IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)
e-ISSN: 2278-2834,p- ISSN: 2278-8735. Volume 7, Issue 5 (Sep. - Oct. 2013), PP 78-85
www.iosrjournals.org

Implementation of error correcting methods to the asynchronous Delay Insensitive codes with reduced delay and area

B. Shilpa Reddy[1], S.M.K. Sukumar Reddy[2] (Phd)
[1]*Post Graduate Student, Vaagdevi Institute of Technology &Sciences, proddatur, India -516360*
[2]*Asst.Professor, Department of ECE, Vaagdevi Institute of Technology &Sciences, proddatur, India-516360*

Abstract: *This Paper provides an approach for reducing delay and area in asynchronous communication.* ***.*** *A new class of error correcting Delay Insensitive (ie., unordered) codes is introduced for global asynchronous communication.It simultaneously provides timing-robustness and fault tolerance for the codes.A systematic and weighted code is targeted. The proposed error correcting unordered (ECU) code, called zero-sum can provide 1-bit correction.The extensions to the zero-sum code are given.The zero_sum⁺ code provides 3-bit error detection,or it can provide 2-bit detection and 1-bit correction.The zero_sum* code support 2-bit correction,while still guaranteeing 2-bit detection under different strategies of weight assignments. Zero_sum* code provides 2-bit correction coverage (50 % to 70%) of all 2-bit errors. The proposed method reduces delay occurred, due to the transfer of corrupted bits in a packet on the channel by the removal of timer and also reduces the area with the proposed Completion Detector (CD).*

Keywords : *Asynchronous communication, Four phase protocol , error-correcting codes, delay insensitive and unordered.*

I. Introduction

In past, significant effort has been spent in designing efficient codes for detection and correction of symmetric, unidirectional and asymmetric errors. Application of such codes to asynchronous buses has also been explored. 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 codes, such codes are also used for the error control.

There are mainly three classes of errors that can be handled [3].They are unidirectional, asymmetric and symmetric errors. Unidirectional errors may include either $0 \rightarrow 1$ or $1 \rightarrow 0$ type bit corruptions, but only one type may occur within any given codeword; the type need not be known in advance [3]. Asymmetric errors involve only one error type for all codewords ($0 \rightarrow 1$ or $1 \rightarrow 0$), where the type must be known in advance [4]. Symmetric errors may simultaneously include both $0 \rightarrow 1$ and $1 \rightarrow 0$ bit corruptions within the same codeword [3]. The main focus of this paper is on handling the symmetric errors.

The contributions of this paper are centered around a family of error-correcting unordered (ECU) codes for use in asynchronous global communication, called Zero-Sum [1], [2].These codes simultaneously targets two types of robustness: *timing-robustness*, allowing variability in the arrival times and orders of individual bits in a codeword on a channel, by the use of DI encoding, and *fault tolerance*, providing error detection and correction capabilities.

All proposed Zero-Sum codes are *systematic* [3], [4], where data appears in unaltered form in each codeword and can be directly extracted by the receiver without any decoding hardware. They are also *weighted*, where the check field is computed as the sum of data index weight. When compared to the best previous systematic ECU code, the new code provides significant reduction in transition power for most field sizes,with better or comparable coding efficiency.

II. System Model

2.1. Point-to-Point Asynchronous Communication

An asynchronous communication channel [5] is the means by which information is transmitted. Fig. 1 gives an example of point-to-point communication.

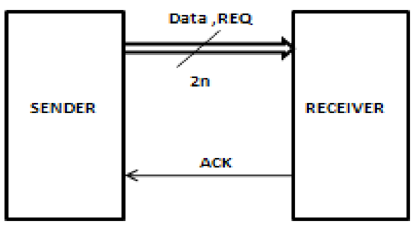
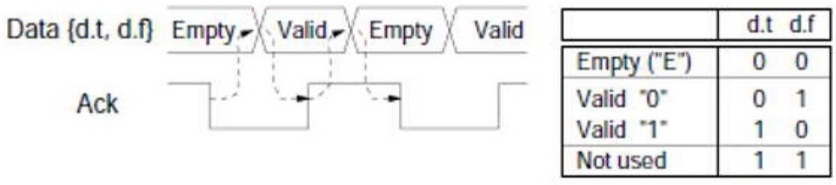


Fig.1: Block diagram of Point-to-point asynchronous communication

If the sender issues a request signal (*REQ*) to the receiver and the receiver in turn provides an acknowledgment signal (*ACK*) to the sender. If the sender passes actual data to the receiver, the REQ is typically replaced by the encoded data, as shown in the fig.1. The ACK indicates that data has been received by the receiver and new data can eventually be sent.

