LOGIC REPAIR AND SOFT ERROR RATE REDUCTION USING APPROXIMATE LOGIC FUNCTIONS

By

Adeola Adeleke

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2012

Nashville, Tennessee

Approved by:

Professor Bharat L. Bhuva

Professor Lloyd W. Massengill

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Prof. Bharat L. Bhuva for his constant insight and guidance through my pursuance of this Master's degree. Without his support, my progress in this research would not have been as expedited. Additionally, I would like to thank Brian D. Sieraswki whose initial research in this topic served as a springboard for this thesis. He also provided me with enough information to continue his research, and assisted me in getting past impasses. Much credit also goes to Andrew Sternberg and Jugantor Chetia for their inputs in this project. I would also like to acknowledge Dr. Lloyd W. Massengill for serving on my thesis committee.

On a more personal note, I am extremely indebted to my family, Adeniyi Adeleke, Funmilayo Adeleke, Gbemisola Adeleke, Olufunmilayo Adeleke Jr., and Ayodeji Adeleke for their prayers, emotional support, and occasional academic insight.

Finally, I am grateful to the School of Engineering and the Graduate School for providing me with a fellowship to study at Vanderbilt, without which this project would have been impossible.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Moore's Law, which stipulates that transistor density will double every eighteen months, has been the driving force in advancing complementary metal-oxide-semiconductor (CMOS) technology fabrication for the past 50 years [1]. Moore's Law is achieved by reducing the feature size of transistors, allowing more transistors to fit on the same die area as previous technology nodes. This transistor scaling principle is behind Intel's "Tick" of the "Tick-Tock" model [2], and also governs industry-wide advancements in very large scale integration (VLSI). While transistor scaling results in high architectural performance, high transistor density and low power consumption, it also increases the vulnerability of integrated circuits (IC) to single event transients (SET) and single event upsets (SEU) [4-7].

SEUs typically occur when highly energetic radiation particles, such as protons, neutrons, alpha particles or other heavy ions (an ion with an atomic number $Z > 1$), strike a sensitive circuit node in an IC, generating an accumulation of electron-hole pairs (EHP) at the node. If the accumulated charge is greater than the critical charge Q_{crit}, the minimum charge required to upset a circuit node, the nodal voltage is altered [8-9, 11]. The resulting voltage alteration either generates an SEU, if the affected node is a storage node in a memory element (latch or flip-flop), or an SET, if the affected node is a combinational element node. SETs that propagate through the combinational cloud may be latched by a memory element thereby resulting in an SEU. If an SEU propagates to the primary output(s) of a design, it becomes a soft error. While SEUs/soft errors are not permanently damaging to a circuit, they have adverse effects on the architectural reliability of application specific

integrated circuit (ASIC) designs, and can be extremely harmful in field programmable gate array (FPGA) applications [10].

Due to the increasing susceptibility of today's deep sub-micron technologies to radiation-induced faults, and the fact that repairing an entire system is exorbitant, it is pertinent to incorporate hardware robustness in new designs. Several methods have been discussed in literature, many of which provide robustness in the form of hardware redundancy [12-15]. The amount of redundancy implemented may range from individual components to entire systems. However, these redundancy techniques require significant power and area overhead (>2X area increase in the Triple Modular Redundancy (TMR) approach). Also, with the exception of the N-modular redundancy (NMR) family, most of these methods deviate from real life by inherently assuming that the protection of one output guarantees the protection of other outputs. As this is hardly ever the case, many of these techniques are not useful on a modular level. Other methods that have been explored include error detection and correction (EDAC) solutions typically used in memories, flip-flops (FF) and latches [16-19]. While these methods are reliable and cost efficient, they are not suitable for combinational implementations. Device-level techniques [17, 20-21] have also been studied extensively; however they require trade-offs in operating speed, area, and/or power, and sometimes an expensively extensive revamp of the entire IC fabrication process.

This thesis, a continuation of the idea put forth by Sierawski, et. al [3], presents a novel technique that uses partial logical masking for logic repair by generating approximate logic functions for each output based on other outputs of the design. Unlike other design approaches for providing hardware robustness, the technique proposed in this thesis provides the designer flexibility in choosing the level of logic repair required while balancing out power, speed and area penalties. Also, based on certain design restrictions, the designer may decides which of three proposed methods will

be used to generate the approximate functions.

This organization of this thesis is as follows. Chapter II presents a detailed discussion on radiation-induced faults and how they affect system reliability metrics. Chapter III discusses existing techniques used for SER reduction and logic repair. Chapter IV proposes a new approach for SER reduction and logic repair. Chapter V details the implementation of the proposed technique and the experimental setup for testing the proposed technique on benchmark circuits. Simulation results are presented and discussed in Chapter VI. Chapter VII recaps this thesis and discusses possible improvements as future work. The appendix provides the supporting scripts used for area synthesis and fault simulation analysis. Due to page-limit constraints, the core codebase of this project is not included in the appendix. However, core files are available upon request.

CHAPTER II

EFFECTS OF RADIATION-INDUCED FAULTS ON ARCHITECTURAL VULNERABILITY

Sources of Radiation Particles

The Big Bang theory, the prevailing cosmological model for explaining the universe, stipulates that the universe expanded from a singularity to high energy subatomic particles such as protons, neutrons, and electrons about 13.7 billion years ago [23]. In 1998, S. Perlmutter et.al discovered the continuing accelerating expansion of the Universe by studying distant supernovae. These supernovae produce extremely luminous radiation-laden cosmic ray explosions capable of outshining a galaxy. Supernovae that occur close enough to earth, or produce far-traveling radiation release high energy cosmic ray particles into the earth's atmosphere as shown in figures below [24-27]. Other stars, most relevantly, the sun, also produce cosmic ray particles during solar flare events [30].

Figure 1 [28] shows the flux of cosmic rays in space. Figure 1(a) presents a plot of the flux cosmic ray particles against their kinetic energies, showing helium particles to have the highest flux, and beryllium particles the lowest. Figure 1(b) compares the intensity detected by various cosmic ray injection detection methods against the energies of the detected particles. Figure 2 [29] shows how cosmic ray particles travel from space to sea level. Figure 3 [28] is a plot of the flux of particles at sea level against their energies. In this figure, neutrons appear to be most prominent among sea level particles with energies below 0.1GeV, while muons dominate among sea level particles with energies above 0.1GeV.

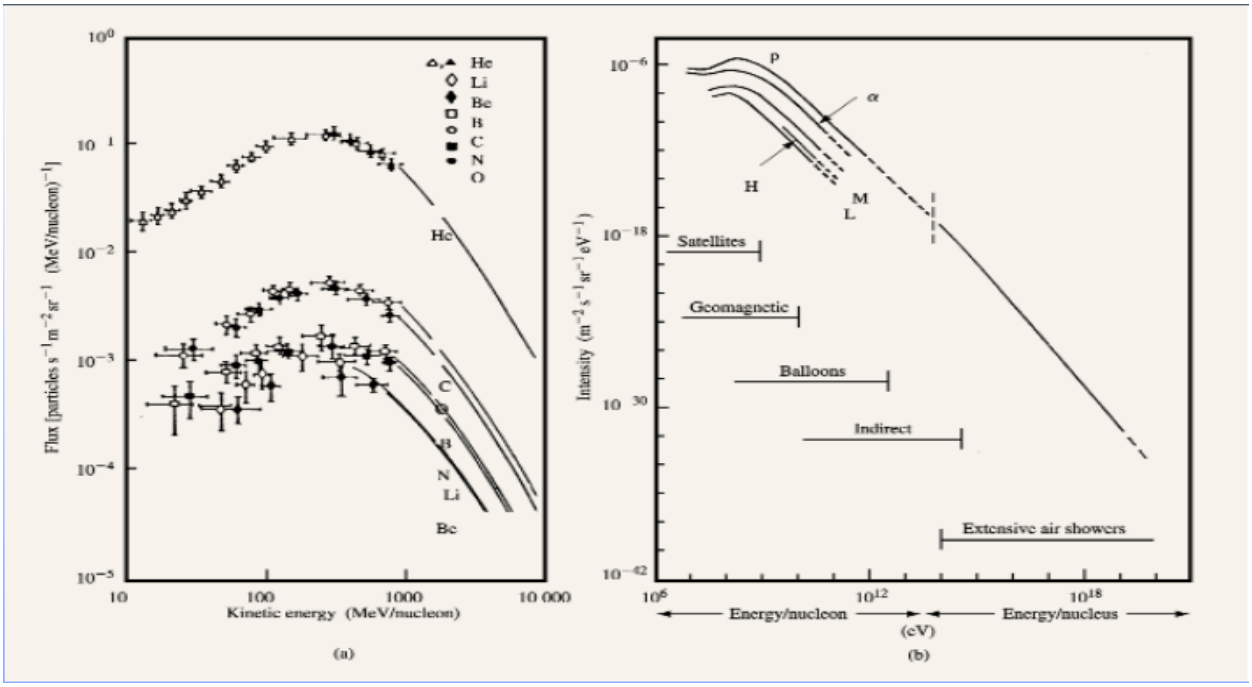


Figure 1 [28]: Flux of Cosmic Rays in Space

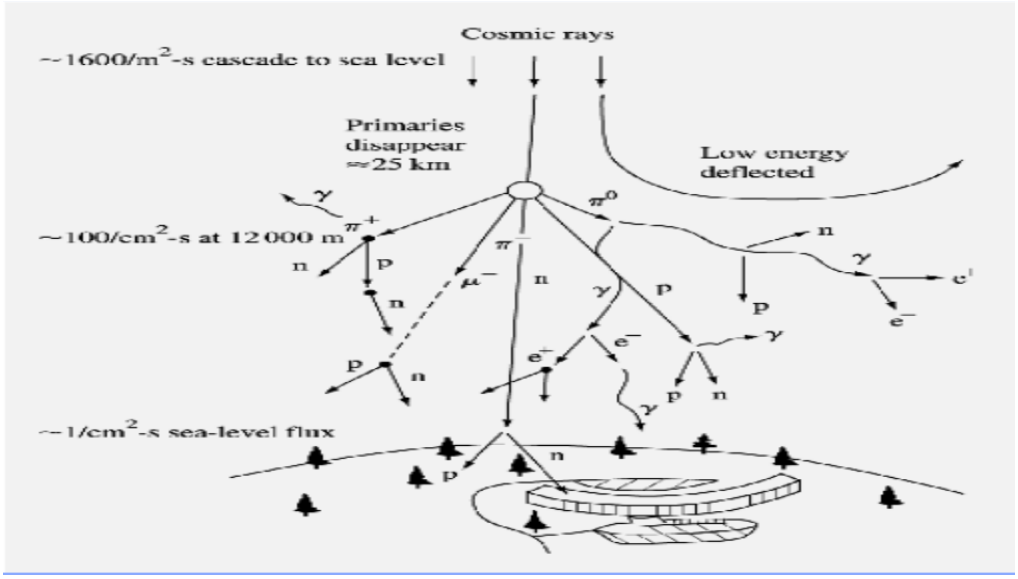


Figure 2 [29]: Particles deposited in the atmosphere after the impact of cosmic ray

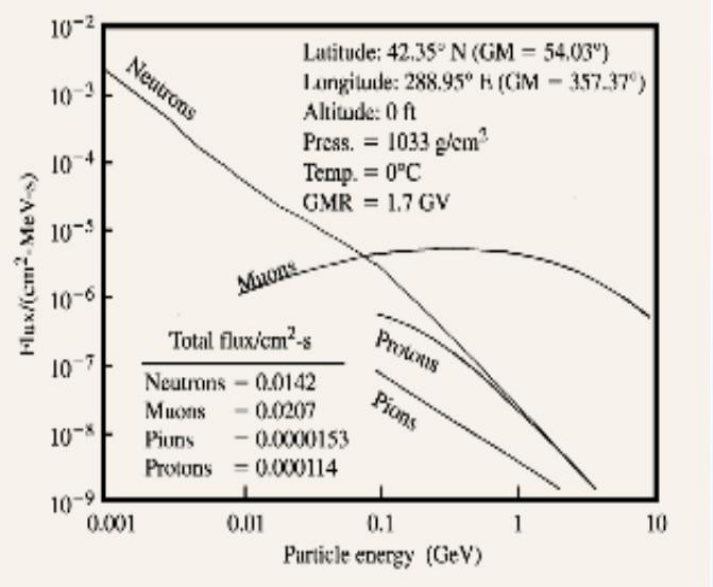


Figure 3 [28]: Particles deposited at sea level

Radiation particles have been a source of concern for space electronics applications since several years ago when bit error anomalies observed in circuits in satellites orbiting earth were first attributed to cosmic ray ionization [31-33]. As a result, much research has been conducted to study the effect of ionizing radiation on extraterrestrial electronics. However, since cosmic ray particles are more prominent in outer space, not much effort has been put into studying the effect of radiation on electronics on earth until recently. Recent studies have proven that device scaling in successive technology nodes exacerbates the effect of high energy radiation particles on terrestrial electronic circuits causing increased unreliability and performance degradation. The SET rate of combinational circuits has also been shown to be frequency dependent, thus recently manufactured devices, which tend to operate at high frequencies, are usually more soft-error prone [1, 4-7].

Neutron, proton, alpha and heavy-ions particles, all present in cosmic rays, are thought to be responsible for most of the radiation effect events that impact ICs. Neutron, proton, heavy-ion, and to a lesser extent, alpha particles also invade our atmosphere via the Van Allen radiation belt [34].

Semiconductor packaging materials can also introduce alpha particles into electronic devices [35].

Origin of SEUs

Charge Deposition

When high energy radiation settles on an IC, electric charge can be generated primarily in two possible ways:

Direct Deposition: On contact with a transistor node in a CMOS device, high energy radiation particles migrate through the device freeing up EHPs and losing energy in the process before coming to rest after energy exhaustion. Linear energy transfer (LET) is a term that defines the energy transferred to the device as an energetic particle travels through it. The unit, MeV-cm^2/mg, is obtained by normalizing the energy loss per unit length (MeV/cm) by the material's density (mg/cm^3). In Silicon (Si) devices, an LET of about 97 MeV-cm^2/mg is equivalent to a charge deposition of 1 pC/μm. Figure 4 [4] shows the charge generation per unit distance traveled for heavy ions in Silicon. Heavy ions tend to generate charge through direct deposition, while lighter ions such as protons usually do not possess enough energy to produce charge by direct deposition. Some studies however suggest that as ICs become increasingly susceptible, protons might cause upsets by direct ionization [9, 36-37].

Indirect Deposition: Indirect deposition is the primary mechanism through which light particles deposit electric charge. When a high energy light particle strikes a transistor, an inelastic collision could occur with the affected nucleus setting off any one of the following nuclear reactions.

1) Emission of alpha and gamma and the recoil of the resulting nucleus (eg., Si produces alpha particles and a recoiling Mg nucleus as shown in figure 5(a) [38])

2) Spallation reactions in which the affected nucleus splits into two fragments each of which can recoil (e.g., Si splits into C and O ions as seen in Figure 5(b) [38]).

3) Elastic Collisions that produce Si nucleus recoils.

The resulting product of any one of these reactions is capable of depositing energy by direct ionization. However the resulting products tend to have relatively low energies and do not travel far from their point of impact [9].

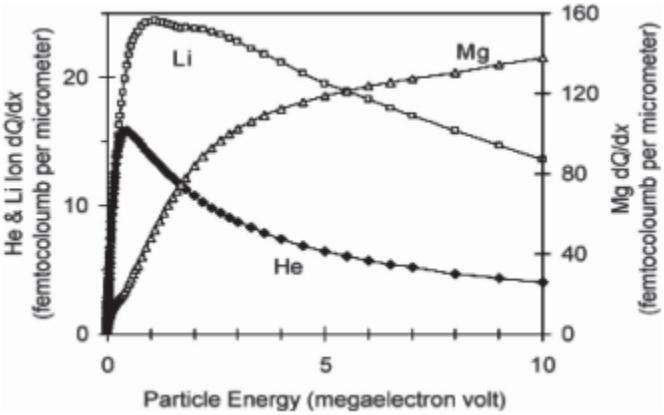
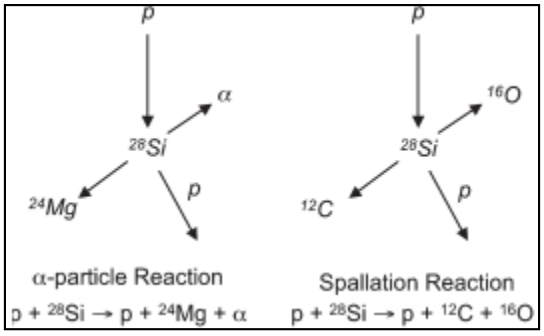


Figure 4 [4]: Charge generation per linear distance for some ions in Silicon



(a) (b)

Figure 5 [38]: Indirect Ionization Reactions in Silicon

Charge Transport and Collection

After electric charge has been deposited in a semiconductor material via any of the methods discussed above, the charge may drift into regions with electric field, diffuse to neutral regions, or recombine with other mobile carriers in the semiconductor lattice. Any of these transport processes causes current to flow possibly resulting in SEUs, however reverse biased p-n junctions are the most sensitive regions in CMOS devices due to the high electric field present in reverse biased junction depletion regions. Charge deposited at a reverse biased p-n junction is collected through drift mechanisms generating transient current in the process. Particle strikes that occur close to the depletion region can also result in transient currents as ions can diffuse towards the depletion region where they are collected [6, 9].

Figure 6 [6] illustrates the principle behind charge transport and collection. Figure 6(a) shows a cylindrical track densely populated with EHPs generated from the incident ionizing radiation impacting the drain of a transistor. Figure 6(b) shows how the electric field present in the depletion region collects charge via drift. Also noticeable in the figure is a funnel shape which aids in drift charge collection by extending the depletion region into the substrate. Figure 6(c) shows ions diffusing towards the depletion region dominating the previous ion drift collection process after a few picoseconds. The transient current generated due to the charge transport is plotted against time in figure 6(d).

