Low Power VLSI implementation of error detection and correction codes using OLS codes

G. Rajesh, PG Student [DECS]
Dept. of ECE, Gudlavalleru Engineering College, Gudlavalleru, Andhra Pradesh, India

T. Venkata Lakshmi,
Associate Professor, Dept. of ECE, Gudlavalleru Engineering College, Gudlavalleru, Andhra Pradesh, India

M. Kamaraju,
Professor& Head, Dept. of ECE, Gudlavalleru Engineering College, Gudlavalleru, Andhra Pradesh, India

***Abstract*— Error correction codes (ECCs) are regularly used to secure memories from errors. As multi bit errors become additional frequent, single error correction codes aren't enough and additional advanced ECCs are required. Accuracy is changing into a challenge for advanced electronic circuits as the number of errors due to manufacturing variations, radiation and reduced noise margins increase as technology scales. In the existed method the error detection and correction are achieved by using Bloom filters (BFs). These filters give a quick and economical way to check whether or not a given element belongs to a set. Bloom filters will effectively correct single error within the associated set. In the proposed method the error detection and correction are enforced by using Orthogonal Latin square codes. The major advantage of OLS codes is that they arrange a good vary of selections for the block size and also the error correction capabilities. These codes will correct multiple errors. This proposed coding system has been enforced to optimize the power for OLS codes.**

Keywords*— Bloom filters (BFs), error detection and correction, and Orthogonal Latin square codes.

I. INTRODUCTION

Error could be a condition once the output information doesn't match with the input information [1]. Throughout transmission, digital signals suffer from noise which will introduce errors within the binary bits movement from one system to another. That means a 0 bit might change to 1 or a 1 bit might change to 0.

Whenever a message is transmitted, it might get push by noise or data may get corrupted [8]. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if an error occurred throughout transmission of the message. A straight forward example of error-detecting code is parity check.

Along with error-detecting code, we are able to pass some data to work out the original message from the corrupt message that we received [3]. This type of code is termed as an error-correcting code. Error-correcting codes also deploy the similar strategy as error-detecting codes however in addition; such codes also detect the exact location of the corrupt bit.

In error-correcting codes, parity check encompasses an easy way to detect errors along with a sophisticated mechanism to work out the corrupt bit location. Once the corrupt bit is found, its value is reverted (from zero to one or one to zero) to urge the original message. To detect and correct the errors, extra bits are super imposed to the data bits at the time of transmission. The extra bits are referred to as parity bits [7]. They permit detection or correction of the errors. The data bits in conjunction with the parity bits form a code word.

Parity checking is employed for error detection. It is the best technique for detecting and correcting errors. The MSB of an 8-bits word is employed as the parity bit and the remaining 7 bits are employed as data or message bits. The parity of 8-bits transmitted word is either even parity or odd parity. Even parity suggests that the amount of 1's within the given word as well as the parity bit should be even (2, 4, 6 ...). Odd parity means the number of 1's in the given word including the parity bit should be odd (1, 3, 5 ...). The parity bit is set to zero and one depending on the type of the parity needed [10]. For even parity, this bit is set to one or zero such that the no. of "1 bits" in the entire word is even. For odd parity, this bit is set to one or zero such that the no. of "1 bits" in the entire word is odd.

II. OVERVIEW OF BLOOM FILTERS

A Bloom filter could be a compact data structures used for probabilistic illustration of a set in order to support membership queries ("Is element x in set X?"). The cost of this compact illustration could be a small probability of false positives. The structure sometimes incorrectly acknowledges an element as member of the set; however usually this is a convenient trade-off. Bloom filters were developed in the 1970's and are used since in database applications to store massive amounts of static data (Mullin, 1990). Bloom's motivation was to scale back the time it took to lookup data from a slow storage device to faster main memory. And hence could

dramatically improve the performance. However, they were found to be notably helpful in data management for modeling, storing, indexing, and querying data and services hosted by numerous, heterogeneous computing nodes.