2.2. Four-phase Communication Protocol

Given an asynchronous communication channel, a protocol is needed to transfer information from sender to the receiver.The most widely used protocol is four-phase or return-to-zero (RZ) [6], is shown in fig(2).



	d.t	d.f
Empty ("E")	0	0
Valid "0"	0	1
Valid "1"	1	0
Not used	1	1

Fig.2: Four-phase return-to-zero protocol.

The request signal is encoded into the data signals using two wires per bit of information that has to be communicated. The information is encoded as follows: {x.f,x.t} ={1,0} for a logic 0 and {x.f,x.t} = {0,1} for a logic 1 represent valid data and {x.f,x.t} = {0,0} represents no data. The codeword {x.f,x.t} = {1,1} is not used, and a transition from one valid codeword to another valid codeword is not allowed.The protocol has two operations. 1) Evaluate and 2) Reset. During evaluate operation, the sender issues a valid codeword, the receiver absorbs the codeword and sets acknowledge high.At this point the reset operation begins, the sender responds by issuing the empty codeword,and the receiver acknowledges this by taking acknowledge low. At this point the sender may initiate the next communication cycle.

An alternative to the RZ protocol is two-phase or non-return to zero protocol [7],which avoids reset phase. This also use the dual-rail coding,contains even phase and odd phase. The codes designed for this typically have large overheads and it has more complex structure. Therefore, four-phase communication protocol is used for communication in this paper.

2.3. DI Codes

When asynchronous communication is used, data must be suitably encoded so that the receiver can identify when a packet has been received. DI codes [5] (i.e.,unordered codes) are insensitive to propagation delays on individual bits in a codeword. A code *C* is said to be *unordered* or *delay-insensitive* (DI) when no code word is covered by another code word. When an unordered code is used, arrival of a code word can be unambiguously recognized by the receiver, in the presence of arbitrary delays in the wires.

Definition 1 (Unordered Code): A codeword $X = x1x2 \ldots xn$ *covers* another codeword $Y = y1y2 \ldots yn$ if and only if, for each bit position *i*, if $yi = 1$ then $xi = 1$. In these cases, *Y* is *covered* by *X*, or $Y \leq X$. Codewords *X* and *Y* are unordered if *X* _ and *Y* _ *X*. A code, *C*, is *unordered* iff each pair of codewords in *C* is unordered.

Example 1: Given codewords, *X* = 001, *Y* = 100, *Z* = 011, the DI pairs are {*X, Y*} and {*Y, Z*}. *X* is not unordered when compared to *Z*.

There are two types of DI codes: Systematic and non-systematic codes. A systematic code [8] contains separate data and check fields. For asynchronous communication, the check field provides extra bit to indicate the entire code is DI. Types of systematic codes are Berger [9] and Knuth [10] codes. In this no hardware decoders are required for the extraction of data because the original data directly appears in codeword.In non systematic codes [5], there are no separate data and check fields. Data is encoded in a unified field, which ensures delay insensitivity. Dual-rail, 1-of- 4 and general class of *m*-of-*n* codes are example for these codes.

.

2.4. Basic Zero_sum code

The basic Zero-Sum code, uses the single index weight assignment only as discussed in Berger [7] and Bose [3] codes. The code provides 1-bit error correction or 2-bit error detection for symmetric errors, as well as delay-insensitivity, thus forming an ECU code.

Code construction combines aspects of both Hamming codes [11] and Berger[7] codes. In particular, as in Berger codes, the pattern of 0 bit in the dataword is used to generate the DI field. However, while the Berger method counts the number of 0s, the Zero-Sum approach adds the 0-bit index weights. Similarly, Zero-Sum uses the bit index numbering scheme used in Hamming codes, but typically requires extra bits.

