SELF-HEALING CELLULAR AUTOMATA TO CORRECT SOFT ERRORS IN DEFECTIVE EMBEDDED PROGRAM MEMORIES

by

Varun Voddi

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Jacob Gunther Committee Member
Prof. Paul Israelsen Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

Self-Healing Cellular Automata to Correct Soft Errors in Defective Embedded Program Memories

by

Varun Voddi, Master of Science

Utah State University, 2009

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

Static Random Acess Memory (SRAM) cells in ultra-low power Integrated Circuits (ICs) based on nanoscale Complementary Metal Oxide Semiconductor (CMOS) devices are likely to be the most vulnerable to large-scale soft errors. Conventional error correction circuits may not be able to handle the distributed nature of such errors and are susceptible to soft errors themselves. In this thesis, a distributed error correction circuit called Self-Healing Cellular Automata (SHCA) that can repair itself is presented. A possible way to deploy a SHCA in a system of SRAM-based embedded program memories (ePM) for one type of chip multi-processors is also discussed. The SHCA is compared with conventional error correction approaches and its strengths and limitations are analyzed.

(61 pages)

To my parents.

Acknowledgments

I would like to thank my advisor, Dr. Aravind Dasu, for his guidance throughout my research. I would also like to thank Dr. David Peak for his guidance.

I thank all my friends for their moral support throughout my graduate studies at Utah State University. In particular, I would like to acknowledge Jeff Carver, Saranya Chandrashekaran, and B. J. Mayers for contributing towards my research. Last, but not the least, I would like to thank my parents for their love and support.

Varun Voddi

Contents

List of Figures

Figure Page

Acronyms

ASIC	Application Specific Integrated Circuit
BCH	Bose Chaudhuri Hocquenghen
BISR	Built in Self Repair
BIST	Built in Self Test
BRAM	Block RAM
CA	Cellular Automata
CIB	Contour Information Bank
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip MultiProcessor
DEC-TED	Double Error Correction Triple Error Detection
ePM	Embedded Program Memories
ECC	Error Correction Code
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
LUT	Look Up Table
LDePM	Large and Distributed set of Embedded Program Memories
m-ECC	Multi Bit Error Correction Code
MFPT	Mean First Passage Time
MRF	Markov Random Field
MTTF	Mean Time to Failure
NMR	N Modular Redundancy
OECT	Other Error Correction Techniques
RAM	Random Access Memory
RS	Reed Solomon
RT	Rule Table
SEC	Single Error Correction
SEC-DED	Single Error Correction Double Error Detection

SHCA	Self-Healing Cellular Automata
SRAM	Static RAM
TMR	Triple Modular Redundancy
VLSI	Very Large Scale Integration

Chapter 1

Introduction

Low-to-medium performance circuits, such as those used in mobile systems, have benefited from an aggressive scaling of supply-voltage, because it has a quadratic relationship with power. Some authors suggest that this will continue well into the sub-threshold region [1–3]. From an energy savings point of view, this is probably the most effective strategy. Hanson *et al.* [3] show that 65 nm-based SRAM cells (if correctly manufactured) can work quite well in the sub-threshold region, for supply voltages as low as 70 mV and at room temperature. However, as the feature size decreases, maintaining precise doping levels will become increasingly unlikely [4]. The resulting parametric variations can cause changes in threshold voltages, which can adversely affect the stability of SRAM flip-flops operating in the sub-threshold regime. Simulations of the effect of thermal noise on such devices (shown in Li *et al.* [5]) indicate that under a supply-voltage of 0.2 V, a 0.025V variation in threshold voltage results in the meantime for a bit flip error that is low enough (10^{-8} seconds) to be a concern. In order to mitigate the effects of "massive amount of widely distributed random defects" [6], this work proposes an error correction technique that relies on distributed information processing to correct errors, which is also capable of self-healing.

The rest of the thesis is organized as follows. In Chapter 2, papers on ECC-based error correction for memory systems are reviewed. In Chapter 3, cyclic codes are analyzed for performance in the presence of single D-FF corruption in the encoder and decoder circuits. Chapter 4 discusses the implementation of the proposed SHCA technique along with the potential to explore instruction redundancies between programs, and how that can be used by a cellular automata system for error correction. In Chapter 5, we will present an analysis of the strengths and limitations of the SHCA and compare it qualitatively (and quantitatively where appropriate) with Hamming, Reed-Solomon, and Hierarchical TMR approaches. The contributions of the thesis is summarized in the conclusion chapter.

Chapter 2

Background and Related Work

2.1 Techniques for Enhancing SRAM Reliability

Error correction in binary logic-based circuits and systems, is a very well studied field. In particular, the memory circuits and aerospace communities have a long history of publications in this area. The reliability of memory products has been an issue of great concern due to transistor downscaling and increase in density. Chin-Lung Su *et al.* [7] suggest a method to enhance the reliability of SRAM, by using (a) single error correction (SEC) hamming code, (b) a built in self test (BIST), (c) a reconfigurable circuit (RC), and (d) redundant rows and columns. Through these features the system is capable of correcting both soft and hard errors. Results show that on an average, the proposed system results in a meantime to failure (MTTF) of 30K hours with a 1.3% increase in footprint. Hung *et al.* [8] present a technique for multi-bit error correction in caches by using (a) BIST, (b) Built in Self Repair (BISR), (c) Redundant rows, and (d) single error correction-double error detection (SEC-DED) decoder. Each cache line is composed of several SEC-DED blocks, which are classified as g (good), t (single defect), and b (multi-defect). The b-blocks are replaced by redundant rows and assurance update is done on the t- blocks. To further enhance the overall reliability data swapping is done, where clean data is written into t-blocks and dirty data into g-blocks. For a processor with 16KB L1 caches (both instruction and data) and 512KB unified L2 cache, the performance degradation is very small at a defect rate of 0.005.

More recently however, there has been an increased emphasis on multi-bit error detection and correction techniques. Sun *et al.* [9] present a technique for multi-bit error correction. To protect a united L2 cache (integrated instruction and data caches), two additional fully associative caches are used, one to store multi-bit ECC (m-ECC) check bits and the other to store the most recently decoded multi-bit ECC code word. The latter is called a pre-decoding buffer (pDB). The L2 cache is partitioned into sub-blocks g, s, and m (good, single-defect, and multi-defect). A hamming single error correct, double error detect (SEC-DED) code is used on the s sub-blocks. A Bose Chaudhuri Hocquenghen (BCH) based double error correction-triple error detection (DEC-TED) code is used on the m-sub blocks. Results show that for a 1MB L2 cache, 16KB of M-ECC cache and 4KB pDB have been used. The system allows for a defect density up to 0.5%. Naseer and Draper [10] present a parallel BCH-DEC code-based approach with a single cycle implementation of DEC decoder (based on XOR trees) rather than the conventional multi-cycle iterative decoding. Experimental results indicate that the for an average increase of 1.5x in latency, 2x in encoder area, and 9x in decoder area, an increase in reliability by 4x magnitude is achieved as compared to SEC-DED Hsiao Code [11]. Kim *et al.* [12] present a 2-D error coding technique capable of tolerating multi-bit errors in caches. The key feature of their technique is the combination of vertical error correction with the conventional per-word horizontal error coding. They have used error detecting codes (EDC) along with interleaving of bits to achieve multi-bit error correction capability. 2-D error coding was used for L1 (64KB) data caches and L2 (16MB/4MB) shared caches in fat and lean baseline chip multiprocessor (CMP) systems. EDC8 along with 4-way bit interleaving was used for 64-bit words in L1 caches and EDC16 with 2-way bit interleaving was used for 256-bit words in L2 cache. This combination can correct clustered errors of up to 32 × 32 bits. The average performance loss (in instructions per cycle) is 2.9% and 1.8% for the fat CMP and lean CMP, respectively, with an overhead of only 5% and 6% for L1 and L2 caches compared to SEC-DED protection. In addition to these publications, the reader can refer to Dutta and Touba [13], Suzuki *et al.* [14], Zhang [15], Slayman [16], and Jeffery *et al.* [17] for a wider range of techniques to correct soft errors in caches.

Besides reviewing traditional error correction techniques targeted towards relatively sparse (radiation induced) soft errors, it is worthwhile to note some recent papers that look into large scale errors due to manufacturing defects in nano-scale circuits. Nepal *et al.* [18] demonstrate the error tolerance for nano-scale circuits with low-operating voltages based on propagation of state probabilities over a Markov random field (MRF) network. CMOS circuits at very low supply voltages (0.1V to 0.5V) are claimed to fail due to crosstalk, thermal noise, and threshold variations. Though the MRF framework of CMOS circuit design provides error tolerance at extreme noise condition, the area overhead incurred is overwhelming. MRF implementations of an inverter and a NAND gate have an area overhead of 1000% and 1500%, respectively, compared to the conventional CMOS implementation. The delays of the MRF inverter and NAND gate operated at 0.15V, normalized to the delay of a CMOS inverter are 7.1 and 8.6, respectively, which is due to the increased level of circuitry and existence of feedback paths.

In another publication, Nepal *et al.* [19] discuss the likelihood of error correcting logic getting corrupted at extremely noisy operating conditions. This work addresses perhaps the most important problem: can an error correction technique correct errors in a target circuit/system if its contents/circuit itself malfunction? After all, all transistors are equally susceptible to process variations and defects. They propose a modular ECC technique based on probabilistic MRF to reduce the occurrence of errors in correcting logic.

It can be observed that most of the papers in the area of ECC protected caches have focused on correcting low levels of soft error rates, through the use of relatively high rate codes. But as technology scaling continues, it is worth considering the question posed by Spica and Mak [6], "Will simple ECC techniques be effective enough to heal large scale errors?" An inherent challenge with using high rate codes is that the complexity of the error correction circuits increases considerably, leaving them susceptible to defects. Therefore while specifically considering caches, it is worth looking into inherent redundancies that may already be present, allowing one to take advantage of low-rate codes that straddle the theoretical Blokh-Zyablov bound [20].

2.2 Plant Stomata

The proposed method is inspired by the way plant stomata deal with change in surrounding atmospheric conditions. Stomata are microscopic, variable-aperture pores that permit the exchange of gases between the plant and the atmosphere (shown in fig. 2.1). In most plants the stomata are 30 to 60 micrometers long and are distributed at a density of 50 to 200 per square millimeter [21]. This change in the aperture is done as a response to change in local environmental conditions like light intensity, humidity, and CO_2, to achieve, it is conjectured, maximum CO_2 intake (to optimize photosynthesis) while mitigating water evaporation loss (to avoid desiccation) [22]. This problem solving capability of the plant leaves is mathematically formalized as constraint optimization problem [22]. Each stomata pore changes its aperture depending on external atmospheric conditions (light, humidity, CO_2) quickly and autonomously. The changing of the aperture happens on a time scale of minutes. This often results in a broad distribution of apertures, most of which are close to the "optimum" aperture, but with outliers that are too open or too close. Most of the times the stomata are uniformly open or close, but in some cases patches appear (with a certain region open and the neighboring region closed) which in rare instances persists for hours. This leads to stomatal patchiness. This phenomenon contradicts the traditional assumption that stomata are autonomous units that respond to change in environmental conditions independently. This phenomenon can be visualized in images of chlorophyll fluorescence (shown in fig. 2.2).

But on a more leisurely time scale, the aperture distribution is observed to narrow down (patchiness disappearing), with the outliers becoming more like the initial majority value. Narrowing down of the apertures of the outliers, in the absence of neural tissue capable of coordinating widely separated parts, must emerge from the interactions of neighboring stomata only [23]. This is an example of "real-life," self-organized, collective error correction. This mechanism can be applied to error correction in Embedded Program Memories, by formulating the stomatal aperture as a Cellular Automaton (CA). By formulating a stomata as a Cellular Automaton, the apertures are quantized into "more open"(1) or a "more closed" (0), with the continuous stomatal adjustments approximated as occurring in discrete time steps. Stomata network can be interpreted as Rule Tables (RTs) connected locally in toroidal fashion. To mimic self-healing by stomata, the distributed iterative dynamics encoded by the rule table must identify the (presumably erroneous) minority state

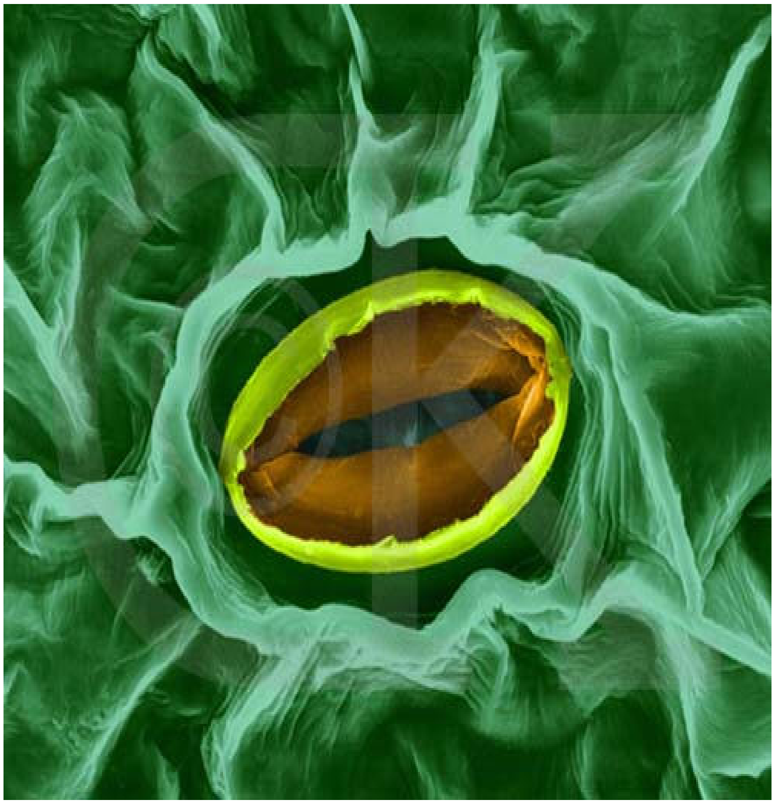


Fig. 2.1: Plant stomata.

present immediately after the network is initialized, then change all minority states to the (correct) majority state.

