International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:268
Vol.2, No.4, April 2013 DOI:10.15693/ijaist/2013.v2i4.80-87

VLSI Design of Fault Tolerant System with Reconfigurable Codes

Vedhanayagi.P
PG Scholar /VLSI
Sethu Institute of Technology
Virudhunagar, India

Mr.V.Karthik
Assistant Professor/Department of ECE
Sethu Institute of Technology
Virudhunagar, India

Dr.R.Ganesan
Professor and Head of VLSI
Sethu Institute of Technology
Virudhunagar, India

\

***Abstract*- Fault is one of the most challenging problems of VLSI testing. In any system, Single piece of data is lost or misintrupted, the meaning of large blocks of related data can completely change. And single event upsets (SEUs) changes the state of the digital circuits are becoming a major concern for memory applications. It's enable reliable Majority logic decodable codes are suitable for memory applications due to their capability to correct a large number of errors. However, they require a large decoding time that impacts memory performance. This paper presents an error-detection and correction technique with reconfigurable codes .The proposed fault-detection method significantly reduces memory access time when there is no error in the data read. The technique uses the Filtering Engine and Exact – matching flow engine itself to detect failures, which makes the area overhead minimal and keeps the extra power consumption low.**

***Index Terms* – ECC codes, SEUs, majority logic, LDPC, memory.**

I. INTRODUCTION

History of computing sounds like a magic trick-squeezing more and more power into less and less space-it is! What made it possible was the invention of the digital Integrated circuits in 1958. Growing technology improvement, hundred's ,thousands, millions, or even billions of electronic components are combined and create multiple job onto tiny chips of silicon no bigger than a fingernail. In over five decades of aggressive scaling, CMOS transistor technology has rapidly progressed towards nanotechnology- scale feature sizes. However, this continuous shrinking of device dimensions has also strongly affected the circuit performance [2]. Especially, SRAM memory failure rates are increasing significantly, therefore posing a major reliability concern for many applications. Initially, the data words are encoded and then stored into the memory. When the memory is read, the codeword is sent to the output for further processing. At that receiving end an error will be occur on that data. An error is a mismatch between the expected circuit outputs and the actual circuit outputs. Embedded memories are more challenging to test and diagnose. Some commonly used mitigation techniques are:

• Triple modular redundancy (TMR);
• Error correction codes (ECCs).

TMR is a special case of the von Neumann method [3] consisting of three versions of the design in parallel, with a majority voter selecting the correct output. As the method suggests, the complexity overhead would be three times plus the complexity of the majority voter and thus increasing the power consumption. For memories, it turned out that ECC codes are the best way to mitigate memory soft errors [1].
For terrestrial radiation environments where there is a low soft error rate (SER), codes like single error correction and double error detection (SEC–DED), are a good solution, due to their low encoding and decoding complexity. However, as a consequence of augmenting integration densities, there is an increase in the number of soft errors, which produces the need for higher error correction capabilities [2], [4]. The usual multierror correction codes, such as Reed–Solomon (RS) or Bose–Chaudhuri– Hocquenghem (BCH) are not suitable for this task. The reason for this is that they use more sophisticated decoding algorithms, like complex algebraic (e.g., floating point operations or logarithms) decoders that can decode in fixed time, and simple graph decoders, that use iterative algorithms (e.g., belief propagation). Both are very complex and increase computational costs [5]. Among the ECC codes that meet the requirements of higher error correction capability and low decoding complexity, cyclic block codes have been identified as good candidates, due to their property of being majority logic (ML) decodable [6], [7]. A subgroup of the low-density parity check (LDPC) codes, which belongs to the family of the ML decodable codes, has been researched in [8]–[10].

This paper explores the idea of using the ML decoder circuitry as a fault detector so that read operations are accelerated with almost no additional hardware cost. The results show that the properties of RC-LDPC enable efficient fault detection.

The remainder of this paper is organized as follows. Section II gives an overview of existing ML detector/decoder; Section III presents the proposed method; Section IV Filtering engine; Section V discusses about the exact-match flow and VI the results obtained for the different versions in respect to effectiveness, performance, and area and power consumption. Finally, Section VII discusses conclusions and gives an outlook onto future work.

II.EXISTING MAJORITY LOGIC DETECTOR/DECODER (MLDD)

A.PLAIN ML DECODER

MLD was first mentioned in [7] for the Reed–Muller codes. Then, it was extended and generalized in [8] for all types of systematic linear block codes that can be totally orthogonalized on each codeword bit. The main reason for using ML decoding is that it is very simple to implement and thus it is very practical and has low complexity.