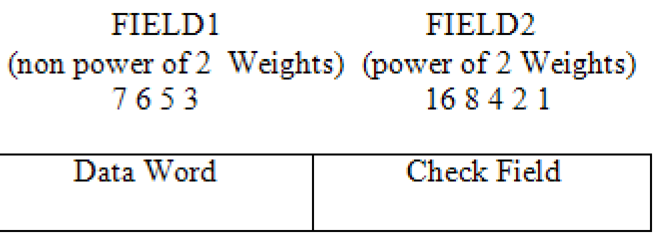


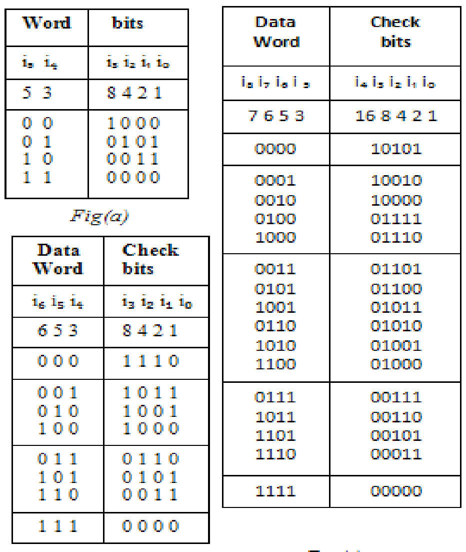
Fig 3: 4-bit zero-sum codeword structure

2.4.1. Overview of Code: Fig.3 shows the codeword structure for a Zero-sum ECU code. It contains two fields: 1) data word, and 2) check field. Each bit position is given an index. The check bits index weights are powers-of-two (for non negative exponents), and the data word index weights are given power of two weights starting from the smallest check bit index. The index weights assigned to the dataword field is called the weight set. For example, given a 3-bit dataword field, the weights of the check bits are 1, 2, 4, and 8 and dataword weights are consecutively assigned integers of 3, 5, and 6. Fig.4 shows examples of Zero-Sum ECU codes for 2-bit, 3-bit,and 4-bit information fields.

2.4.2. Check Field Generation: Checkfied of Zero-sum Code is the binary representation of the arithmetic sum of the dataword index weights whose bit is 0, hence the name of the codes, Zero-Sum.

2.4.3. Formal Calculation of Code Length: The check field must be large enough to support the binary representation of the sum of all of the data field index weights, i.e., to handle the extreme case where all data bits are 0. Therefore, the total number of check bits allowed is the [log2(_dataword index weights)] + 1. A closed-form equation for the check field length k in terms of the data field length n is [7]:
$k = \_log2([(n + \mu)(n + \mu + 1)/2] - 2\mu + 1)$, (1) where μ is determined by $2\mu-1 < n+\mu < 2\mu$, (2)

Example 2: For the 4-bit dataword 1010 in Fig. 4(c), the sum of those dataword index weights whose bits are set to 0 (i.e., weights 6 and 3) is 9. Therefore, the corresponding check field value is the binary representation of 9, which is 01001.



Word	bits
i_5 i_4	i_3 i_2 i_1 i_0
5 3	8 4 2 1
0 0	1 0 0 0
0 1	0 1 0 1
1 0	0 0 1 1
1 1	0 0 0 0

Fig(a)

Data Word	Check bits
i_6 i_5 i_4	i_3 i_2 i_1 i_0
6 5 3	8 4 2 1
0 0 0	1 1 1 0
0 0 1	1 0 1 1
0 1 0	1 0 0 1
1 0 0	1 0 0 0
0 1 1	0 1 1 0
1 0 1	0 1 0 1
1 1 0	0 0 1 1
1 1 1	0 0 0 0

Fig(b)

Data Word	Check bits
i_8 i_7 i_6 i_5	i_4 i_3 i_2 i_1 i_0
7 6 5 3	16 8 4 2 1
0000	10101
0001	10010
0010	10000
0100	01111
1000	01110
0011	01101
0101	01100
1001	01011
0110	01010
1010	01001
1100	01000
0111	00111
1011	00110
1101	00101
1110	00011
1111	00000

Fig(c)

Fig 4: Examples of Zero_sum ECU codes (a)2-bit ECU code (b)3-bit ECU code (c)4-bit ECU code.