2.3 Cellular Neural Networks

Cellular Neural Networks (CNN) are parallel computing paradigm similar to neural networks [24, 25], with the difference being that communication is allowed between neighboring units only. The "standard" CNN is an analog computational system defined by the dynamical equations

$$y_{ij} = f(x_{ij}), \tag{2.1}$$

$$\tfrac{dx_{ij}}{d\tau} = -x_{ij} + \sum_{kl} A_{kl} y_{kl} + I_{ij}, \tag{2.2}$$

where each node (ij) has an internal state x and an output state y; x can be a vector of

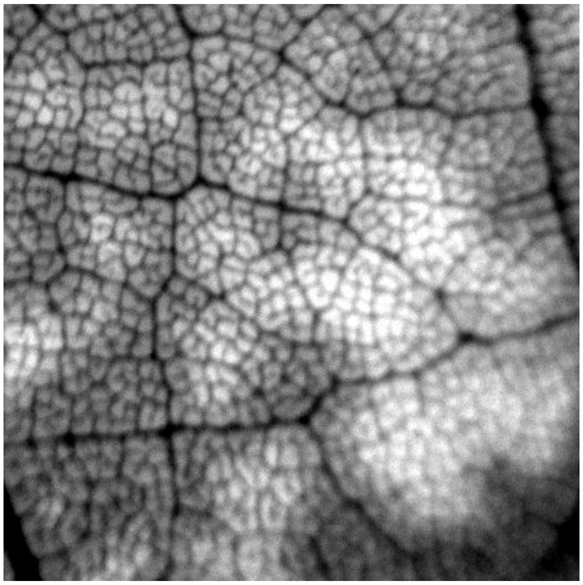


Fig. 2.2: Chlorophyll fluorescence image showing stomatal patchiness, with dark areas representing open stomata and light areas representing closed stomata.

any real values, while the components of y are restricted to [-1, 1]; f in (2.2) is a vector-valued function that returns -1 if the corresponding component of x is less than -1, +1 if the component is greater than +1, and the component of x otherwise; the sum in (2.3) involves the outputs of node (ij) and the nodes to which it is connected, weighted by the set of connection strengths A (the "A template"); I is a set of external inputs to node (ij); and t is dimensionless time. Provided the As and Is are chosen correctly (the CNN "program"), the network relaxes to attracting states, x^s, and the corresponding outputs, y^s, achieve a desired, target configuration (encoded in the As and Is). The target configuration is a self-organized collective condition, attained without the aid of a central controller.

2.4 Cellular Automata

Cellular Automaton (CA) is a grid of identical elements, each made up of a finite number of states, which are discrete in -state, -space and -time [26]. The grid can be 1, 2, 3, or n dimensional grid. Figure 2.3 shows a 2-D CA grid. The output of a cell at time instant t is a function of its neighboring cells and itself at time instant *t-1*. Each node iteratively executes a program whose output depends on the instantaneous state of that node, as well

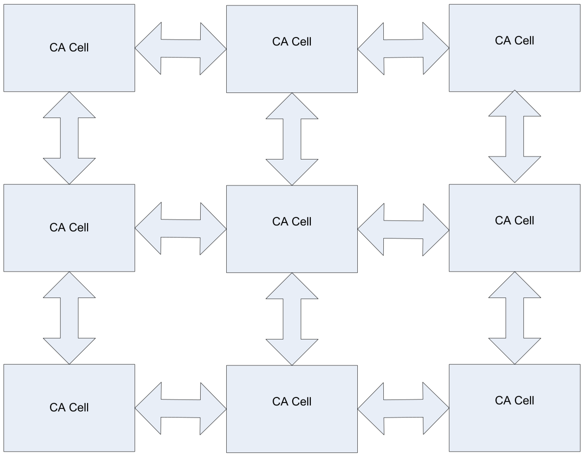


Fig. 2.3: 2-D CA grid.

as the states of the nodes to which it is connected. The states in a CA are updated in discrete time via a look-up table, where as in a Cellular Neural Network (CNN) the states are updated via a continuous integral. In either case, the iterative updating eventually produces a network state configuration that can often be interpreted as "performing a task." Because task performance by CNNs or CAs is distributed over all of the network nodes, it can be remarkably insensitive to a wide range of temporal ("soft") and spatial ("hard") errors.

The CA cells in the grid are connected to the neighboring cells, but the connections of the cells on the edges depend on the problem that is being addressed using the CA network. In my implementation of the CA network, I have used a toroidal arrangement (shown in fig. 2.4), meaning the cells on the top edge would connect to the ones at bottom and the ones at the right would connect to the ones at left and vice versa. This would give the cells at the edges the same number of neighbors as the ones in the middle of the network. This would lead to a uniform rule implementation in all the CAs.

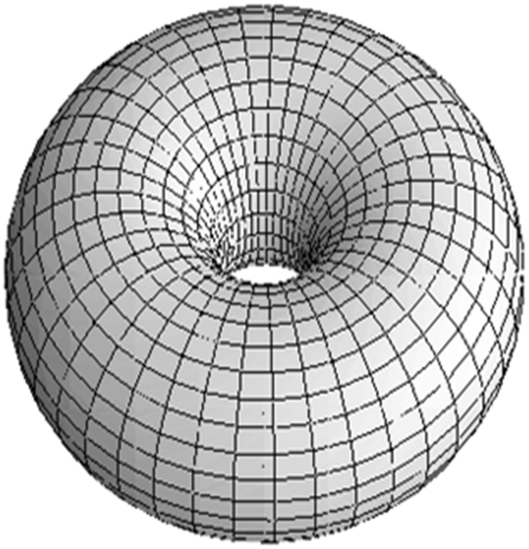


Fig. 2.4: A toroidal structure.

Chapter 3

Analysis of Effects of Flip-Flop Corruption in Cyclic Encoder Decoder

3.1 Error Correction Codes

Error Correction Codes (ECC) are of great importance in maintaining data integrity in storage devices and across noisy channels. The main function of ECC is to detect and correct errors, which is achieved by introducing additional bits called check bits, to the original data bits. The number of check bits that need to be introduced depend on the number of data bits and the error correction capability of the code. ECC is used in a wide variety of fields, few of them being TCP/IP stack, Deep-Space communication, Satellite Broadcasting, and Data Storage.

3.2 Cyclic Codes

Cyclic codes are a sub-class of a much larger class called the linear block codes. A linear block code is said to be cyclic code if it remains unchanged by cyclic shift. The difference between a shift and a cyclic shift has been shown with an example below. Let the code word vector be denoted by

$$C = (c_0, c_1, c_2, c_3, c_4, \ldots c_{n-1}), \tag{3.1}$$

which has a polynomial representation as follows

$$c(x) = c_0 + c_1x + \ldots + c_{n-1}x^{n-1}. \tag{3.2}$$

A non-cyclic shift gives a polynomial of the form

$$c(x) = c_0x + c_1x^2 + \ldots + c_{n-1}x^n. \tag{3.3}$$

So basically a zero is appended to the left most position in the vector in a non-cyclic shift, but with a cyclic shift the last element is shifted to the first element in the sequence. This can also be obtained by dividing $xc(x)$ by $x^n - 1$ using the usual polynomial division.

$$c(x) = c_{n-1} + c_0x + c_1x^2 + \ldots + c_{n-2}x^{n-1} \tag{3.4}$$

If *c(x)* is assumed to be an element of the GF(q)[x]/(x^{n-1}), $xc(x)$ is a cyclic shift since operations in the field are defined modulo $x^n - 1$. This would mean that the multiples of the code word are also codeword's. For a *(n, k)* code with k message bits there exists a *g(x)* polynomial called the generator polynomial, which has a degree of *n-k* . The value of *n-k* denotes the redundancy of the code.

$$g(x) = g_0 + g_1x + g_2x^2 + \ + g_{n-k}x^{n-k} \tag{3.5}$$

There exists a message polynomial which contains information about the message bits, with a degree of *k-1*.

$$M = [m_0, m_1, m_2 \ldots m_{k-1}] \tag{3.6}$$

$$m(x) = m_0 + m_1x + m_2x^2 + \ldots + m_{k-1}x^{k-1} \tag{3.7}$$

There also exists a parity check polynomial for a code of length n, which is denoted by a *h(x)* which satisfies the condition *h(x)g(x)* $= x^n - 1$.

3.2.1 Hardware Implementation for Basic Arithmetic Operations

Following are hardware implementations for some basic arithmetic operations:

If *b(x)* = *a(x)* *g(x)*.

Figure 3.1 shows the hardware implementation of a multiplier. The input *a(x)* is fed with the last element (highest order coefficient) going first into the circuit and the output *b(x)* has its last element (highest order coefficient) coming out first.

If *q(x)* = *a(x)*/*g(x)*, similarly for a division operation the last element (highest order coefficient) of *a(x)* is fed into the circuit first and *q(x)* is obtained at the output with the last element (highest order coefficient) coming out first, as shown in fig. 3.2.

3.2.2 Hardware Implementation of an Encoder

The following are steps for encoding the message bits:

- Depending on the length of the code (n) and the number of errors it can correct the generator polynomial *g(x)* is formed.
- Once the *g(x)* polynomial is formed, the generator polynomial is divided by the message polynomial (*m(x)*).
- This gives the parity check polynomial *h(x)*.

One way of implementing the encoder is by using the parity check polynomial coefficients (*h(x)*) instead of the generator polynomial coefficients (*g(x)*) [27]. The hardware implementation for this is shown in fig 3.3. For the initial k cycles gate 1 is closed (connected) and all the message bits are fed into the circuit. So at the end of k cycles all the message bits are in the circuit. Once all the message bits are fed, then for n-k cycles, gate 1 is opened (no connection) and gate 2 is closed (connected) allowing the parity bits to be shifted out of the circuit.

Figure 3.4 shows an alternative implementation of an encoder. With the gate closed (connected) the shifted message bits are fed into the circuit with the switch being in position A. In the second step, once all the bits are fed into the circuit the gate is opened (no connection) and the switch is moved to position B. Then the circuit is clocked for another n-k cycles to shift the parity bits out.

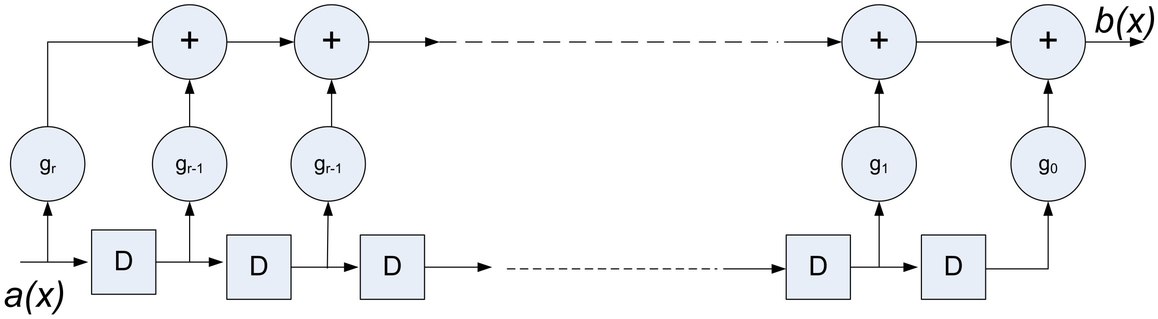


Fig. 3.1: Hardware implementation of a multiplier.

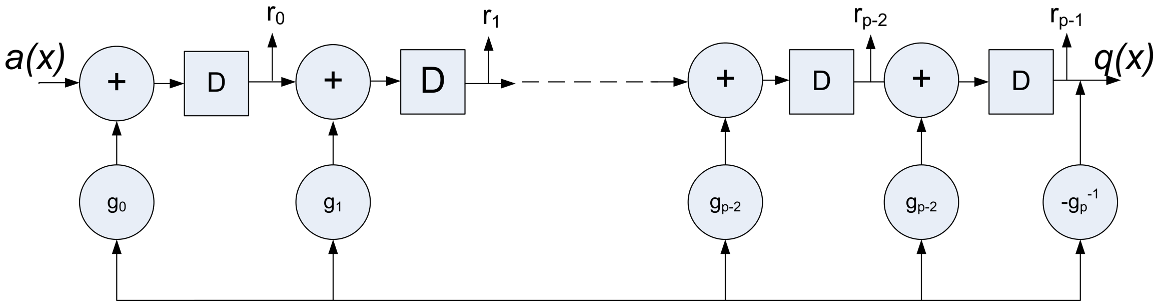


Fig. 3.2: Hardware implementation of a divider.

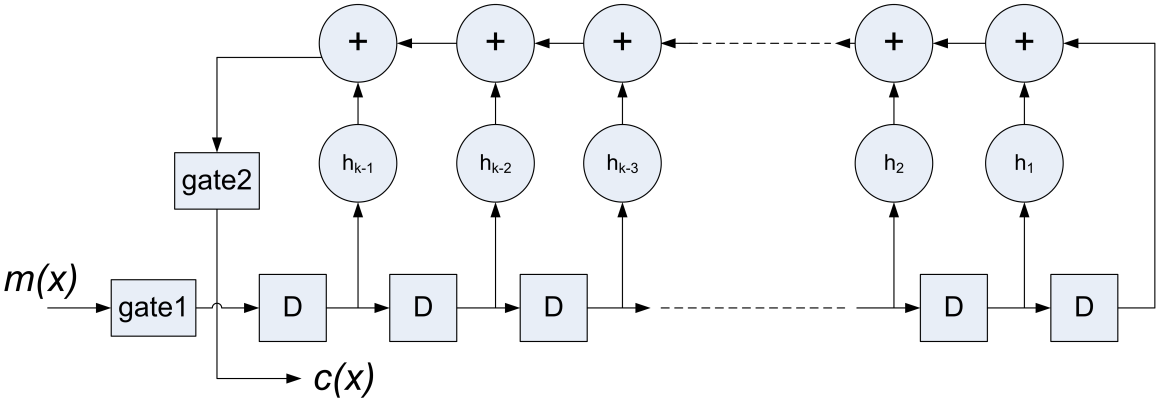


Fig. 3.3: Hardware implementation of an encoder.

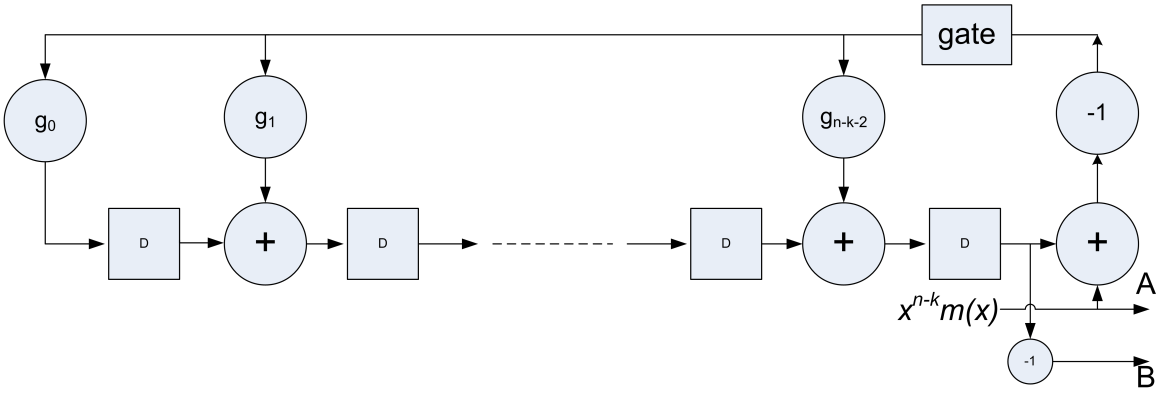


Fig. 3.4: Alternative hardware implementation of an encoder.