Bloom filters are used in large databases (example: Google big table uses it to reduce the disc lookups). Bloom filters are different types like counting bloom filters and traditional bloom filters. Counting bloom filters are introduced to allow removal of elements from bloom filters. Traditional bloom filters are used to optimize the transmission over the network. In recent times bloom filter (biff) codes which are based on bloom filters have been proposed to perform error correction in big data sets.

The approach is based on the concept of algorithm based fault tolerance which is proposed to reuse existing properties or elements of the system to implement fault tolerance. In this approach traditional bloom filters can also be used. But in this case the percentage of errors that can correct is much lower. The major goal of bloom filter is to correct single bit error using compressed bloom filters. The primary step is to achieve error correction is to detect errors. This can be done by checking the parity bits. Once an error is detected a correction procedure is triggered. If error occurs in the compressed bloom filter, it can be corrected by clearing the compressed bloom filter and reconstructing it using the data set.

Compressed Bloom Filters

Compressing Bloom filter improves performance once the Bloom filter is passed as a message between nodes, notably once information should be transmitted repeatedly, and its transmission size could be a limiting factor. To illustrate, Bloom filters have been suggested as a means for sharing Web cache information. During this setting, proxies don't share the exact contents of their caches, however instead periodically broadcast Bloom filters representing their cache. However, if we elect the optimal value for k to minimize the false probability as calculated above, then p =1/2. Beneath our assumption of independent random hash functions, the bit array is actually a random string of 0's and 1's, with every entry being 0 or 1 with probability 1/2. It would therefore seem that no gain in compression when sending such Bloom filters.

III. PROPOSED METHOD

OLS codes are nothing but Orthogonal Latin square codes. These codes are multiple error correcting codes. This class of codes is derived from a set of mutually orthogonal Latin squares. A Latin square of size m is an m × m matrix that has permutations of the digits 0, 1…… and m − 1 in each its rows and columns. Two Latin squares are mentioned to be orthogonal if once they are superimposed each ordered combine of elements seems just once.

The block sizes for OLS codes are $k = m^2$ data bits and 2tm parity bits, where t is the range of errors that the code will correct and m is an integer. For a given combine of values of t and m, the corresponding OLS code exists only if there are minimum of 2t OLS of size m. Orthogonal Latin square codes are one-step majority logic decodable and have recently thought about to shield caches and memories. The basic block diagram of OLS codes are shown in Fig. 1.

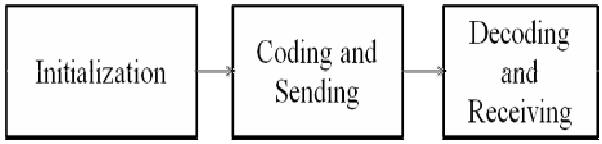

Fig. 1. Block diagram of OLS codes

In the above block diagram there are three major steps namely Initialization, encoding and decoding. In this encoding procedure OLS codes require more parity bits than the other codes to correct same number of errors. The algorithm for OLS codes encoding is as follows.

A. *Algorithm for OLS codes encoding*

1) Initialization of bits.
2) Write bit numbers in matrix type.
3) Send data by two varieties of encoding techniques.
   i) If r_1r_2=00 or 11, then there's no error, otherwise there's an error.
   ii) If r_1r_2=01 or 10 condition, there's an error.

B. *Orthogonal Latin Squares Decoding*

Orthogonal Latin square codes are decoded exceptionally high speed due to simple decoding procedure. The key parameters of an OLS code are its data block size k and also the range of errors which will be corrected t. The block sizes are typically a power of four i.e.; k=16, 64, 256. The decoding of OLS codes is completed using majority logic. In particular, the decoded bit can be obtained from the values of 2t recomputed parity check bits and the bit itself. To illustrate for the code with k = 16 and t = 2, four parity check bits and also the bit to be decoded are used. The majority vote can be enforced in two other ways. The primary option is to re-compute the four parity check bits and take a majority vote among them. Once there's a majority of ones, an error has occurred and also the bit is corrected. The correction can be finished with a xor gate.