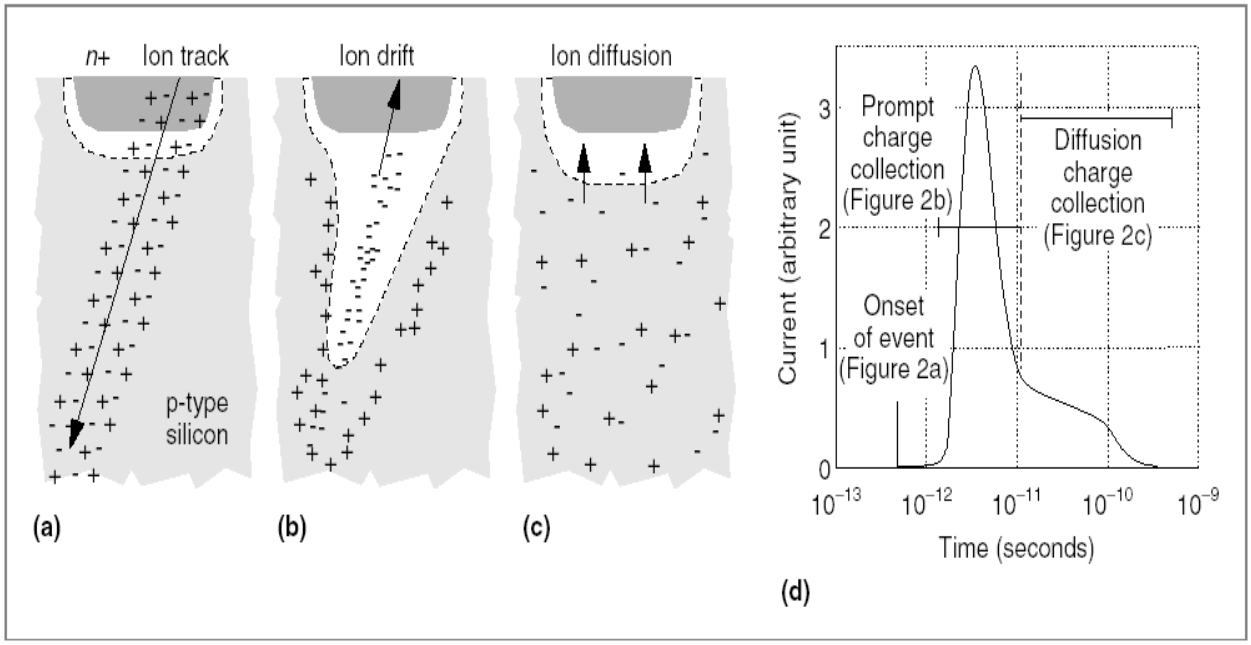


Figure 6 [6]: Charge collection and resulting transient current

SEUs in Integrated Circuits

Digital systems are typically divided into two subsystems - combinational systems and sequential systems (composed of memory elements). Figure 7 [17] shows how these subsystems are typically connected together. On a clock edge, the data from FF U1 is fed into the combinational subsystem which performs some Boolean algebra before providing its result to the input of FF U2. Combinational logic and memory elements are affected by ionizing radiation quite differently; however both contribute to the SEU rate of digital systems. As stated in Chapter I, there are two mechanisms through which SEUs can occur in ICs one attributed to each subsystem. SEUs induced by memory elements can occur when radiation particles strike memory elements directly. However, SEUs induced by combinational elements can occur only if the SETs generated when radiation particles strike a vulnerable combinational node, are latched by a memory element. Due to the differences in the SEU generation mechanism for memory elements and combinational circuits, it's

imperative to discuss both mechanisms albeit succinctly.

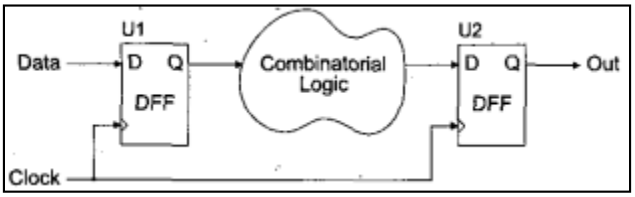


Figure 7 [17]: Sample connection between combinational and sequential elements

SEU in Memory Elements

SEUs in DRAMs: DRAMs are a class of memory elements that use a storage capacitor to passively store digital information. The information stored degrades over time and stored signals need to be refreshed periodically. Since DRAMs lack a discernible regeneration path, any radiation-particle-strike-induced alteration to the information stored will persist until an external circuitry refreshes the signal; this makes DRAMs especially prone to SEUs. Usually a particle strike in or near the storage capacitor or the source of the transistor or a bit-line, as shown in figure 8 [9], is enough to cause an upset. If the charge collected at the node is greater than Q_{crit} and the noise margin, the stored signal is overwritten and is usually observed as a 1 -> 0 transition. 0 -> 1 transitions can also be observed but are usually less likely to occur [9].

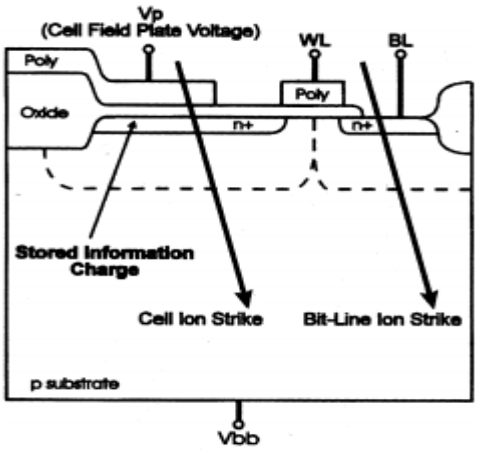


Figure 8 [9]: Ion Strikes to the bit-line and storage cell of a 1T-DRAM

SEUs in SRAMs/Latches/FFs: Unlike DRAMs, SRAMs exhibit data remanence, meaning their storage signals do not need to be refreshed periodically. This is because SRAMs use a bistable latching circuitry to store bit information [39]. Figure 9 [40] shows a typical SRAM with a back to back inverter configuration used to regenerate its signals. A particle strike on any of the nodes may cause the node to transition. If this transient glitch propagates through the inverters, it causes the wrong value to be latched. When this happens, external circuitry is needed to rewrite the nodal value [5]. Latches also contain a regenerative path similar to SRAMs, thus latch-induced SEUs are largely based on the same principles as SRAM-induced SEUs. Flip-flops typically consist of two latches in a master-slave setup shown in figure 9, thus radiation particle strikes on FFs have similar effects as in latches and SRAMs.

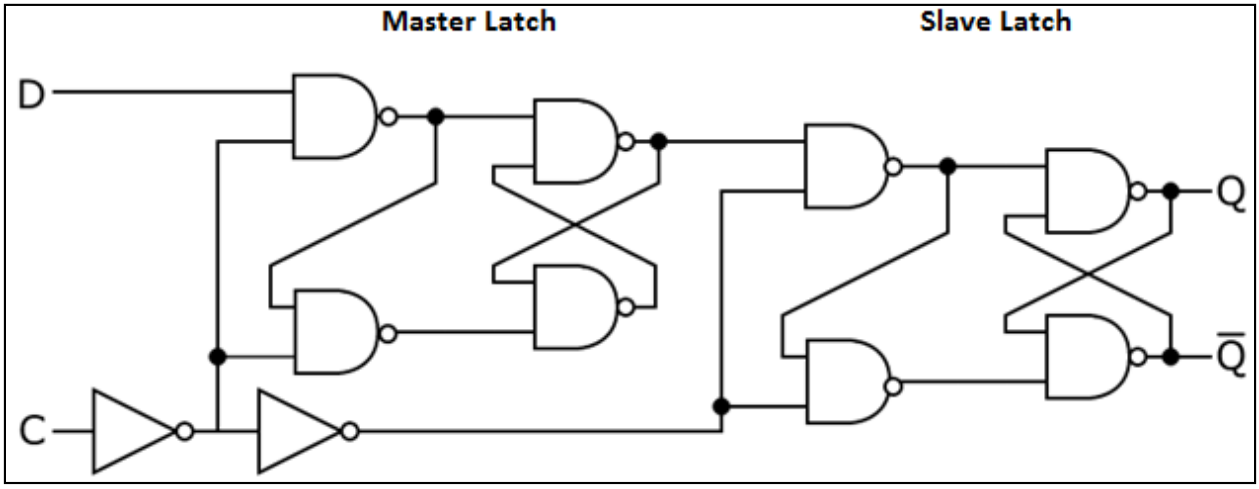


Figure 9 [40]: Master-Slave D Flip-Flop arrangement

Combinational Logic Induced SEUs

As in memory elements, charge collection occurs at a combinational node affected by a radiation particle strike. If the collected charge exceeds the Q_{crit}, a 100 to 200 picosecond (ps) wide transient voltage glitch is generated. The transient voltage may propagate through combinational

elements and be latched by a memory element as seen in figure 10 [41]. However, three masking effects present in digital circuits generally prevent transients from becoming SEUs [5, 9, 17, 41]. These effects are illustrated in figure 11 [41].

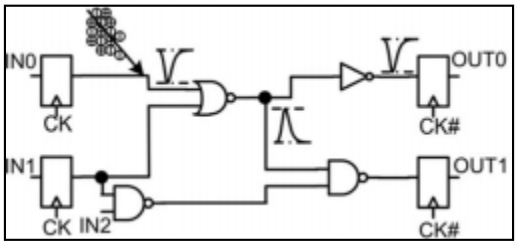


Figure 10 [41]: SET propagating through a circuit

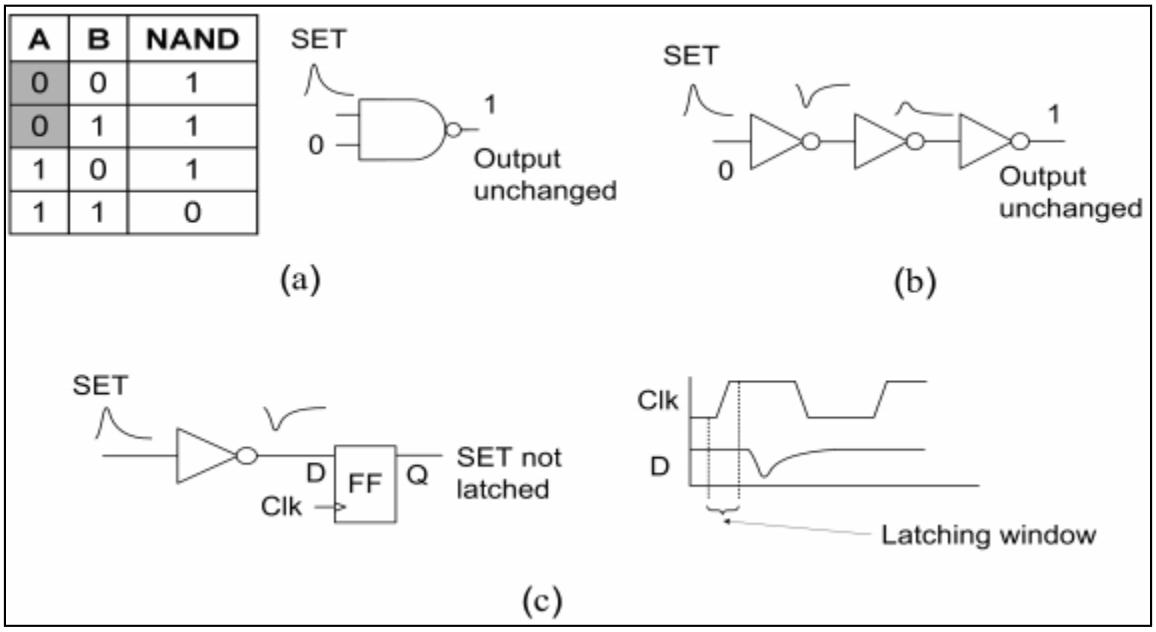


Figure 11 [41]: Different SET masking effects

Logical Masking: The logical masking effect can be described with the NAND gate seen in figure 11(a). If a particle strikes one of the input nodes of the NAND gate, but the other input remains in the controlling state (0 in this case), the output of the gate will not change and the SET will be completely masked. For an SET to propagate through a combinational logic element, a

sensitive path must exist from the affected node to the output of the logic element.

Electrical Masking: All CMOS circuits have limited bandwidths. SETs with bandwidths higher than the cut-off frequency of a circuit will be attenuated. This causes the amplitude of the SET to reduce, the rise and fall times to increase, and, eventually, the pulse to disappear as it propagates through logic elements as seen in figure 11(b).

Latching Window Masking: This effect, also known as Temporal Masking, can be described with figure 11(c). As the SET moves towards the D input node of the FF, it might show up outside the latching window of the FF preventing it from being latched, thereby preventing an SEU from occurring. In FFs and latches, setup and hold time requirements make up the latching window. As the operating frequency for subsequent technology nodes increases, the latching window decreases due to a resulting decrease in setup and hold time requirements, thus the effect of latching window masking decreases, effectively causing an increase the SEU rate.

Contribution of Memory Elements and Combinational Logic to SEU rate

We have determined the mechanisms through which combinational logic and memory elements generate SEUs, however it is important to understand the relative contribution of combinational and sequential elements to be able to choose a suitable protection technique against SEUs. Conventional wisdom suggests that the SEU rate of CMOS designs is largely dominated by sequential elements for low frequency circuits primarily because the three masking effects observed in combinational logic often prevent SETs from being latched by a memory element. Another reason for the domination of sequential element SEUs is that memory elements usually have a lower Q_{crit} than combinational elements, thus a radiation particle is more likely to cause an SEU in a sequential circuit than in a combinational circuit. However, the distinction between the relative contribution of

memory elements and combinational logic to SEU rate is not always as clear-cut as implied. The combinational logic area is typically greater than the sequential area; this makes combinational elements more likely to be struck by radiation particles. Therefore, it is possible that increasing the combinational logic area exponentially can cause combinational elements to dominate SEU rate.

Research studies have long proven that the SEU rate of ICs is frequency dependent. Radiation effects experts postulate that as the operating frequency of CMOS devices continues to increase, an increase in SEU rate is observed [1, 3, 4-7, 21-22, 42]. Figure 12 [22] corroborates this hypothesis in a plot of error rate vs. frequency. In this figure, we observe that for low frequency applications, sequential logic errors dominate combinational logic errors. However, as sequential logic error rate is relatively independent of frequency, combinational logic error rate dominates for high frequency ICs, increasing the overall error rate (represented by "sum" in the figure) in the process. It should be noted that SEU rate is directly related to error rate; an increase in SEU rate consequentially means an increase in error rate, as will be proven in the next section, thus our utilization of figure 12 is justifiable.

Much of this increase in combinational logic induced SEUs can be attributed to a weakening of the latching window masking effect at high frequencies. Since set-up and hold time, which typically makes up the latching window, has to be less than clock frequency, latching window decreases with an increase in clock frequency. Another reason for combinational logic error domination at high frequencies is that as transistors switch faster, the effects of electrical masking diminishes. For the aforementioned reasons, it has become quite essential to protect high operating frequencies ICs against radiation particles. In particular, techniques for protecting combinational logic elements are quite invaluable.

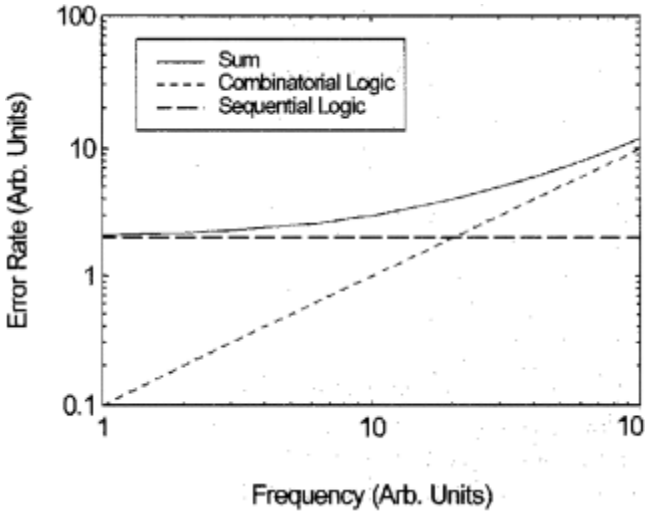


Figure 12 [22]: Error rate dependence on frequency

Soft Errors & Architectural Vulnerability

Not all SEUs propagate to the final output(s) of a design. The masking effects highlighted in previous sections continue to have an effect on SEU signals as they propagate through a circuit preventing most of the signals from making it to the output even after being latched by a memory element. Thus, the probability of an SEU showing up at the output of a design is largely dependent on the length of the circuit path between the initially affected node and the final output node. An SEU that shows up at the primary output of a design is known as a soft error. Soft error rate (SER) represents the rate at which a device experiences soft errors. It is usually measured in terms of mean time to failures (MTTF) and failures-in-time (FIT). MTTF is defined as the average time elapsed between failures in a system and is expressed in years of device operation. FIT is the reciprocal of MTTF and equivalent to 1 error per 10^9 hours of device operation.

For soft faults to be noticed, they have to cause an observable error in program execution otherwise they are not considered when quantifying architectural vulnerability. Architectural

Vulnerability Factor (AVF) is the metric used to compute the unreliability of architecture in correctly executing programs. If a soft fault occurs on a node that has no effect on the current program execution, such as a tri-stated node, the soft fault does not cause a visible error in the program execution and is not included in AVF calculations. Also, a fault that occurs in a branch predictor will go unnoticed. Soft faults that occur on control bits are especially harmful as they can affect the flow of program execution. For instance, if the control bit for a RISC instruction becomes corrupted, the wrong instruction might be executed.

CHAPTER III

SER REDUCTION AND LOGIC REPAIR

To counter the harmful effect of radiation particles on devices, several techniques aimed at reducing SER have been proposed. The obvious solution is to eliminate the sources of radiation particles. While extraterrestrial cosmic ray generating phenomena cannot be eradicated, other controllable sources of radiation particle emissions can be purged. For instance, IC manufacturers use very high purity materials and processes to reduce alpha particle emissions. Another way to decrease alpha particle emissions is to separate sensitive nodes of a circuit from materials that emit alpha particles. Also, coating chips with a thick polymide layer helps reduce the effects of alpha emissions. However, these methods typically only reduce the effect of radiation particles found in IC packaging materials and are ineffective against cosmic ray particles from space.

Process techniques for mitigating SER have also been researched. Some researchers advocate reducing the depth of charge collection by using specific doping profiles or substrate structures. Charge collection can also be reduced by using multiple-well isolation. Well based mitigation technologies as well as silicon on insulator (SOI) substrate structures have also been suggested for logic circuits. In general, applying any of these process techniques typically yields a minimal reduction in SER, at the expense of increased manufacturing cost.

Another way to reduce SER is by increasing the Q_{crit} of a manufactured design while maintaining or decreasing the collected charge Q_{coll}. For instance, in a 6T-SRAM, the Q_{crit} at the storage node is dependent on the node capacitance and the voltage as well as the restoring charge supplied by the pull-up/pull-down transistors. The restoring charge is in turn dependent on cell's

switching frequency, and the load transistor current. By decreasing the SRAM's switching frequency or increasing the strength of the load transistor's current drive, the SER rate can be reduced considerably. However decreasing the switching frequency is unacceptable for high frequency designs. Under normal circumstances, this technique is typically avoided as the area, power and speed penalties incurred can be astronomical.

Another technique that has been largely explored and proven effective is to include extra circuitry for error detection and correction. Error detection is typically achieved by using a parity bit to store the parity of each data word. Upon retrieving the data word, an algorithm compares the parity of the data obtained with the stored parity bit. A mismatch signifies that an error has occurred. This technique is particularly cost efficient as only one additional bit is needed for error detection. However, it is usually more useful to correct errors rather than just detect them, thus error detection and correction techniques are more suitable for application. Single error correction can be achieved by adding extra parity bits to each data word and encoding the data such that the information distance is 3. For a 64-bit-wide memory, 8 bits are required for single error correction. Since most soft errors are single bit errors, EDAC protection reduces SET rate drastically. However, implementation of EDAC introduces design complexity, additional memory, and increased latency. Also, EDAC is typically most suitable for memory applications and not for logic elements. [1, 6, 9]

Logic repair typically refers to the practice of fixing IC logic failures. In a sense, logic SER reduction techniques are all forms of logic repair, however logic repair is usually used to mean correction techniques that utilize some level of redundancy as shown in figure 15. As the transistor density continues to increase for new technology nodes, the probability of transistor failures increases. It only takes a few transistors to radically alter the functionality of an IC. When this happens, it is beneficial to have some form of redundancy to replace the failing IC. At the lowest

level, each individual component may be duplicated. If the failing transistors in an IC can be identified, the device can be redesigned for future use. However, since it is impossible to replace individual transistors in an IC of over 1 billion transistors, it is necessary to consider redundancy at a higher level. At a higher level, sub-circuits that house the faulty transistors may be replaced, as it is much easier to identity failing hardware modules. At the highest level, an entire system may be replaced; however this is a cost inefficient and is usually avoided. Triple modular redundancy (TMR), a widely used form of N-modular redundancy, involves triplicating a hardware module, typically a logic block, and feeding each of the three outputs into a voter circuit to produce a final output as shown in figure 13. A typical voter circuit such as figure 14 ensures that single bit errors are masked from the final output. For instance, if we assume that A is corrupted and flips from a correct value of 1 to a faulty value of 0, B and C should not be affected and should still have a value of 1, resulting in a final output value of 1. Multiple bit errors are also masked provided that they do not occur at the same node in the three circuits. TMR systems are very efficient and typically reduce SER rate to 0 so long as the voter circuit is heavily protected from radiation strikes, otherwise an error in the voter circuit will corrupt the final output. As with other logic repair and SER reduction techniques, TMR implementation introduces increased area (>200 %), speed, and power penalties.