3.2.3 Hardware Implementation of a Decoder

The received codeword which is represented by *r(x)* is fed into the decoder circuit shown in fig. 3.5. For the initial n cycles gate 1 is closed (connected) and gate 2 is opened (not connected). With this setup, the codeword bits are fed into the Syndrome register and the Buffer register. The number of flip-flops in the Syndrome register are n-k, and that in the Buffer register are n. After n cycles, when all the codeword bits are fed into the Syndrome and Buffer registers gate 1 is opened (not connected) and gate 2 is closed (connected), allowing the looping back of the buffer register. At the same time the output from the Error Pattern Detection Circuit is added to loopback contents of the Buffer register. Once the Error Pattern Detection Circuit detects an error, it generates a 1, which is then added to the contents of the Buffer register to toggle the corrupted output. This is validated through the example provided below:

Taking the example of a *(n, k)* = (7,4) single-bit error correction hamming code

$$\text{g(x)} = x^3 + x + 1 \text{ and } m = [0, 1, 1, 1] \ , \tag{3.8}$$

$$\text{h(x)} = x^7 - 1 \ / \ x^3 + x + 1 \ . \tag{3.9}$$

The circuit for this encoder is shown in fig. 3.6. Initially the contents of the register are set to 0. For k cycles (k being 4 in this example), contents are shifted through the register. After k cycles, once all the message bits are fed into the encoder circuit, gate 1 is opened

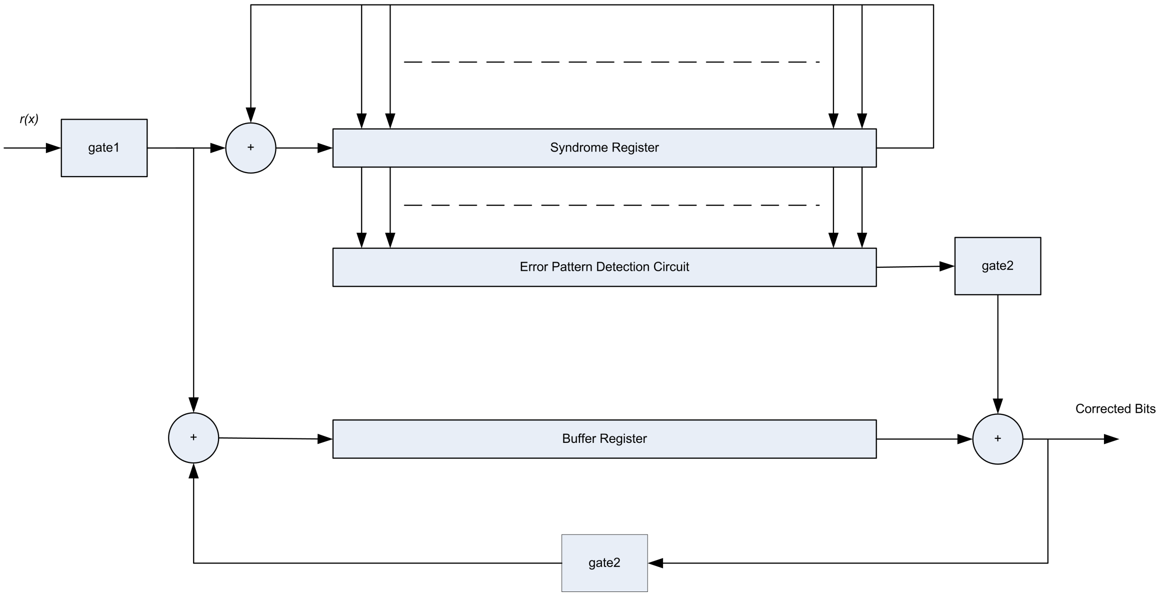


Fig. 3.5: Hardware implementation of a decoder.

(no connection) and gate 2 is closed (connected) as shown in fig. 3.7. For another n-k cycles, the parity bits are shifted out of the encoder circuit. So after a total of n cycles (n being 7), the codeword containing the message and parity bits is formed. The state of the encoder after n cycles is shown in fig. 3.8. The bits in bold indicate the message bits and the bits in italics indicate the parity bits. Now assuming that there is a one-bit corruption of the code word, the code word gets converted to 0110111 with corruption in the second bit form the left, which was supposed to be a 0 (shown in red in fig. 3.9). This corrupted codeword is supplied to the decoder for error correction.

After n cycles the corrupted codeword is loaded into the decoder buffer register. The circuit for the decoder is shown in fig. 3.9. Now gate 1 is closed and gate 2 is opened and the contents of the Buffer register keep shifting to the right. The state of the decoder after n cycles, when the codeword gets loaded into the Buffer register is shown in fig. 3.10. Once the corrupted bit reaches the right most bit position, the Syndrome register generates a syndrome value, which when given as input to the Error Pattern Detection Circuit detects the presence of an error and generates a 1, which is added to the corrupted bit to toggle it back to the uncorrupted value as shown in fig. 3.11.

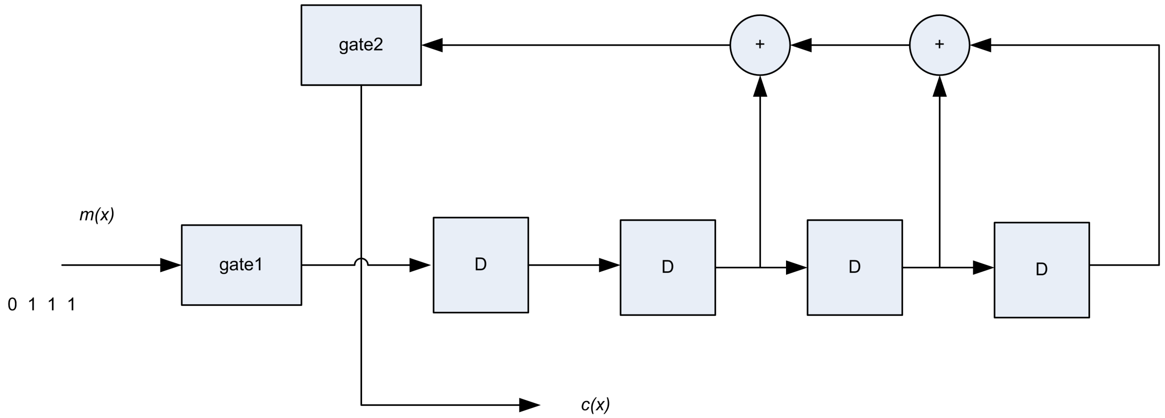


Fig. 3.6: Encoder implementation.

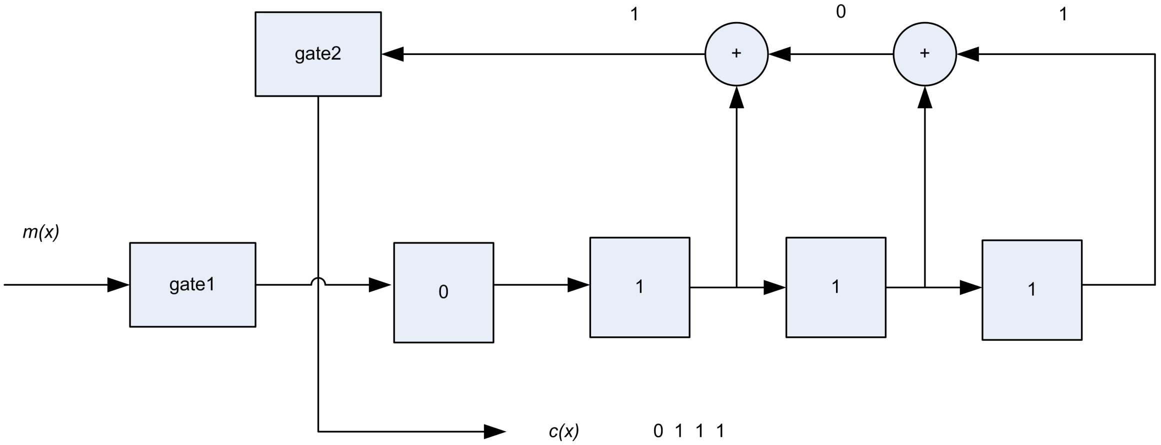


Fig. 3.7: Encoder after k cycles.

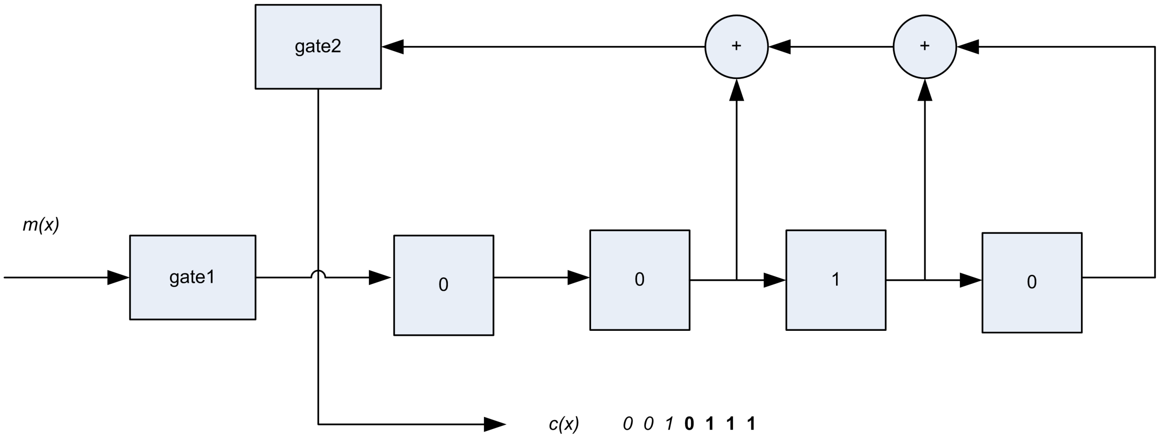


Fig. 3.8: Encoder after n cycles.

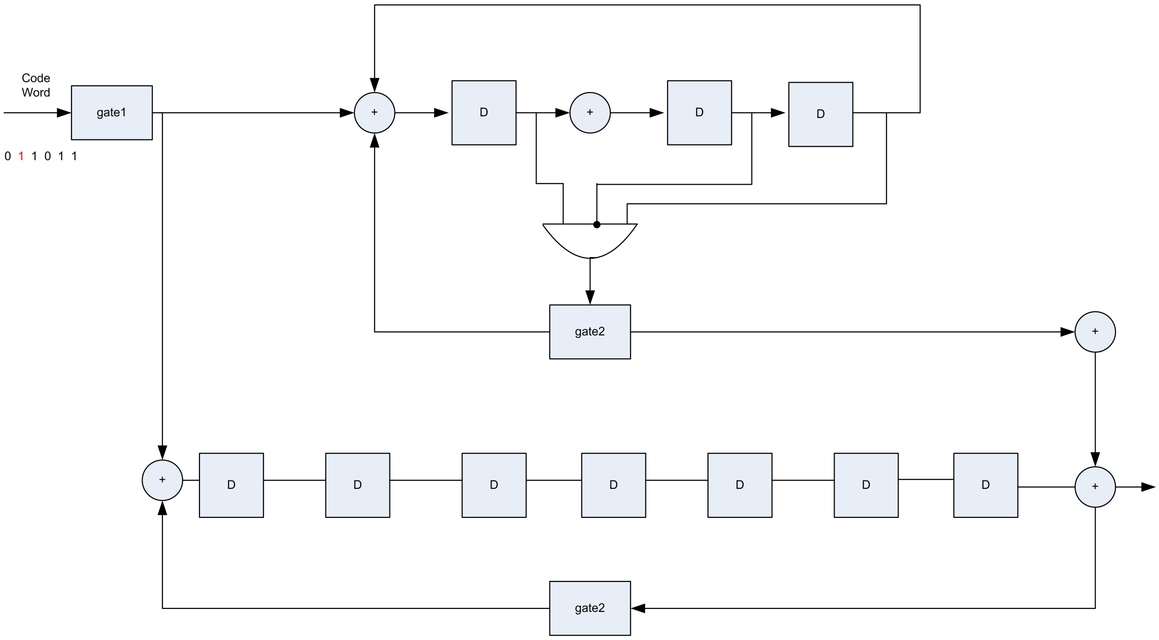


Fig. 3.9: Decoder implementation.

So after 2n cycles, when the shifting of all the bits in the codeword is done, the corrected codeword 0010111 is given as output.

3.3 Analysis on the Effect of Flip-Flop Corruption in Cyclic Encoder Decoder

Most of the error correction codes are not designed to handle corruption of the flip-flop in encoder decoder circuits. In this section, I have analyzed the effect of flip-flop corruption on working of a (31, 26) single-bit error correction hamming encoder decoder circuit. In this experiment only one D flip-flop, either from the encoder or the decoder, is corrupted at any given time. Figures 3.12 and 3.13 show a (31,26) hamming code encoder and decoder circuit, respectively.

3.3.1 Single Flip-Flop Corruption

The experiment was done for all possible valid single flip-flop corruption in the encoder. The term "valid single flip-flop corruption" means corrupting the flip-flop only when there is a message bit in the flip-flop. For example in 1st cycle, the corruption of the 1st D flip-flop

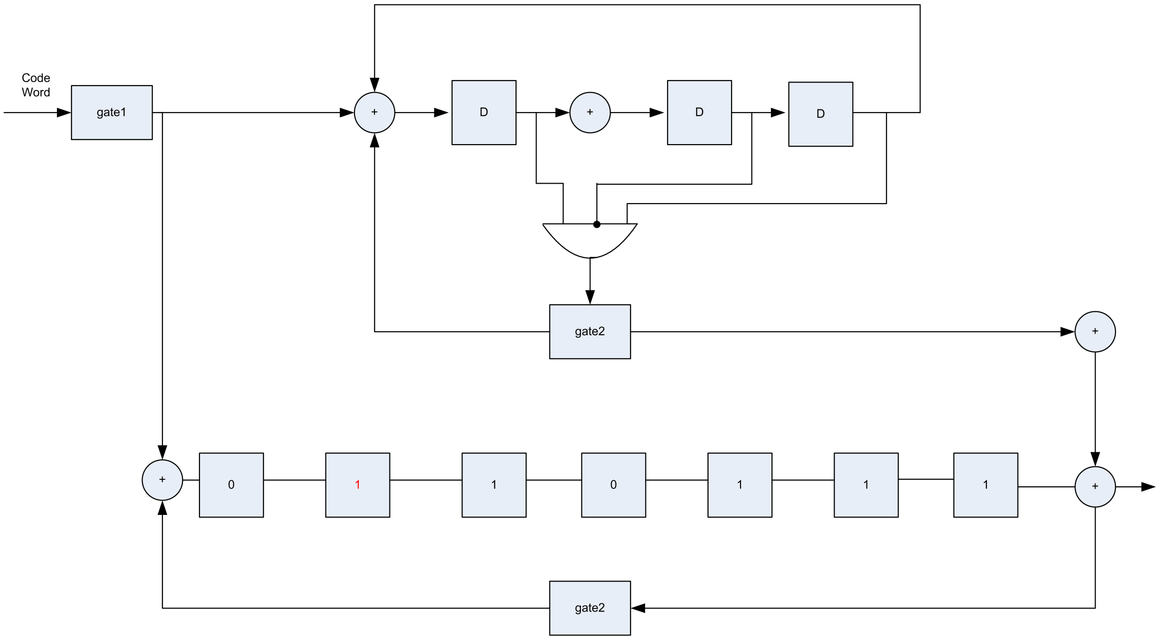


Fig. 3.10: Decoder after n cycles.

