# An LDPC Decoding Method for Fault-Tolerant Digital Logic

Yangyang Tang*, Chris Winstead†, Emmanuel Boutillon*, Christophe Jego‡ and Michel Jezequel§
*Universite de Bretagne Sud, UMR CNRS 3192 Lab-STICC, Lorient, France
†Dept. of Electrical and Computer Engineering, Utah State University, Logan, Utah 84322
‡Institut Polytechnique Bordeaux, UMR CNRS 5218 Lab-IMS, Bordeaux, France
§Institut TELECOM/TELECOM Bretagne, UMR CNRS 3192 Lab-STICC, Brest, France
Email: yangyang.tang@univ-ubs.fr

*Abstract*— **A decoding algorithm and logic implementation is proposed for fast, low-complexity error correction in environments with a high rate of transient faults as well as hard errors. The circuit is able to correct a single error in one clock cycle, making it suitable for mitigating faults in pipelined digital logic systems. The proposed method is also resilient against internal transient gate errors that may occur within the decoder itself. In the presence of a high input error rate (0.001) and high internal gate fault rate ($10^{-5}$), the new decoding algorithm is able to reduce the error probability by two orders of magnitude. An asynchronous implementation is also presented for the new algorithm, which performs iterative error-correction with reduced latency compared to synchronous algorithms.**

## I. Introduction

In the emerging nano-scale era, electronic devices are increasingly susceptible to logic errors that can degrade the reliability and longevity of integrated electronic systems. Logic errors may arise from short transient events induced by noise or exogenous interference. Errors may also arise from permanent manufacturing defects, device damage [1], or from long-time transient events [2]. All of these error varieties are expected to increase in post-CMOS nano-scale devices. They are also expected to pose challenges in three-dimensional integrated systems, where high thermal density increases the likelihood of transient upsets due to thermal noise, and permanent defects due to heat damage.

There is consequently an increased interest in developing fault-masking techniques that provide tolerance for both momentary upsets and long-term defects. It is also important for a fault-masking technique to be tolerant of internal errors within its own logic, since any embedded fault-masking solution is implemented using the same devices as the logic it protects. Researchers previously investigated the performance of a variety of error-correcting schemes under the influence of intrinsic faults. One of the authors (Winstead) proposed an LDPC-coded Fault Compensation Technique (LFCT) [3] which achieves reliable operation in the presence of transient and permanent defects. The LFCT method uses Gaudet and Rapley's theory of stochastic decoding [4], [5] to correct errors that appear at the output of some logic computation.

In this paper, we present a new LDPC Stochastic Decoding (LSD) architecture, which can be considered as a circuit-level implementation of the LFCT concept. The LSD algorithm is similar to the well-known Gallager-B algorithm [6]. Where the Gallager algorithm uses majority logic, the new decoding method uses a cascade of Muller C-element gates [7]. The C-element is a flip-flop with two inputs $a$ and $b$, and a single output $c$. When $a = b$, the output is latched so that $c = a = b$. When $a \neq b$, the output $c$ holds its value. In the LSD approach, C-elements are used to provide state memory along the cascade. The memory suppresses transient upsets at each point in the cascade, ultimately leading to a reduced rate of uncorrected errors. We also introduce an asynchronous architecture that implements the LSD algorithm without clocked iterations.

The remainder of this paper is organized as follows: Section 2 presents the LSD algorithm. Section 3 presents the asynchronous architecture implementation. Experimental results are given in Section 4. Discussion and conclusions are given in Section 5 and Section 6, respectively.

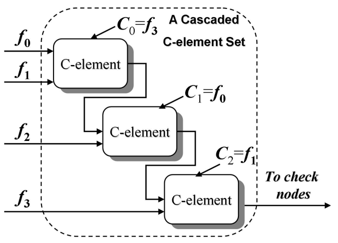
## II. The New LDPC Stochastic Decoding Technique

### A. Code Structure and Decoding Algorithm

The proposed decoding method is based on the Tanner graph [8] of a parity-check code. As usual, the graph contains two sets of nodes – variable nodes $v_i$ and parity-check nodes $p_j$. During the decoding process, binary messages are exchanged between the two sets of nodes. The message passed from node $v_i$ to $p_j$ is written $y_{ij}$, and the returning message from $p_j$ to $v_i$ is written $f_{ji}$. The parity-check nodes perform a modulo-2 summation over their input messages, as is usual with stochastic and Gallager-style decoders [3], [6]. These operations are implemented by using a cascade of XOR gates. In the proposed LSD algorithm, C-element gates are used to implement the variable nodes, as is done with conventional stochastic LDPC decoders [4], [5]. Unlike stochastic decoders, our LSD method introduces a new two-phase operation for the variable node, described as follows.