Memory ICs can also implement robustness by having spare memory rows/columns that can be switched into a design to replace faulty memory rows/columns. It is usually more difficult to replace entire memory cells as a large design overhead is incurred, thus designers replace entire rows/columns. As memory blocks tend to have similar cells, it is exponentially easier to identify and replace faulty rows and columns in memory ICs than in combinational logic.

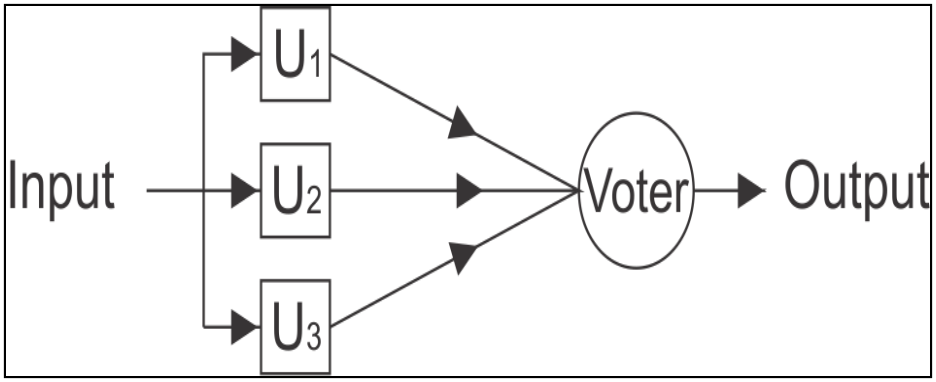


Figure 13: Triple Modular Redundancy

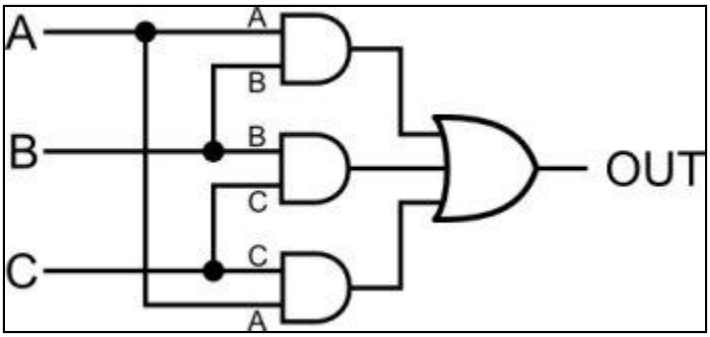


Figure 14: Voter Circuit

Due to the penalties associated with implementing logic repair and SER reduction techniques, it is imperative to provide designers with flexibility in selecting the protection level of ICs while balancing out design trade-offs. Furthermore, most of the currently available modi operandi for implementing hardware robustness apply to, or have been optimized for, memory elements, thereby presenting a dire need for strategies that target combinational logic. This thesis presents a technique that addresses the aforementioned issues by using approximate logic functions to take advantage of the logical masking effect discussed in Chapter II.

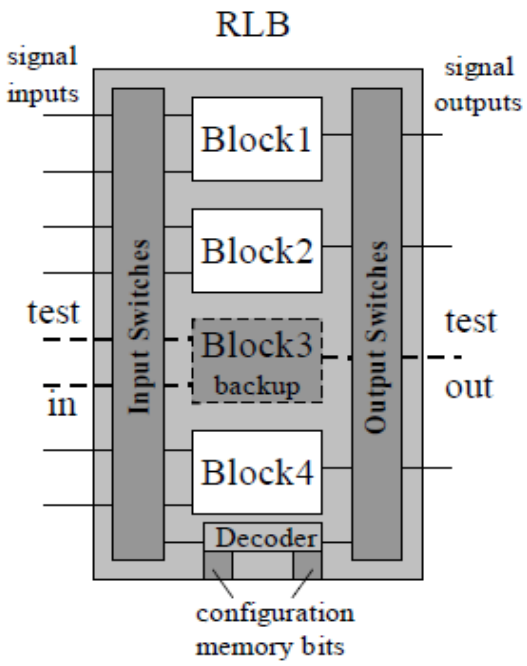
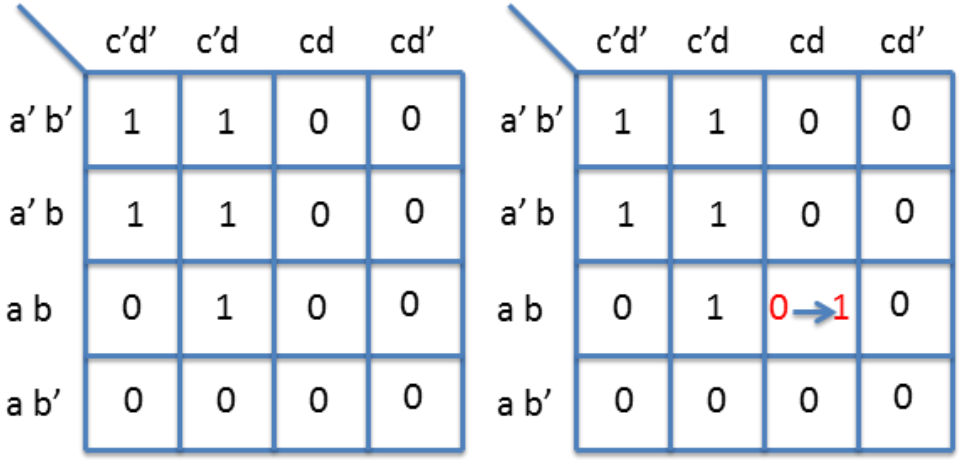


Figure 15: Conventional configuration for logic repair in digital circuits

CHAPTER IV

LOGIC REPAIR AND SER REDUCTION USING APPROXIMATE LOGIC FUNCTIONS

Combinational circuits are designed from Boolean functions which are made up of minterms/maxterms and represented by truth tables. Minterms represent product terms in which each of the input variables appear once, while maxterms represent sum terms in which each of the input variables appear once. Based on the corresponding minterms or maxterms, Boolean functions are minimized to reduce the overall area of the resulting combinational circuit implementation. This minimization is achieved using Karnaugh Maps (K-map). Essentially, when the input combinations represented by 2^n minterms (or maxterms) are sufficiently close enough, meaning they only differ by 1 variable, the function can be minimized. When such a circuit is struck by radiation particles and produces a soft error in form of a bit flip in the final output, the original Boolean function represented by the combinational circuit changes, for as long as the error persists on the output. Figure 16 illustrates such a scenario with a K-map representing a four-variable Boolean function.



(a) (b)

Figure 16: K-maps representing a Boolean function in (a) the original state (b) the error

corruption state with an upset minterm

The Boolean equation for the function represented in the K-map in figure 16(a) can be given as

$$F = a'c' + bc'd \quad \ldots (1)$$

and it represents the original circuit in an unperturbed state. Assume a transistor in the circuit malfunctions due to a device level phenomenon or radiation particle strike modifying the output from logic 1 to logic 0 for one of the minterms. For instance, consider the case in figure 16(b) where the minterm abcd becomes faulty and changes from logic 0 to logic 1, the circuit implementation implements the Boolean equation in (2), which is radically dissimilar to the

$$F = a'c' + abd \quad \ldots (2)$$

original function and causes the circuit to malfunction, possibly affecting the entire system. FPGA designs are particularly susceptible to such errors, as they allow circuit implementations to be reprogrammed, thus a malfunction might be misjudged by the FPGA as a reprogramming.

Due to the perceived difficulty in repairing combinational circuits, research publications in the area of robustness are often deplete of strategies for combinational logic repair. Unlike memory ICs that are extensively composed of identical sub-blocks, combinational logic circuits rarely utilize similar blocks. Furthermore, any transistor in a circuit may be the cause of an IC failure, thus it is impossible to predict the type of failure that will occur. For instance, any of the minterms or maxterms in figure 16(b) that produce an output of logic 0 might become logic 1. As it is impossible to determine before-hand how many minterms will fail, many researchers assume that the likelihood of developing efficient logic repair strategies is low.

Single-Output Logic Repair

Fundamentally speaking, when an IC design experiences a fault, only the failing minterms

need to be replaced as the other unaffected minterms will continue to function properly unless they also encounter faults. Devising strategies to hinder failed minterm(s) from propagating to the final output of a design is paramount to efficient logic repair. As in the logical masking effect discussed in Chapter II, failed minterms can be prevented from corrupting the output of a design by desensitizing logical paths that correspond to the affiliated minterm. This is achievable by integrating logic repair circuits with the original design using novel hardware techniques. Therefore, once a failure is detected, the logic path from the transistor failure site to the output of the design needs to be desensitized and rerouted with a properly functioning path. TMR is an extreme form of logic repair via logical masking as it requires that the entire logic block be reproduced.

This approach enables designers to implement hardware robustness for combinational circuits, without having to replace entire logic blocks, by altering the intended path to output of the failed minterm and replacing it with a functionally accurate logical path that yields the correct output. This error detection and replacement method is achieved by adding extra detection and correction circuits whose sizes are largely dependent on the size of the original logic circuit. In a nutshell, this strategy is based on utilizing Boolean functions that represent under and over approximations of the original logic circuit. Considering the Venn diagram in figure 17 [3], assume that G is a function that represents a logic circuit and contains the entire input space for the combinational circuit, and that the rectangular box represents the entire input space and contains 2^n set elements for n input variables. In this case, the area inside the G circle represents the input space for which the output of the original function is logic 1 (the members of this input space are the minterms for function G), and the area outside the G circle represents the input space for which the output of the original function is logic 0 (the members of this input space are the maxterms for function G). For this function G, the function F represents an under-approximation of G, and the

function H represents an over-approximation of G such that $F \leq G \leq H$. In other words, if an input vector $\vec{x}$ is a minterm of G, it must also be a minterm of H. Furthermore, any maxterm of H is also contained in G. This ensures that F only evaluates to 1 when G evaluates to 1, while H only evaluates to 0 when G evaluates to 0. By using strong F and H approximate functions, F and H will only differ from G for a small number of inputs. In a nutshell, approximate functions F and H are incomplete, additional sets of minterms and maxterms that can be used to replace any failing minterms/maxterms.

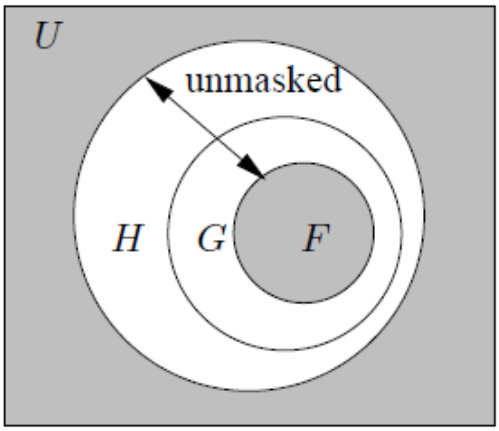


Figure 17 [3]: Over-approximation (H) and under-approximation (F) for an original function (G)

This strategy allows us to accurately assert the value of the minterms of G (based on the minterms of F), and the maxterms of G (based on the maxterms of H) ensuring that we can accurately determine the logic value of G. Even in the presence of an error on a minterm in G (as in a transistor failure in the original logic circuit that results in a 1 -> 0 output bit flip), F should contain a minterm that effectively logically masks out this error. Likewise, maxterms in H can be used to mask any maxterm errors in G. Under-approximation functions may be chosen from one of the large cubes that exist in the original design, while over-approximation functions may be chosen by

expanding a subset of minterms of the original design to form a larger cube. Since cubes contain fewer literals than original functions, using large cubes as approximate functions is an excellent way to minimize the area overhead of the additional logic repair circuitry. This will be discussed in more depth in Chapter V and corroborated with results in Chapter VI.

The effectiveness of the approximate functions used in masking errors is dependent on the number of minterms and maxterms F and H contain, thus if F and H contain most of the minterms and maxterms of G, then the failures in G can be tolerated more efficiently. In fact, failures only propagate to the output if F and H loosely bound G for the affected minterm or maxterm. Thus, for a protection circuit to be efficient, F U G and F U H must be very large sets. TMR is essentially an extreme example of this approach where approximate functions are strongest, guaranteeing that ever minterm/maxterm error will be masked. Weakening the approximations used in TMR decreases the associated penalties, while ensuring a reasonable amount of protection.

As in TMR, a decider circuit (voter circuit in TMR) is required to detect and correct errors masked by F and H. Figure 18 presents an and-or structure that essentially decides what the correct output should be.

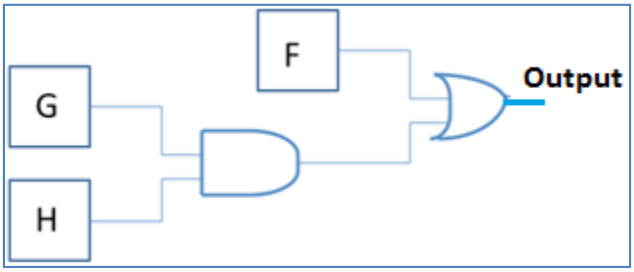


Figure 18: And-Or decider circuit

F and H will mask any error in G provided that the affected minterm/maxterm is not an element in the set represented by the unmasked region shown in figure 17. Errors are unmasked only when F

evaluates to logic 0, G evaluates to logic 1, and H evaluates to logic 1. Based on the and-or structure in figure 18, when this happens, an error on node G that causes it to evaluate to logic 0 will result in an output of logic 0 which is considered an error as the correct value of G is a logic 1. Thus it is crucial to minimize the unmasked area as much as possible.

It is necessary to introduce a new term - repair coverage - to quantify what proportion of a logic circuit is covered against failures. In other words, a logic circuit will not have a repair-coverage of 100% unless all its minterms and maxterms are protected from errors. Considering the circuit represented in figure 16(a), we notice that the Boolean equations F = a'c' and H = c (a'+b') can be used as repair circuits. Since F covers 4 minterms of G, and H covers 10 maxterms of G, a total of 14 out of 16 minterms and maxterms are covered. The failed maxterm shown in figure 3 will be covered by H resulting in a correct output. This approach is more advantageous than the conventional block replacement strategy in figure 15 due to the flexibility provided to the designer. Typically, the strength of approximate functions F and H is also a function of area and power tradeoffs thus by slightly relaxing repair-coverage requirements, repair circuits of negligible area can be generated. Thus, this approach can be applied to all functional blocks without overly compromising design tradeoffs.

Computing Repair-Coverage Factor

The repair capacity of a circuit, and the resulting complexity of F and H designs, depends on the coverage of G provided by F and H. As there are many approximate F and H functions for every given logic circuit, it is important to evaluate all these functions to choose the most appropriate approximation that will maximize repair coverage and minimize design penalties. Repair-Coverage is the premier factor for evaluating approximate functions. The repair-coverage factor is basically the

number of minterms and maxterms of G masked by F and H. ||F|| is a notation used to express the number of minterms of function F. F and $\bar{H}$ are necessarily disjoint as illustrated by the gray regions in figure 17. Thus, the repair-coverage factor is the number of minterms contained in the gray portion divided by the entire input space represented by the rectangle, and is given by

$$\gamma = \frac{(||F||+||\bar{H}||)}{2^{n}} \quad \text{.... (3)}$$

where n represents the number of variables/inputs. For example, if $\gamma = 0.65$, the approximate functions can repair 65% of the possible minterms/maxterm failures. In the degenerate case of the strongest approximations, $\gamma = 1$ indicates that any failure is covered. However, the weakest approximation, $\gamma = 0$ indicates that no errors are masked and all failures propagate to the output. In real life fault injection experiments, the observed repair-coverage is likely to slightly different from the calculated estimate for several reasons. For one, radiation particle strikes can also impact F and H circuit implementations, thus they have to be accounted for. In this case, G becomes a repair circuit and since G represents the correct circuit implementation, the repair-coverage factor is slightly increased [3].

Multiple-Output Logic Repair

Since combinational logic blocks typical possess multiple outputs, or multi-bit outputs, it is important to modify the single-output logic repair strategy presented in the previous section for application in multiple-output logic blocks. Many of the techniques for logic repair fail to account for this and protect individual outputs selectively. However, for a combinational block to be considered fully protected from errors, every output needs to be protected for any input combination error. It is no use protecting one output and leaving the other output unprotected even if the protected output is more prone to soft errors. If the probability that the unprotected output produces a soft error

is non-zero, the overall vulnerability of the block increases. For this reason, it is important to protect outputs with respect to other outputs. Three methods are presented for application in multiple-output combinational blocks.

Non-Optimized Method

This is by far the crudest of the three methods to be explored. In essence, this method is an extrapolation of the single output repair technique. For each output, the single output technique is applied to generate F and H approximate functions. The F and H circuits can then be synthesized together to decrease the overall area increase. While this method is the easiest to implement, it is also the least effective for the following reasons. Assume that the two different outputs of the combinational block in figure 19 are represented by the K-maps in figure 20.

Figure 19: Black-box representation of a combinational circuit

	c'd'	c'd	cd	cd'
a'b'	0	0	1	1
a'b	0	0	0	1
ab	0	0	1	1
ab'	0	0	1	1

G(0)

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	0	0	0
ab	0	0	0	0
ab'	0	1	1	0

G(1)

Figure 20: K-maps representing two outputs of a logic block

The output functions can be split into minterm and maxterm functions and written as:

$$G(0)_{min} = ac + cd' + a'b'c, \quad G(0)_{max} = c\,(a+b'+d') \qquad \ldots. (4)$$

$$G(1)_{min} = a'b' + ab'd, \qquad G(1)_{max} = b'\,(a'+d) \qquad \ldots. (5)$$

Possible approximate functions for output G(0) and G(1) are given in figure 21 and 22 respectively. F(0), seen in figure 21, provides protection for 4 minterms of G(0) - abcd, abcd', ab'cd, and ab'cd' - and is represented with the function F(0) = ac, while H(0) covers 8 maxterms of G(0) and is represented with the function H(0) = c. Thus, a total of 12 out of 16 minterms/maxterms are protected by F(0) and H(0), resulting in a repair-coverage factor of 0.75.

	c'd'	c'd	cd	cd'
a'b'	0	0	0	0
a'b	0	0	0	0
ab	0	0	1	1
ab'	0	0	1	1

F(0)

	c'd'	c'd	cd	cd'
a'b'	0	0	1	1
a'b	0	0	1	1
ab	0	0	1	1
ab'	0	0	1	1

H(0)

Figure 21: K-maps representing approximate functions for G(0)

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	0	0	0
ab	0	0	0	0
ab'	0	0	0	0

F(1)

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	0	0	0
ab	0	0	0	0
ab'	1	1	1	1

H(1)

Figure 22: K-maps representing approximate functions for G(1)

The K-maps in figure 22 represent the approximate functions, F(1) and H(1), for the function G(1). F(1) protects 4 of the 6 minterms of G(1) and can be represented by the equation F(1) = a'b', while H(1) provides protection for 8 of 10 maxterms of G(1) and can be represented by the equation H(1) = b'. Thus, a total of 12 out of 16 terms (minterm/maxterm) of G(1) are protected by F(1) and H(1), resulting in a repair-coverage factor of 0.75. Imagine that a fault occurs when the input abcd to the combinational block is 1001 (corresponds to minterm ab'c'd and maxterm (a'+b+c+d')), propagates to both outputs G(0) and G(1), and causes G(0) to transition from 0 to 1 and G(1) to transition from 1 to 0. If this happens, H(0) provides adequate cover for G(0) and the error is masked from the output. However, F(1) fails to mask this error from G(1) thus the fault propagates to the output as an error. Therefore, even though G(0) is protected, G(1) is left exposed, resulting in a combinational block failure. This is the worst pitfall of this method.