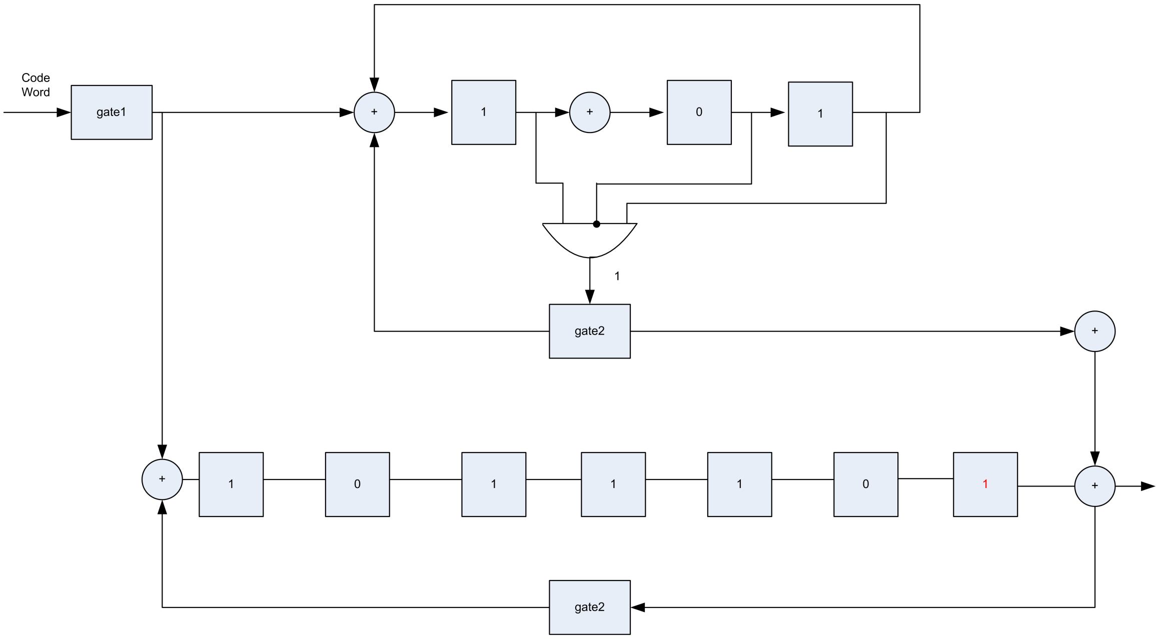


Fig. 3.11: Correction of the error bit.

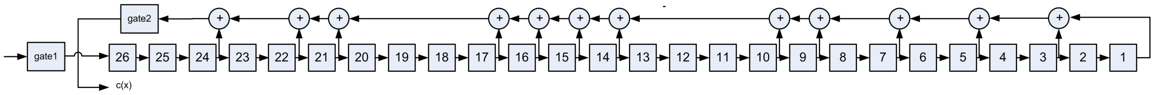


Fig. 3.12: (31,26) Hamming encoder.

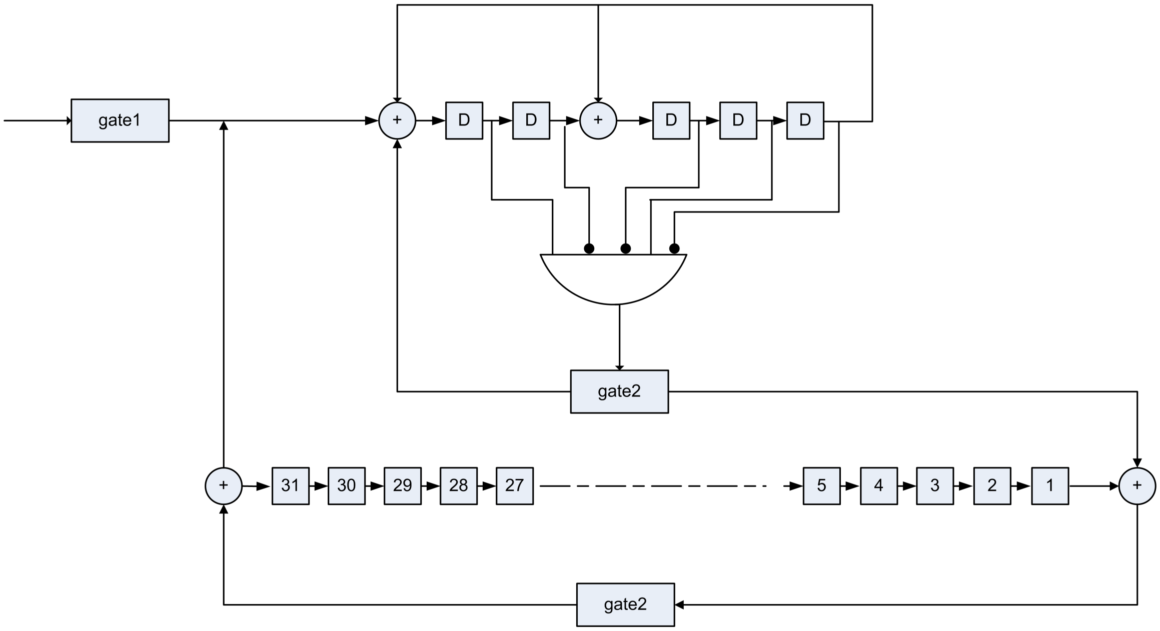


Fig. 3.13: (31,26) Hamming decoder.

in the encoder is not a valid corruption because it does not have message bit in it, but corrupting the 26th flip-flop is a "valid single flip-flop corruption" because it has message bit in it. The way the message recovery was determined was based on the comparison of the input message bits given to the encoder with that of the decoded message bits. If the message bits after decoding are same as the message bits before encoding, it is assumed as a recovery. 30001 different message words were used for every valid corruption. In this experimental setup it is assumed that there is no corruption over the channel. A total of 24,180,806 cases were run for corruption in the encoder. After the decoding process the message words were recovered 4,799,890 times, which is 19.85% of the total cases (shown in fig. 3.14).

Similar experiment was conducted for 1-bit corruption in the decoder. 50001 different message words were used for every valid single flip-flop corruption. In total, 15,500,310 different message words were used and the message word was recovered 8,950,179 times, which is around 58% of the total number of cases (shown in fig. 3.15).

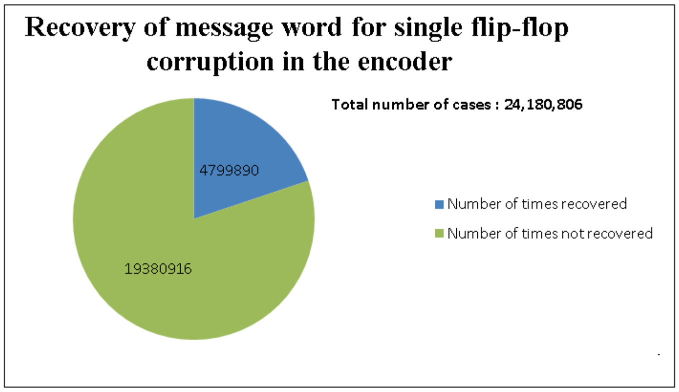


Fig. 3.14: Pie chart showing the number of times the message words were recovered for single flip-flop corruption in the encoder.

3.3.2 Multiple Flip-Flop Corruptions

In this experiment, two flip-flops were corrupted in the encoder or the decoder at any given time. For the encoder corruption a total of 674,963,600 cases were run, out of which the message word was recovered for 6,535,338 times, which is 10.06% of the total cases (shown in fig. 3.16). When a similar experiment was done on the decoder in which the total number of cases were 35,508,000, the message was recovered 12,956,869 times which is 36.49% of the total cases (shown in fig. 3.17).

Further analysis was done by increasing the flip-flop corruption to three. For this case of three flip-flop corruptions in the encoder a total of 523,606,616 cases were run, out of which the message word was recovered 38,206,768 times, which is only 7.8% of the total cases (shown in fig. 3.18). Similarly when three flip-flops were corrupted in the decoder for a total of 29,791,000 cases, the message word was recovered for 9,547,397 times, which is 32.04% of the total cases (shown in fig. 3.19).

These experiments show that for a (31, 26) hamming encoder decoder, the message word recovery percentage falls drastically as the number of flip-flops that are corrupted increase. This performance would further degrade if the corruption of the codeword over the channel is considered, as that would mean higher percentage of corruption. This experiment shows that there is a need for an error tolerant (in D flip-flop) error correction technique, which is addressed in this thesis.

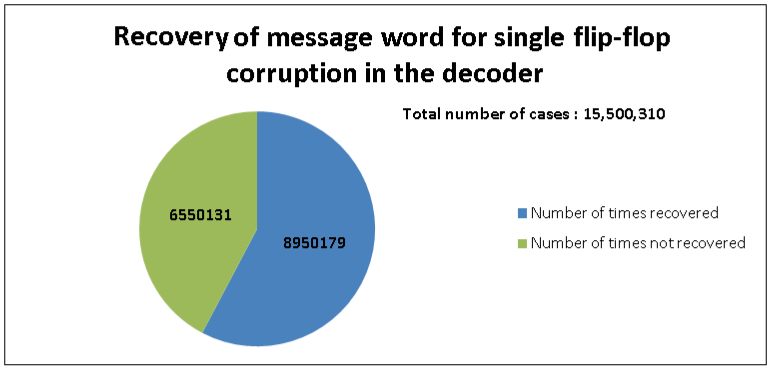


Fig. 3.15: Pie chart showing the number of times the message words were recovered for single flip-flop corruption in the decoder.

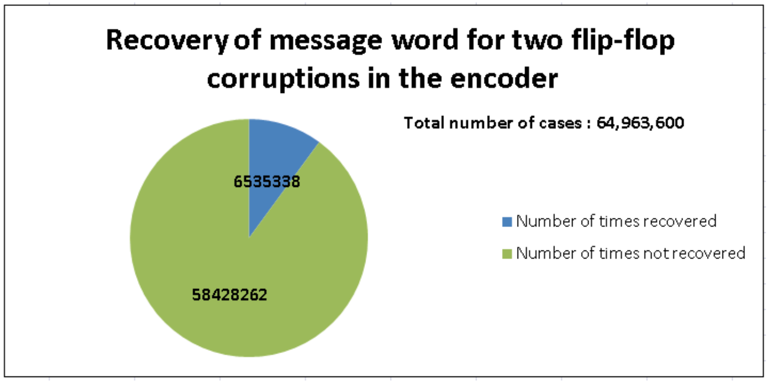


Fig. 3.16: Pie chart showing the number of times the message words were recovered for two flip-flop corruptions in the encoder.

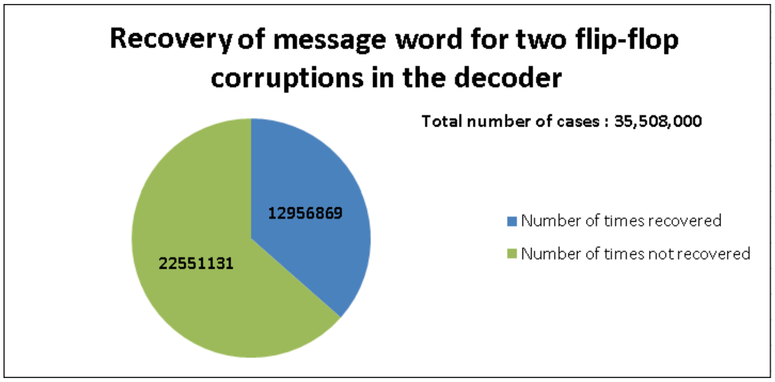


Fig. 3.17: Pie chart showing the number of times the message words were recovered for two flip-flop corruptions in the decoder.

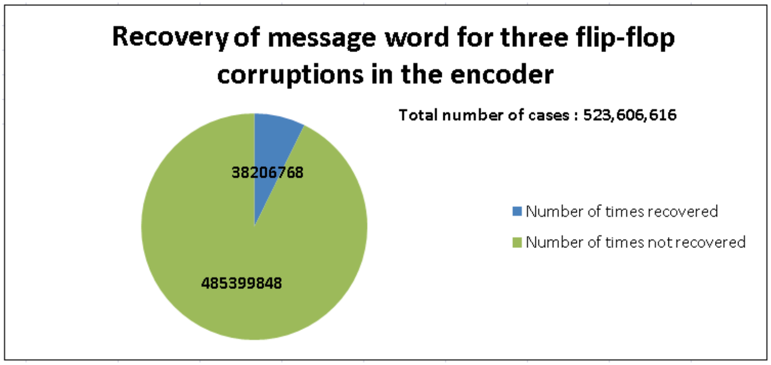


Fig. 3.18: Pie chart showing the number of times the message words were recovered for three flip-flop corruptions in the encoder.

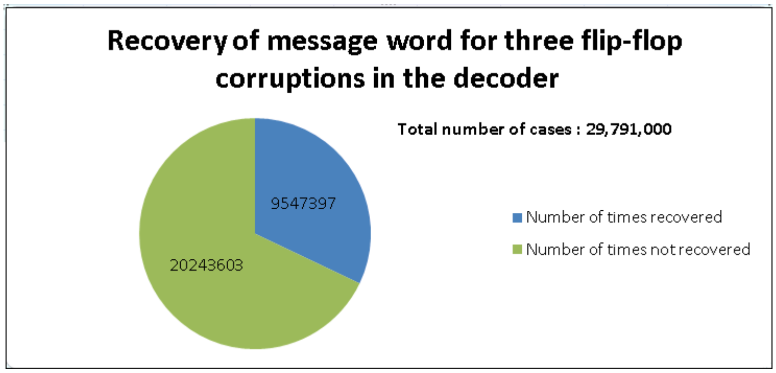


Fig. 3.19: Pie chart showing the number of times the message words were recovered for three flip-flop corruptions in the decoder.