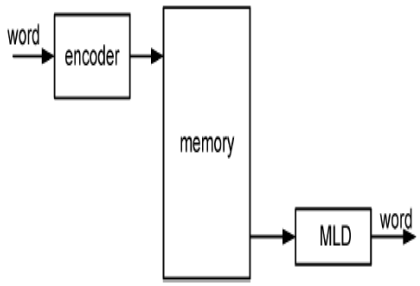


Fig.1.Memory system schematic with MLD

A generic schematic of a memory system is depicted in Fig.1. for the usage of an ML decoder. Initially, the data words are encoded and then stored in the memory. When the memory is read, the codeword is then fed through the ML decoder before sent to the output for further processing. In this decoding process, the data word is corrected from all bit-flips that it might have suffered while being stored in the memory.

The drawback of ML decoding is that, for a coded word of N-bits, it takes cycles in the decoding process, posing a big impact on system performance. This algorithm needs as many cycles as the number of bits in the input signal, which is also the number of taps, in the decoder. This is a big impact on the performance of the system, depending on the size of the code. For example, for a codeword of 73 bits, the decoding would take 73 cycles, which would be excessive for most applications.

B.ML DETECTOR/DECODER

This section presents a modified version of the ML decoder that improves the designs presented before. Starting from the original design of the ML decoder introduced in, the proposed ML detector/decoder (MLDD) has been implemented using the difference-set cyclic codes (DSCCs). This code is part of the LDPC codes, and, based on their attributes, they have the following properties:

- Ability to correct large number of errors;
- Sparse encoding, decoding and checking circuits synthesizable into simple hardware;
- Modular encoder and decoder blocks that allow an efficient hardware implementation;
- Systematic code structure for clean partition of information and code bits in the memory.

An important thing about the DSCC is that its systematical distribution allows the ML decoder to perform error detection in a simple way, using parity check sums. However, when multiple errors accumulate in a single word, this mechanism may misbehave, as explained in the following.

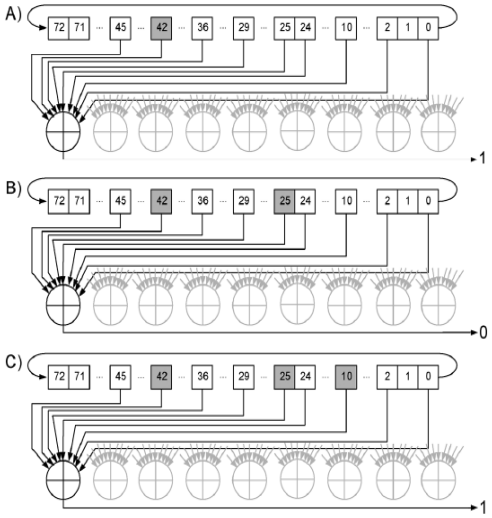


Fig.2. Single check equation of a N=73 ML decoder A) One bit-flips B) Two bit-flips C) Three bit-flips

In the simplest error situation, when there is a bit-flip in a code word, the corresponding parity check sum will be "1," as shown in Fig. 2(A). This figure shows a bit-flip affecting bit 42 of a code word with length N=73 and the related check sum that produces a "1." However, in the case of Fig.2 (B), the codeword is affected by two bit-flips in bit 42 and bit 25, which participate in the same parity check equation. So, the check sum is zero as the parity does not change. Finally, in Fig.2(C), there are three bit-flips which again are detected by the check sum (with a "1"). As a conclusion of these examples, any number of odd bit flips can be directly detected, producing a "1" in the corresponding B_J. The problem is in those cases with an even numbers of bit-flips, where the parity check equation would not detect the error. In this situation, the use of a simple error detector based on parity check sums does not seem feasible, since it cannot handle "false negatives" (wrong data that is not detected). However, the alternative would be to derive all data to the decoding process (i.e., to decode every single word that is read in order to check its correctness), as explained in previous sections, with a large performance overhead.

Since performance is important for most applications, we have chosen an intermediate solution, which provides a good reliability with a small delay penalty for scenarios where up to five bit-flips may be expected. This proposal is one of the main contributions of this project.

As described before, the ML decoder is a simple and powerful decoder, capable of correcting multiple random bit-flips depending on the number of parity check equations. It consists of five parts: a cyclic shift register; an XOR matrix; a majority gate; an XOR gate, Control unit and tristate buffer as illustrated in Fig. 3.