For each variable node $v_i$, we associate a set of C-element gates $C_k$, $0 \leq k < (d_v - 1)$, where $d_v$ is the variable degree. Each C-element gate $C_k$ contains a single-bit storage element $c_k$. The error correction algorithm is:

1) Initialize $y_{ij} = x_i$ (the channel-side message), for all $i$.
2) Compute $f_{ji} = \oplus_{m \in P_{j \setminus i}} y_{mj}$ for all $j$.



|  |  |  |  |  | Initialized |  |  | After 1 iteration |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $c_0$ | $c_1$ | $c_2$ | $c_0$ | $c_1$ | $c_2$ |
| Error-free | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Error in $f_0$ | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Error in $f_1$ | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Error in $f_2$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Error in $f_3$ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Fig. 1. Local implementation of a variable node. The circuit is a cascade of C-element gates, modified for the initialization of the state memory. In this figure, $d_v = 4$. Each C-element has three inputs; the left-side inputs are the usual inputs, and the top-side input is the initial state for the C-element's memory. Moreover, a table that illustrates the behaviors of $c_k$ is given.

3) For a given variable node $v_i$, let $f_0, \cdots, f_{d_v}$ be the locally received messages for this node, where $f_0 = x_i$ (the channel-side message), and $f_1, \cdots, f_{d_v}$ are the message from parity-check nodes. Initialize each memory as $c_k = f_m$, where $m = (k + d_v) \bmod (d_v + 1)$.
4) The C-element's port connections are as follows. For $C_0$, the inputs are $f_0$ and $f_1$, and the output is $c_0$. For the remaining $C_k$, the inputs are $c_{k-1}$ and $f_{k+1}$, and the output is $c_k$.
5) Iterate steps 2 and 4 during a fixed number of iterations. The initialization in step 3 is performed only during the first iteration.
6) The corrected output is $z_i = c_{(d_v - 1)}$.

The algorithm is able to correct a single error during one iteration, and can correct many multi-error patterns as well. Fig. 1 shows the structure of a variable node, such as, a cascaded C-element set.

## B. Error-correction Analysis

An error appearing at some $f_k$ may originate from an internal error within the decoder. The C-element cascade helps to stop the propagation of such errors. For example, if a transient upset occurs in one of the C-element's state memories, this error is masked by the subsequent C-element in the cascade. In the rest of this section we show that single error events are corrected by the LSD algorithm, regardless of where those errors originate.

*1) A Single Error Apprears at $x_i$:* If a single error occurs at some input $x_i$, then the error is initially propagated along several messages $y_{ij}$. The error is further propagated to the messages $f_{jm}$ for all nodes $m \in P_j$. If the code has no four-cycles (i.e. cycles of length four), no variable node receives more than one erroneous $f_{jm}$ message. Suppose that erroneous messages are sent from variable node $i$ to distinct check nodes $j_1$ and $j_2$, causing both $j_1$ and $j_2$ to send erroneous messages to the same variable node $m$. Then, a cycle has to link nodes $i \rightarrow j_1 \rightarrow m \rightarrow j_2 \rightarrow i$, which has length four. If four-cycles are excluded, then this cannot occur. Furthermore, no erroneous message is propagated back to variable node $i$ because all other nodes carry correct values. Therefore, each variable node receives at most a single erroneous message among the locally-received messages $f_k$. As shown in the next subsection, all such single errors are corrected.

*2) A Single Error Apprears at $f_k$:* If variable node $v_i$ receives one erroneous message $f_k$, then $c_m$ is initialized to an erroneous value, but all other $c_k$ $(k \neq m)$ are initialized to correct values. Then, for each C-element, there are three possible cases:

1) The C-element has two correct inputs and an initially erroneous memory state. In this case, the memory state is corrected because the inputs agree.
2) The C-element has a correct memory state but one incorrect input. In this case the correct memory state is retained because the inputs do not agree. Hence, the fault is masked by the C-element's memory.
3) The C-element has two correct inputs and an initially correct memory state. In this case there is no error.

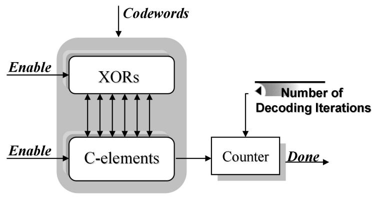


Fig. 2. Asynchronous iterative decoding architecture.