2.4.4. Detecting and Correcting 1-Bit Error: Hamming code provides a syndrome which is a vector of individual check bits (i.e., one for each parity group) [11], the Zero-Sum ECU code provides a unified syndrome which is a single positive integer: the absolute value of the difference between the appended check field and a newly calculated check field. Error detection and correction uses a modification of the Hamming approach. In both Zero-Sum and Hamming codes, nonzero syndrome indicates an error. However, in Zero-Sum, the syndrome is computed differently: the receiver creates a regenerated check field C' from its datafield ,and compares C' to the actual received check field C. The resulting Zero-Sum syndrome, is the absolute difference of$|C' - C|$. If the difference is zero, there is no error. The syndrome is also used to correct a 1-bit error: its value is the index of the corrupted bit, as in Hamming codes. However, unlike Hamming, the Zero-Sum appended check field also ensures delay-insensitivity, as will be proven below.

Example 3: Following Example 2 above, suppose there is an error in transmitting the 4-bit dataword 1010, due to a flip in the data bit with index 7 (i.e., erroneous dataword 0010), transmitted with the original error-free check field (i.e., 01001). The newly calculated check field, based on the corrupted dataword, is 16 (i.e., 7 + 6 + 3). Therefore, the syndrome is 16 − 9 = 7, which is nonzero and not a power-of- two. This syndrome therefore precisely identifies the index (i.e., 7) of the corrupted data bit. In contrast, if a single check bit has an error, the syndrome will be a power-of-two and identify the corresponding index of the corrupted check bit.

Theorem 1 (zero-sum code delay – insensitivity [5]): Every zero sum code is unordered

Proof: By definition1, it is sufficient to show that zero-sum code, c, is unordered if each pair of code words is unordered. Given two data words *X* and *Y*, by definition 1 if *X* covers *Y*, then *X* has more 1's and *Y*, and *Y* has more 0's than *X*. The check field is computed as the sum of the data field indexweight which are `0'. The sum of index weights of check field must be greater than the sum of index weights of data word.

Theorem 2(Zero-sum code error detection): Every zero-sum code provides 2-bit detection.

Proof: The proof is immediate from Theorem 2. Since all 1-bit errors can be corrected, the code must have minimum distance of at least 3; therefore, all 2-bit errors can be detected.

III. Extending The Zero-Sum Code

3.1. Zero-Sum+ Code

It is an enhancement of Zero-sum code, provides two alternative modes. In one mode, up to 3-bit errors and all odd errors can be detected. In a second mode, both correction and detection can be handled together: every 2-bit error can be detected and every 1-bit error can be corrected.

FIELD1	FIELD2	FIELD 3
(non power of 2 Weights)	(power of 2 Weights)	(0)
7 6 5 3	16 8 4 2 1	0

Data Word	Check Field	Parity

Fig 5: 4-bit zero-sum⁺ code word structure

.

DataWord	Check bits	parity
$i_8\ i_7\ i_6\ i_5$	$i_4\ i_3\ i_2\ i_1\ i_0$	i_0
7 6 5 3	16 8 4 2 1	0
0000	10101	1
0001	10010	1
0010	10000	0
0100	01111	1
1000	01110	0
0011	01101	1
0101	01100	0
1001	01011	1
0110	01010	0
1010	01001	0
1100	01000	1
0111	00111	0
1011	00110	1
1101	00101	1
1110	00011	1
1111	00000	0

Fig 6: 4-bit zero_sum⁺ code

3.1.1. Overview: Zero-Sum+ code is essentially a Zero-Sum code with a parity bit attached,as shown in fig (5). Both the data word and check fields of a Zero-Sum+ code are identical to those of a basic Zero-Sum code. The new field is the parity bit, which provides even parity. Its index weight is 0. The fig.6 shows the zero-sum$^+$code for 4-bit data

3.1.2. Error Detection: Error detection mode can detect up to 3-bit errors and all odd numbers of bit errors. As shown in Table 1, each error type is detected by its parity and syndrome values. The case of 0 errors in the received codeword is distinct, where the parity is correctly set to even and the syndrome is equal to 0. In the remaining cases, the parity is either odd or the syndrome is nonzero. When a 1-bit error occurs, there are two possibilities, each resulting in odd parity. If a 1-bit error occurs in either the data word or the checkfields, the syndrome will be nonzero; if a 1-bit error occurs in the parity bit, the syndrome will be zero. When a 2-bit error occurs, the parity of the received codeword will be correctly set to even; however, the syndrome is guaranteed to be nonzero. Finally, any odd number of errors results in odd parity, with either a nonzero or a zero syndrome.

Table 1:Zero_sum$^+$error detection classification.