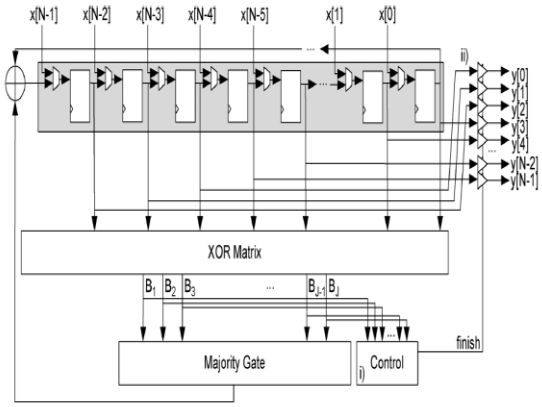


Fig.3. Schematic of MLDD

The figure shows the basic ML decoder with an N-tap shift register, an XOR array to calculate the orthogonal parity checksums and a majority gate for deciding if the current bit under decoding needs to be inverted. Those components are the same as the ones for the plain ML decoder shown in Fig. 3. The additional block to perform the error detection is illustrated in Fig.3.as:

1) The control unit which triggers a finish flag when no errors are detected after the third cycle and

2) The output tristate buffers are always in high impedance unless the control unit sends the finish signal so that the current values of the shift register are forwarded to the output *y*.

The control unit manages the detection process. It uses a counter that counts up to three, which distinguishes the first three iterations of the ML decoding. In these first three iterations, the control unit evaluates the by combining them with the OR1 function. This value is fed into a three-stage shift register, which holds the results of the last three cycles. In the third cycle, the OR2 gate evaluates the content of the detection register. When the result is "0," the FSM sends out the finish signal indicating that the processed word is error-free. In the other case, if the result is "1," the ML decoding process runs until the end.

This clearly provides a performance improvement respect to the traditional method. Most of the words would only take three cycles (five, if we consider the other two for input/output) and only those with errors (which should be a minority) would need to perform the whole decoding process. The schematic for this memory system is very similar to Fig.1, adding the control logic in the MLDD module.

Given a word read from a memory protected with DSCC codes, and affected by up to five bit-flips, all errors can be detected in only three decoding cycles. This is a huge improvement over the simpler case, where N decoding cycles are needed to guarantee that errors are detected. The proof of this hypothesis is very complex from the mathematical point of view. Therefore, two alternatives have been used in order to prove it, which are given here.

- Through simulation, in which exhaustive experiments have been conducted, to effectively verify that the hypothesis applies. It's explained in this section.
- Through a simplified mathematical proof for the particular case of two bit-flips affecting a single word. It's explained at Different-set Cyclic Codes. For simplicity, and since it is convenient to first describe the chosen design, let us assume that the hypothesis is true and that only three cycles are needed to detect all errors affecting up to five bits. This is proved in MLDD algorithm.

III.PROPOSED METHOD

Error detection and correction is one of the most challenging problems of VLSI testing. In this paper, fault free data's are achieved using a novel architecture on reconfigurable platform. This architecture produces a fault free output on time of the process. So there is no delay in this novel architecture. It's used to detect and correct all types of possible faults. After detection of error, all the faults are stored in the fault database with respect to their location. Existing method having lot of computational complexity. In novel architecture reduces the complexity. Architecture contains two major blocks; Filtering Engine and Matching Engine.

The filtering engine is a front-end module responsible for filtering out secure data efficiently and indicating to candidate positions that patterns possibly exist at the first stage. The exact-matching engine is a back-end module responsible for verifying the alarms caused by the filtering engine. Only a few unsaved data need to be checked precisely by the exact-matching engine in the second stage.

Both engines have individual memories for storing significant information. For cost reasons, only a small amount of significant information regarding the patterns can be stored in the filtering engine's on-chip memory. In this

case, we used a 32-kB on-chip memory for the fault database, which contained more than 30 000 fault codes and localized most of the computing inside the chip.

Conversely, the exact-matching engine not only stores the entire pattern in external memory but also provides information to speed up the matching process. Our exact-matching engine is space-efficient and requires only four times the memory space of the original size pattern set. The size of a pattern set is the sum of the pattern length for each pattern in the given pattern set; in other words, it is the minimum size of the memory required to store the pattern set for the exact-matching engine. In this case, 8 MB of off-chip memory was required for the fault database (2 MB).