## III. Implementation of the LSD method

### A. Asynchronous Iterative Decoding Architecture

In this subsection, we present an asynchronous iterative decoding architecture that uses handshaking signals to control the iterative operations. This approach allows the decoder to operate at its maximum local speed, which is likely to be faster than the main system clock. The asynchronous decoder architecture is shown in Fig. 2. The XOR and C-element blocks are triggered by an external *Enable* signal. An internal *Enable* flag then ripples through each gate in the respective cascades, controlling the flow of operations. Once all cascade operations are finished in one block, the other block is enabled. Each time the C-element block completes its operations, an iteration counter is incremented. Once the counter reaches a pre-defined number of iterations, it asserts a *Done* signal to indicate that decoding is finished.

### B. Handshaking signal flow

Fig. 3 shows the flow of handshaking signals in the asynchronous LSD architecture. Its operations are detailed as follows:

- During the initialization phase, $Rst = 0$. During this phase, the input codeword bits are loaded into the XORs, and signal *Flag* is set as low. After the XOR computation are complete, the *Flag* signal is toggled to indicate that data is available for the C-elements. The first C-element of each variable node is enabled by *Flag*. When the first C-element operation is complete, it propagates the enabling *Flag* to the second C-element, and so on until the cascaded operations are complete. The last C-element

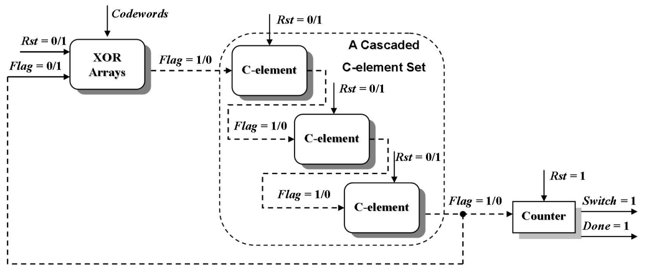


Fig. 3. Asynchronous handshaking signal flow for the LSD algorithm.

passes its *Flag* to the iteration counter. The counter then responds with a signal $Switch = 1$ to indicate the end of initialization phase.

- Once the *Switch* signal is asserted, the error-correction phase is initiated by asserting $Rst = 1$. The propagation of handshaking signals occurs as in the initialization phase. When the iteration counter reaches a pre-set number of iterations, it asserts the *Done* signal to indicate that decoding is complete.

The signal flow presented in Fig. 3 represents a handshaking specification. The circuit-level implementation of this handshaking procedure can be synthesized by following well-known rules for delay-insensitive circuits [9].

### *C. Application of the LSD architecture in fault-tolerant logic.*

In this subsection, we apply the LSD architecture to achieve fault-resilience in a digital logic circuit. The LSD algortihm is applied to a digital computation as shown in Fig. 4. A logic function $F(x)$ is implemented using a digital technology that is subject to errors at its output. The original function $F(x)$ is augmented by the addition of a redundant parity-generator module, $E \cdot F(x)$, where $E$ represents the encoding function that generates parity bits codeword space at the output of $F(x)$ as in [3]. The *systematic* output word $s$ with a length $K$ from $F(x)$ is then concatened with the parity outputs $r$ from $E \cdot F(x)$, yielding a complete codeword $[s\ r]$ of length $N$. If there are no errors, then the resulting codeword should satisfy the standard parity-check constraint, $[s\ r]\,H = 0$, where $H$ is the parity-check matrix that defines the error-correction code.

According to the code's $H$ matrix, the LSD architecture is synthesized by XOR and C-element modules, which are indicated by $\boxed{+}$ and $\ominus$, respectively. Based on the code's Tanner Graph, the circuit's interconnect corresponds to an interleaver, which is synthesized from well-known rules [8]. The system's final decisions $z$ are taken from the C-elements modules.