For a combinational block to be immune to a certain term error, every output of the block has to mask the error. Thus, repair-coverage of the combinational block is estimated to be the product of the repair-coverage factors of the outputs. This implies that the overall repair-coverage of a logic block is less than the least of the per-output repair-coverage factors. This estimate used for determining the overall repair-coverage factor of a logic block is a tad inaccurate as it assumes that the term space set will be exhausted, whereas in real life, only a few term errors might occur. In addition, the repair-coverage factor of a block is very much dependent on the common protected terms between the approximate functions used for the outputs. For example, if the approximate functions of G(0) protect a certain term, and the approximate functions of G(1) also protect that term, then the combinational block is protected from that term error. This estimation formula is more accurate when the per-output repair-coverage factors are extremes, 0 or 1, and is slightly less accurate for non-extreme per-output repair-coverage factors. Despite the expected shortcomings of

this estimation formula, simulation results discussed in Chapter VI have proven that the formula is sufficient enough to serve as a baseline for quantifying the efficiency of the non-optimized method, as this method consistently underperforms the other two methods to be discussed. Based on the estimation formula, it can be implied that the repair-coverage factor is dependent on the number of outputs; an increase in the number of outputs results in a decrease in the repair-coverage of the logic block. Thus, the non-optimal method of logical masking is likely to be impractical for microprocessors and other IC blocks that often have multiple outputs and multiple-bit outputs.

Shared Minterm Method

To address the many issues that arise from the non-optimized method, it is essential to provide designers with other logic repair strategies that effectively utilize approximate functions. One of such methods, the shared minterm method, requires collectively examining the multiple outputs of a logic block, to determine which outputs share minterms, and which of the possible shared minterm combinations is most appropriate for implementation. The basic idea behind this method is that outputs with shared minterms expose minterms that have to be protected. For example, consider the K-maps in figure 23.

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	0	0	1
ab	0	0	1	1
ab'	0	0	1	1

G(0)

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	0	0	0
ab	0	0	0	0
ab'	0	1	1	0

G(1)

Figure 23: Outputs of a logic block represented with K-maps

On the basis that cubes are used for approximate functions to reduce area, there are three major candidates for F(0) – a'b', cd', and ac -, and two major candidates for F(1) – a'b', and ab'd. The cube a'b' represents the shared minterms between G(0) and G(1). Thus, using the shared minterm method, F(0) = F(1) = a'b'. The benefit of picking the shared minterm cube as the approximate function for both outputs is quite obvious. If instead F(0) = cd', and F(1) = a'b', then errors that occur on G(0) and G(1) when a = 0 and b = 0 (minterm a'b') will most likely cause a failure in the logic block. This is because for this input combination, G(1) will always be protected while G(0) will only be protected for 1 of the 4 possible input combinations, thus an error can occur 75% of the time even though the repair-coverage of G(0) does not change irrespective of which one of the three F(0) candidate functions is used. The repair-coverage of a block that utilizes this method cannot be lower than the percentage of terms masked by the shared minterms. Thus, this method is more beneficial for logic blocks with outputs that share many minterms.

There are some difficulties in implementing this method. The major stumbling block is the fact that brute-force search might be required to determine the best shared minterms to use. For a combinational block of n outputs, the time complexity of this algorithm is exponential - $O(2^n)$, as every possible output combination has to be considered. In general, $(2^n - n - 1)$ combinations are examined for shared minterms. We subtract n and 1 from the total possible combination of 2^n to eliminate the single outputs and no outputs "combinations" respectively. This exponential time complexity is very poor, especially for an application that is likely to have many outputs, therefore it is imperative to improve on this algorithm. This can be achieved by providing more hardware computation resources in the form of multi-processing units and multithreading the application program that implements this scheme. By dividing up the tasks into multiple processing units, the computation process can be speed up in accordance with Amdahl's Law. The time complexity can

also be reduced by eliminating output combinations that are not expected to share any minterms based on previous computations. To achieve this, output combinations need to be processed in the order of least number of outputs. For instance, if a block has 5 outputs – a, b, c, d, e, 2-output-combinations should be examined first to determine if any of such combinations is sparse for shared minterms. If, for example, it is determined that the combination of outputs a and b produce 0 shared minterms, then 7 output combinations - abc, abd, abe, abcd, abce, abde, abcde – can be eliminated from the original set of 26 (2^5-5-1) output combinations. The number of eliminated output combinations is given by the equation:

$$¥ = 2^{(n-X)} - 1 \quad \text{…. (6)}$$

Where n is the total number of outputs, and X is the number of outputs in the no-shared-minterm combination. We can verify equation (6) with the above example where $¥ = 2^{(5-2)} - 1 = 7$. The new equation for calculating the total number of output combinations is derived below:

$$Ô = 2^n - n - 1 - \sum_{k=2}^{n} ¥ * v_k - L\ [v_k \neq 0] \quad \text{…. (7)}$$

$$Ô = 2^n - n - 1 - \sum_{k=2}^{n} (2^{n-k}\text{-}1\text{-}m)\ v_k - L\ [v_k \neq 0] \quad \text{…. (8)}$$

In the above equations, v_k represents the number of distinct output combinations with vector size k that contain no shared minterms, m represents the number of output combinations that have been eliminated due to previously evaluated output combinations, and L represents the number of overlapping output combinations to be eliminated for output combinations with vector size k. This guarantees that for each k, the output combination eliminations will be distinct. Take the above example for instance. Assume that when k = 3, the output combination bcd was discovered to contain no shared minterms. Thus, 3 output combinations – bcd, abcd, abcde – are to be eliminated. However, since abcd and abcde have already been eliminated, care must be taken to ensure that they are not double counted as output combinations can only be eliminated once.

By eliminating output combinations that are not expected to contain shared minterms, the computational time can be reduced drastically, provided that the minterms of the outputs are not strongly correlated. Nonetheless, while loose correlation might be good for area and computational time, it reduces the number of candidates that could be selected hereby essentially resulting in a lower repair-coverage factor. Also, additional memory requirements are imposed on a designer for storing the results of previous output combination evaluations. Additionally, the efficiency of output-combination-elimination decreases as the number of outputs increases.

After generating the shared minterms for every output combination, the most effective output combinations have to be decided upon. This is not a trivial affair as there are often many possibilities to choose from. Generally speaking, the ideal candidate for a given combinational block is any output combination that contains the most outputs and the most shared minterms. This ensures that the same output combination can be utilized for multiple outputs. Since ideal candidates rarely exist, output combinations that balance out the number of outputs they contain with the number of shared minterms they protect are usually chosen. The problem of determining which output combinations will produce the optimal solution is NP-hard. It should be noted that additional terms can be added to the shared minterm as this can slightly increase the repair-coverage factor at the expense of additional area.

Bestrc Method

An additional method was considered for logic repair to address the inadequacies of the first two methods. This method requires obtaining the best approximate functions (one F function and one H function), based on the single-output logic repair method, from the approximate functions chosen for each output. The same F and H functions are then used to determine which output terms will be

protected, such that when the output, from which the approximate functions were chosen, is protected from failures, the other outputs will also be protected. Consider a 4-input-3-output block with inputs a, b, c, d and outputs G(0), G(1), G(2) represented with the K-maps in figure 24.

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	0	1	0	1
ab	0	0	0	0
ab'	0	0	0	0

G(0)

	c'd'	c'd	cd	cd'
a'b'	1	0	0	0
a'b	0	1	0	1
ab	0	1	0	0
ab'	1	0	1	1

G(1)

	c'd'	c'd	cd	cd'
a'b'	0	1	0	1
a'b	0	0	1	0
ab	1	0	0	1
ab'	1	1	0	1

G(2)

Figure 24: K-maps for a 4-input-3-output combinational block

The best approximate functions for this combinational block, a.k.a. bestrc, is F(0) = a'b' and H(0) = a, and is obtained from the output G(0) as it contains the most cubes. The next step is to guarantee that errors on other outputs will be masked for the terms covered by F and H, as shown in figure 25.

	c'd'	c'd	cd	cd'
a'b'	1	0	0	0
a'b	X	X	X	X
ab	0	1	0	0
ab'	1	0	1	1

F(1)&H(1)

	c'd'	c'd	cd	cd'
a'b'	0	1	0	1
a'b	X	X	X	X
ab	1	0	0	1
ab'	1	1	0	1

F(2)&H(2)

Figure 25: Approximate functions for G(1) and G(2) using bestrc method

The approximate functions for G(1) and G(2) are chosen by using the original function and replacing minterms/maxterms that are not covered by F(0) and H(0) with don't care sets. The don't-care terms can be replaced with cubes obtained from G(1) and G(2) or with any other logic values that aid the formation of large cubes.

This method is the often the most efficient in increasing the repair-coverage factor of combinational blocks, as it requires replicating most of the terms of the original function at the expense of increased area requirements. However, if the best approximate functions that are selected do not contain enough terms, the repair-coverage decreases. Nonetheless, this method is always better than the non-optimized methods, again at the expense of increased area. The overall repair-coverage factor of the block cannot be less than the repair-coverage factor of the bestrc.

In Chapter VII, the shared minterm and bestrc methods are combined to generate a theoretical optimal method.

CHAPTER V

IMPLEMENTATION OF THE APPROXIMATE LOGIC FUNCTIONS TECHNIQUE

Software programming is vital to logic repair as it speeds up the process of choosing and simulating repair circuits. Thus, for this project, a software tool was developed to assist in selecting and simulating the logic repair circuits generated from each of the three methods outlined in Chapter IV. This tool was written in C++, in a Linux environment, and it utilizes multiple freely available libraries for software development. The flowchart in figure 26 is a high-level representation of the implementation process.

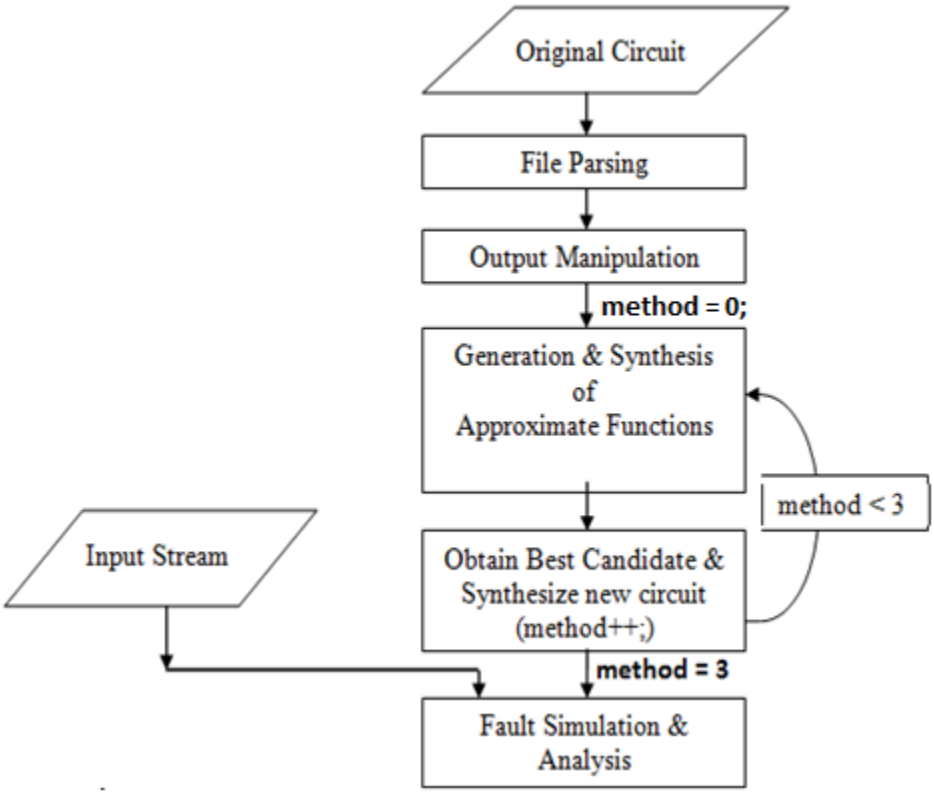


Figure 26: Flow chart of Implementation Process

In summary, the software tool is invoked from the command line of a Linux/UNIX system and a programmable logic array (PLA) filename is encoded in the argument list that the tool receives. The pla file represented by the filename is then parsed to generate an appropriate software representation, and to populate the data structures used by the software program. After parsing the file, the program carries out certain manipulations (more data structure object declarations and populations, and output synthesis) on every output of the combinational block defined in the file. Next, approximate functions are generated for each output based on any one of the three methods outlined in the previous chapter. Subsequently, the best approximate function is selected for each output, from the many possibilities, and synthesized with the original circuit to generate information about the overall area of the new circuit for the particular method. The above steps after output manipulation are repeated until all three methods have been exercised. After the best approximate functions have been selected for each method per output and synthesized to generate three new circuits (one for each method), fault simulation and analysis is performed on each of the three circuits to accurately quantify the actual repair-coverage factor and compare it with the theoretical repair-coverage proposed in Chapter IV. Finally, based on area, power, speed, and repair-coverage factor requirements, the designer must decide which of these three circuits is most appropriate for implementation in hardware. The subsequent sub-sections will provide an in-depth description of the input file format, the data structures used in representing combinational logic outputs, the strategy used by the computer program in selecting approximate functions from large cubes, and the fault simulation process.

Input File Format

Inputs files have to conform to the PLA format before they can be accepted by the logic tool.

While the logic repair tool does contain a method stub for Berkeley logic interchange format (BLIF) files, the implementation is still in the inchoate state due to a lack of interest in this feature. However, we do intend to complete the BLIF portion of the tool in the future. To understand the PLA format, it is helpful to consider how a circuit might be implemented in this format. For instance, given the circuit in figure 27 [43], we can generate a PLA description by following these steps [43]. Each step should begin on a new line.

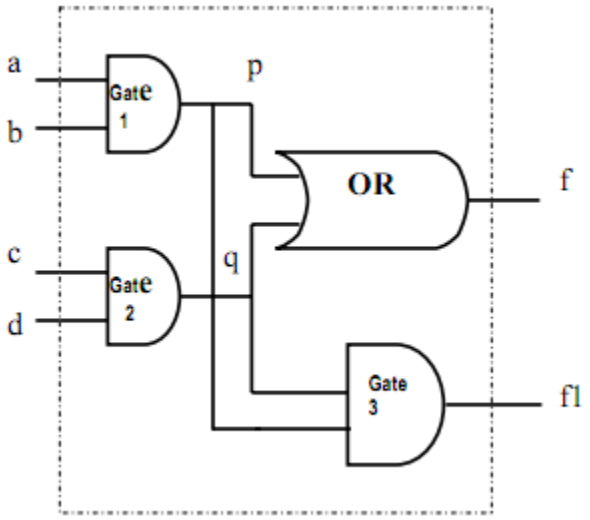


Figure 27 [43]: Logic block to be implemented in PLA format

1) Specify the number of inputs - 4 in this case - using the notation ".i".
2) Specify the number of outputs - 2 in this case - using the notation ".o".
3) Specify the names of the input wires - a, b, c, d in this case - using the notation "ilb".
4) Specify the names of the output wires - f, f1 in this case - using the notation "ilb".
5) Leave a blank line before Step 6
6) Specify the truth table of the circuit. The number of terms in the truth table can be specified using ".p". However, specifying the number of terms is optional. For this example, the original truth table is given in figure 28 [43].

A	B	C	D	F	F1
1	1	-	-	1	0
-	-	1	1	1	0
1	1	1	1	1	1

Figure 28 [43]: Truth table for example function

7) Indicate the end of file using ".e"

Based on the above prescription, the PLA file for the circuit in figure 27 is given in figure 29 [43].

```
.i 4
.o 2
.ilb a b c d
.ob f f1

11-- 10
--11 10
1111 11
.e
```

Figure 29 [43]: Final PLA file for example function

Despite the fact that the modus-operandi for describing hardware modules in software is by using hardware description languages (HDL) like Verilog and VHDL, the PLA format is preferred for this project for several reasons. Firstly, Verilog and VHDL programs are usually dependent on multiple libraries, thus the file space required for the proper execution of such programs tends to occupy more memory than other file formats. Also, the simplicity of the PLA format, make PLA files easily developable, understandable and parsable for use in software programs. Furthermore, logic blocks are normally developed from truth table, and since the PLA format is principally a truth table representation, it is, in some sense, more universal than other hardware representation formats.

Given the choice, designers are likely to favor having to create PLA files over having to learn a more complicated HDL. However if a design already exists in the HDL form, a PLA file can be created by following these steps.

1) Program and run a script to generate the first four lines based on the PLA format, and the input and output information contained in the HDL description. The output of the script execution should be logged into a file.
2) Develop a test bench - in your HDL language of choice - that exercises every input combination. Simulate the design using the generated test bench. Log the test bench inputs and corresponding outputs into the file from step 1 according to the PLA format. In adherence to the PLA format, ensure to leave a blank line before logging.
3) If don't care terms exist in the log file, replace "X" characters with "-" for inputs and "~" for outputs.
4) Append .e to the file, in accordance with the PLA format, and save the updated file with the .pla extension. The resultant PLA file is tenderable to the logic repair tool in its un-minimized form. However, to reduce the overall area obtained from synthesis, any PLA minimization software, such as UC Berkeley's espresso tool located at [44], can be used to eradicate redundant terms that might be present in the PLA file.

File Parsing

There are three primary ways in which useful information can be extracted from an input file and passed on to a software program for execution. The first method requires that the target software program reads in the file directly, using very slow file I/O command calls, and executes the required instructions without requesting a supporting program from the programmer. In some cases, another

program might be written to provide support for the main software program. Typically, the main application program that contains the core execution procedures of the software is developed using a compiled language, or on rare occasions an interpreted language, whereas the supporting program is often implemented using scripting languages. Scripting languages are quite efficient in string handling, which is why they are very used for handling input files, and are almost always interpreted. At this point, it is important to distinguish between compilation and interpretation to fully grasp the third method. Compilation refers to the process of directly translating source code (program) into machine executable code. On the other hand, interpretation refers to the process of having another program (interpreter) execute source code directly. Figure 30 [44] presents a pictorial view of the differences between compilation and interpretation.