The proposed exact-matching engine also supports data prefetching and caching techniques to hide the access latency of the off-chip memory by allocating its data structure well. The other modules include a text buffer and a text pump that prefetches text in streaming method to overlap the matching progress and text reading. A load/store interface was used to support bandwidth sharing. The architecture of reconfigurable testing shown in fig.4.

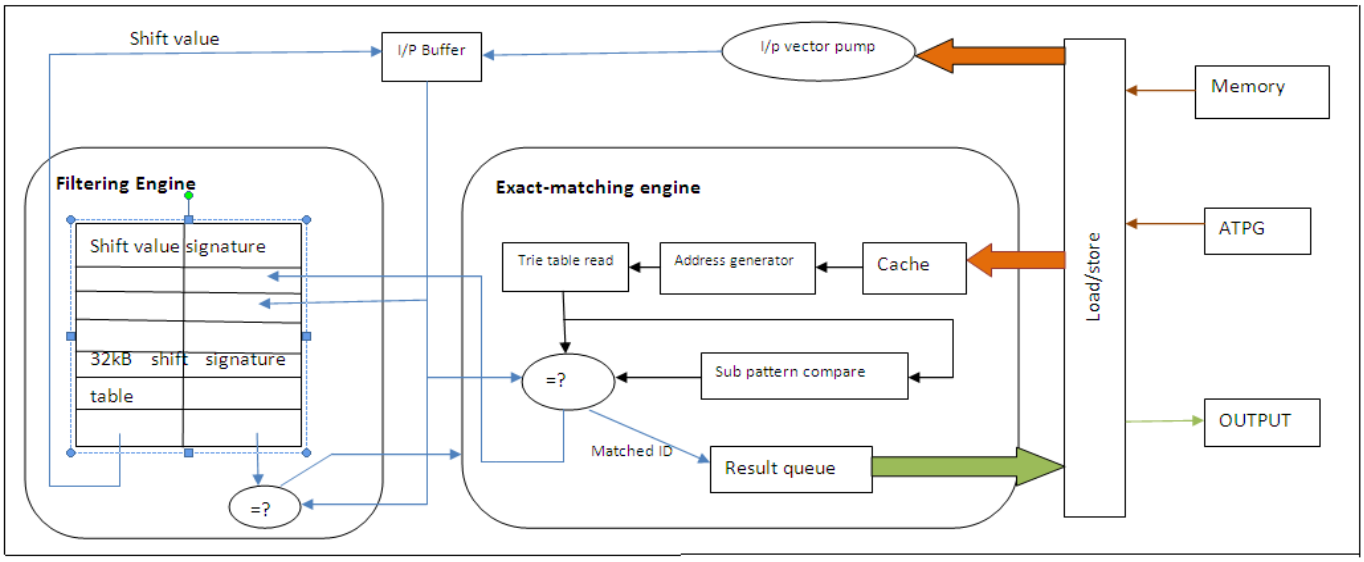


Fig.4. Reconfigurable Testing Architecture

This proposed architecture has six steps shown in Fig. 5 for finding patterns. Initially, a pattern pointer is assigned to point to the start of the given word at the filtering stage. Suppose the pattern matching processor examines the word from left to right. The filtering engine fetches a piece of word from the word buffer according to the pattern pointer and checks it by a shift-signature table. If the position indicated by the pattern pointer is not a candidate position, then the filtering engine skips this piece of word and shifts the pattern pointer rights multiple characters to continue to check the next position. The shift-signature table created by the data structure used the Bloom filter algorithm, and it provides layer filtering. If layer is missing their filter, the processor enters the exact-matching phase. The next section has details about the shift-signature table.

After filtering engine filters the word, the exact-matching engine precisely verifies this word by retrieving a trie structure [11]. This structure divides a pattern into multiple sub-patterns and systematically verifies it. The exact-matching engine generally has four steps for each check. First, the exact-matching engine gets a slice of the word and hashes it to generate the trie address. Then, the exact-matching engine fetches the trie node from memory. This step causes a long latency due to the access time of the off-chip memory.

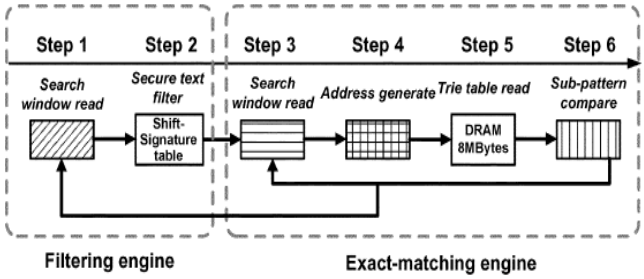


Fig.5. Two-Phase execution

Finally, the exact-matching engine compares the trie node with this slice. When this node is matched, the exact-matching engine repeatedly executes the above steps until it matches or misses a pattern. The pattern matching

then backs out to the filtering engine to search for the next candidate. The details of table generation and matching flow are explained in the following sections.