Chapter 4

Proposed Technique

4.1 Proposed CA Implementation

The proposed CA network consists of grid of identical RTs connected to each of its four neighbors and itself. The RTs are connected using a mesh network with loop around at the edges forming a toroidal arrangement. The contents of the RTs in the SHCA network are identical. Figure 4.1 shows a scalable mesh-interconnect with wrap around for a 3 x 3 CA network.

4.1.1 CA Cell Implementation

A CA RT consists of a Look Up Table (LUT), a D flip flop (D-FF) and a 2-to-1 multiplexer. Each LUT has five inputs and one output. N, E, S, W, and C denote the inputs from the North, East, South, West neighbors, and Self, respectively, as shown in figs. 4.2 and 4.3. The logic embedded in the LUT is given by the Boolean expression

$$C_{t+1} = [C \times (N + E) + \bar{C} \times S \times W]_t. \tag{4.1}$$

In the actual Virtex4 LX60 FPGA implementation, the 5-input LUT is implemented as two 4-input LUTs and a 2-to-1 multiplexer as shown in fig. 4.4. One of the LUT is a 3-input LUT and the other one is a 4-input LUT. The inputs to the 3-input LUT are E, N, and C, and the inputs to the 4-input LUT are S, E, N, and C. The outputs of the two LUTs are multiplexed using a 2-to-1 multiplexer, whose select line is the W input. The output of the LUT is given as input to the D flip flop, whose output is then given as one of the two inputs of the multiplexer. The switching on and switching off of the CA is controlled by

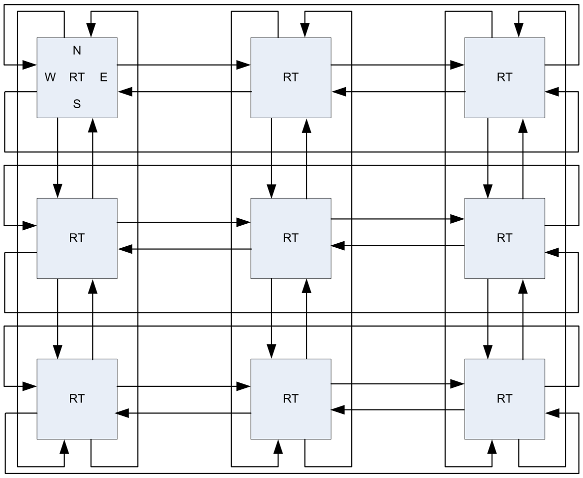


Fig. 4.1: A 3x3 CA network.

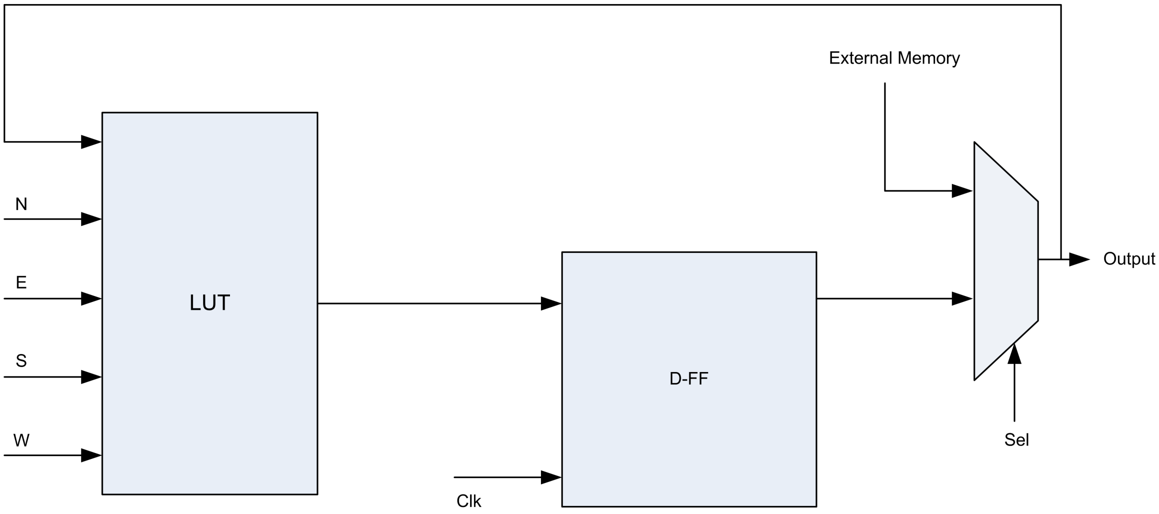


Fig. 4.2: CA logic block diagram.

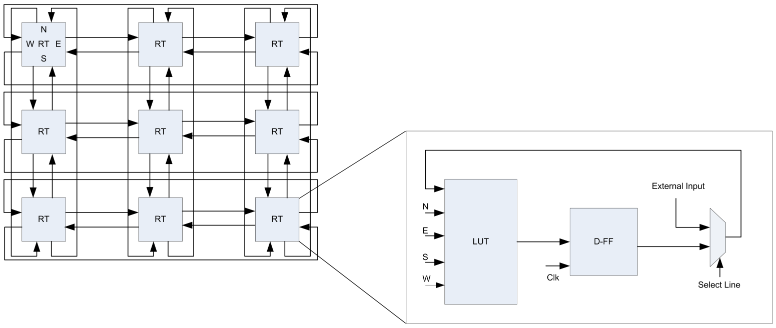


Fig. 4.3: CA network with the boolean logic in each RT.

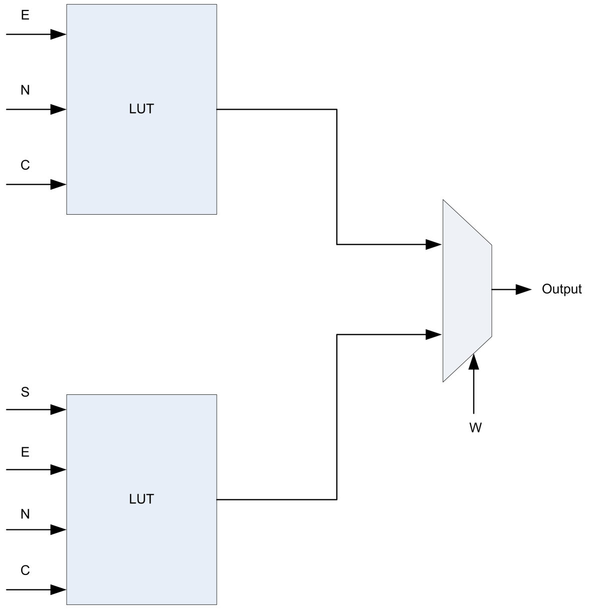


Fig. 4.4: Implementation of 5-input logic in two 4-input LUTs in Virtex4 LX60 FPGA.

the select line of the multiplexer. Initially, the bits that need to be corrected are given as inputs to the RTs from the external input port of the multiplexer. These external inputs (instructions) that need to be corrected are loaded into the CA network by setting the Select line (Sel) of the multiplexer high. Once the bits are loaded into the network, the select line (Sel) of the multiplexer is set low, and the correction process is started. The number of iterations required to converge all the minority state bits to the majority state depends on the percentage of minority state bits. After multiple iterations all the minority state bits converge to the majority state (correct state).

4.1.2 Self-Healing Cellular Automata

Traditional ECC-based circuits assume that the ECC encoder decoder contents itself do not get corrupted, hence they can focus on correcting errors that occur in defective memory cells. However, such assumptions cannot hold true in noisy nano-scale circuits, because these error-correcting circuits are also built from the same defective components and equally exposed to thermal noise. Therefore, there is a need to build self-healing error correcting circuits as shown in the previous chapter. The Self-Healing Cellular Automata

(SHCA) was implemented using Virtex4 LX60 FPGA. For this implementation the SliceM of CLBs were used. The LUTs in SliceM can be implemented as 16 x 1-bit synchronous RAM called distributed RAM. The distributed RAM can be implemented in two configurations single-port configuration and dual port configuration. For the SHCA implementation dual port configuration has been used. In this mode the write operations are synchronous and the read operation asynchronous.

The LUT implemented using SliceM of virtex4 LX60 is shown below in fig. 4.5. The port SPO outputs the content from the memory location pointed by Address [3:0] and the port DPO outputs the contents from the memory location pointed by DPRA [3:0]. Input ports WCLK and WE are the clock and the write enable inputs. DPO port is used for the healing of the external bits and the SPO port is used for the healing of the SHCA. This is because the DPO output port is driven by C, N, E, S, and W ports, where as the SPO is driven by the Address [3:0] port, from which the contents of the Rule Table (RAM) can be accessed for self-healing. So when the self-healing is on, the contents in the RAM are accessed through the SPO port by giving Address [3:0] as input.

The Self-Healing Cellular Automata node is shown in fig. 4.6. All the three outputs from the LUT are given to the flip-flops. The registered outputs are given to the given to a 4-to-1 multiplexer. The multiplexer has four inputs: (1) External Input, (2) Opcode Healing Port, (3) Self-Healing Port 1, and (4) Self-Healing Port 2. The external opcode bits to be healed are supplied to the SHCA through the External Port. Initially the external Opcode bits that are to be healed are fed into the SHCA network by setting the Select line [1:0] to a 00.

Once the external Opcode bits are fed into the SHCA network, the Select line [1:0] is set to 01. Upon setting the Select Line to 1, the External Port is disabled and Opcode Healing port is selected. The SHCA network heals the corrupted Opcode bits supplied to it over multiple iterations. The number of iterations for all the bits to converge to the majority state (correct bit value) depends on the percentage of minority state (error bit value).

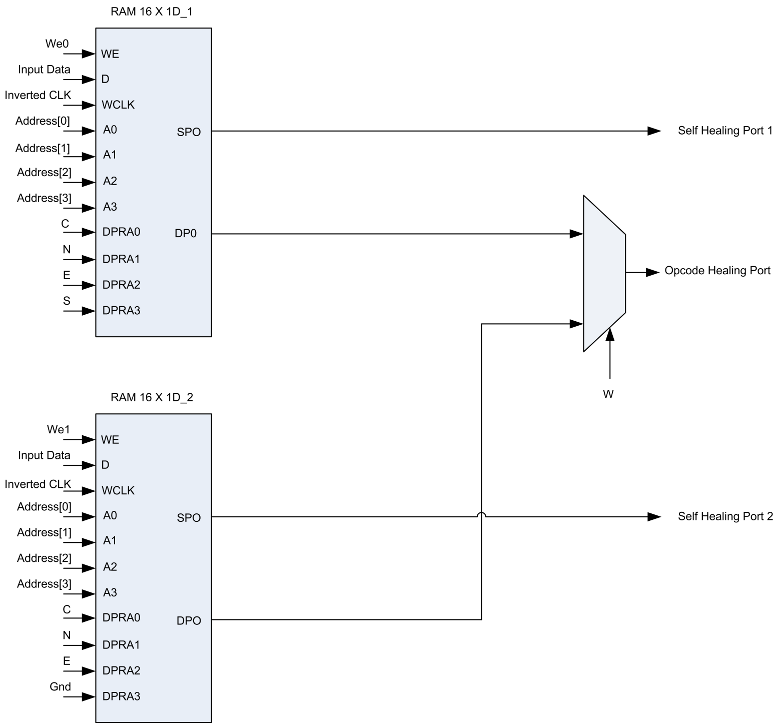


Fig. 4.5: RT implementation in SliceM of Virtex4 LX60 FPGA.

For repair process of the SHCA, Self-Healing Port 1 and Self-Healing Port 2 are used. The LUT in fig. 4.6 is implemented in two RAM16 × 1D (shown in fig. 4.5). The self-healing process of these two RAMs is done separately. This is possible because of the inherent redundancy present in SHCA network. Since the SHCA is made of local interconnections of RTs, which have the same contents, all the bits from the same address location from all the distributed RAMs can be fed into the SHCA for healing. The repair process is similar to the healing of the external Opcode bits, except that the Select Line of the multiplexer is either set to 10 or 11, depending on which of the two RAMs is being repaired. Supposing that the repair process is done for RAM 16 × 1D_1, the Select Line is set to 11, which selects the Self-Healing Port 1. The Self-Healing Port 1 has contents from address location pointed by Address [3:0] from RAM 16 × 1D_1. Over multiple iterations the repair process is done and the contents are written back to the RAM using Input Data port by setting We0 (write Enable port of RAM 16 × 1D_1) high. Similarly repair technique can be applied to the RAM 16 × 1D_2.

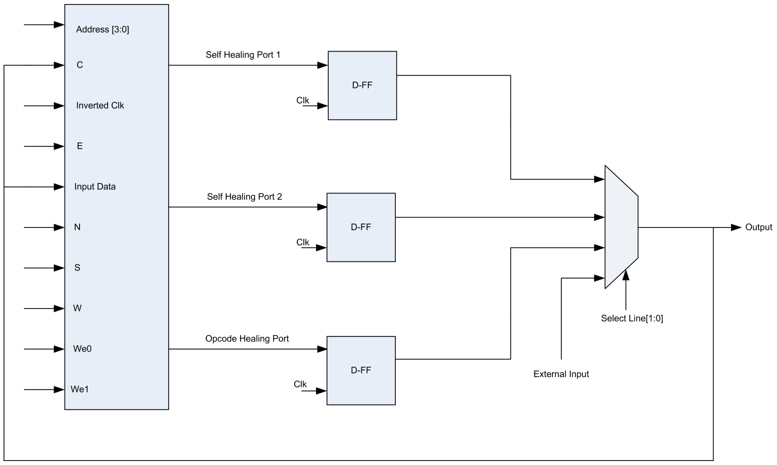


Fig. 4.6: SHCA cell implementation on a Virtex4 LX60 FPGA.

