detection in Combinational and Sequential Switching Circuits. *IBM Journal of Research*, 9 (mar 1965), 90–99.

[8] Kohavi, Z. *Switching and Finite Automata Theory.* Tata McGraw-Hill, 1978. Chapter 4 and Chapter 6.

[9] Martin, A. J. The Limitations to Delay-insensitivity in Asynchronous Circuits. In *Advanced Research in VLSI : Proceedings of the Sixth MIT Conference.* MIT Press, Mar. 1990.

[10] Mead, C. A., and Conway, L. *An Introduction to VLSI Systems.* Addison Wesley, 1980. *Chapter 7, entitled "System Timing".*

[11] Miller, R. E. *Switching Theory Volume II: Sequential Circuits and Machines.* John Wiley & Sons, 1965. Chapter 10: Speed Independent Switching Circuit Theory.

[12] Nanya, T., Ueno, Y., Kayotomi, H., Kuwako, M., and Takamura, A. TITAC: Design of a Quasi-Delay-Insensitive Microprocessor. *IEEE Design and Test of Computers 11*, 2 (June 1994), 50–63.

[13] Piestrak, S. J., and Nanya, T. Towards totally self-checking delay-insensitive systems. In *Digest of papers: The* 25th *Int. Symp. Fault-Tolerant Comp.* (1995), pp. 228–237.

[14] Tallini, L., Merani, L., and Bose, B. Balanced codes for noise reductionin VLSI systems. In *Digest of papers: The* 24th *Int. Symp. Fault-Tolerant Comp.* (June 1994), pp. 212–218.

[15] Unger, S. H. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, 1969.

[16] Varshavsky, V., Kishnivsky, M., Markhovsky, V., Peschansky, V., Rosenblum, L., Taubin, A., and Tzirlin, B. *Self-Timed Control of Concurrent Processes.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

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