# of errors	parity	Syndrome value
0	even	Zero
1	Odd	Zer or nonzero
2	Even	Nonzero
3(or odd)	odd	Zero or nonzero

3.1.3.Error Correction: Similar to the Zero-Sum code, the Zero-Sum+ code also guarantees that 1-bit errors can be corrected. As an additional parity bit is added in the Zero-Sum+ approach, a variant set of correction methods is needed. As shown in Table 2, When 0 errors occur, the parity is even and syndrome values is correct. Therefore, no error handling method is applied. When 1 error occurs, there are two cases. If the error occurs in either the data word or the check fields, the syndrome value is nonzero and the parity is incorrectly set to odd. The error correction technique used is the 1-bit correction strategy for a Zero-Sum code presented in Section 2.4.4; toggle the bit indicated by the nonzero syndrome value. If a 1-bit error occurs in the parity bit, the syndrome value is zero. Correction is performed by toggling the parity bit.

Table 2:zero_sum$^+$ error correction classification

of errors	Parity	Error handling
0	Even	None
1	Odd	1-bit correction method
1	Odd	Toggle parity bit

3.2. Zero-Sum* Code:

3.2.1 Overview: The Zero-Sum* codes support heuristic multi-bit correction by strategically varying the index weight assignment. The structure of the code is similar to a Zero-Sum+ code.The data word field is assigned non-power-of-two values *{*3, 7, 11, 5*}*. The check field indices are assigned the ordered values of power-of-two weights *{*16, 8, 4, 2, 1*}*.The fig.7 shows the 4-bit zero-sum* code.

Table 3:zero_sum* error correction classification

#of errors	Parity	Syndrome value	Error handling
0	Even	Zero	None
1	Odd	Nonzero	1-bitcorrection method
1	Odd	Zero	Toggle parity bit
2	even	nonzero	2-bitcorrection method

3.2.2.Method for 2-Bit Correction:

The Zero-Sum* code can correct all 1-bit errors and some instances of 2-bit errors,as shown in Table 3. Errors are corrected in a two-step procedure. First, the error type is classified according to the syndrome and parity of the received codeword. Next, the appropriate method for correcting the error is selected. When 0 errors occur, the parity is even and syndrome is zero. Therefore, no error handling method is applied. A 1-bit error is corrected by either toggling the parity bit or using the 1-bit correction method as discussed in Section 2.4.4. For the 2-bit error correction,the unoptimized method is proposed.

The unoptimized approach performs 2-bit correction by generating and pruning the syndromes of all possible candidate code words which are at distance 2 from the received codeword. Each candidate that has nonzero syndromes is discarded, as it could not have been the original codeword. If there exists exactly one valid candidate codeword with syndrome zero. then correction can be performed in 2-bit flips. This single remaining codeword is the original sent codeword. When multiple Code words remain, there is ambiguity in how the received codeword was reached, since any one could have been the originally sent codeword.

Data Word	**Check bits**	**parity**
i_8 i_7 i_6 i_5	i_4 i_3 i_2 i_1 i_0	i_0
3 7 11 5	16 8 4 2 1	0
0000	10101	1
0001 0010 0100 1000	10010 10000 01111 01110	1 0 1 0
0011 0101 1001 0110 1010 1100	01101 01100 01011 01010 01001 01000	1 0 1 0 0 1
0111 1011 1101 1110	00111 00110 00101 00011	0 1 1 1
1111	00000	0

Fig 7: 4-bit Zero_sum* code

3.2.3. Strategies for Assigning the Dataword Index Weights:

The choice of weight set directly impacts the coverage of 2-bit correction that can be obtained. When selecting the weight set, there are two points to consider. The first is to consider the possible combinations (i.e., sum and difference)between pairs of index values in the weight set Second, a careful balance must be maintained between the selected weights and the length of the check field. Taking these points into consideration,the dataword index weight set is generated such that each pairwise combination (i.e., sum or difference) of dataword index weights in the weight set is unique. Hence, each distinct 2-bit error occurring solely within the dataword field and check field has a unique syndrome. This approach ensures no aliasing of any distinct 2-bit errors lying entirely within the data word field or check field, but does not eliminate aliasing for some 2-bit errors that span the two fields.