In order for the LSD method to function properly, the encoding logic must be folded into the duplicate functions in a way that avoids correlated errors among the parity bits. Fig. 4 shows the *functional* form of $E \cdot F(x)$ as a cascade of the function $F(x)$ with the encoding function $E$. If the architecture were physically implemented, then an error in $F(x)$ might propagate through $E$ to generate several errors in the parity bits, leading to an uncorrectable pattern. The

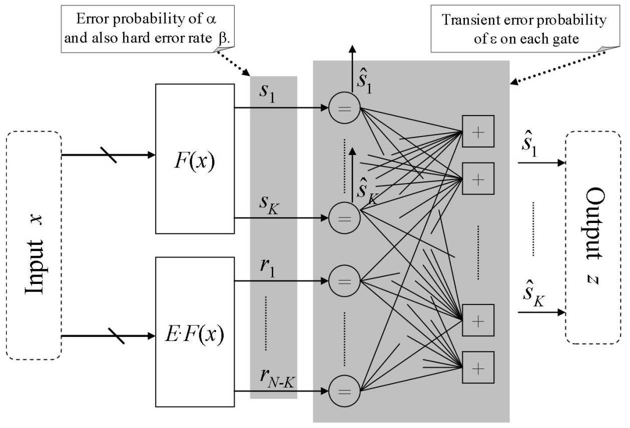

Error probability of α and also hard error rate β.
Transient error probability of ε on each gate
Input x
F(x)
EF(x)
$s_1$
$s_K$
$r_1$
$r_{N-K}$
$\hat{s}_1$
$\hat{s}_K$
Output z

Fig. 4. The proposed fault-tolerant architecture. The shaded area represents an asynchronous LSD decoder implementation.

composite function $E \cdot F(x)$ must therefore be synthesized as a flat Boolean function to avoid correlated error patterns. The most reliable approach is to use a flat truth-table synthesis, as is done with cross-bar logic arrays [10]. This approach guarantees that error events occur independently on all the codeword bits, so that correlated error bursts are precluded.

## IV. Simulation Results

To evaluate the proposed architecture, a system composed of five systematic regular (4, 8) LDPC codes was simulated. All codes are rate 1/2, meaning that there is one redundant parity bit for each systematic bit, i.e. $N = 2 \cdot K$. Moreover, the codeword sizes $N$ are set as 64, 128, 256, 512, and 1024. Longer codes are expected to provide better error protection but are also more complex to integrate.

In our simulations, the output bits $s$ and $r$ from $F(x)$ and $E \cdot F(x)$, respectively, are assumed to have a uniform independent error probability of $\alpha$. The XORs and C-elements that comprise the LSD architecture are assumed to be faulty boolean operations. Each gate in the LSD architecture has a uniform error probability of $\epsilon$. To take into account of the impact of hard defects,"stuck-at" faults were also inserted in uniformly random positions in $[s\ r]$ with a fault rate of $\beta$, here, $\beta = 0.001$.

The Bit Error Rate (BER) results for the LDPC codes applying LSD algorithm are shown in Fig. 5. As long as $\alpha < 0.05$, the output $s$ from $F(x)$ can be recovered with a significantly reduced error probability. As $\alpha$ is reduced below $\beta$, the performance becomes dominated by the gate-level fault probability $\epsilon$. The results show that, in the case of $\beta = 10^{-5}$, the LSD architecture introduces gains about two orders of magnitude by comparison with uncoded data (output $s$ from $F(x)$). Consequently, the proposed deocding method is able to suppress the resulting error probability to a level equal to the decoder's internal fault rate. For the sake of asychronous iterative decoding, the extra time cost of each iteration is considered as the latency of one XOR operation plus $(d_v - 1)$ C-element operation. Note that the results are obtained after five asynchronous iterations of the LSD architecture.

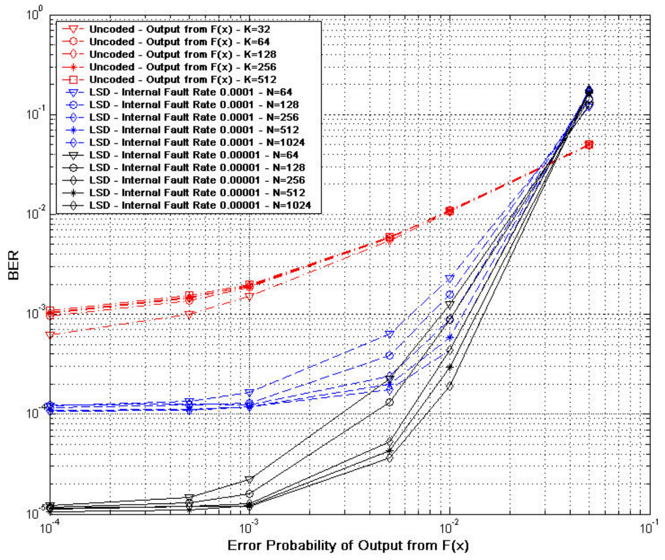


Fig. 5. Simulation results for rate-1/2 LDPC codes based on LSD architecture with five iterations. The hard-fault rate $\beta$ is 0.001.