4.1.3 Alternative Implementation of the SHCA

The other possible implementation of SHCA is shown in fig. 4.7. This realization is composed of 32 bi-stable flip-flops sandwiched in between a multiplexer and a de-multiplexer (built using a cascade of pass transistors). The cell can be initialized (configured upon power on) into a 32 entry LUT by setting the Sel-a, Sel-b, and Sel-c lines high, choosing a specific combination of the N, S, E, W, and C' ports to target a specific flip-flop and asserting a valid entry (based on eq. 4.1) onto the C-bit line. Once all the flip-flops are configured, the Sel-a line is dropped low. This disables the pass transistor, hence blocking any writes to the flip-flops. In this configured state, a large number of LUTs can be connected using a mesh network with wrap around forming a SHCA. In order to allow the configured SHCA to work on a set of bits (to find the majority type), the Sel-c line is switched to create the feedback loop, and Sel-d is set to choose a registered bit from an external source (External Input). This immediately influences the output of each LUT in the network. Then the Sel-d line is switched off, from this point onwards a SHCA takes several iterations to converge and the decision is available at the output of each LUT, which can then be tapped out. However, when LUTs in the SHCA need to heal themselves, bits stored in LUT flip-flops are supplied into the network as bits to be healed. Since all the LUTs in the network contain the same

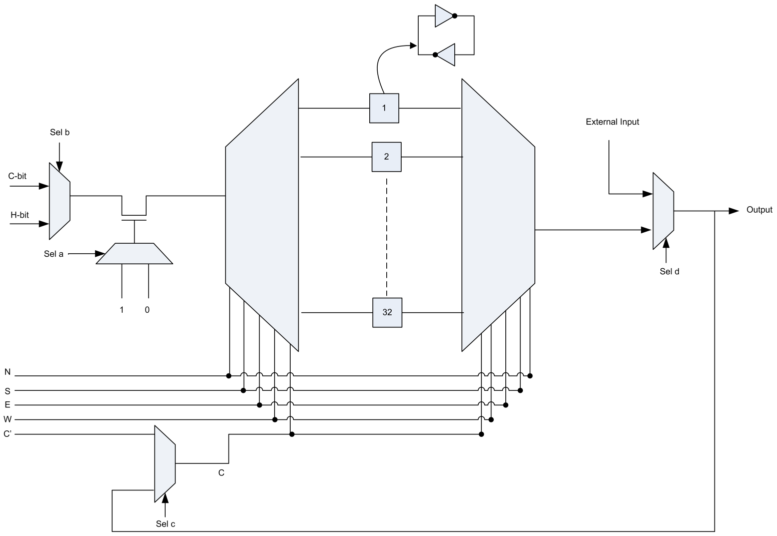


Fig. 4.7: Circuit of a reconfigurable LUT composed of simple multiplexers and bi-stable flip-flops.

32 bits, when one bit from equivalent flip-flops in all LUTs are supplied to the network, the resultant decision would be the majority correct value of that cell value. This can then be written back to all LUTs by appropriately setting the H_bit, Sel_a, and Sel_b lines. This process is followed to heal each of the 32 sets of flip-flops. During this self-healing process, however, a SHCA is incapable of correcting an external set of Opcode bits supplied from the External Port.

4.2 Applying SHCA to Homogenous ePMs

The hypothesis of this thesis is that chips made from nanoscale devices will have high levels of defects, which can result in distributed and unpredictable soft errors scattered all over a chip and perhaps even large clusters of bit flips due to thermal hotspots. If these errors affect Embedded Program memories (ePMs), executables are vulnerable to corruption. Here Program Memories refer to L1 cache of a processing unit that is composed of SRAM cells. In all error correction techniques redundancy is the key to mitigating the effects of errors. However in some cases, it is perhaps useful to examine if redundancy inherently exists in the system, as in the case of Homogenous Embedded Program Memories. To demonstrate and validate the error correction capability of SHCA for instructions in Homogenous Embedded

Program Memories (ePMs), a frame work which consists of simple $N \times N$ processor cores that are assumed to execute, in parallel, variants of a counter program is designed. Each core is an 8-bit RISC micro-controller based on the Xilinx PicoBlaze architecture [28, 29] that is connected to a unique program memory (PM) module (simulated by a Block RAM on the FPGA).

Picoblaze:

Picoblaze microcontroller is a compact, capable 8-bit RISC microcontroller. It takes 104 slices on a Virtex4 LX60 FPGA which is 0.39% of the total number of slices. The reason for choosing a Xilinx PicoBlaze micro controller is the compactness of the micro controller. Since this frame work needs $N \times N$ (N taking a value of 12) core implementation, the resource utilization for these cores needs to be as small as possible. PicoBlaze micro controller is a compact yet capable 8-bit micro controller. The other reason for choosing a PicoBlaze micro controller is that it is provided as a free source-level VHDL file which can be modified according to requirements. The salient features of PicoBlaze micro controller are as follows:

- 16 8-bit general purpose data registers,
- 1K instruction capability,
- 8-bit Arithmetic Logic Unit,
- 64-byte internal scratch pad RAM.

The block diagram of a PicoBlaze micro controller is shown in fig. 4.8.

IN_PORT [7:0] is the port through which valid input data can be supplied to the PicoBlaze micro controller through INPUT instruction. INSTRUCTION [17:0] is the port through which instructions are supplied to the PicoBlaze core. OUT_PORT is the output port of the PicoBlaze micro controller. The other ports in the block diagram are for connecting external logic.

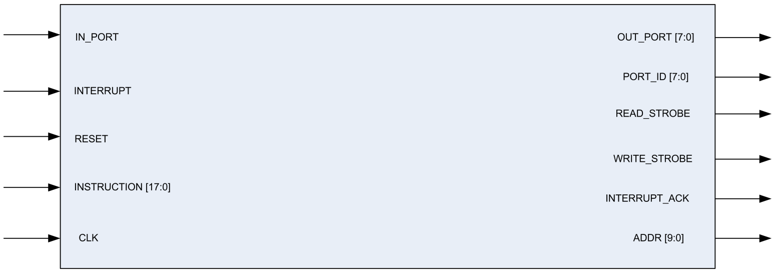


Fig. 4.8: Block diagram of a Picoblaze microcontroller.

In this framework, we consider soft errors to affect only the opcode bits of ePMs. It is also assumed in the framework that the cores are independent and parallel execution units, but the SHCA cells associated with each core's PM are connected in a mesh network with wraparound as shown in fig 4.9. Each SHCA cell participates in the network by operating on one bit from its PM. Therefore, the SHCA network operates on one bit-plane. In the uncorrupted state, in the example described here, each bit in a bit-plane is identical. This is referred to as homogeneous bit-plane (HBP).

Let us now consider the prototype code (shown in fig. 4.10) loaded into each PM of the system. The left-most column indicates the address, the middle column is the instruction in hexadecimal, and the right-most column is the assembly language of the program. It is a simple program intended to count from 0 to 255. Variants loaded across the $N \times N$ PMs include changes in start and end values of the counter and the use of different registers. In order to explain the concept of HBPs for this example, let us consider the matrix of bits shown in fig. 4.11.

The elements of a row of this matrix are formed by concatenating the bits of the first instruction (Load sx, kk) with those of the second and so on. Vectors thus obtained for each core's PM are then stacked to form the matrix. The simplest types of HBPs are those columns in the matrix that have identical bits. If the height of such a column is equal to $N \times N$, then all the CA cells in the network draw their bit from the same bit position of a specific address location in each PM. Such an HBP can exist for instance if all the CA cells are obtaining their bit from the MSB of the LOAD instruction, since its most significant bit (MSB) is always “0” (based on Picoblaze Opcode for LOAD instruction) and if all programs

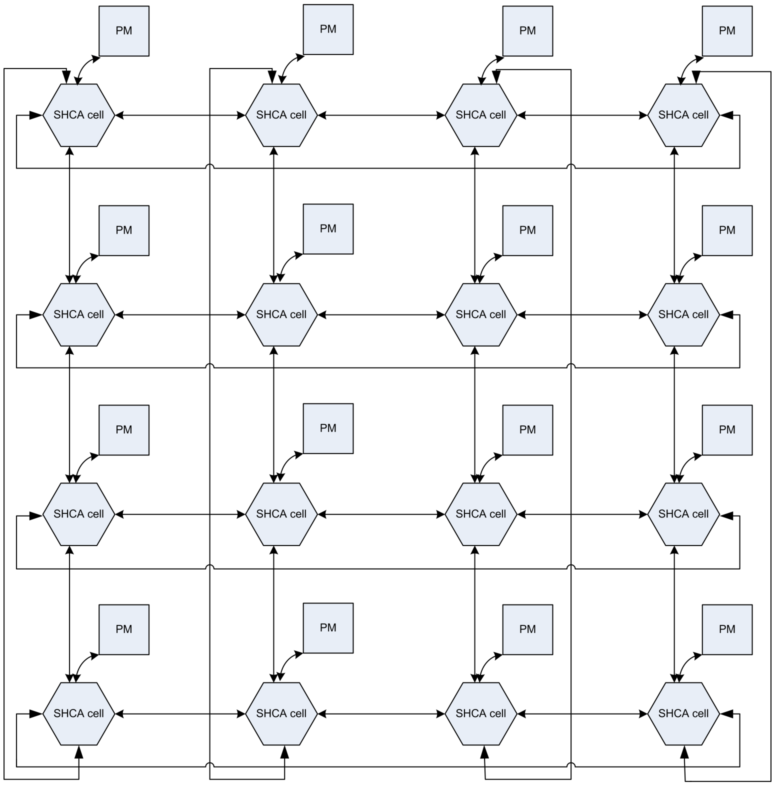
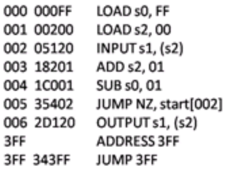
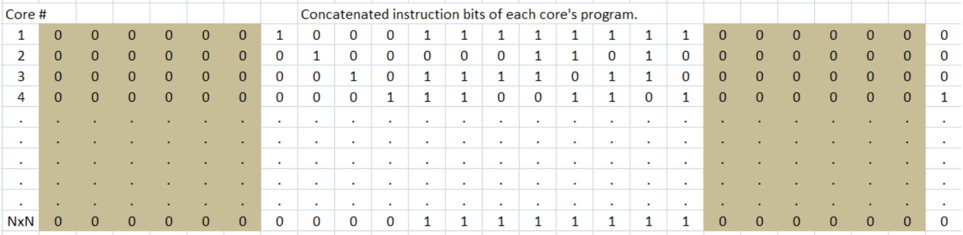


Fig. 4.9: Framework of cores and SHCA cells.



```
000  000FF      LOAD s0, FF
001  00200      LOAD s2, 00
002  05120      INPUT s1, (s2)
003  18201      ADD s2, 01
004  1C001      SUB s0, 01
005  35402      JUMP NZ, start[002]
006  2D120      OUTPUT s1, (s2)
3FF             ADDRESS 3FF
3FF  343FF      JUMP 3FF
```

Fig. 4.10: Sample counter code loaded onto the cores.



Core #								Concatenated instruction bits of each core's program.																
1	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
NxN	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0

Fig. 4.11: A sample of bits-of-instructions concatenated for all cores to allow for the search of HBPs (shaded group of bits).

are assumed to have load as their first instruction.

A closer inspection reveals that this automatically leads to five other consecutive HBPs (shown in shaded box of fig. 4.11). Therefore if variants of the sample program are such that their opcodes do not change, then several such groups of HBPs can be found. The more interesting case of finding HBPs across operands is still a topic of investigation, and hence not considered in this work. One SHCA network can work (healing process) on one bit per ePM at a time. Therefore to take advantage of available parallelism, there can be b $N \times N$ SHCA networks, with b representing the number of Opcode bits in an instruction. ePMs are designed to function with dual read and single write ports. The PicoBlaze core retains the address of the next instruction to be fetched for two clock cycles on its program counter port (PC). Due to this latency, it is possible to design a fairly simple state machine (SM) circuit that generates a copy of the PC supplied address, but offset by two. This future instruction, available at the second read port of the PM, is separated into two sets. The opcode bits which are supplied to the CA cells for error correction. The remaining bits which are the operand bits are concatenated with Opcode bits emerging from the CA network and are fed back to the ePMs. When the ePMs is switched into write mode, the offset address is still held at the second address port, allowing the corrected instruction to overwrite a presumably corrupted one. This state machine considers the state of the zero flags in the processor to synchronize with the jump instructions. The framework for this implementation is shown in fig. 4.12.

4.3 Applying SHCA to Heterogeneous ePMs

Instructions in these LDePMs framework belong to more than one instruction set architecture (ISA). If we consider a sub-set of these LDePMs (i.e., those that are affiliated with the same ISA) there is a possibility of finding redundancies within and across ePMs. To investigate this matter we considered a simple example of three such ePMs that are loaded with RISC instructions to execute different programs such as a linear algebra kernel (matrix multiplication), a signal processing kernel (FIR filter), and an image-processing kernel (integer transform). It was quickly obvious that across the three programs, there were some common instructions (ADD, LOAD, JUMP, etc.) and some that were not common (shown in fig. 4.13). We then inspected a variety of mixtures of relatively small programs and observed that there are often some instructions that form part of contours spanning multiple programs or all programs and there are some that do not. This led to the hypothesis that if contours can span all programs in LDePMs and cover all instructions, they can provide a level of embedded redundancy that can be taken advantage of for the purpose of error correction. However, traditional compiling of binaries do not have any reason to embed this form of redundancy when targeting LDePMs, unless multiple instances of a program are created to exploit program level parallelism or power-performance trade-offs [30]. Therefore to establish contours for all instructions in all ePMs, two exploratory and fairly simple approaches were taken: (1) First one program was hand-coded and the instructions used were noted. Then the second program was created, by reusing as much as possible the instructions used in the first program. This process was followed for every subsequent program to maximize the use of previously used instructions. Despite this approach, it was still not

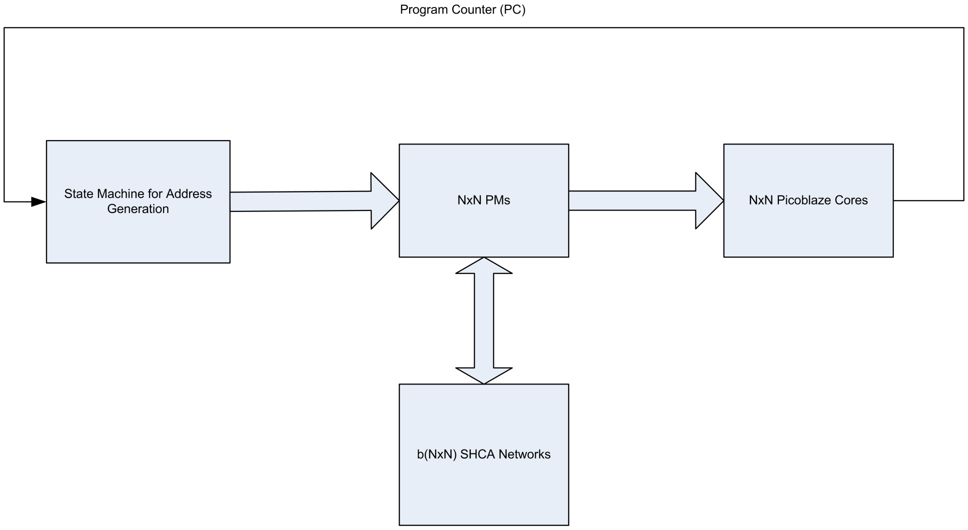


Fig. 4.12: SHCA for homogeneous ePMs .