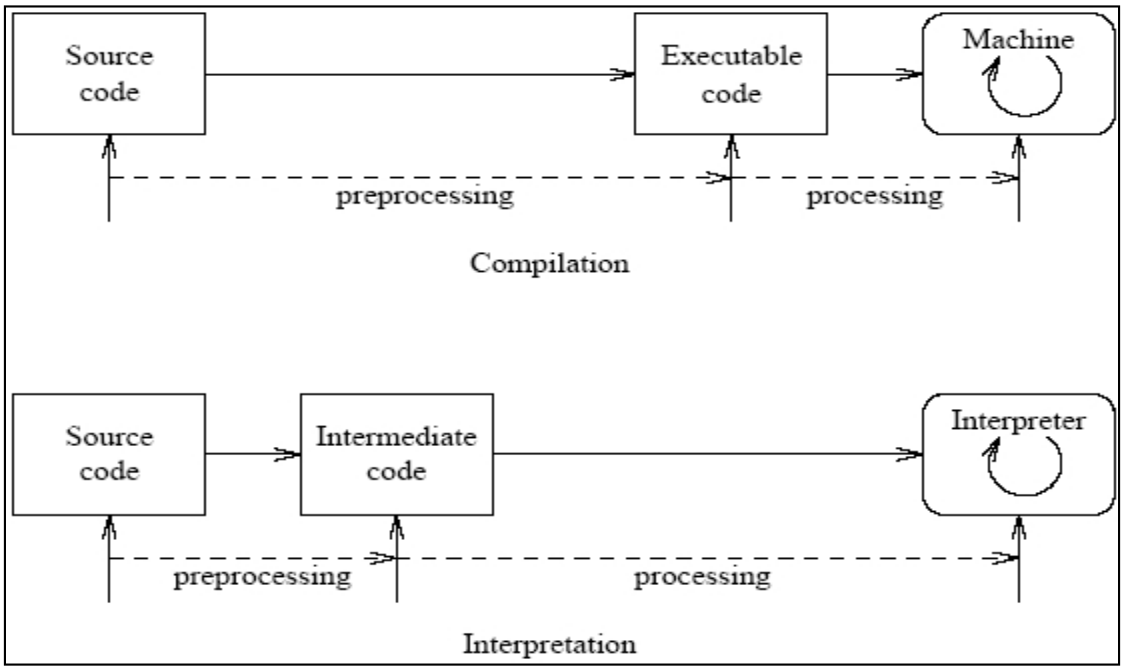


Figure 30 [44]: A comparison between compilation and interpretation

Although compilation requires spending relatively more time on preprocessing the source code, compiled programs execute much faster than interpreted programs (in ratios as high as 10:1), thus the preprocessing step is often justified. The third alternative for extracting information from input files borrows heavily from the preprocessing techniques utilized by compilers. It involves using a lexical analyzer to disintegrate the input file into a collection of tokens and parsing the token collection via a parser that recognizes the tokens. Because file I/O calls are typically slow, and scripted programs are slow as well, the viable option for obtaining information from input files is by using the same preprocessing techniques that cause compiled programs to execute so fast.

Processing Input PLA File

In general, text files are made up of a sequence of characters that conform to a set of rules imposed by a particular formal language. For example, this thesis document is essentially composed of strings of symbols (words) that hopefully conform to the rules (grammar) of English language ¿no? Thus, this document can be represented in a tree-like structure that depicts the grammatical composition of words. Figure 31 [57] shows how the sentence - "John hit the ball." – might be parsed. In the figure, V stands for verb, N stands for nouns, and D stands for determiner.

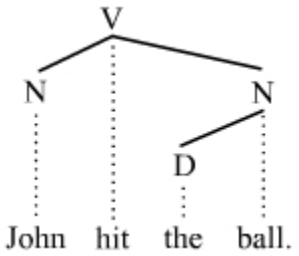


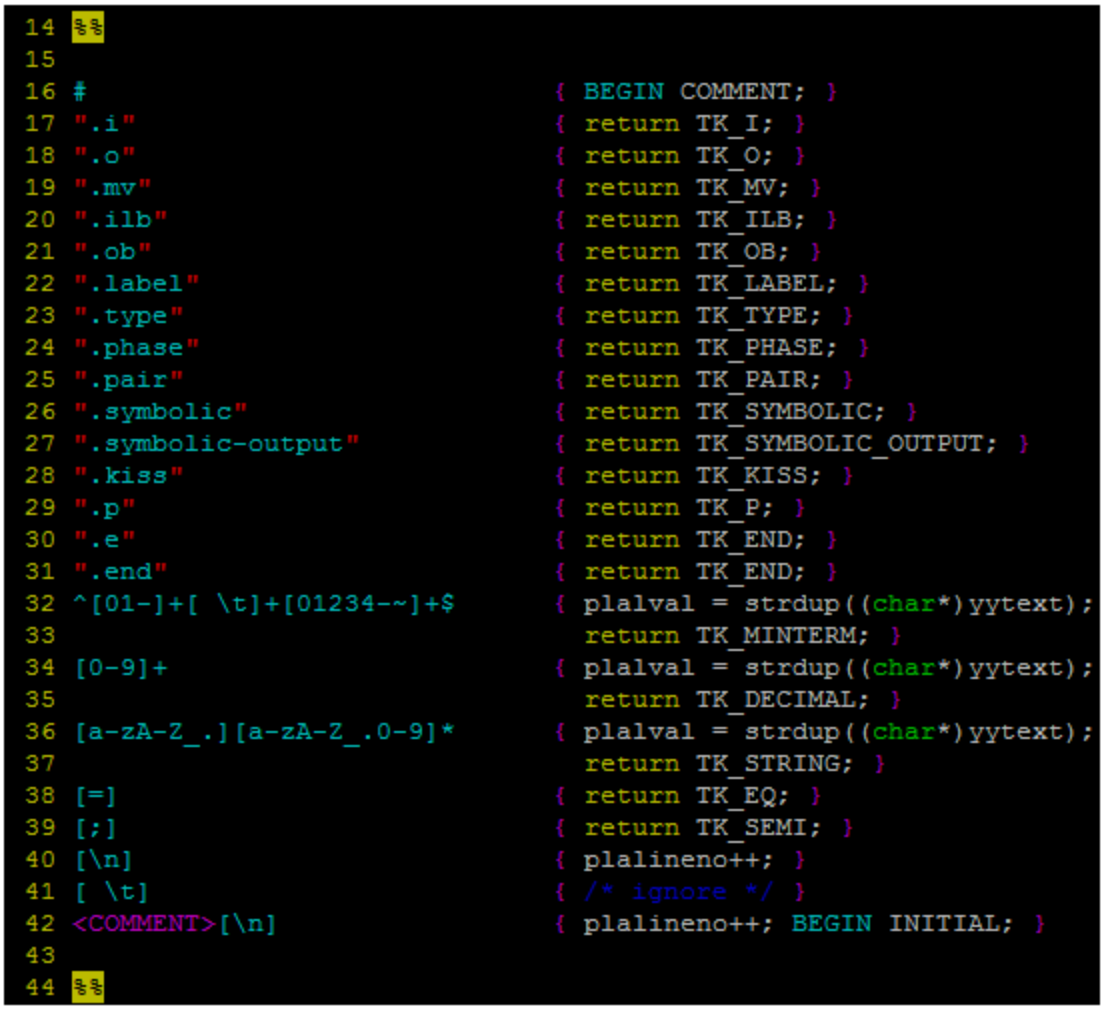
Figure 31 [57]: Sample parse tree

Similarly, PLA files are expected to conform to the rules defined in the previous sub-section and as

such can be depicted as a grammatically accurate composition of symbols. There are two steps involved in extracting information from a file via parsing.

Lexical Analysis: This refers to the process of converting a sequence of characters into a sequence of tokens. A token is a string of characters that is recognizable as a symbol under the rules that describe formal language [45]. For instance, the ".i" notation present in PLA files can be categorized and represented with the symbol INPUT. A lexer is a program that performs the character sequence to token sequence conversion based on a set of rules. Lexers may be programmed manually or generated automatically using custom tools. For this project, we utilize the lex program [46], a tool readily available on UNIX systems, for generating a PLA lexer.

Lex: The Lex program reads a file (.l file extension) that maps string(s) of characters to their associated token symbols and generates C code for a lexer. The lexer matches strings in an input stream, based on the table, and converts the strings to tokens. When the lexer encounters matches in an input stream, it enters them in a symbol table. The symbol table might also contain other information such as the data type and location of the variables in memory. Further references to string identifiers attempt to access the appropriate symbol table index [48]. Figure 32 shows the string characters are mapped in the input file to the lex for PLA format.



```
14 %%
15
16 #                              { BEGIN COMMENT; }
17 ".i"                           { return TK_I; }
18 ".o"                           { return TK_O; }
19 ".mv"                          { return TK_MV; }
20 ".ilb"                         { return TK_ILB; }
21 ".ob"                          { return TK_OB; }
22 ".label"                       { return TK_LABEL; }
23 ".type"                        { return TK_TYPE; }
24 ".phase"                       { return TK_PHASE; }
25 ".pair"                        { return TK_PAIR; }
26 ".symbolic"                    { return TK_SYMBOLIC; }
27 ".symbolic-output"             { return TK_SYMBOLIC_OUTPUT; }
28 ".kiss"                        { return TK_KISS; }
29 ".p"                           { return TK_P; }
30 ".e"                           { return TK_END; }
31 ".end"                         { return TK_END; }
32 ^[01-]+[ \t]+[01234~-]+$       { plalval = strdup((char*)yytext);
33                                  return TK_MINTERM; }
34 [0-9]+                         { plalval = strdup((char*)yytext);
35                                  return TK_DECIMAL; }
36 [a-zA-Z_.][a-zA-Z_.0-9]*       { plalval = strdup((char*)yytext);
37                                  return TK_STRING; }
38 [=]                            { return TK_EQ; }
39 [;]                            { return TK_SEMI; }
40 [\n]                           { plalineno++; }
41 [ \t]                          { /* ignore */ }
42 <COMMENT>[\n]                  { plalineno++; BEGIN INITIAL; }
43
44 %%
```

Figure 32: Code snippet for input file to lex

Sections are separated using “%%” thus line 14 signifies the beginning of a section, and line 44 signifies the end. Token symbols are mapped to “.i”, “.o”, “.ilb”, “.ob”, “.p”, “.e” in lines 17, 18, 20, 21, 29 and 30. Other strings like “.label” and “type” are also mapped to token symbols even though most PLA files do not use these directives. Line 32, 34, and 36 make use of regular expressions in mapping character strings to tokens. Line 32 maps minterm characters as specified in the PLA format (“0”, “1” or “-” on the input side followed by a space (or a tab \t), followed by “0”, “1”, “2”, “3”, “4”, “-“, or “~” on the output side) to a

minterm token. Line 34 maps decimal numbers (used when specifying number of inputs and ouputs using ".i" and ".o") to a decimal token. Line 36 maps actual strings (used when specifying the input and output names using ".ilb" and ".ob") to a string token. For more information on regular expressions, examine [49]. The variable "yytext" holds the current value of the character string being processed. In lines 32, 34, and 36, the "yytext" value is essentially assigned to the "plalval" which is visible to the parser. This will be discussed in more detail in the subsequent section.

To generate the lexer, we issue the command below in a UNIX terminal.

lex filenames.l

The program generates a lexer file named lex.yy.c which can be used to disintegrate PLA files into strings of token symbols. Next we study how the parser comes into play.

Parsing: Also known as syntactic analysis, parsing is the process of analyzing a text, made up of a sequence of tokens to determine its grammatical structure with respect to a certain formal language [50]. For instance, to specify the number of inputs in a design, ".i XX" is used, where XX represents a decimal number. If, for example, a PLA file contains a line ".i ab", this is grammatically incorrect and should generate a parse error. A parser, or syntax analyzer, is a program that performs grammatical verification based on a set of rules. A parser reads in the output of a lexer and imposes a hierarchical structure on the token symbols. Like lexers, parsers may be programmed manually or generated automatically using custom tools. For this project, we utilize the yacc program [46], also readily available on UNIX systems, for generating a PLA parser.

Yacc: Yacc is a parser generator developed for UNIX. The name is an acronym for "Yet Another Compiler Compiler." The grammar of a formal language is specified in a text

file (.y file extension). Yacc reads this grammar file and generates C code for a parser. The parser uses grammar rules, inherited from the grammar file, to analyze tokens sent from the lexer and create a parse tree (a.k.a. syntax tree). The tree imposes a hierarchical structure on the token as in figure 31. Figure 33 depicts the hierarchical structure of the token symbols contained in a PLA file.

```
18 %token TK_I TK_O TK_MV TK_ILB TK_OB TK_LABEL TK_TYPE TK_PHASE TK_PAIR
   MI
19
20 %%
21
22 pla: input_decl output_decl optional function end
23
24 optional:
25          | mv_decl label optional
26          | type optional
27          | phase optional
28          | symbolic optional
29          | symbolic_output optional
30          | kiss optional
31          | pair optional
32          | products optional
33          | input_list optional
34          | output_list optional;
35
36 input_decl: TK_I TK_DECIMAL {
37          myPla->m_dNumberOfInputs = atoi($2);
38 };
39
40 output_decl: TK_O TK_DECIMAL {
41          myPla->m_dNumberOfOutputs = atoi($2);
42 };
43
44 input_list: TK_ILB string_list {
45          /* Parse the list of inputs are store them individually */
46          unsigned int start = 0;
47          unsigned int index = 0;
48          unsigned int input = 0;
49          char *list = $2;
50          unsigned int len = strlen(list);
```

Figure 33: Code snippet for input file to parser

In this input file, the tokens are first declared using the %token keyword as seen in line 18. Next, the hierarchical structure is defined. Line 22 declares 5 possible commands that can be contained in the PLA file. For instance the string - .i 5 - is a command that specifies the number of inputs in the combinational logic and as such qualifies as an input_decl command. Other commands are output_decl, optional (for optional commands), function (for parsing minterms), and end (for the last line of a PLA file). The next step is to define command

structure. Line 36 defines the input_decl command as any string of characters that contain the input token (.i) followed by a decimal token. Once this command is encountered, the value of the decimal token ($2, for value of second token) is assigned to a member function (m_dNumberOfInputs) of an object instance (myPla) of a class we defined for parsing PLA files. After the PLA file has been parsed, this function can be accessed by the main program to obtain the number of inputs defined in the PLA file.

To generate the parser, we issue the command below in a UNIX terminal.

yacc -d filename.y

The yacc program generates a parser file named y.tab.c which can be used to parse token stream obtained from a lexer. The "-d" option causes yacc to generate token definitions and place them in file y.tab.h.

After generating the lexer - lex.yy.c - and the parser - y.tab.c -, we compile/link both files, to obtain the final executable, by issuing the command

cc lex.yy.c y.tab.c -o pla_parse

This generates an executable - pla_parse - that can be used to parse a PLA file by simply running the executable and passing the PLA file name as an argument as shown below

./pla_parse filename.pla

Figure 34 [48] presents a high-level overview of the file parsing process.

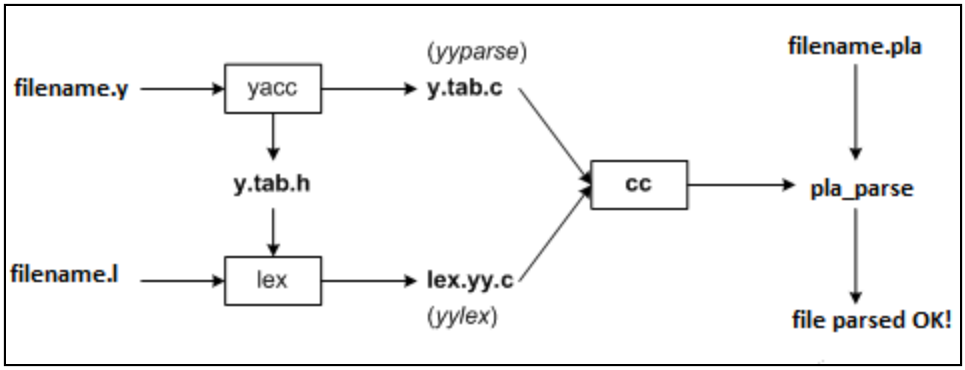


Figure 34 [48]: Overview of file parsing process

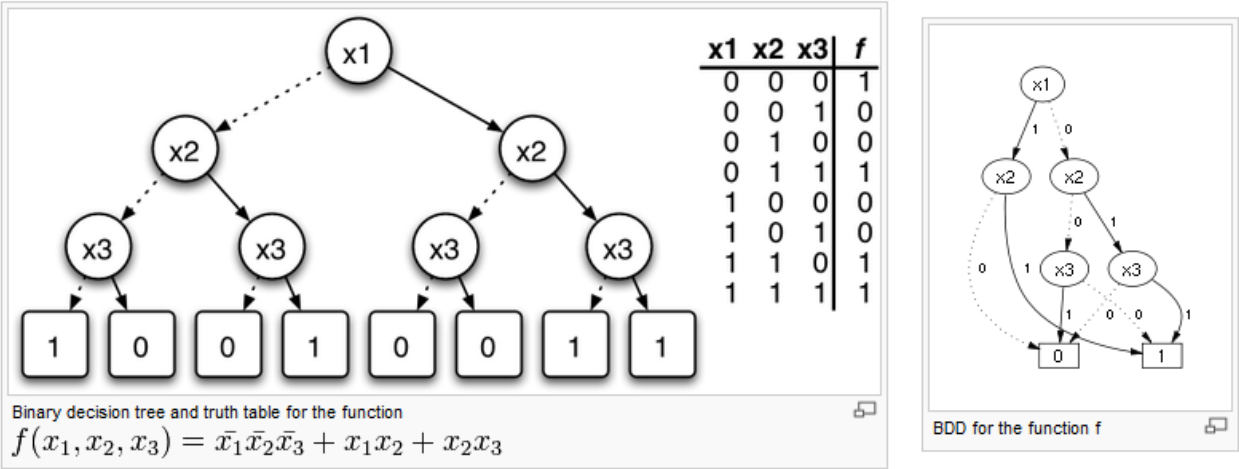
The document located at [47] provides a comprehensive tutorial on how to use lex and yacc.

Output Manipulation

After parsing an input PLA file, information about the combinational block it describes is contained in the logic repair program's memory space in a usable format. Minterms of the combinational block are parsed in line by line and stored in a C++ vector data structure as a collection of characters as defined in the PLA file. To properly and effectively utilize software in generating approximate functions, outputs need to be represented in a format that can be easily operated on by Boolean operators. An argument can be made for bitwise operators, however PLA files can contain a considerable amount of minterms and operating on that many minterms can be time consuming. Furthermore, it's very difficult to perform K-map manipulations on strings even when the strings represent minterms.

For the above reasons, an algorithmic representation optimized for the specific purpose of Boolean manipulations is preferred. This project utilizes binary decision diagrams (BDD) to implicitly represent every function. A BDD is a data structure that is used specifically to represent Boolean functions. In general, a Boolean function can be represented with a graph, which consists of

decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each decision node represents a Boolean variable and has two child nodes that represent logic 0 output and logic 1 output. Essentially, an edge from a decision node to a logic 0 (logic 1) child node represents a 0 (1) assignment to the associated variable. A BDD can be reduced to decrease the memory size. Typically when the term BDD is used, it refers to a reduced BDD. The same applies to this project. Figure 35 [51] presents an example of how a function might be represented by a BDD.