## V. Discussion

To evaluate the redundancy cost of the proposed architecture, we suppose $F(x)$ is a large block of crossbar logic (i.e. gate array logic) representing a minimized sum-of-products expression. The crossbar fabric may consist, for example, of a layer of AND operations followed by a layer of OR operations, which may be used for flat truth-table synthesis. This style of logic is generally not optimal in terms of device count, but it is useful in our analysis for three reasons. First, crossbar logic is quite popular in research on next-generation, post-CMOS logic implementation, and reliability is a key concern for crossbar implementations [10].

The second reason is that a single gate-level fault will typically propagate only to a single output of $F(x)$. In alternative combinational styles, a single gate fault may propagate catastophically to many outputs. The LSD architecture is likely to fail in such cases of sudden error bursts. Therefore, we have to constrain its application to logic styles such as crossbar fabrics where such bursts are precluded.

The third motivation for considering crossbar logic is that it provides a precise strategy for synthesizing the auxilliary function $E \cdot F(x)$. For arbitrary functions $F(x)$, the average complexity in terms of logic gates to synthesize $E \cdot F(x)$ is close to the average number of logic gates necessary to synthesize $F(x)$. Hence the LSD method requires an average extra complexity of approximately two, plus the additional cost by implementing the decoder.

For a given LDPC code with length $N$, the number of C-elements assigned for the design of the decoder is approximately $N \cdot d_v$, where $d_v$ is the average variable node degree. The number of XOR gates is approximately $N \cdot R \cdot d_c$, where $R$ is the code's rate ($R = 1/2$ in our example simulations), and $d_c$ is the parity-check node degree. In one of the considered codes, $N = 1024$, $R = 1/2$, $d_v = 4$ and $d_c = 8$. In this case the approximate total hardware complexity is 8k logic gates.

One interesting result of our experimentations is that the LSD method provides similar performance across a wide range of code lengths. Error-correcting benefits may therefore be obtained for short codes, such as the $N = 64$ example, which have a length compatible with data words in current 64-bit processors. As suggested in [3], the LSD method can also achieve a reduced net redundancy if several parallel operations are packaged together, yielding a larger code size. In this "bundled-function" case, the relative hardware complexity of the decoder is reduced compared to the complexity of the functions themselves.

## VI. Conclusion

In order to correct the internal hardware errors of a combinational logic block, we proposed a new LDPC Stochastic Decoding (LSD) technique as a forward error correction mechanism. The originality of LSD method is the use of C-elements to implement the symbol node computations. C-elements are able to block the propagation of internal faults within the decoder, yielding a design that is robust to transient errors. The resulting algorithm is similar to the well-known Gallager-B method, but with additional error protection from the C-elements' state memory. Our simulation results show that LSD approach is able to reduce the error probability $\alpha$ at the output of a combinational function by multiple orders of magnitude.

## Acknowledgment

This work was supported by the US National Science Foundation under award ECCS-0954747.

## References

[1] J. Sequra and C. F. Hawkins, *MOS Electronics: How it works, how it fails*, Wiley-IEEE Press, 2004.
[2] R. C. Baumann, "Soft errors in advanced semiconductor devices-part i: the three radiation sources," *IEEE Transactions on Device and Materials Reliability*, vol. 1, no. 1, pp. 17–22, 2001.
[3] C. Winstead and S. Howard, "Probabilistic LDPC-coded fault compensation technique for reliable nanoscale computing," *IEEE Transactions on Circuits and Systems II – Express Briefs*, vol. 56, no. 6, pp. 484–488, June 2009.
[4] V.C. Gaudet and A Rapley, "Iterative decoding using stochastic computation," *Electronics Letters*, vol. 39, no. 3, pp. 299–301, 2003.
[5] C. Winstead, V. C. Gaudet, A. Rapley, and C. Schlegel, "Stocahstic iterative decoders," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2005.
[6] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 599–618, 2001.
[7] D.E. Muller and W.S. Bertky, "A theory of asynchronous circuits," in *Proc. International Symposium on the Theory of Switching, Part 1*, 1959, pp. 204–243.
[8] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.
[9] Chris J Myers, *Asynchronous Circuit Design*, Wiley-IEEE, 2003.
[10] Wenjing Rao, A. Orailoglu, and R. Karri, "Logic mapping in crossbar-based nanoarchitectures," *Design Test of Computers, IEEE*, vol. 26, no. 1, pp. 68 –77, jan.-feb. 2009.