possible to establish contours for all instructions, which required us to take an additional step. (2) Instructions that were missing (i.e., not used) in a particular program, but present in other programs were identified and inserted after the last instruction (RETURN instruction as shown in fig. 4.14). This step is somewhat similar to inserting NOOPs and resulted in code bloating. For some mixture of programs, such as three instances each of 5-tap FIR filter, Matrix multiply, and an image processing integer transform, (assuming each instance works on a different data set of data) the average extent of code bloating was in the range of 70%. However, for some other combinations of programs such as (one instance of bubble sort, selection sort, quick sort, shell sort, DWT lifting kernel, heap sort, h.264 4x4 integer transform, quick sort, and cocktail sort) the average code bloating was very high (500%). Such wide-ranging results indicate that it is not yet possible to draw strong conclusions on the quality of these approaches, since they were implemented manually with little effort on optimization. Nonetheless, they do show that for certain mixtures of programs it is possible to identify large levels of inter-program redundancy in the form of contours. Once this redundancy is exploited the SHCA network can be applied to the contours in the LDePMs for error correction. The physical realization of such a CA-based self-healing system that can correct errors in an LDePM system, is based on the diagram shown in fig. 4.15. The

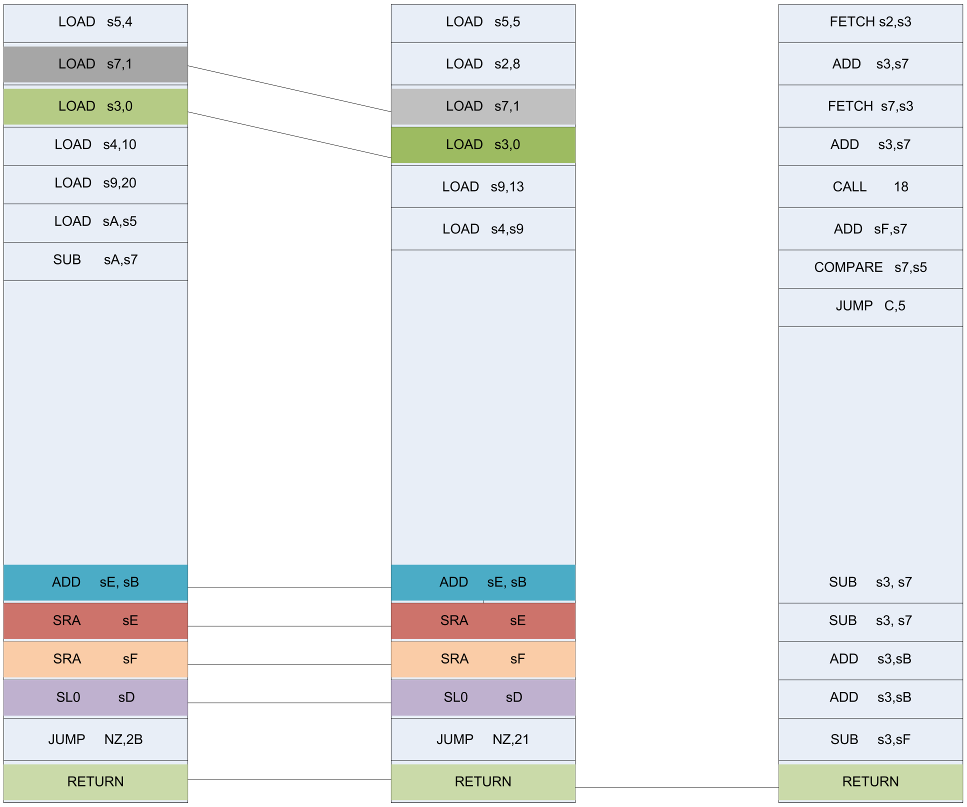


Fig. 4.13: Initial contour map for three program memories.

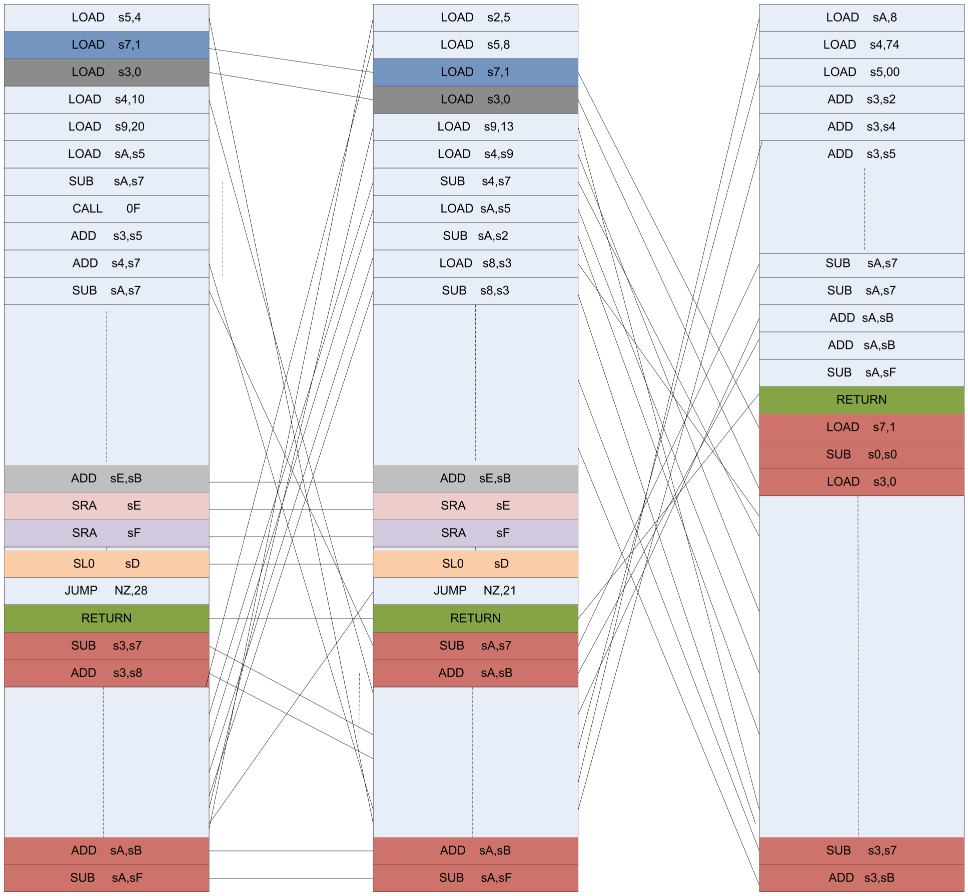


Fig. 4.14: Countour formation after register renaming and instruction insertion.

system is composed of a set of $N \times N$ ePMs, each having m or fewer instructions (of b bits each). This implies that the number of contours (m) is bound by the ePM that has the largest number of lines of code. A contour has $N \times N$ set of addresses, each $log_2(m)$ bits wide. These address bits are stored in a Contour information bank (CIB). Each ePM is a dual port memory, with one port supplying code to a processor, and the other port addressed by the CIB to supply the SHCA with potentially corrupted instruction bits. This process enables a non-intrusive healing of ePMs. Each time the CIB puts out addresses to the LDePMs, they deliver $N^2 \times b$ bits that are to be healed (corrected). These bits are delivered in b bit-planes to a corresponding set of b SHCA networks (acting concurrently on all the bit-planes). During the healing period, which could take several iterations, the addresses for all the ePMs are held steady by the CIB. The physical time for healing is

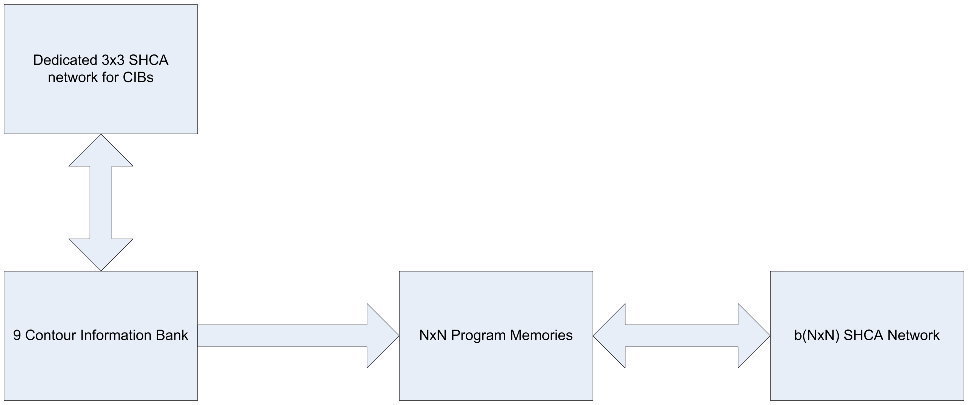

Dedicated 3x3 SHCA network for CIBs
9 Contour Information Bank
NxN Program Memories
b(NxN) SHCA Network

Fig. 4.15: SHCA system for heterogeneous ePMs.

set by an upper bound estimate of the number of iterations needed by an SHCA to heal a maximum percentage of errors. After the bits are healed, the CIB asserts a write enable to all the ePMs allowing healed instructions to overwrite corrupted instructions. However, despite the ability of LUTs in the SHCA to heal themselves, the introduction of a CIB exposes a potential weakness: the vulnerability of information stored in the CIB itself. To overcome this hurdle, the use of a set of nine CIBs (eight copies of redundancy) is proposed, such that they are connected to a separate and dedicated SHCA to heal themselves. The specific reason to choose a set of nine CIBs is that a 3 × 3 SHCA is the smallest network capable of healing errors. The results and comaparisons are presented in the next chapter.

Chapter 5

Results and Analysis

5.1 SHCA Analysis

Simulations are conducted to analyze the effects of localized corruption in a $N \times N$ ePm system (provided by Dr. David Peak). For concreteness, it is assumed that the system consists of 12x12 memories and that bit corruption occurs in an $n \times n$ ($n < 12$) subsystem patch ("hot spot"). In fig. 5.1, the data "SHCA PERFECT" show that when the SHCA LUT is uncorrupted, the SHCA heals all corrupted bits in a 7x7 memory array hot spot, where up to 20% of the instruction bits are corrupted. Each data point represents 1000 random initial configurations, healed for a maximum of eight iterations. This actually under-represents the healing power of the SHCA: as long as the SHCA remains uncorrupted, it can perfectly heal hot spots in which even 100% of the bits are corrupted, provided the corrupted fraction is less than 50% of all possible bits. In other words, the SHCA strategy is at least as good as any other error-correction method, for cases where both produce perfect outputs.

More interesting perhaps is the case where the error-correction technology is itself vulnerable to corruption. For some "local" methods (i.e., those applied separately cell-by-cell), a wrong output bit disables correction and might actually increase the error density. As shown in fig. 5.1, however, even when the SHCA suffers the same corruption level as the instructions it is designed to correct, it still manages to clean up damaged bits. For example, at a relatively high level of 5% corruption in the hot spot, the damaged SHCA (with 5% of its outputs flipped) corrects over 75% of the initial bit errors. Because all bits are equal to the SHCA, a damaged SHCA can be directed onto itself to "self-repair." Figure 5.1 shows the result of borrowing four of the eight iteration cycles to first do SHCA self-repair. Even this unsophisticated scheme leads to remarkable improvement after self-repair. For the 5%

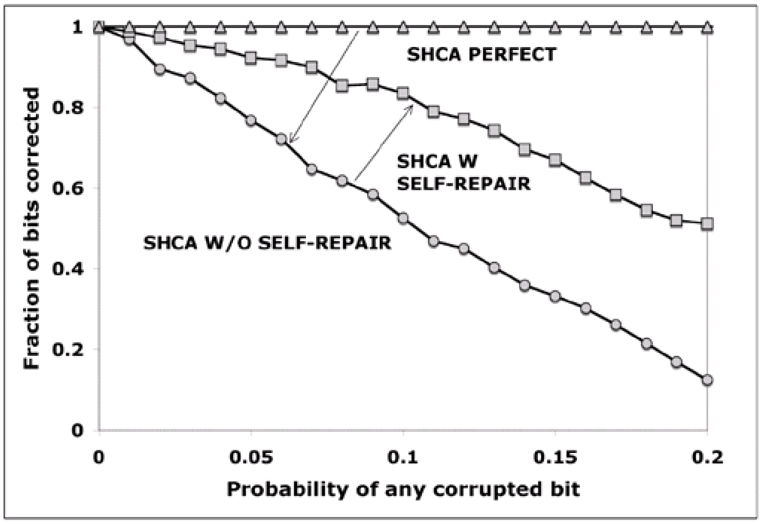


Fig. 5.1: Plot showing the ability of 12x12 SHCA to correct a set of 144 bits in eight cycles.

corruption case, for example, the self-repaired SHCA increases its correction rate to over 92%. This is not a trick standard error-correcting techniques can replicate.

Figure 5.2 examines how time-to-heal depends on corruption level. The perfect SHCA heals very rapidly, requiring no more than three iterations for levels up to 20%. Corrupted SHCA takes longer to correct all bits when it is able to do so. The data show that little additional improvement in self-repair is possible after a small number of iterations, so borrowing more time is not necessarily useful. Instead, further enhancement of self-repair will require subtler use of the inherent redundancy of the SHCA network.

Since the proposed CA network does the task of converging the minority bits to the value of majority bits, it is appropriate to compare the effectiveness of our technique to that of traditional majority voters like the Triple Modular Redundancy (TMR) voter and a 5-input Voter CA (VCA), which is basically a 5-input majority voter. It is a known fact that TMR voter and VCA perform exceptionally well when the contents of the LUT themselves are not corrupted and when the total percentage of errors is within the permissible limit. But the thing of interest would be what if the LUTs for each of the technique get corrupted. Figure 5.3 shows that rule corruption has significant detrimental effect on all three strategies, with TMR suffering worse than the two CAs. CA continues to outperform both TMR and

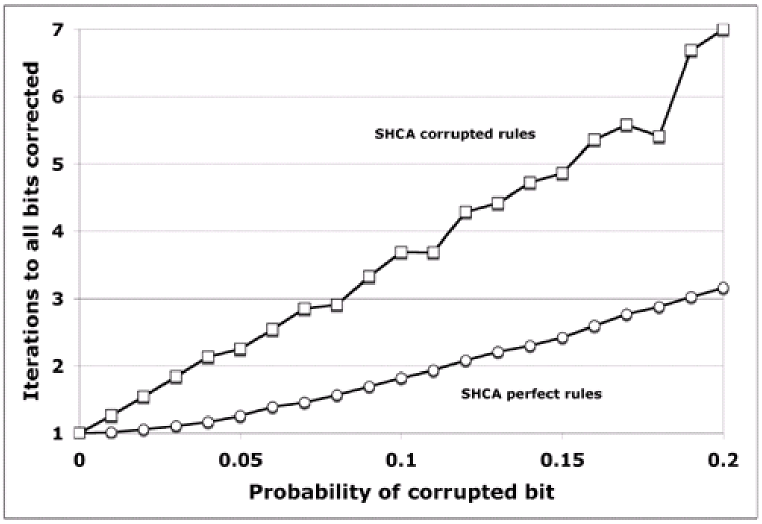


Fig. 5.2: Plot showing the time taken by a 12x12 SHCA to correct a set of 144 bits.