(a) (b)

Figure 35 [51]: BDD representation of a Boolean function

Figure 35(a) shows a binary decision tree (BDT) and a truth table, each representing a three input function. The dotted lines represent a logic 0 edge. We can determine how the BDT works by comparing it with the truth table. For instance, to determine the value of the output (f) when x1 = 1, x2 = 0, and x3 = 1 (based on the truth table, f = 0), begin at the root node (x1) transverse along the solid line (logic 1 edge), arrive at node x2, transverse along the dotted edge (logic 0 edge), arrive at node x3, transverse along the solid line (x3 = 1) and arrive at the output 0, which is as expected. Figure 35(b) is BDD, which is essentially a reduced BDT. Carrying out the same experiment on the

BDD yields a similar result. However since BDDs are reduced, they contain less nodes, less edges and as a result occupy less memory. Tools for representing functions using BDDs handle BDD reduction and reordering. For the purposes of this project, BDDs are implemented using the CU decision diagram (CUDD) package.

CUDD

The CUDD package, which is freely available at [52], provides users with functions for manipulating BDDs. To develop a program using CUDD, the following libraries should be included to the C/C++ source file as shown, and should link libraries *libcudd.a, libmtr.a, libst.a, and libutil.a* to the executable.

```
#include "util.h"
#include "cudd.h"
```

CUDD uses data structures for storing useful information and manages the structures efficiently.

BDD nodes are represented in CUDD using DdNode. A DdNode is a structure with many fields such as the variable index, the reference count and the value. The other fields are pointers that connect nodes together. The index field contains the name of the variable that the node represents. It also reflects the order in which a variable was created. Therefore, the index value of a node is permanent. On reordering BDDs (to reduce size), variables might change in order, but indices remain constant. CUDD depends on garbage collection to free up memory used by BDDs that are no longer in use. This is achieved by maintaining a reference count each time a node is accessed. When a node is created, its reference count must be increased using *Cudd_Ref*. Similarly, once a node is no longer needed, *Cudd_RecursiveDeref* must be called to recycle the nodes of the BDD. The value field represents the logical value of a node. This only applies to leaf nodes.

The CUDD manager coordinates creation, and utilization of BDDs. A DdManager is a data

structure that controls this coordination using unique tables. Unique tables ensure that every node is unique; that is there are no other nodes that have the same variable name and the same child nodes. Before a BDD can be created, the manager must be initialized using *Cudd_Init* as shown in figure 36 below.

```
manager = Cudd Init(0,0,CUDD UNIQUE SLOTS,CUDD CACHE SLOTS,0);
```

Figure 36 [52]: CUDD Manager Initialization

The appropriate parameters must be passed on to *Cudd_Init*. Refer to [52] for more information on initializing the manager. Before exiting the program, *Cudd_Quit* must be called to free up memory. Figure 37 [52] shows a sample program that builds a BDD for the function $f = x'_0 x'_1 x'_2 x'_3$.

```
1.      DdManager *manager;
2.      DdNode *f, *var, *tmp;
3.      int i;
4.
5.       manager = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
6.
7.      f = Cudd_ReadOne(manager);
8.      Cudd_Ref(f);
9.      for (i = 3; i >= 0; i--) {
10.         var = Cudd_bddIthVar(manager,i);
11.         tmp = Cudd_bddAnd(manager,Cudd_Not(var),f);
12.         Cudd_Ref(tmp);
13.         Cudd_RecursiveDeref(manager,f);
14.         f = tmp;
15.     }
```

Lines 1 to 3 declare the manager, several nodes and an integer variable for future use. Line 5 initializes the manager. Line 7 calls a function that essentially assigns the logic 1 value to node f. In line 8, node f is referenced. Lines 9 to 15 contain a loop that repeats the instructions from lines 10 to 14. Line 10 calls a function - *Cudd_bddIthVar* - which checks to see if the projection function exists and returns a reference; otherwise a projection function is created for the index. Line 11 calls the

Cudd_Not for performing the NOT Boolean operator on variable "var". Next, *Cudd_bddAnd* performs the Boolean AND operator on the variables f and var'. Note that the *tmp* variable is required for assigning the result of the AND operation. If a program attempts to assign a new value to a referenced variable (f in this case), the old variable will be lost and there will be no way to free up its nodes.

Upon generating BDDs for each output of the combinational block, design synthesis is carried out to obtain area statistics. Area information is required to determine how much area increase approximate functions need. Synopsys design compiler is the tool of choice for synthesis. It accepts PLA files, among other formats, and generates a gate-level representation of the circuit. Design Compiler only requires that an input file called ".synopsys_dc.setup" - be provided to specify what libraries should be used for implementation. In this project, the Synopsys technology independent class.db library was used to synthesize all designs. Figure 38 is a code fragment of the .synopsys_dc.setup file used for synthesis.

```
1 set designer "Adeola Adeleke"
2 set company "Institute for Space and Defense Electronics"
3 set search_path { . ${search_path} ./src ./db }
4 set link_library { *  /usr/local/isde/synopsys/syn/C-2009.06-SP2/libraries/s
  yn/class.db }
5 set target_library {  /usr/local/isde/synopsys/syn/C-2009.06-SP2/libraries/s
  yn/class.db }
6 set symbol_library {  /usr/local/isde/synopsys/syn/C-2009.06-SP2/libraries/s
  yn/generic.sdb }
7 define_design_lib WORK -path ./work
```

Figure 38: .synopsys_dc.setup file

To specify commands for the design compiler, a shell script may be written. The script can call design compiler and echo commands to it. In the appendix section, we provide every script utilized in area synthesis.

A class called Bound was created for function manipulation. For every function, a bound

object is instantiated to ease the process handling of functions. The Bound class handles BDD creation, synthesis of the original function and approximate functions, and output dumping for writing the output function to a PLA file.

Generation & Synthesis of Approximate Functions

This is arguably the most important step in logic repair. Similarly to how the original output functions are represented, approximate functions are represented using BDDs. CUDD provides functions that can be used to generate approximate functions by selecting cubes. Let us explore how approximate functions are selected for each of the three methods discussed in Chapter IV.

Single-Output Logic Repair/Non-Optimized Method

This section presents the building blocks for selecting approximate functions using CUDD. Preferably, logic repair circuits should result in a high repair-coverage, yet maintain a small area to minimize cost. In the same vein, approximate functions are expected to contribute significantly to the repair-coverage factor while requiring little area for implementation. This can only be achieved by selecting functions that cover a large number of minterms and contain few variables and gates, essentially large cubes. CUDD provides functions capable of choosing the large cubes of a function and/or adding extra terms to the function to improve the quality of generated cubes.

In this project, two major CUDD functions are utilized for selecting candidate cubes; other CUDD functions which represent Boolean operators are also used to facilitate processing. *Cudd_SubsetShortPaths*, one of the two CUDD functions most invaluable to this project, extracts a heavily populated subset from a BDD. This function tries to keep the shortest paths of the input BDD, since they contribute many minterms and contain few nodes. *Cudd_SubsetShortPaths* accepts

5 parameters and returns a pointer to the resulting BDD as shown below.

```
DdNode *
Cudd_SubsetShortPaths(
  DdManager * dd, manager
  DdNode * f, function to be subset
  int  numVars, number of variables in the support of f
  int  threshold, maximum number of nodes in the subset
  int  hardlimit flag: 1 if threshold is a hard limit
)
```

Figure 39: *Cudd_SubsetShortPaths* function definition

Calling this function multiple times with a different *threshold* parameter causes it to generate distinct BDDs, which are basically under-approximation functions. The complementary CUDD function, *Cudd_SupersetShortPaths*, instead chooses a dense superset from a BDD. This function is similar to *Cudd_SubsetShortPaths* except that it accepts the complement of the given function and essentially generates a subset of the complement function which corresponds to a superset of the original function and represents over-approximation functions. *Cudd_SupersetShortPaths* accepts and returns the same parameter types as *Cudd_SubsetShortPaths*. Figure 40 indicates how CUDD functions are used in designing the logic repair tool.

```
166     for(float threshold = 1; threshold > 0; threshold -= 0.1) {
167         if((int)(threshold * Cudd_DagSize(g)) > 1) {
168             DdNode *tmp = Cudd_SubsetShortPaths(manager, g, Cudd_SupportSize(manager, g), (int)(threshold * Cudd_DagSize(g)), 0);
169             if(tmp != F.back()) {
170                 Cudd_Ref(tmp);
171                 F.push_back(tmp);
172             }
173         }
174     }
175
176     // Generate H
177     H.push_back(Cudd_ReadOne(manager));
178
179     for(float threshold = 1; threshold > 0; threshold -= 0.1) {
180         if((int)(threshold * Cudd_DagSize(g)) > 1) {
181             DdNode *tmp = Cudd_SupersetShortPaths(manager, g, Cudd_SupportSize(manager, g), (int)(threshold * Cudd_DagSize(g)), 0);
182             if(tmp != H.back()) {
183                 Cudd_Ref(tmp);
184                 H.push_back(tmp);
185             }
186         }
187     }
```

Figure 40: Using *Cudd_SupersetShortPaths* and *Cudd_SubsetShortPaths*

Lines 168 and 181 show these functions being called with appropriate parameters, after which resulting BDDs are assigned to DdNode pointers and stored in a C++ list for further processing. Many under and over approximation functions are generated for each output. Every possible combination of under-and-over approximation function is considered for selection. Each combination of F&H is referred to as a bound.

The next step is to determine the quality of every bound by synthesizing and calculating associated repair-coverage factors, otherwise known as masking factor. Masking factor can be computed by using equation 3 as discussed in Chapter IV. Essentially, summing up the number of minterms of F and the number of maxterms of H, and dividing the resulting sum by the total number of terms produces the masking factor of the associated bound. By determining the masking factor of each bound, approximate functions that contain more nodes but do not yield an increase in masking

factor may be discarded. Disposing unnecessary bounds minimizes the number of BDDs that have to be synthesized. This speeds up the process of generating approximate functions. CUDD provides the function *Cudd_CountMinterm* for determining the number of minterms present in a BDD. The number of maxterms can be obtained by passing the complemented of the original function as a parameter to *Cudd_CountMinterm*. Complements are obtained by calling *Cudd_Not*. BDD synthesis can be carried out on approximate functions as discussed in the prior section.

Shared Minterm Method

Much of the steps taken in the previous sub-section also apply when obtaining approximate functions using the shared minterm method. However, there is a critical difference in how the logic repair tool implements the shared minterm method. Instead of directly generating approximate functions from output functions, the tool calls a function that recursively spurns for-loops for evaluating the number of minterms. Since it is virtually impossible to determine the optimal shared minterm, the tool smartly selects minterms good candidates based on the number of outputs and shared minterms they contain. After selecting a good minterm sharing output combination, the tool logically ANDs the associated outputs to obtain the shared minterm function. Because shared minterms do not always protect as many minterms as required, it is beneficial to protect more minterms in each output function without interfering with the protection offered by shared minterms. This can be achieved by ANDing the concerned output function with the complement of the shared minterm function for F approximations, and ORing the concerned output function with the shared minterm function for H approximations. Consider how this might work by assuming Figure 41(a) and 41(b) represents two output functions G(0) and G(1) respectively.

	c'd'	c'd	cd	cd'
a'b'	0	1	0	0
a'b	1	0	0	0
ab	1	1	1	1
ab'	1	1	1	1

(a)

	c'd'	c'd	cd	cd'
a'b'	1	0	1	1
a'b	0	1	1	1
ab	1	1	1	1
ab'	1	1	1	1

(b)

	c'd'	c'd	cd	cd'
a'b'	0	0	0	0
a'b	0	0	0	0
ab	1	1	1	1
ab'	1	1	1	1

(c)

Figure 41: Example output functions G(0) and G(1)

A shared minterm analysis of both outputs will likely generate the function represented in figure 41(c). It can be observed that the shared minterm function is equivalent to the result of logical ANDing both output functions. To obtain the largest cube of the output function without affecting shared minterms, the shared minterm function (figure 41(c)) is complemented (logical NOT). Next the complemented shared minterm function (figure 42(a)) is ANDed with the original output function to zero out shared minterm positions. The resulting function (figure 42(b)) can used to obtain a large cube (figure 42(c)) for further minterm protection.

	c'd'	c'd	cd	cd'
a'b'	1	1	1	1
a'b	1	1	1	1
ab	0	0	0	0
ab'	0	0	0	0

(a)

	c'd'	c'd	cd	cd'
a'b'	1	0	1	1
a'b	0	1	1	1
ab	0	0	0	0
ab'	0	0	0	0

(b)

	c'd'	c'd	cd	cd'
a'b'	0	0	1	1
a'b	0	0	1	1
ab	0	0	0	0
ab'	0	0	0	0

(c)

Figure 42: Intermediate functions

Finally, ORing the large cube with the shared minterm function produces the required under-approximate function (figure 43).

	c'd'	c'd	cd	cd'
a'b'	0	0	1	1
a'b	0	0	1	1
ab	1	1	1	1
ab'	1	1	1	1

Figure 43: Final under-approximation function for G(1)

Notice that the new under-approximate function protects 12 of 14 minterms compared to the shared minterm function which only protects 8 of 14 minterms. The same process is repeated to obtain the over-approximation function. However, care should be taken to use the appropriate Boolean operators for obtaining over-approximation functions, as dealing with maxterms is different from dealing with minterms.

Bestrc Method

To obtain approximate functions using this method, the same procedures used in obtaining approximate functions for the non-optimized method apply. However in this case, only the best bound for the entire design is chosen. Subsequently, this bound is ANDed with each output to obtain a bound per output. By obtaining more cubes from this bound as in the steps used in the shared minterm method, replacing the shared minterm function with the corresponding output approximate function, more bounds may be obtained for each output from its original bound.

Selection of Best Candidate & Synthesis of New Circuit

This is the last stage that involves the logic repair tool. Many bounds are generated for each of the methods employed in obtaining logic repair circuits. Even in the shared minterm and bestrc methods where a bound is initially generated for each output, the successive step of selecting cubes to protect more terms guarantees that more possible bounds are generated per output. As a result, a modus operandi must exist for determining which bound is the best candidate per output. Furthermore, designers should be provided with the option to decide how much area overhead is affordable, as the masking factor typically scales accordingly.

In this research project, the equation below was used in selecting the best candidate

$$\chi = \alpha(1-\gamma) + \beta(\text{area incr.}) \quad \ldots. (6)$$

where α is the masking factor multiplier and β is the area increase multiplier. By varying α and/or β, a designer can control area increase penalties and repair-coverage protection. Since the best candidates are functions with high repair-coverage factor γ and low area increase, the ideal candidate is one with the lowest χ. For each output, the software tool stores all possible bounds in a C++ map data structure. Members of a C++ map are added and accessed using keys and values. Bounds are stored using χ as the key and the BDD node as the value. Elements of a C++ map are sorted based on keys such the element with the lowest key occurs first in the map. Thus, after entering every bound of an output into a map, the first element is the best candidate since it represents the bound with the lowest γ. Once the best candidate bound has been determined, the tool dumps a PLA file which contains a description of the F and H functions of the chosen bound per output.

Synthesizing the new circuit is the final step in obtaining a logic repair circuit, and the penultimate step in the overall process of logic repair using approximate functions. Synthesis allows area comparisons to be made between the original circuit and the new circuit. For an n-output

combinational logic block, figure 44 shows what new circuit has to be synthesized.

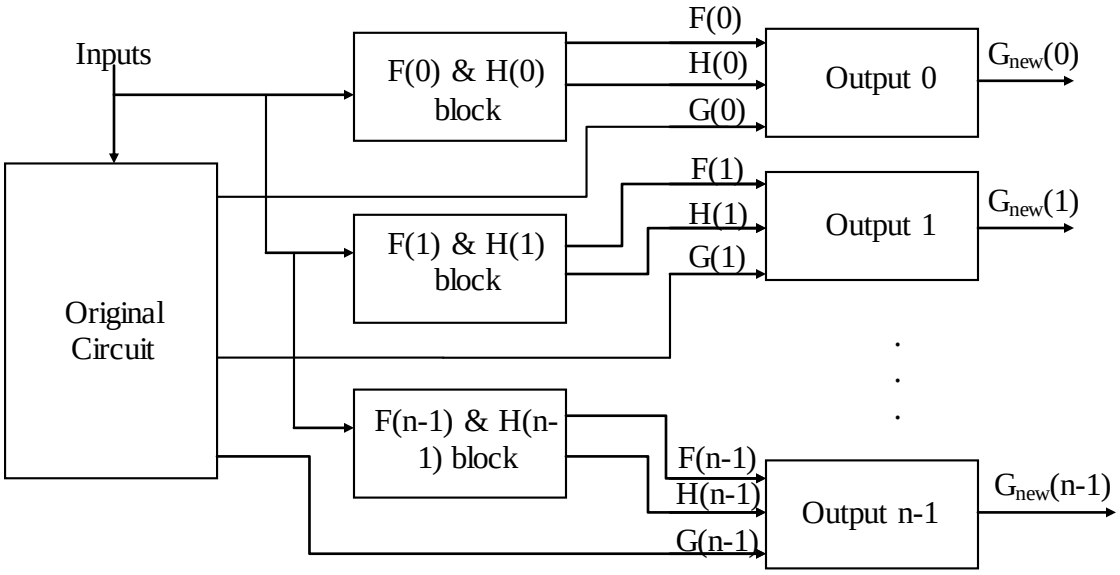


Figure 44: New circuit with logic repair/SET reduction protection

As expected, F and H represent the under and over approximation functions for each output, G represents the original output function, and G_{new} symbolizes the new output function after introducing soft error protection. The black boxes (Output 0 to Output n-1), which are ironically white, contain the selected decider circuit mechanism, which could be the voter circuit, or the and-or structure discussed in Chapter IV. The primary difference between the and-or structure and voter circuit is that the and-or structure synthesizes to slightly less area, but often offers slightly less protection when compared to the voter circuit. Thus, it is up to the designer to decide which structure more suitable for application.

A script was developed to aid in synthesizing the new circuit. The script includes the PLA files that represent each output's best candidate bound, as well as PLA files that represent each output of the original logic block. After including the necessary PLA files, the script invokes DC

compiler on a Verilog file (bounds.v) that describes connections for the new circuit as shows in figure 44. It should be noted that PLA files included in the script must contain *.design XXX* on the first line, where XXX is the module name of the block it describes. By doing this, bounds.v can effectively instantiate a copy of each module block as required. Figure 45 below presents the bounds.v file used for a 7-input-10-output block. In this example, the Voter circuit structure was preferred and implemented over the and-or structure.

```
module BOUNDS (fanin,gp);

input [6:0] fanin;
wire [9:0] g;
output [9:0] gp;
wire [9:0] f;
wire [9:0] h;

SYN_g_   I0 (fanin,g);
SYN_o_0_ U0 (fanin,f[0],h[0]);
SYN_o_1_ U1 (fanin,f[1],h[1]);
SYN_o_2_ U2 (fanin,f[2],h[2]);
SYN_o_3_ U3 (fanin,f[3],h[3]);
SYN_o_4_ U4 (fanin,f[4],h[4]);
SYN_o_5_ U5 (fanin,f[5],h[5]);
SYN_o_6_ U6 (fanin,f[6],h[6]);
SYN_o_7_ U7 (fanin,f[7],h[7]);
SYN_o_8_ U8 (fanin,f[8],h[8]);
SYN_o_9_ U9 (fanin,f[9],h[9]);

assign gp[0] = (f[0] & g[0]) | (g[0] & h[0]) | (f[0] & h[0]);
assign gp[1] = (f[1] & g[1]) | (g[1] & h[1]) | (f[1] & h[1]);
assign gp[2] = (f[2] & g[2]) | (g[2] & h[2]) | (f[2] & h[2]);
assign gp[3] = (f[3] & g[3]) | (g[3] & h[3]) | (f[3] & h[3]);
assign gp[4] = (f[4] & g[4]) | (g[4] & h[4]) | (f[4] & h[4]);
assign gp[5] = (f[5] & g[5]) | (g[5] & h[5]) | (f[5] & h[5]);
assign gp[6] = (f[6] & g[6]) | (g[6] & h[6]) | (f[6] & h[6]);
assign gp[7] = (f[7] & g[7]) | (g[7] & h[7]) | (f[7] & h[7]);
assign gp[8] = (f[8] & g[8]) | (g[8] & h[8]) | (f[8] & h[8]);
assign gp[9] = (f[9] & g[9]) | (g[9] & h[9]) | (f[9] & h[9]);

endmodule
```

Figure 45: Bounds.v file for a 10-input logic block

Note that gp in figure 45 is analogous to G_{new} in figure 44. SYN_g_ is the module name for the block that describes the outputs of the original circuit, while SYN_o_?_ is the module name of the block that describes the F and H approximation functions for each output. This means that on inspecting the PLA file that represents the outputs, the first line will read *.design SYN_g_*. Synthesizing the new

circuit produces a gate level description of the new circuit as specified in the synthesis script.