IV. FILTERING ENGINE (FE)

Designs that feature filters indicate that the action behind these filters is costly and necessary. In this work, the overall performance strongly depends on the filtering engine. Providing a high filter rate with limited space is the most important issue. We introduce a classical filtering algorithm for pattern matching in the following sections. We then show how to merge their structures in the same space to improve the filter rate.

A. Bloom Filter Algorithm

A Bloom filter is a space-efficient data structure used to test whether an element exists in a given set. This algorithm is composed of different hash functions and a long vector of bits. Initially, all bits are set to 0 at the preprocessing stage. To add an element, the Bloom filter hashes the element by these hash functions and gets positions of its vector. The Bloom filter then sets the bits at these positions to 1. The value of a vector that only contains an element is called the signature of an element. To check the membership of a particular element, the Bloom filter hashes this element by the same hash functions at run time, and it also generates positions of the vector. If all of these bits are set to 1, this query is claimed to be positive, otherwise it is claimed to be negative. The output of the Bloom filter can be a false positive but never a false negative. Therefore, some pattern matching algorithms based on the Bloom filter must operate with an extra exact-matching algorithm. However, the Bloom filter still features the following advantages: 1) it is a space-efficient data structure; 2) the computing time of the Bloom filter is scaled linearly with the number of patterns; and 3) the Bloom filter is independent of its pattern length.

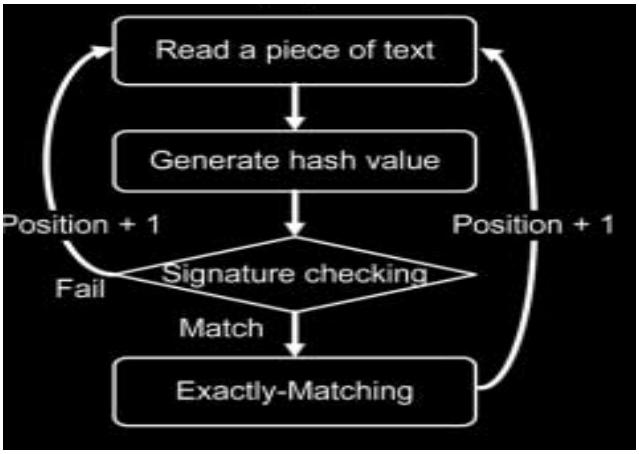


Fig.6. Bloom Filter matching process

Fig. 6 describes a typical flow of pattern matching by Bloom filters. This algorithm fetches the prefix of a pattern from the word and hashes it to generate a signature. Then, this algorithm checks whether the signature exists in the bit vector. If the answer is yes, it shifts the search window to the right by one character for each comparison and repeats the above step to filter out safe data until it finds a candidate position and launches exact-matching.

B. Shift-Signature Algorithm

The proposed algorithm re-encodes the shift table to merge the signature table into a new table named the shift-signature table. The shift-signature table has the same size as the original shift table, as its width and length are the same as the original shift table. There are two fields, S-flag and carry, in the shift signature table. The carry field has two types of data: a shift value and a signature. These two data types are used by two different algorithms. Thus, the S-flag is used to indicate the data type of a carry. The filtering engine can then filter the word using a different algorithm while providing a higher filter rate. The method used to merge these two tables is described as follows.

First, the algorithm generates two tables, a shift table and signature table, at the preprocessing stage. The generation of the shift table is the same as in the normal filter algorithm. The shift table is used as the primary filter. The signature table could be considered a set of the bit vector of the Bloom filter, and it is used for the second-level filtering. The signature table's generation is similar to the Bloom filter but is not identical; it hashes the tail characters of patterns to generate their signatures instead of the prefix. Generated signatures are mapped onto the signature table and indexed by bad-characters, which have shift values of zero in the shift table. In other words, a pattern is assigned a zero shift value in the shift table by its last characters, and it uses the same index to locate its signature in the signature table. After the shift table and signature table are generated, the algorithm re-encodes the shift value into two fields: an S-flag and a carry in the shift-signature table. The S-flag is a 1-bit field used to indicate the data type of the carry. Two data types, shift value or signature, are defined for a carry. The size and width of the shift-signature table are the same as those of the original shift table. To merge these two tables, the algorithm maps each entry in the shift table and signature table onto the shift-signature table. For the non-zero shift values, the S-flags are set, and their original shift values are cut out at 1-bit to fit their carries. Conversely, for the zero shift values, their S-flags are clear, and their carries are used to store their signatures. In this method, all of the entries in the shift-signature table contribute to the filtering rate at run time. Because of the address collision of bad-characters, most entries contain less than half of the maximum shift distance for a large pattern set. Therefore, although this method sacrifices the maximum

shift distance, the filter rate is not reduced but rather improved.