IV. Hardware Support:

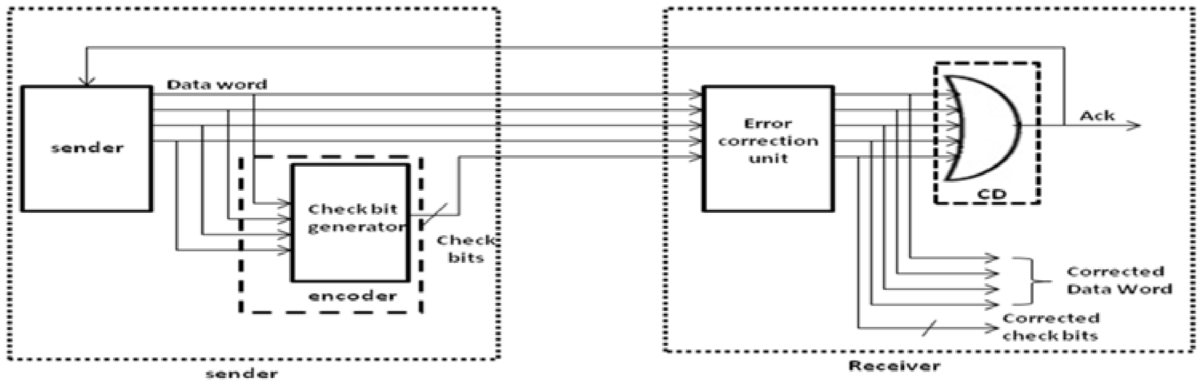


Fig.8: Architectural block diagram

Fig. 8 shows the architectural block diagram for a zero-sum code. The system mainly contains an encoder, an error correction unit, and a completion detector. An encoder generates the check field, which is appended to the data word and transmitted on the four-phase asynchronous communication channel between sender and receiver. At receiver the corresponding data word and check bits are given to the error correction unit, it generates the corrected data word and the check bits. Then, the CD generates the Ack which again given back to the sender.

An encoder design for a 4-bit dataword is shown in Fig. 9. It consists of a row of selectors (i.e., multiplexors) followed by adders. There is one selector for each data bit, for the data bit"0" hardcoded index weight is selected and for"1" zero is selected.Finally,they are added to get the check bits, which is the binary representation of obtained value.

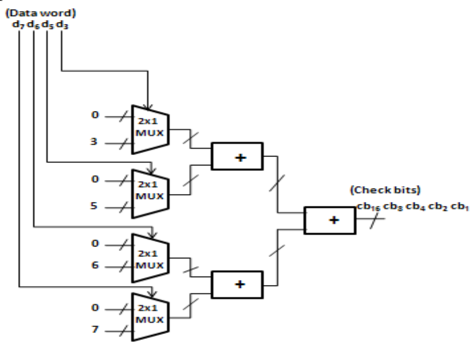


Fig 9:Basic Encoder

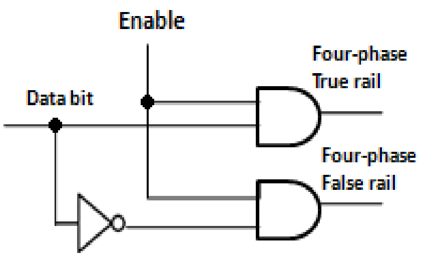


fig 10: 4-phase encoder bit slice

The fig.10 shows the simple four phase encoder block, having a row of AND gates. Each bit contains two wires with a common enable signal. When enable is high, one and only one of each pair of data rails will then go high; when enable is low, the encode block will reset all of the data rails to zero.

The 4-bit error corrector unit, shown in Fig. 11 is divided into a syndrome generator and corrector. The syndrome generator produces the syndrome by performing the operations of comparison and subtraction. First, given the received dataword, an encoder generates a new check field. Next, the syndrome is generated by finding the mathematical difference between the received and newly generated check fields. A magnitude comparator is used to perform the absolute value function. The top-most multiplexor selects the larger of the two values, while the lower multiplexor selects the smaller of the two. The second part performs the correction operation. A C-element and 2-input XOR gate are allocated for each bit of a codeword. The input to the C-element is the syndrome, and for each bit, a nonzero syndrome which uniquely identifies when an error occurs in that particular bit. The corresponding XOR gate corrects the faulty bit by performing bit-inversion. Given an error, exactly one C-element and XOR gate will be enabled. A bank of latches are included to ensure a glitch-free transaction.