Fault Simulation & Analysis

How can we certain that the selected bounds will provide the required protection against radiation particles? After all, their estimated repair-coverage factors are only theoretical and have to be proven in a real world scenario, or at least in simulations of a real world scenario. Furthermore, the theoretical repair-coverage factors do not provide a resolute answer to the question, "which of the three methods for selecting approximate functions is the best?" Frankly, this very much depends on the type of circuit being analyzed. For these reasons, it is necessary to carry out fault simulation on the protected circuit for each of the three methods to determine what the "real" repair-coverage factors are and to ease the process of selecting which method is best for the specific circuit application. Fault simulation is also performed on the original circuit for comparison with the protected circuit and evaluation of performance improvement.

In this project, fault simulation is achieved by using the statistical fault injection (SFI) technique presented by Corey Toomey et al. [53]. The SFI technique offers many advantages that make it preferable over other fault simulation strategies. The following advantages of the SFI technique are of utmost importance to this project:

1) It is non-intrusive to the design code
2) Existing test benches can be used for fault simulation.

Figure 46 [53] presents an overview of the SFI technique.

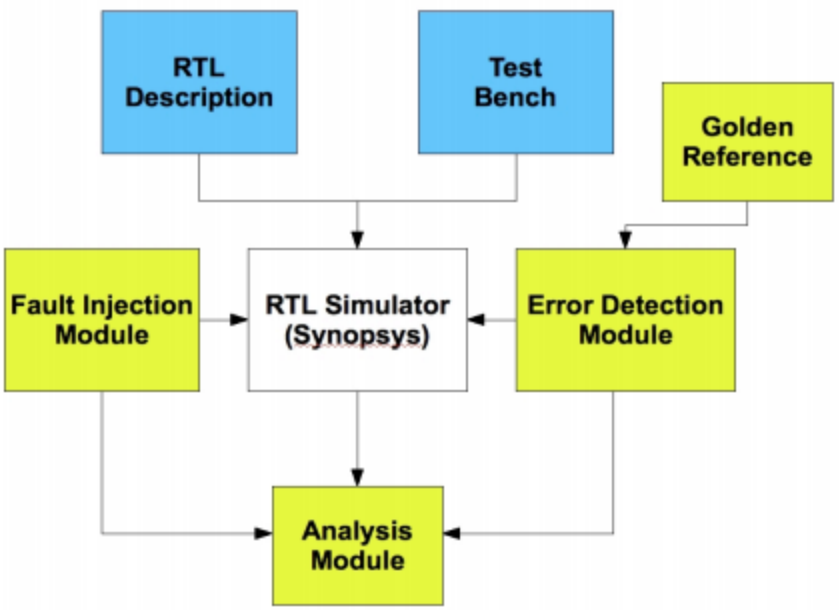


Figure 46 [53]: SFI approach flow chart

Blue boxes represent input files supplied to the design. The RTL description of the logic block to be simulated and the test bench to be used for simulation, fall under this category. The white box represents the chosen simulator. In this project, the Synopsys Verilog Compiler Simulator (VCS) was used to perform simulations. However this technique does not rely on the simulator and works just as well with any other RTL simulator. Yellow boxes represent files that are used to implement the SFI procedure. The relevant files are the fault injection module, the error detection module, the analysis module, and the golden reference file.

The RTL description is obtained from synthesizing the protected circuit as outlined in the previous section. The test bench is generated by simulating every possible input case for combinational block in question. However, in some cases, only a few input combinations are typically exercised. As the generated test bench might not be in sync with what happens in real life, it is more advantageous for a designer to provide a suitable test bench that is more reflective of how the combinational block is connected and utilized in a larger module. This ensures that repair-

coverage factor obtained from simulation will be more accurate for the specific case.

The fault injection module was written in C/C++, using the Verilog Procedural Interface (VPI), by Brian Sierawski of the Institute for Space and Defense Electronics. The VPI is an interface that allows Verilog code to invoke C functions and C functions to invoke standard Verilog system tasks [54]. An in-depth tutorial on the VPI can be found in this book [55]. The fault injection module randomly selects a net, driven by a gate or a primitive, in the RTL description and injects a fault by flipping the bit value of that net (0 -> 1, 1 -> 0). This is essentially a high-level simulation of the real life effect of radiation particles on susceptible nodes. Fault injections occur at a random time during the simulation of the test bench. As this project is concerned with protecting combinational logic, the VPI function used in generating faults simulates SETs, in stark contrast to the VPI function used in [54], which simulates SEUs by injecting faults in registers.

Let us assume that a test bench module – alu4_tb – instantiates a combinational logic block module with the instance name U0, and runs for 16383 time cycles. The fault injection VPI function can be invoked at a random time step as shown in the code snippet in figure 47.

```
$singleEventInit();
#($pseudoRandom(16383*1000)/1000.0);
pulseWidth=$pseudoRandom(500)+500;*/
$singleEventTransient(alu4_tb.U0,pulseWidth);
```

Figure 47: Code snippet of fault injection

The *$singleEventInit()* function initializes the fault injection module and generates a random seed for randomizing the location and time of injection.

The *$pseudoRandom(max_value)* function is called to generate a random number between0 and the *max_value*. Prefixing the function call with '#' causes the simulator to impose a time delay

equal to the random number generated.

Line 3 causes the simulator to generate a random number between 0 and 500, add it to 500, and assign the sum to the pulse width variable. Thus, in essence, the pulse width is assigned a value between 500 and 1000.

Finally, the *singleEventTransient(DUT, pulse_width)* function call injects a faulty pulse with a random pulse width (specified by the *pulse_width* parameter), in a random net, and at a random time. The *DUT* argument parameter is the hierarchical path of the instance that is to receive fault injections. Information about the generated fault such as the fault location, fault generation time, and the type of injection (0 -> 1, 1 -> 0) is logged to the standard output monitor.

The Error detection module, also written in C and VPI, is tasked with logging the outputs of the design after fault injection and comparing the logged output in fault-injection mode against a golden copy obtained in the fault-free mode. To maintain consistency, the simulator must use the same test bench and RTL description when simulating in either of the two modes. The error detection module is invoked on every rising and falling edge of the main clock of the module by calling the *create_output_log(DUT)* function in an *always* Verilog procedural block that is sensitive to the positive and negative edge of the clock. As implied above, there are two modes in which the *create_output_log* function can be invoked. In the faulty mode, the function simply logs the outputs of the combinational logic block into a file which becomes the golden reference copy. In the fault-free mode, the function compares the outputs of the logic block to the golden reference copy. On detecting any discrepancies, *create_output_log* prints out a message indicating the presence of an output error. The message log contains information about where the fault was injected, what time it was injected, what time an output error was observed, and which outputs were affected by the error. Typically, a log file is created for each simulation to facilitate error message logging. The mode of

execution can be influenced by setting the environment variable *COL_FIRST* to true for fault-free simulations and false for faulty simulations.

The analysis module is a script, or in some cases a batch of scripts (excuse the pun), that is invoked post-simulation. Typically, for a given design, one fault-free simulation is run to generate a golden copy, after which multiple faulty Monte Carlo simulations are run to generate multiple output log files. The analysis script traverses through output log files, and counts the number of times that an error occurred on at least one output. It also logs the total number of simulations that were performed. The result obtained from dividing both numbers is the repair-coverage factor.

By carrying out this process for the original circuit and the protected circuit, associated repair-coverage factors can be obtained and contrasted.

CHAPTER VI

RESULTS AND DISCUSSION

The proposed technique of logic repair using approximate functions was tested on the LGSynth93 benchmarks [56], a public domain suite from the 1993 International Logic Synthesis Workshop. Simulation was performed on a single core 2.4 GHz AMD Opteron Processor 250 machine with 11.2 GB of memory running the Linux operating system. The following data presents the area increase and masking factor (γ) information for the best bounds of each output for each method. Each circuit was simulated with the parameters $\alpha = 1$ and $\beta = 0.3$. By doing this, the masking factor is assigned 3.33 times more weight than area. Once again, these parameters should be modified according to the designer's discretion. For each method, 3000 simulations were carried out.

5xp1

This combinational circuit contains 7 inputs and 10 outputs and is described in the 5xp1.pla file.

Non-Optimized Method

This method required 56 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 1 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
5xp1	o_0_	14	**50**	73.9	**91.47**
	o_1_	24	**104.17**	67.3	**96.67**
	o_2_	21	**114.29**	62.13	**85.9**
	o_3_	16	**87.5**	65.03	**90.2**
	o_4_	11	**136.36**	69.6	**96.3**
	o_5_	10	**70**	75.07	**97.13**
	o_6_	5	**100**	77.97	**96.03**
	o_7_	3	**100**	74.83	**92.9**
	o_8_	1	**100**	77.43	**96.07**
	o_9_	6	**66.67**	74.57	**89**
	Overall	96	**110.42**	50.37	**65.27**

Table 1: Table showing results for the 5xp1 circuit for the non-optimized method

There are two scopes for analyzing the results. By considering each output independently, we notice that every output experiences a significant increase in the repair coverage, sometimes at the expense of minimal area. For example, the output o_0_ has a repair coverage % of 91.47% for a 50% area increase, whereas with TMR, a 200% area increase is required to achieve a repair coverage % of 100%. Other outputs such as output o_3_, o_5_, and o_9_, also produce remarkable results. In a sense, every output produced great results considering the fact that the maximum area increase % is 136.36% for output o_4_, which is still some ways off the 200% required by TMR.

When we consider the combinational block as a whole (overall row), the result is not as impressive. A 65.27% repair factor % is observed at the expense of 110.42% area increase. While this is better than TMR, by extrapolation, the designer will be better off with improved results.

Under normal circumstances, the shared minterm method and the bestrc method are expected to produce better statistical outcomes.

Shared Minterm Method

This method required 68 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 2 shows the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
5xp1	o_0_	14	**92.86**	73.9	**95.2**
	o_1_	24	**75**	67.3	**98.3**
	o_2_	21	**109.52**	62.13	**91.47**
	o_3_	16	**150**	65.03	**96.43**
	o_4_	11	**136.36**	69.6	**96.7**
	o_5_	10	**90**	75.07	**95.73**
	o_6_	5	**120**	77.97	**95.67**
	o_7_	3	**100**	74.83	**97.03**
	o_8_	1	**100**	77.43	**96.53**
	o_9_	6	**33.33**	74.57	**86.6**
	Overall	96	**115.63**	50.37	**70.03**

Table 2: Table showing results for the 5xp1 circuit for the shared minterm method

Since the goal of this method is to optimize the repair coverage factor for the overall block, analyzing each output independently is not required. However on first glance, we notice that some of the outputs experience a slight decrease in repair coverage % and a noticeable increase in area.

Nevertheless the results per output are still impressive, although this might not always be the case.

Considering the overall results, a repair coverage % of 70.03% is observed at the cost of an area increase of 115.63%. This result is much better than the result obtained from the non-optimized method and consequently better than the results obtainable from TMR. Still, the bestrc method has to be analyzed to observe how it fares against the other two methods.

Bestrc Method

This method required 61 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. The results are presented in table 3.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
5xp1	o_0_	14	**107.14**	73.9	**95.47**
	o_1_	24	**120.83**	67.3	**93.27**
	o_2_	21	**152.38**	62.13	**89.27**
	o_3_	16	**168.75**	65.03	**90.57**
	o_4_	11	**136.36**	69.6	**92.53**
	o_5_	10	**110**	75.07	**96.73**
	o_6_	5	**100**	77.97	**96.87**
	o_7_	3	**100**	74.83	**97.57**
	o_8_	1	**100**	77.43	**96.13**
	o_9_	6	**100**	74.57	**96.97**
	Overall	96	**133.33**	50.37	**78.47**

Table 3: Table showing results for the 5xp1 circuit for the bestrc method

We notice that, in general, the outputs experience a high increase in area compared to the shared minterm method, while barely experiencing any increase in repair coverage factor. Although one can argue that at the >95% repair coverage %s obtained in the shared minterm method for most outputs, it is unexpected that individual output repair coverages can be improved any further.

When examining the overall results, a repair coverage % of 78.47% is observed at the cost of an area increase of 133.33%. This result is better than the result obtained from the shared minterm method, and a result better than the result obtainable from TMR and that obtained from the non-optimized method. While the bestrc method comes out top in this situation, this might not always be the case as we will observe in the next circuit.

Based on the results presented for the three methods, the bestrc method is the most suitable for the 5xp1 circuit. In the following results per circuit, we will refrain from discussing per-output results for the shared minterm method and the bestrc method unless absolutely necessary.

Clip

This combinational circuit contains 9 inputs and 5 outputs and is described in the clip.pla file.

Non-Optimized Method

This method required 25 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 4 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Clip	o_0_	26	**50**	71.93	**90.27**
	o_1_	34	**155.88**	65.43	**86.63**
	o_2_	40	**167.5**	61.17	**82.7**
	o_3_	28	**185.71**	63.57	**86.87**
	o_4_	27	**100**	74.1	**91.33**
	Overall	88	**98.86**	56.23	**57.77**

Table 4: Table showing results for the clip circuit for the non-optimized method

By considering each output independently, we observe that the area increase % obtained in this case is much higher than in the previous circuit. Also, the resulting repair coverage % is not as good as in the previous circuit. Nevertheless, some outputs such as o_0_ and o_4_ produce great results. If we discard the area of the voter circuit used in TMR, outputs o_2_ and o_3_ produce worse results than TMR. Even without discarding the area of the voter circuit, the bounds of these outputs are not nearly good enough. This is one of the few cases where it might be possible to obtain better per-output results from the other methods.

The result is even worse when examining the overall protection circuit. The repair coverage % only increases by 1.54 units for an area increase % of 98.86%. This is clearly unacceptable and facilitates the need to examine the other methods.

Shared Minterm Method

This method required about 35 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 5 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Clip	o_0_	26	**138.46**	71.93	**92.87**
	o_1_	34	**152.94**	65.43	**92.5**
	o_2_	40	**160**	61.17	**92.19**
	o_3_	28	**200**	63.57	**94.93**
	o_4_	27	**125.93**	74.1	**94.93**
	Overall	88	**153.41**	56.23	**80.23**

Table 5: Table showing results for the clip circuit for the shared minterm method

As the per-output results obtained in the previous method were not up to par, it is important to consider the per-output results. On comparing these results to the previous per- output results, it is observed that outputs o_1_ and o_2_ performed better using this method. On the other hand, the other outputs either perform as bad as or much worse than the previous method. Output o_2_ is the black sheep of the bunch, requiring a 200% area increase for a <100% repair-coverage %.

This method helps significantly improve the repair coverage factor over the non-optimized method. However the inordinate area means that this method produced similar results to TMR for this circuit. It is beneficial to consider the third method to determine if better results can be obtained.

Bestrc Method

About 32 minutes are required to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Results appear in table 6 below.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Clip	o_0_	26	**111.54**	71.93	**95.47**
	o_1_	34	**129.41**	65.43	**93.47**
	o_2_	40	**97.5**	61.17	**93.5**
	o_3_	28	**121.43**	63.57	**89.7**
	o_4_	27	**114.81**	74.1	**95.27**
	Overall	88	**138.64**	56.23	**75.1**

Table 6: Table showing results for the clip circuit for the bestrc method

By examining the results in table 6, we observe that the per-output bounds are better than those selected in the shared minterm method. However, the overall result is not much better producing an area increase to repair coverage ratio that borders on TMR.

Of the 5 circuits that were studied, the clip circuit is the only one that does not greatly benefit from any of the three methods of logic repair using approximate functions. While the per-output results are awe-inspiring, results for the overall block are displeasing to say the least.

Alu4

Alu4 contains 14 inputs and 8 outputs and is described in the alu4.pla file.

Non-Optimized Method

This method required 122 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 7 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Alu4	o_0_	19	**100**	85.23	**94.57**
	o_1_	14	**71.43**	87.9	**98.5**
	o_2_	45	**137.78**	84.7	**92.4**
	o_3_	66	**119.70**	83.57	**96.57**
	o_4_	107	**173.83**	81.03	**93.3**
	o_5_	50	**112**	81.73	**98.3**
	o_6_	49	**130.61**	82.37	**95.7**
	o_7_	296	**112.84**	82.47	**96.1**
	Overall	549	**119.67**	78.37	**86.73**

Table 7: Table showing results for the alu4 circuit for the non-optimized method

From the results presented in table 7, it is observed that the per-output γ % is much higher than in the previous two circuits. Recalling discussions in Chapter II about masking effects, the probability of an SET being masked is inversely proportional to the area of the affected design. Thus the high γ % observed can be attributed to the relatively large area of the alu4 circuit. Nonetheless, the γ % increases across the board for the protection circuit except for output o_4_, which has a γ % and area increase % that is almost as bad as TMR. However, particularly noteworthy is output o_1_ which produces a γ % of 98.5% at the expense of a 71.43% area increase.

The overall block also has a high γ % without including logic repair. By adding logic repair circuit to the original block, a γ % of 86.73% is observed. Although much of this high γ should be attributed to the original circuit's inherent ability to mask SETs, the logic repair circuit still plays a significant role in increasing the γ, albeit at the expense of a 119.67% area increase.

Shared Minterm Method

This method required 151 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 8 shows the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Alu4	o_0_	19	**100**	85.23	**96.33**
	o_1_	14	**100**	87.9	**94.3**
	o_2_	45	**140**	84.7	**95.23**
	o_3_	66	**196.97**	83.57	**96.63**
	o_4_	107	**97.2**	81.03	**96.93**
	o_5_	50	**134**	81.73	**98.07**
	o_6_	49	**108.16**	82.37	**97.87**
	o_7_	296	**97.64**	82.47	**98.6**
	Overall	549	**91.62**	78.37	**84.3**

Table 8: Table showing results for the alu4 circuit for the shared minterm method

This method does not produce any significant increase in the γ %. If implementing this method for the per-output case, the designer has to be wary of output o_3_ which requires a large increase in area for implementation. However considering the overall block, we notice that even though the γ % decreases slightly, the required area decreases by a higher magnitude, thus this is a essentially a better method to implement for the alu4 circuit.