IV. SIMULATION RESULTS

After successful code execution in XILINX 13.1 version, simulation results and power analysis of OLS codes are taken and compared with Bloom filters.

The simulation results for Bloom filters are shown in Fig. 2

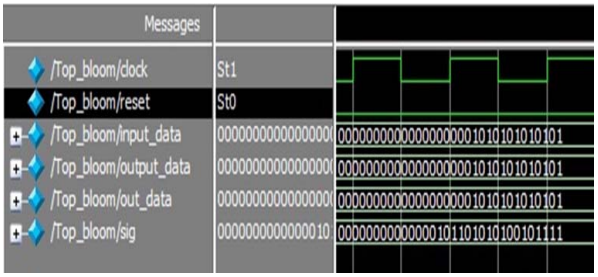

Fig. 2. Simulation results for bloom filters

The input provided in Fig. 2. is '00000000000000000001010101010101' and corresponding output is same as input. Here bloom filters can correct single error.

The simulation results for OLS codes when r_1r_2=00 are shown in Fig. 3.

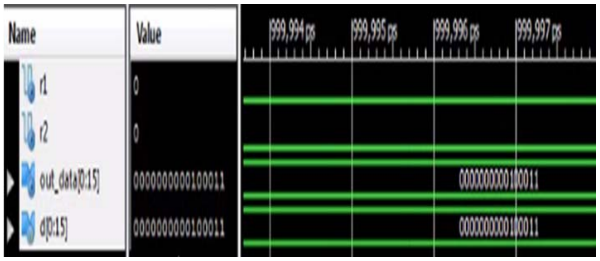

Fig. 3. Simulation results for OLS codes when r_1r_2=00

As per algorithm, if r_1r2 =00 or 11, then there is no error otherwise there is an error.

The input provided in Fig. 3. is '0000000000100011' and corresponding values of r1 and r2 are '00' i.e. r1r2=00, so the output is same as input i.e. '0000000000100011'.

The simulation results for OLS codes when r_1r_2=01 are shown in Fig. 4.

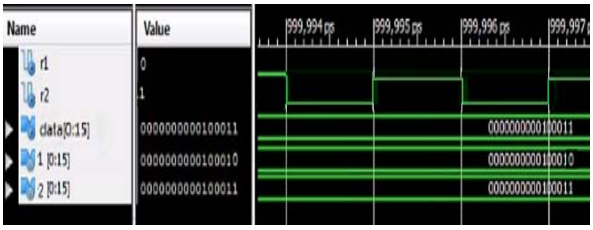

Fig. 4. Simulation results for OLS codes when r_1r_2=01

The input provided in fig. 4. is '000000 0000100011' and corresponding encoded inputs are serial encoding as '0000000000100010' and parallel encoding as '0000000000100011'. The corresponding values of r1 and r2 are '01' i.e. r1r2=01, so the output varies from input.

By using OLS codes, it can correct less than $\frac{(m+1)}{2}$ errors, where m is an integer. Here m=4, so it can correct 2 errors. But that is not in the case of bloom filters which corrects only one bit. So from simulation results it is clear that multiple error correction can be achieved by using OLS codes more effectively by utilizing less power.

The power report for OLS codes are shown in Fig. 5.

Name	Power (W)	Used	Total Available	Utilization (%)
Signals	0.000	16	---	---
IOs	0.003	34	66	51.5
Total Quiesce...	0.034			
Total Dynamic...	0.003			
Total Power	0.037			

Fig. 5. Power report for OLS codes

After generating power report as shown in Fig. 5. Total power is the summation of quiescent power and dynamic power. The total power utilization of OLS Codes is 0.037 W.

V. PERFORMANCE ANALYSIS

Table 1 shows the comparison values of Bloom Filters and OLS codes.

TABLE 1
COMPARISON OF BLOOM FILTERS AND OLS CODES

Parameter	***Bloom filters***	***OLS codes***
Power(W)	0.046	0.037
Delay(ns)	7.199	4.632
Memory(KB)	262260	163848

After generating power report as shown in Fig.5. The total power utilization of OLS Codes is 0.037 W which is 20% (approx.) less when compared to the Bloom Filters which utilizes 0.046 W.