V.EXACT-MATCH ENGINE (EME)

The EME must verify the false positives when the filtering engine alerts. It also precisely identifies patterns for upper-layer applications. Most exact-match algorithms use the two kinds of trie structures shown in Fig. 7, loose and compact tries, to establish their pattern databases. Both trie structures have their merits. The AC algorithm uses loose tries, which check each input character in a constant amount of time because of their fan-out states for all possible input characters. Thus, the input data do not affect the AC-based algorithm's performance, but their memory requirements increase exponentially with pattern size. Unlike loose tries, compact tries construct pattern databases with two pointers, sibling and child, to reduce their memory requirements. However, this method has potential performance problems because it may redundantly search link lists formed by sibling pointers. Despite this limitation, compact tries are still highly practical because, in practice, attack texts are not easy to generate. Attacks can be avoided by removing patterns that cause attacks before constructing the pattern database. For this reason, we use compact tries as our exact-matching engine's algorithm, and we propose several solutions to mitigate the effect of algorithmic attacks. Fig.8. shows exact-matching flow.

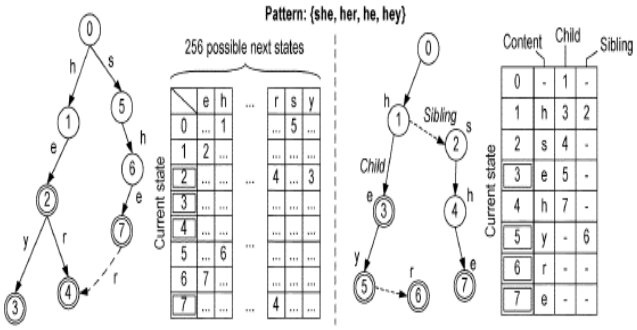


Fig.7. a) Compact trie b) FSM of AC algorithm

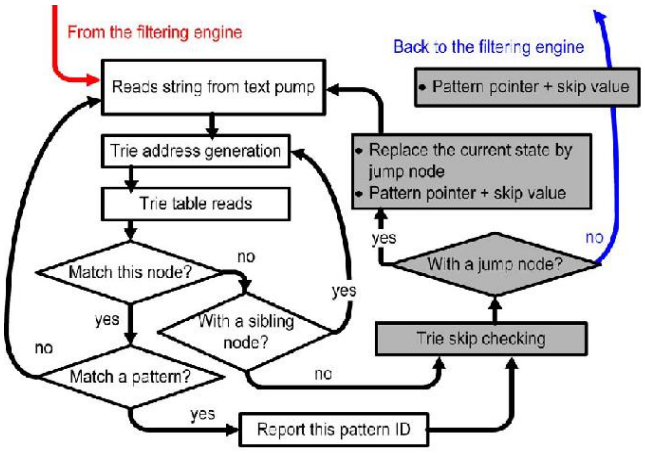


Fig.8. Exact-matching Flow

VI.RESULTS

A.AREA

As mentioned several times, this is compensated with a clear savings in area. To study this, the three designs have been implemented in VHDL and synthesized, for different values of N, using a XC65LX75 device. The obtained results are depicted, in number of equivalent gates, in Table 1.

No. of Bits	Area utilization for SFD	Area utilization for MLD	Overhead Reduced %	Area utilization for MLDD	Overhead Reduced %
21	59	53	11.32	46	37.20
73	201	180	11.66	157	28.02
273	754	643	17.26	584	29.10

Table.1. synthesis results for three designs with different code lengths

No . Of bits	Area utilization for SFD	Area Utilization for RFTS	Overhead reduced %
21	59	39	51.28
73	201	136	47.79
273	754	523	44.16

Table.2. synthesis results for RFTS with different code lengths

The conclusions on the area results are given as follows.

• The MLD design requires large area compared with the other two designs. However, as seen before, the performance results are not very good.

•The MLDD version has a very similar performance to SFD; however it requires a much lower area overhead, ranging from 29.1% to 31.2%.