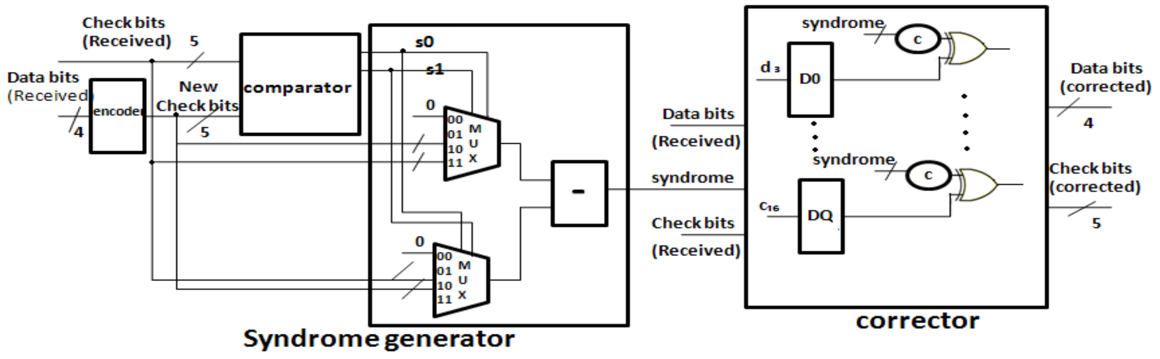
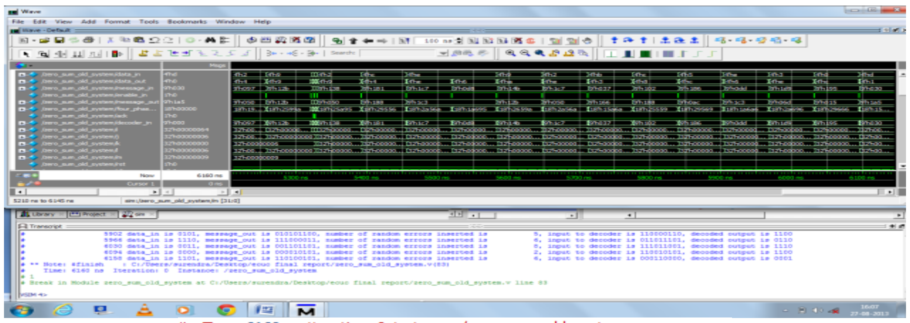


Fig.11. Microarchitecture of 4-bit zero-sum error-corrector unit design

The output of the Error Correcting Unit is given to the CD, which contains a single multi input OR gate which combines the results to produce the final Ack. Given CD replace the usage of 16 C-elements as in [1] with a single multi input OR gate. This reduces the area with less requirement of hardware components.

By the removal of timer block we are reducing the execution time i.e., reducing the delay with 1-bit detection and correction for the zero-sum code, when compared to [1] .

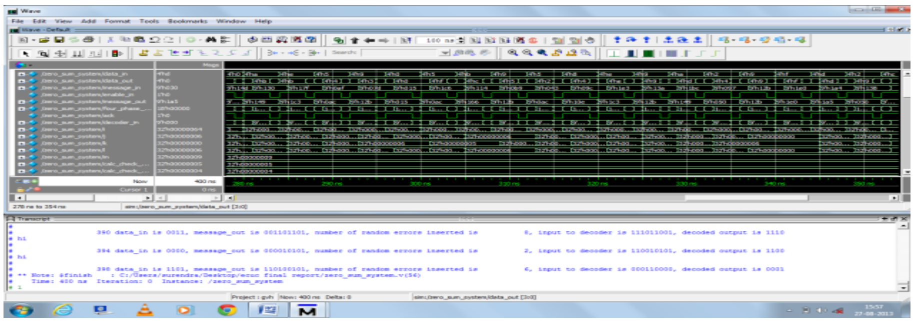
The hardware overhead of Zero-sum code was reduced by eliminating the timer-out unit in terms of both delay and area. Interestingly, as shown in fig 12& fig 13 the trend shows a strong and consistent reduction in execution time (6160ns to 400ns) from zero-sum-old system to the modified zero-sum system.



```
#            5902 data_in is 0101, message_out is 010101100, number of random errors inserted is          5, input to decoder is 110000110, decoded output is 1100
#            5966 data_in is 1110, message_out is 111000011, number of random errors inserted is          6, input to decoder is 011011101, decoded output is 0110
#            6030 data_in is 0011, message_out is 001101101, number of random errors inserted is          8, input to decoder is 111011001, decoded output is 1110
#            6094 data_in is 0000, message_out is 000010101, number of random errors inserted is          2, input to decoder is 110010101, decoded output is 1100
#            6158 data_in is 1101, message_out is 110100101, number of random errors inserted is          6, input to decoder is 000110000, decoded output is 0001
# ** Note: $finish    : C:/Users/surendra/Desktop/ecuc final report/zero_sum_old_system.v(83)
#    Time: 6160 ns  Iteration: 0  Instance: /zero_sum_old_system
# 1
# Break in Module zero_sum_old_system at C:/Users/surendra/Desktop/ecuc final report/zero_sum_old_system.v line 83
VSIM 4>
```