Total Delay of OLS codes is 4.632 ns which is 36% (approx) less when compared to bloom filters which has delay of 7.199 ns.

Memory utilization of OLS codes is 163848 KB which is 62% (approx) less when compared to bloom filters 262260 KB.

Table I clearly shows that OLS codes are effectively show greater performance in terms of power, delay and memory when compared to bloom filters.

VI. Conclusion

Low power VLSI implementation of error detection and correction for OLS codes has been successfully implemented and multiple error detection and correction is achieved. The obtained results are compared with bloom filters and came to a conclusion that OLS codes perform better in aspects of power, delay and memory usage.

Acknowledgment

We would like to thank our HOD Dr. M. Kamaraju, in department of Electronics and Communication Engineering in Gudlavalleru Engineering College for his great encouragement and facilities provided for this project.

References

[1] Pedro Reviriego, Salvatore Pontarelli, Juan Antonio Maestro, and Marco Ottavi "A Synergetic Use of Bloom Filters for Error Detection and Correction" Vol. 23, No. 3, March 2015.

[2] Pedro Reviriego, Salvatore Pontarelli, Alfonso Sánchez-Macián, and Juan Antonio Maestro "A Method to Extend Orthogonal Latin Square Codes" Vol. 22, No. 7, July 2014.

[3] M. Lalita Kiranmani, G S S Prasad "Power Analysis of Concurrent Error Detection in Orthogonal Latin Squares Codec" International Journal of Research in Computer and Communication Technology", Vol 3, Issue 11, November – 2014.

[4] Kazuteru Namba and Fabrizio Lombardi "Concurrent Error Detection of Binary and Nonbinary OLS Parallel Decoders" Vol. 14, No. 1, March 2014.

[5] Pedro Reviriego, Salvatore Pontarelli, and Juan Antonio Maestro "Concurrent Error Detection for Orthogonal Latin Squares Encoders and Syndrome Computation" Vol. 21, No. 12, December 2013.

[6] S. Pontarelli and M. Ottavi, "Error detection and correction in content addressable memories by using bloom filters," IEEE Trans. Comput.vol. 62, no. 6, pp. 1111–1126, Jun. 2013.

[7] M. Mitzenmacher and G. Varghese, "Biff (Bloom Filter) codes: Fast error correction for large data sets," in Proc. IEEE ISIT, Jun. 2012.

[8] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," IEEE Trans. Knowl. Data Eng., vol. 22, no. 5, pp. 651–664, May 2010.

[9] S. Elham, A. Moshovos, and A. Veneris, "L-CBF: A low-power, fast counting Bloom filter architecture," IEEE Trans. Very Large Scale Integer. (VLSI) Syst., vol. 16, no. 6, pp. 628–638, Jun. 2008.

[10] Pedro Reviriego, Salvatore Pontarelli, Adrian Evans, and Juan Antonio Maestro "A Class of SEC-DED-DAEC Codes Derived From Orthogonal Latin Square Codes" Vol. 23, No. 5, May 2015.

Author Profile

G. Rajesh is currently pursuing master's degree program in Digital Electronics and Communication Engineering of Electronics and Communication Engineering, Gudlavalleru Engineering College, Gudlavalleru, Andhra Pradesh, India. He has completed B. Tech from NRI Institute of Technology, Agiripalli, Andhra Pradesh, India, in 2013.

T. Venkata Lakshmi is Associate Professor in Electronics and Communication Engineering department of Gudlavalleru Engineering College. She has about 13 years of experience and presented papers in several International and National Conferences and published papers in International Journals. Her area of interest includes Low power VLSI & Embedded Systems.

M. Kamaraju is Professor & Head of Electronics and Communication Engineering department of Gudlavalleru Engineering College. He has about 22 years of experience and presented papers in several International and National Conferences and published papers in International Journals. His area of interest includes Microprocessors, Microcontrollers, VLSI Design. He is the Chairman of IETE, Vijayawada.