• The Reconfigurable Fault Tolerant System (RFTS) version has a very similar performance to MLDD; however it requires a much lower area, ranging from 44.16% to51.28% of overhead reduced.

These conclusions can be extrapolated to power. The overhead reduced by RFTS, contrary to the MLDD case.

B. SPEED & MEMORY ACCESS TIME

The memory read access time of the plain MLD is directly dependent on the code size, i.e., a code with length 73 needs 73 cycles, etc. Then, two extra cycles need to be added for I/O. On the other hand, the memory read access delay of the MLDD is only dependent on the word error rate (WER). If there are more errors, then more words need to be fully decoded. RFTS is not dependent on size and error.

In Table 3, a comparison of the MLDD and RFTS techniques is provided for several values of N. Although this is only a best case scenario, because it is assumed that all words come without errors, it gives the idea of how much speed-up can be obtained in an ideal situation. In a real situation, a fraction of the words would have bit-flips. This fraction is represented by the WER. Since MLDD needs five cycles to handle correct words N+5 and for erroneous words, the average performance would be

MLDD – performance = (1-WER).5+ WER.(N+5).

Using this expression, the performance of the three techniques has been studied for different values of the WER.

The first comment on these results is that the MLD technique has the worst performance, whose value is independent of the WER (i.e., it needs the same number of cycles to handle correct and erroneous data). And MLDD is very similar in this aspect, since both values are very close.

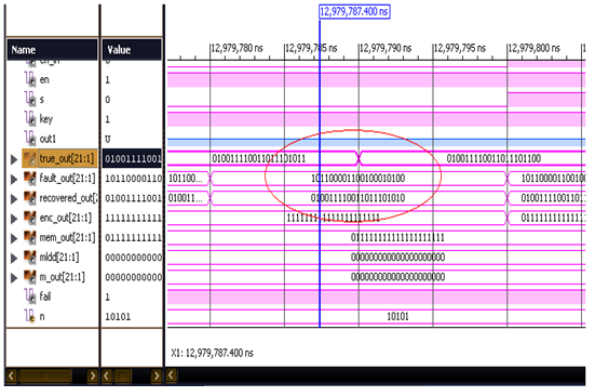
	MLDD		Reconfigurable output	
No of Bits	Area Overhead reduced in %	Speed in MHz	Area Overhead reduced in %	Speed in MHz
21	37.20	879.27	51.28	879.27
73	28.02	780.39	47.79	780.39
273	29.10	879.27	44.16	879.27

Table.3. Speed-up for different code lengths

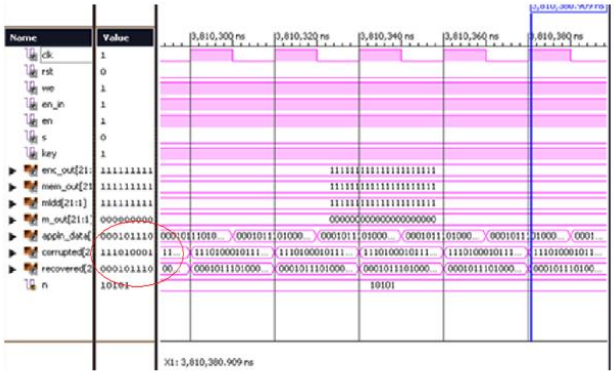
This small performance difference is compensated for with the area savings that MLDD provides. This difference is even smaller for large values of N and WER. Both the technique only detects and corrects fixed type of faults only. But our proposed technique (RFTS) is detected and corrects all type of faults. So it's independent of word size and fault types.

C. SIMULATION RESULT

OUTPUT FOR 21 BIT-EXISTING



OUTPUT FOR 21 BIT-PROPOSED



VII. CONCLUSION

In this paper, Fault –detection mechanism, RFTS, has been presented based on filtering engine and Exact-match engine. Exhaustive simulation test results show that the proposed technique is able to detect any pattern of faults at on-time process. This improves the performance of the design with respect to the traditional MLD and MLDD approach.

On the other hand, the RFTS error detector module has been designed in a way that is independent of the code size. This makes its area overhead quite reduced compared with other traditional approaches such as Majority logic decoder(MLD) and Majority logic detector/decoder(MLDD).we have proposed a reconfigurable architecture error control framework to address the performance, energy and reliability issues in a variable fault and dense environment. With the assistance of the reconfigurable framework, configuration of the error control codec used in the VLSI environment can consider both the fault conditions and delay time. As a result, delay, area, energy and reliability are simultaneously managed. In

future, this approach is motivated to solve the traffic and better to manage the fault injected in Network on Chip (NOC) links.