Time: 6160 ns Iteration: 0 Instance: /zero_sum_old_system

Fig 12: zero_ sum _old system with out modified diagram.



```
#
#            390 data_in is 0011, message_out is 001101101, number of random errors inserted is          8, input to decoder is 111011001, decoded output is 1110
# hi
#
#            394 data_in is 0000, message_out is 000010101, number of random errors inserted is          2, input to decoder is 110010101, decoded output is 1100
# hi
#
#            398 data_in is 1101, message_out is 110100101, number of random errors inserted is          6, input to decoder is 000110000, decoded output is 0001
# ** Note: $finish    : C:/Users/surendra/Desktop/ecuc final report/zero_sum_system.v(56)
#    Time: 400 ns  Iteration: 0  Instance: /zero_sum_system
# 1
```

Time: 400 ns Iteration: 0 Instance: /zero_sum_system

Fig 13:zero_sum system of modified block diagram

V. Conclusion:

A novel Zero-sum family was introduced, which supports the design of asynchronous global communication which simultaneously provides timing robustness and fault tolerance. Two enhancements has been proposed, Zero-sum+ code extends error detection to 3-bit errors and zero-sum* code provides 2-bit detection and correction .The correction coverage of 2-bit errors are upto 50% to 70%. We identified the simulation results for zero-sum code and reduction of delay and area with a modification to the block level architecture.

References

[1] M.Y. Agyekum, and S. M. Nowick,“Error-Correcting Unordered Codes and Hardware Support for Robust Asynchronous Global Communi-cation”,proc DATE,Jan 2012,PP 75-88.

[2] M. Y. Agyekum and S. M. Nowick, “An error-correcting unordered code and hardware support for robust asynchronous global communication,” in *Proc. DATE*,Mar. 2010, pp. 765–770.

[3] B. Bose, “Unidirectional error correction/detection for VLSI memory,” in *Proc.ISCA*, 1984, pp. 242–244.

[4] N. K. Jha, “Separable codes for detecting unidirectional errors,” *IEEE Trans.Comput.-Aided Des.*, vol. 8, no. 5, pp. 571–574, May 1989

[5] T. Verhoeff, “Delay-insensitive codes: An overview,” *Distrib. Comput.*, vol. 3, no.1, pp. 1–8, 1988.

[6] M. Cannizzaro, W. Jiang, and S. M. Nowick, “Practical completion detection for 2-of-n delay-insensitive codes,” in *Proc. IEEE ICCD*, Oct. 2010, pp. 151–158.

[7] M. Blaum and J. Bruck, “Unordered error-correcting codes and their applications,” in *Proc. FTCS*, Jul. 1992, pp. 486–493

[8] B. Bose, “On unordered codes,” *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 125–131,Feb. 1991.

[9] D. A. Anderson and G. Metze, “Design of totally self-checking check circuits form-out-of-n codes,” *IEEE Trans. Comput.*, vol. 22, no. 3, pp. 263–269, Mar. 1973

[10] A. Chandrasekaran and K. Boahen, “A 1-change-in-4 delay-insensitive interchiplink,” in *Proc. ISCAS*, May 2010, pp. 3216–3219.

[11] R. W. Hamming, “Error detecting and error correcting codes,” *Bell Syst. Tech. J.*,vol. 29, no. 1, pp. 147–150, 1950.

[12] J. M. Berger, “A note on error detecting codes for asymmetric channels,” *Inform.Contr.*, vol. 4, no. 1, pp. 68–73, 1961.