VCA, and even under highly damaged conditions still manages to correct more erroneous bits than TMR and VCA. The meaning of the "negative" corrected fraction in fig. 5.3 is that damaged TMR actually adds more corruption to the instructions than was initially present. The y-axis of the plot indicates the fraction of bits corrected out of the total number of corrupted bits and the x-axis indicates the probability of each bit getting corrupted, with bits in the LUTs are corrupted with the same frequency as those in the instructions.

5.2 Implementation Overhead

5.2.1 CA-Based Implementation Overhead

The CA-based implementation overhead is given by

$$\begin{aligned} Transistor\ Overhead\ of\ CA\ &= (Transistor\ overhead\ of\ CA\ network\ for \\ &\quad healing\ the\ ePM's) + (Transistor\ overhead\ of \\ &\quad 9\ CIB's) + (Transistor\ overhead\ of\ the\ CIB's \\ &\quad CA\ network). \end{aligned} \tag{5.1}$$

Each RT of the CA-based implementation consists of 114 transistors. So the total overhead of an $N \times N$ CA network is $114 \times (N^2)$. Assuming that there are b bits in an instruction, simultaneous healing of all the bits in an instruction is possible with the use of b SHCA networks operating concurrently on each bit-plane. The overhead for b such SHCA networks therefore is $114 \times (N^2) \times b$. Assuming that the largest ePM has m instructions, the number of bits required to store the address of an instruction in a CIB is $log_2(m)$. Therefore, the transistor overhead of the 9 CIBs is $log_2(m) \times (N^2) \times m \times 54$. In order to heal soft errors in the CIBs, we have a dedicated SHCA network that is composed of 9 LUTs. The transistor overhead of this CIB healing SHCA network is 114× 9 = 1,026. Therefore, (eq. 5.1) evaluates to $[114 \times (N^2) \times b] + [log_2(m) \times (N^2) \times m \times 54] + 1,026$.

5.2.2 SHCA-Based Implementation Overhead

The overhead for the SHCA implementation is same as that of the CA-based implementation except for fact that the CA network is replaced by an SHCA network.

$$\begin{aligned} Transistor\ Overhead\ of\ SHCA\ &= (Transistor\ overhead\ of\ SHCA\ network\ for \\ &healing\ the\ ePM's) + (Transistor\ overhead\ of \\ &9\ CIB's) + (Transistor\ overhead\ of\ the\ CIB's \\ &SHCA\ network). \end{aligned} \tag{5.2}$$

Each RT of the SHCA network is made up of 305 transistors. The total overhead of an $N \times N$ LUT based SHCA network is $305 \times (N^2)$. The transistor overhead of this CIB healing SHCA network is 305× 9 = 2,835. Continuing with the assumptions made for calculating the overhead of the CA implementation, (eq. 5.2) evaluates to $[305 \times (N^2) \times b] + [log_2(m) \times (N^2) \times m \times 54] + 2,835$.

5.3 Comparison with ECC Techniques

5.3.1 Comparison of SHCA with Hamming and TMR

A graph has been plotted comparing the transistor overhead for SEC-DED Hamming

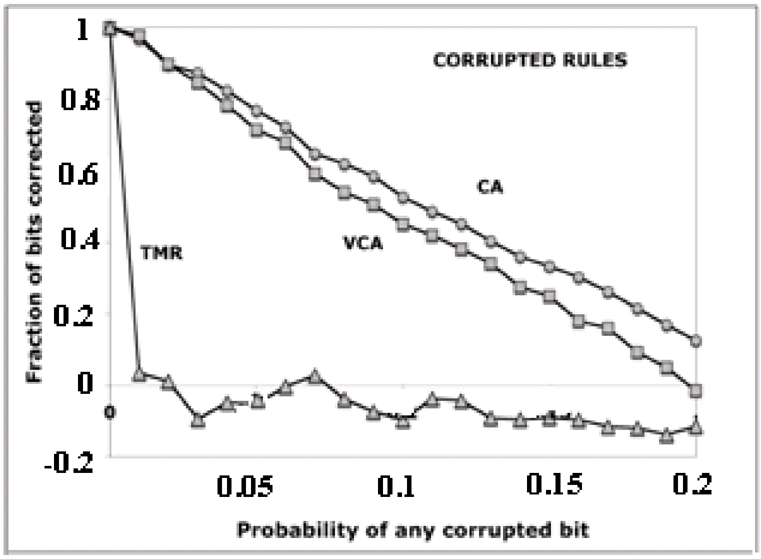


Fig. 5.3: Plot showing the fractions of instruction bit errors corrected by corrupted DMV, VCA, and TMR implementations as a function of error density. Bits in the error-correcting programs are corrupted with the same frequency as those in the instructions.

code, Hierarchical Ternary TMR voter, and the proposed SHCA method. All the techniques where implemented on a Xilinx Virtex4 LX60 FPGA for instruction width of 16 bits. It is evident for the plot (shown in fig. 5.4) that the overhead incurred for the SHCA implementation is high compared to the other two techniques. But as discussed in Chapter 3 SEC-DED Hamming and Hierarchical Ternary TMR voter are not tolerant to errors within the system, whereas the SHCA is capable of self-healing itself. Most of the overhead incurred in the SHCA is due to the CIB's for storing the address, if the CIB's are optimized the overhead would reduce future.

5.3.2 Comparison of SHCA with Reed Solomon

Plot shown in fig. 5.5 gives a overhead comparison for proposed SHCA vs. Reed Solomon code for different error correction capabilities. In this setup 144 embedded program memories were considered, each having 160 instructions with an instruction width of 12. Reed Solomon codes were implemented using Xilinx Intellectual Property Cores (IP cores) on Xilinx Virtex4 LX60 FPGA. The overhead in the plot is in terms of 4-input LUTs.

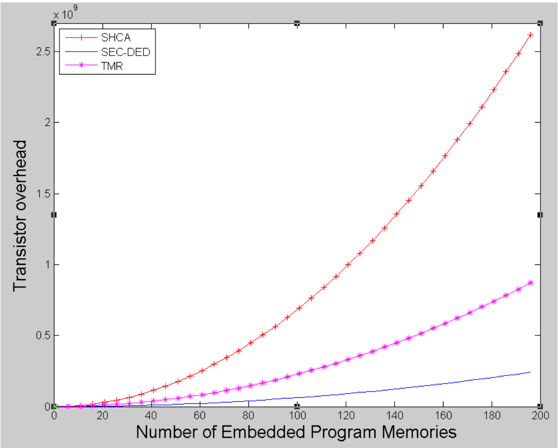


Fig. 5.4: Plot showing the transistor overhead for SEC-DED Hamming code, Hierarchical Ternary TMR voter, and proposed SHCA method.

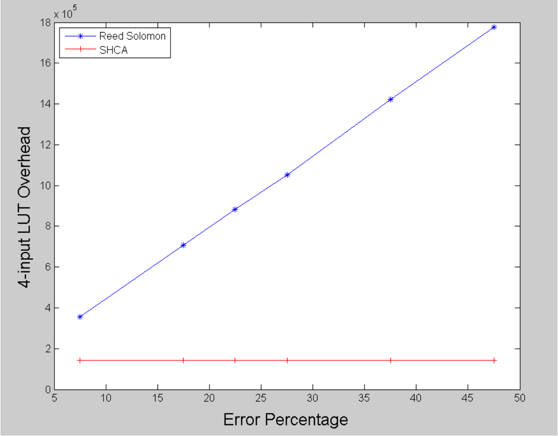


Fig. 5.5: Plot showing 4-input LUT overhead for Reed-Solomon and proposed SHCA for different error correction capabilities.

Chapter 6

Conclusion

In this thesis the potential for large-scale soft errors in nanoscale storage elements, particularly SRAM flip-flops based on a survey of articles in the areas of error protection for memory systems is discussed. While one cannot be sure how reliable such future nanoscale embedded memory systems will be, it cannot however be assumed that large scale soft errors will not occur. A growing number of publications in the area of noisy computations seem to indicate that several researchers expect such problems to be important enough, warranting early investigations. This work also investigates the presence of redundancy that is likely to exist between instruction binaries in a set of large and distributed embedded program memories. Preliminary results indicate that it is worth looking into more sophisticated compiler-based techniques such as register renaming and code motion, to create contours of redundant instructions across programs. This was followed by the discussion of a belief propagation-based cellular automata system that can collectively determine the majority bit in a plane of identical bits infected by large scale and distributed or clustered bit-flips. We then presented a possible circuit realization based on LUT technology well understood in the FPGA community. This realization enables LUTs to heal themselves providing for a more robust error correction circuit. This was followed by a set of results and analysis of an SHCA system and a qualitative comparison with contemporary error correction techniques based on available published data. It can be seen that the SHCA system has the ability to correct a large and distributed set of errors, as well as itself from time to time, but incurs a high cost to securely maintain the contour information in CIBs. This shows that the best possible error correction technique may need to use a combination of MRF gates and SHCA for a set of ePMs.

References

[1] B. Calhoun and A. Chandrakasan, "Analyzing static noise margin for sub-threshold sram in 65nm cmos," in *Proceedings of the 31st European Solid-State Circuits Conference, 2005*, pp. 363–366, Sept. 2005.

[2] K. Granhaug and S. Aunet, "Improving yield and defect tolerance in multifunction subthreshold cmos gates," in *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006*, pp. 20–28, Oct. 2006.

[3] S. Hanson, B. Zhai, K. Bernstein, D. Blaauw, A. Bryant, L. Chang, K. K. Das, W. Haensch, E. J. Nowak, and D. M. Sylvester, "Ultralow-voltage, minimum-energy cmos," *IBM Journal of Research and Development*, vol. 50, no. 4/5, pp. 469–490, June 2006.

[4] H. Tuinhout, "Impact of parametric fluctuations on performance and yield of deep-submicron technologies," in *Solid-State Device Research Conference, 2002. Proceedings of the 32nd European*, pp. 95–102, Sept. 2002.

[5] H. Li, J. Mundy, W. Paterson, D. Kazazis, A. Zaslavsky, and R. Bahar, "Thermally-induced soft errors in nanoscale cmos circuits," *IEEE International Symposium on Nanoscale Architectures, 2007*, pp. 62–69, Oct. 2007.

[6] M. Spica and T. Mak, "Do we need anything more than single bit error correction (ecc)?" in *Records of the 2004 International Workshop on Memory Technology, Design and Testing*, pp. 111–116, Aug. 2004.

[7] C.-L. Su, Y.-T. Yeh, and C.-W. Wu, "An integrated ecc and redundancy repair scheme for memory reliability enhancement," in *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 81–92. Washington, DC, USA: IEEE Computer Society, 2005.

[8] L. Hung, H. Irie, M. Goshima, and S. Sakai, "Utilization of secded for soft error and variation-induced defect tolerance in caches," in *Design, Automation and Test in Europe Conference and Exhibition, 2007*, pp. 1–6, Apr. 2007.

[9] H. Sun, N. Zheng, and T. Zhang, "Realization of l2 cache defect tolerance using multi-bit ecc," in *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems, 2008*, pp. 254–262, Oct. 2008.

[10] R. Naseer and J. Draper, "Dec ecc design to improve memory reliability in sub-100nm technologies," in *15th IEEE International Conference on Electronics, Circuits and Systems, 2008.*, pp. 586–589, Sept. 2008.

[11] M. Y. Hsiao, "A class of optimal minimum odd-weight-column secded codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[12] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, “Multi-bit error tolerant caches using two-dimensional error coding,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 197–209. Washington, DC, USA: IEEE Computer Society, 2007.

[13] A. Dutta and N. A. Touba, “Multiple bit upset tolerant memory using a selective cycle avoidance based sec-ded-daec code,” *VLSI Test Symposium, IEEE*, pp. 349–354, 2007.

[14] T. Suzuki, Y. Yamagami, I. Hatanaka, A. Shibayama, H. Akamatsu, and H. Yamauchi, “A sub-0.5-v operating embedded sram featuring a multi-bit-error-immune hidden-ecc scheme,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 152–160, Jan. 2006.

[15] W. Zhang, “Replication cache: A small fully associative cache to improve data cache reliability,” *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1547–1555, 2005.

[16] C. Slayman, “Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, Sept. 2005.

[17] C. Jeffery, A. Basagalar, and R. Figueiredo, “Dynamic sparing and error correction techniques for fault tolerance in nanoscale memory structures,” in *4th IEEE Conference on Nanotechnology, 2004*, pp. 168–170, Aug. 2004.

[18] K. Nepal, R. I. Bahar, J. Mundy, W. R. Patterson, and A. Zaslavsky, “Designing nanoscale logic circuits based on markov random fields,” *Journal of Electronic Testing*, vol. 23, no. 2-3, pp. 255–266, 2007.

[19] K. Nepal, R. I. Bahar, J. Mundy, W. R. Patterson, and A. Zaslavsky, “Designing mrf based error correcting circuits for memory elements,” in *Proceedings of the conference on Design, automation and test in Europe*, pp. 792–793. 3001 Leuven, Belgium: European Design and Automation Association, 2006.

[20] E. L. Blokh and V. V. Zyablov, *Linear concatenated codes*, 1982.

[21] D. Peak, J. D. West, S. M. Messinger, and K. A. Mott, “Comparing the dynamics of stomatal networks to the problem-solving dynamics of cellular computers,” in *Proceedings of the International Conference on Complex Systems*, 2005.

[22] I. Cowan and G. Farquhar, “Stomatal function in relation to leaf metabolism and environment,” *Symposium of the Society for Experimental Biology 31*, 1977.

[23] D. Peak, J. D. West, S. M. Messinger, and K. A. Mott, “Evidence for complex, collective dynamics and emergent, distributed computation in plants,” in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 4, pp. 918–922, Jan. 2004.

[24] L. Chua and L. Yang, “Cellular neural networks,” in *IEEE International Symposium on Circuits and Systems, 1988*, pp. 985–988 vol. 2, June 1988.

[25] L. Chua and L. Yang, “Cellular neural networks: theory,” *IEEE Transactions on Circuits and Systems*, vol. 35, no. 10, pp. 1257–1272, Oct. 1988.

[26] WolframMathWorld, “Cellular automaton,” [http://mathworld.wolfram.com/CellularAutomaton.html], 2009.

[27] T. K. Moon, “Error correction coding: Mathematical methods and algorithms.” Hoboken, New Jersey: John Wiley & Sons, Inc., 2005.

[28] Xilinx, “Picoblaze 8-bit embedded microcontroller user guide,” [http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf], 2007.

[29] Xilinx, “Kcpsm3 8-bit micro controller for spartan-3, virtex-ll and virtex-llpro,” [http://www.eng.auburn.edu/~strouce/class/elec4200/KCPSM3_Manual.pdf], 2007.

[30] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, “Single-isa heterogeneous multi-core architectures: The potential for processor power reduction,” in *36th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 81–92, Dec. 2003.