REFERENCES

[1] Shih-Fu Liu, Pedro Reviriego, "Efficient Majority Logic Fault Detection With Difference-Set codes for Memory Applications", IEEE Trans.VLSI., VOL.20, NO.1, JANUARY 2012.

[2] M. A. Bajura *et al.*, "Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs," *IEEE Trans. Nucl. Sci.*,vol. 54, no. 4, pp. 935–945, Aug. 2007.

[3] J. von Neumann, "Probabilistic synthesis of reliable organisms from Unreliable components," *Automata Studies*, pp. 43–98, 1956.

[4] R. Naseer and J. Draper, "DEC ECC design to improve memory reliability in sub-100 nm technologies," in *Proc. IEEE ICECS*,2008, pp.586–589

[5] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2004.

[6] I. S. Reed, "A class of multiple-error-correcting codes and the decoding scheme," *IRE Trans. Inf. Theory*, vol. IT-4, pp. 38–49, 1954.

[7] J. L. Massey, *Threshold Decoding*. Cambridge, MA: MIT Press, 1963.

[8] S. Ghosh and P. D. Lincoln, "Low-density parity check codes for error correction in nanoscale memory," SRI Comput. Sci. Lab. Tech. Rep.CSL-0703, 2007.

[9] B. Vasic and S. K. Chilappagari, "An information theoretical framework for analysis and design of nanoscale fault-tolerant memories based on low-density parity-check codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 54, no. 11, pp. 2438–2446, Nov. 2007.

[10] H. Naeimi and A. DeHon, "Fault secure encoder and decoder for NanoMemory applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 4, pp. 473–486, Apr. 2009.

[11] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490–499, 1960.

[12] R. S. Boyer and J. S. Moore, "A fast string searching algorithm,"*Commun. ACM*, vol. 20, pp. 762–772, 1977.

[13] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet patternmatching using TCAM," in *Proc. 12th IEEE Int. Conf. Netw. Protocols*, 2004, pp. 174–183.

[14] P. Piyachon and Y. Luo, "Efficient memory utilization on network processors for deep packet inspection," presented at the ACM/IEEE Symp.Arch. for Netw. Commun. Syst., San Jose, CA, 2006.

[15] C.-C. Wang, C.-J. Cheng, T.-F. Chen, and J.-S. Wang, "An adaptively dividable dual-port BiTCAM for virus-detection processors in mobile devices," *IEEE J. Solid-State Circuits*, vol. 44, no. 5, pp. 1571–1581, May 2009.

[16] Z. K. Baker and V. K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," presented at the ACM Symp. Arch. for Netw. Commun. Syst., Princeton, NJ, 2005.

[17] Chieh-jen cheng,Chao-ching wang, Wei-chun ku, Tie – Fu chen, Jinh-shyan wang "A Scalable High-perfomance Virus Detection Processor Against a large pattern set for Embedded Network Security," IEEE Trans.VLSI Systems,vol.20,No.5 May 2012

Authors Profile

P.VEDHANAYAGI received the **B.E.** degree in Electronics and Communication Engineering from the PSR Engineering College Sivakasi, Anna University, Chennai, India, in 2008. Currently doing **M.E.** in VLSI Design in Sethu Institute of Technology Kariapatti, Anna University Chennai, India. Her research interest includes VLSI Testing, Low Power VLSI, Networking and Wireless Communication.

V. KARTHIK received the B.E. degree in electronics and communication engineering from National Engineering College, Kovilpatti, Anna University, Chennai, and received the M.Tech degree in VLSI Design from SATHYABAMA UNIVERSITY, India, in the year of 2009 and 2011 respectively. He is presently working as Assistant Professor, Department of Electronics and Communication Engineering at Sethu Institute of Technology India. He has published 9 research papers in the National & International Conferences.

Dr.R.Ganesan received his B.E. Instrumentation & Control Engineering from Arulmigu Kalasalingam College of Engineering and ME (Instrumentation) from Madras Institute of Technology in the year 1991 and 1999 respectively. He has completed his PhD from Anna University, Chennai, India in 2010. He is presently working as Professor and head in the department of M.E-VLSI Design at Sethu Institute of Technology, India. He has published more than 25 research papers in the National & International Journals/ Conferences.
His research interests are VLSI design, Image Processing, Neural Networks and Genetic algorithms.