Bestrc Method

This method required 136 minutes to generate and synthesize every possible approximate

function before deciding on and synthesizing the best candidate bounds. The results are presented in table 9.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Alu4	o_0_	19	**110.53**	85.23	**98.57**
	o_1_	14	**178.57**	87.9	**95.23**
	o_2_	45	**126.67**	84.7	**97.1**
	o_3_	66	**156.06**	83.57	**96.7**
	o_4_	107	**92.52**	81.03	**97.17**
	o_5_	50	**168**	81.73	**98.17**
	o_6_	49	**138.78**	82.37	**98.03**
	o_7_	296	**86.49**	82.47	**99**
	Overall	549	**95.26**	78.37	**90.83**

Table 9: Table showing results for the alu4 circuit for the bestrc method

This method addresses the issue with output o_3_ by reducing its area increase % from 196.97% to 156.06% while maintaining the γ %. As a result, a few of the other outputs, such as o_1_ and o_5_, experience an increased area.

For the overall circuit, this method produces a higher γ % at the expense of a slightly higher area. Thus, for the alu4 circuit, the bestrc method is indeed the most desirable method for logic repair.

B12

This combinational circuit contains 15 inputs and 9 outputs and is described in the b12.pla file.

Non-Optimized Method

This method required 93 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Results are presented in table 10 below.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
B12	o_0_	10	**40**	59.47	**88.17**
	o_1_	10	**70**	62.77	**91.27**
	o_2_	15	**73.33**	65.7	**97.13**
	o_3_	3	**100**	74.03	**86.37**
	o_4_	5	**120**	67.83	**94.87**
	o_5_	5	**60**	71.2	**97**
	o_6_	15	**13.33**	64.63	**85.37**
	o_7_	8	**62.5**	66.37	**93.87**
	o_8_	8	**75**	69.57	**85.1**
	Overall	66	**96.97**	54.33	**64.6**

Table 10: Table showing results for the b12 circuit for the non-optimized method

In the most impressive case yet, analyzing the results per output shows many cases of minimal area increase % accompanied by a substantial increase in the γ %. Output o_6_ appears to

have the best result, requiring only an area increase % of 13.33% for a γ % of 85.37%. Even the worst case result, observed in output o_4_ is not as bad as in previous circuits.

Considering the overall circuit, an increase in the γ % is observed for an area increase % of 96.97%.

Shared Minterm Method

This method required 126 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 11 shows the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
B12	o_0_	10	**100**	59.47	**97.6**
	o_1_	10	**30**	62.77	**88.73**
	o_2_	15	**86.67**	65.7	**96.63**
	o_3_	3	**100**	74.03	**98.1**
	o_4_	5	**120**	67.83	**96**
	o_5_	5	**100**	71.2	**95.63**
	o_6_	15	**13.33**	64.63	**86.07**
	o_7_	8	**62.5**	66.37	**89.07**
	o_8_	8	**112.5**	69.57	**96.77**
	Overall	66	**104.55**	54.33	**67.2**

Table 11: Table showing results for the b12 circuit for the shared minterm method

Comparing the results in table 11 to those in table 10, we observe a slight increase in the overall γ % for the shared minterm method over the non-optimized method at the expense of a slight

increase in area. When compared to the results obtained from the non-optimized method, the per-output results for this method do not yield any noticeable area or γ changes.

Bestrc Method

This method required 112 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. The results are presented in Table 12.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
B12	o_0_	10	**140**	59.47	**96.47**
	o_1_	10	**150**	62.77	**94.07**
	o_2_	15	**93.33**	65.7	**94.6**
	o_3_	3	**100**	74.03	**95.2**
	o_4_	5	**100**	67.83	**96.57**
	o_5_	5	**100**	71.2	**95.83**
	o_6_	15	**106.67**	64.63	**94.23**
	o_7_	8	**100**	66.37	**92.97**
	o_8_	8	**112.5**	69.57	**92.87**
	Overall	66	**145.45**	54.33	**69.97**

Table 12: Table showing results for the b12 circuit for the bestrc method

As with the other two methods, this method barely yields any major per-output repair-coverage increases. Although outputs o_1_, o_6_, and o_7_ do produce increases in their repair-coverage %s, the repair-coverage increases are achieved at the expense of high additional area.

Overall, the repair-coverage % is slightly higher for this method than for the others. However, a substantial increase in area is required to achieve the 69.97% repair-coverage %, making it worse off than TMR.

Inc

This combinational circuit contains 7 inputs and 9 outputs and is described in the inc.pla file.

Non-Optimized Method

This method required 17 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 13 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Inc	o_0_	18	**44.44**	63.9	**89.4**
	o_1_	18	**66.67**	59.2	**97.2**
	o_2_	28	**46.43**	57.63	**84.1**
	o_3_	31	**77.42**	53.07	**94.3**
	o_4_	13	**61.54**	59.27	**93.3**
	o_5_	9	**55.56**	68.2	**91.5**
	o_6_	8	**37.5**	76.23	**95.43**
	o_7_	14	**28.57**	71.07	**85.43**
	o_8_	3	**100**	66.57	**98.03**
	Overall	91	**118.68**	35.55	**68.6**

Table 13: Table showing results for the inc circuit for the non-optimized method

Based on the results in the table above, this circuit is very prone to errors as evident from the 35.55% coverage of the original circuit, and many of the outputs have an independent original coverage % of less than 60%. By applying to non-optimized method, the γ % per output jumps up to above 90% for most of the outputs. Output o_6_ appears to be the most positively affected, yielding a coverage % of 95.43% while requiring a meager 37.5% area increase for implementation.

When we consider the overall circuit, things are not as rosy. An area increase % of 118.68% is needed to increase the γ % to 68.6%. Although this relatively stunted increase can be attributed to the circuit's lack of natural masking.

Shared Minterm Method

This method required 27 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 14 shows the obtained results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Inc	o_0_	18	**55.56**	63.9	**91.1**
	o_1_	18	**77.78**	59.2	**93.47**
	o_2_	28	**35.71**	57.63	**88**
	o_3_	31	**70.97**	53.07	**94.33**
	o_4_	13	**100**	59.27	**93.43**
	o_5_	9	**122.22**	68.2	**91.93**
	o_6_	8	**50**	76.23	**95.97**
	o_7_	14	**64.29**	71.07	**87.2**
	o_8_	3	**100**	66.57	**98.37**
	Overall	91	**123.08**	35.55	**69.13**

Table 14: Table showing results for the inc circuit for the shared minterm method

Comparing results for the shared minterm method to results for the non-optimized method, we perceive that, as expected, the non-optimized method is a better implementation for the majority of outputs. In fact, only output o_2_ produces a better bound for the shared minterm method, boasting a lower area increase % of 35.71% for a slightly higher γ %.

The same outcome is observed when considering the overall circuit. In fact, the area increase % increased for the shared minterm method without resulting in a significant increase in γ %. This is most likely due to the fact that the outputs are weakly correlated such that there are hardly any shared minterms to increase the γ % significantly. Also it is safe to state that the increase in area increase % can be partially attributed to the fact that the shared minterms synthesize to larger circuits for the inc circuit.

Bestrc Method

This method required 22 minutes to generate and synthesize every possible approximate function before deciding on and synthesizing the best candidate bounds. Table 15 presents the results.

Circuit	Output	Orig. Area	Area Incr. %	Orig. Cov. %	Rep. Cov. (γ)%
Inc	o_0_	18	**100**	63.9	**95.27**
	o_1_	18	**100**	59.2	**94.5**
	o_2_	28	**110.71**	57.63	**91.93**
	o_3_	31	**158.06**	53.07	**97.13**
	o_4_	13	**76.92**	59.27	**95.2**
	o_5_	9	**100**	68.2	**92.8**
	o_6_	8	**100**	76.23	**96.7**
	o_7_	14	**71.43**	71.07	**97.6**
	o_8_	3	**100**	66.57	**98.8**
	Overall	91	**148.35**	35.55	**80.3**

Table 15: Table showing results for the inc circuit for the bestrc method

When the output results are considered independently, we observe a general increase in area increase % for most of the outputs, and virtually no change in the per-output coverage %s.

This per-output area increase is justifiable though when the overall result is examined. We notice that for the overall circuit, an 80.3% repair-coverage % is obtained at the expense of a 148.35% area increase %. While this result is closer to TMR than other circuits that were simulated, this method was able to produce a better result than the other methods for the overall circuit.

In general, it is often beneficial to interpolate the three methods before selecting the most efficient bounds. While this is optional for a designer interested in independent outputs as he/she might be satisfied with the first bound obtained, it is most helpful for a designer interested in logic block repair (entire block should be protected) to simulate all three methods before selecting the most fitting bound.

CHAPTER VII

CONCLUSION AND FUTURE WORK

This thesis proposed a new technique for achieving logic repair and SER reduction using approximate functions. The proposed technique was developed with the intention of offering better logic repair circuits for minimal additional area. The new logic repair strategy also focuses on providing the designer with flexibility in balancing out area penalties with the associated repair-coverage percentages. A logic repair tool was developed in C/C++ to realize this proposal. The tool essentially considers multiple approximate functions for three methods, obtained by considering large cubes of original functions, and selects the best candidates per method based on the designer's area and coverage factor constraints. The designer can effectively control area and coverage factor trade-offs by modifying the parameters of an equation used for selecting approximate functions. Experiments were performed on five benchmark circuits to test the efficacy of the novel logic repair technique. Three of the five circuits show significant improvements in overall repair-coverage factors at the expense of low area increases. The other two circuits experienced only moderate improvements but are still useful if the designer can afford high area increases. All five circuits produced great results when the outputs are protected independently.

It will be beneficial to perform actual in-field experiments on the protection circuits to compare simulation results with in-field results. Also in the future, this project could be improved by classifying circuits based on which of the three proposed methods is likely to mostly improve the repair-coverage factor. This analysis will also be beneficial to per-output implementations if the designer's goal is to mask each output individually. It is also imperative to determine what causes the

pitfalls for the two circuits that did not experience a great improvement. By doing this, we can improve the algorithms for selecting approximate functions accordingly. This tool could also be further developed to include provision for BLIF and possibly RTL files as this saves designers from the burden, albeit negligible, of having to convert such files to the PLA format.

Another idea to consider for the future is the possibility of developing other methods which are also based on the approximate functions theory. One of such ideas entails combining the shared minterm method with either the bestrc or the non-optimized method. This can be achieved by first generating bounds based on either the bestrc or the non-optimized method. Next, the tool selects a bound for each output using the shared minterms on the available bounds per output. Finally the chosen shared minterm functions are applied to the appropriate outputs. The only potential drawback with this method is the relative increase in computation time not unlike as with the shared minterm method. An SIMD machine might be required to implement such an algorithm. One more glaring improvement that could be made to this project is to expand the shared minterm method to include shared maxterms as well.

It should be noted the time taken to select the best bounds is considerable high because the logic tool considers and synthesizes every possible bound. By discarding certain bounds prior to synthesis, the simulation time could be reduced significantly. For instance, if it can be concluded beforehand that a certain bound will not meet area requirements, then this bound does not need to be synthesized. This predictive disposal can be achieved by evaluating the bound's good factor based on its repair-coverage factor. If it is determined that the bound cannot achieve the area increase required to usurp the current best candidate bound, it can be discarded. To be absolutely thorough in bound selection, we have decided not to discard bounds pre-synthesis.

APPENDIX A

AREA SYNTHESIS SHELL SCRIPTS

Per-Output Area Synthesis Script

```
#/bin/sh

SYN_TOOL=/usr/local/isde/synopsys/syn/C-2009.06-SP2/bin/dc_shell
FILE=$1
SYN_ID=$2
SYN_TOP=$3 #For creating sub-directory in /tmp/ specific to top file
if [ `echo $FILE | egrep .blif$` ]; then
        `~/local/src/blif2v/blif2v.pl < $FILE > $FILE.v`
   FILE="$FILE.v"
        FILE_FORMAT=verilog
elif [ `echo $FILE | egrep .v$` ]; then
        FILE_FORMAT=verilog
else
        FILE_FORMAT=pla
fi

if [ ! -d "/tmp/$3" ]; then
    mkdir /tmp/$3
fi
DC_SCRIPT=/tmp/$3/syn.$SYN_ID.scr
DC_REPORT=/tmp/$3/syn.$SYN_ID.rpt
AREA=/tmp/$3/syn.$SYN_ID.area
echo "free -all" | cat > $DC_SCRIPT
echo "read_file -format $FILE_FORMAT { $FILE }" | cat >> $DC_SCRIPT
echo "uniquify" | cat >> $DC_SCRIPT
echo "remove_constraint -all" | cat >> $DC_SCRIPT
echo "set_max_area 0" | cat >> $DC_SCRIPT
echo "set_structure true -boolean true -timing false" | cat >> $DC_SCRIPT
echo "compile -area_effort high" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite -gen_database" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite -symbol_view" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite -hier_view" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite -schematic_view" | cat >> $DC_SCRIPT
echo "write -f verilog -hierarchy -o /tmp/$3/syn.$SYN_ID.v" | cat >> $DC_SCRIPT
echo "report_area" | cat >> $DC_SCRIPT
echo "quit" | cat >> $DC_SCRIPT
```

```
#dc_shell -f $DC_SCRIPT >& $DC_REPORT
$SYN_TOOL -f $DC_SCRIPT >& $DC_REPORT
grep Combinational  $DC_REPORT | awk '{ print  $3 }' > $AREA
cp $DC_REPORT ./dc_report
cp $AREA ./area
cp $DC_SCRIPT ./dc_script
```

Overall Block Area Synthesis Script

```
#/bin/sh

SYN_TOOL=/usr/local/isde/synopsys/syn/C-2009.06-SP2/bin/dc_shell
SYN_ID=$1
shift
FILE=$@

if [ `echo $FILE | egrep .blif$`  ]; then
        `~/local/src/blif2v/blif2v.pl  < $FILE > $FILE.v`
   FILE="$FILE.v"
        FILE_FORMAT=verilog
elif [ `echo  $FILE | egrep .v$` ]; then
        FILE_FORMAT=verilog
else
        FILE_FORMAT=pla
fi

#echo ${FILE}

DC_SCRIPT=/tmp/syn.$SYN_ID.scr
DC_REPORT=/tmp/syn.$SYN_ID.rpt
AREA=/tmp/syn.$SYN_ID.area
echo "free -all"  | cat > $DC_SCRIPT
echo "read_file  -format $FILE_FORMAT  { $FILE }" | cat >> $DC_SCRIPT
#echo "read_file  -format verilog  { bounds_OverallVoter.v  }" | cat >> $DC_SCRIPT
echo "read_file  -format verilog  { bounds_OverallVoter_fhcombined.v  }" | cat >> $DC_SCRIPT
#echo "read_file  -format verilog  { bounds_TMR.v  }" | cat >> $DC_SCRIPT
echo "uniquify"  | cat >> $DC_SCRIPT
#echo "ungroup  -all -flatten" | cat >> $DC_SCRIPT
echo "ungroup  {J0} -flatten"  | cat >> $DC_SCRIPT
echo "remove_constraint  -all" | cat >> $DC_SCRIPT
echo "set_max_area 0" | cat >> $DC_SCRIPT
echo "set_structure true -boolean true -timing  false" | cat >> $DC_SCRIPT
echo "compile  -area_effort high"  | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite  -gen_database" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite  -symbol_view"  | cat >> $DC_SCRIPT
```

```
#echo "create_schematic -size infinite  -hier_view" | cat >> $DC_SCRIPT
#echo "create_schematic -size infinite  -schematic_view" | cat >> $DC_SCRIPT
echo "write  -f verilog  -hierarchy -o /tmp/syn.$SYN_ID.v"  | cat >> $DC_SCRIPT
echo "report_area" | cat >> $DC_SCRIPT
echo "quit"  | cat >> $DC_SCRIPT
$SYN_TOOL -f $DC_SCRIPT >& $DC_REPORT
grep Combinational  $DC_REPORT | awk '{ print  $3 }' > $AREA
cp $DC_REPORT ./dc_report_all
cp $AREA ./area_all
cp /tmp/syn.$SYN_ID.v syn.$SYN_ID.v
```

APPENDIX B

TEST BENCH INVOCATON AND FAULT SIMULATION ANALYSIS SHELL SCRIPTS

Test Bench Compilation and Fault Simulation Analysis for Overall Block Script

```
#!/bin/sh

rm *.tar.gz
export COL_FIRST=true
vcs +vpi alu4_tbNoFaults.v  -P  /home/adelekaa/logicalMaskingStuff/alu4/col.tab
create_output_log.c
./simv > noInjectionSimulationReport.log
export COL_FIRST=false
vcs +vpi alu4_tbFaults.v  -P /home/adelekaa/logicalMaskingStuff/alu4/col.tab  -P
/home/adelekaa/logicalMaskingStuff/alu4/vcs_pli.tab
/home/adelekaa/logicalMaskingStuff/alu4/libsingleEvent-prerelease.so  create_output_log.c
for (( i = 0; i < $1; i++ ))
do
      ./simv > InjectionSimulationReport$i.log
      mv output_faults.log  output_faults$i.log
done
itr=0;
itr2=0;
itrtot=0;
echo "" > report.log
for (( i = 0; i < $1; i++ ))
do
      if grep -q "SEU ERROR: In Bit Location: " "InjectionSimulationReport$i.log"  ; then
            let itr+=1;
      fi
      if diff --ignore-all-space  output_nofaults.log  output_faults$i.log  >/dev/null  ; then
            let itrtot+=1;
      else #Different files  so an error
            let itr2+=1;
            let itrtot+=1;
      fi
done
echo "${1}  faults were injected  and $itr  errors were generated" >> report.log
echo "${itrtot}  faults were injected and $itr2  errors were generated" > report2.log
```

```
tar -cf InjectionSimulationReport.tar  InjectionSimulationReport*.log
rm -rf InjectionSimulationReport*.log
gzip  InjectionSimulationReport.tar
tar -cf output_faults.tar  output_faults*.log
rm -rf output_faults*.log
gzip  output_faults.tar
```

Fault Simulation Analysis per Output Script

```
#!/bin/sh

#Argument 1 is number of faults generated
#Argument 2 is number of Outputs

if [ -f InjectionSimulationReport.tar.gz  ]; then
    tar -xzvf  InjectionSimulationReport.tar.gz
fi
itr=0;
echo "" > report.log
for (( j = 0; j < $2 ; j++ ))
do
      let itr=0
      for (( i = 0; i < $1; i++ ))
      do
            if grep -q "SEU ERROR: In Bit Location:  gp ${j}"  "InjectionSimulationReport$i.log"  ;
then
                  let itr+=1;
            fi
      done
      echo "${1}  faults were injected  and $itr  errors were generated for output  gp[${j}]"  >>
report.log
done
tar -cf InjectionSimulationReport.tar  InjectionSimulationReport*.log
rm -rf InjectionSimulationReport*.log
gzip  InjectionSimulationReport.tar
```

REFERENCES

[1] C. A. Mack, "Fifty years of Moore's law," *IEEE Trans. on Semicon. Man.*, vol. 24, no. 2, May 2011.

[2] Intel Tick-Tock: http://en.wikipedia.org/wiki/Intel_Tick-Tock.

[3] B. D. Sierawski, B. L. Bhuva, and L. W. Massengill, "Reducing soft error rate in logic circuits through approximate logic functions," *IEEE Trans. on Nucl. Sci.*, vol. 53, no. 6, pp. 3417-3421, Dec. 2006.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Reliab.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[5] R. Koga, S. H. Penzin, K. B. Crawford, and W. R. Crain, "Single event functional interrupt (SEFI) sensitivity in microcircuits," in *Proc. 4th Radiation and Effects Components and Systems (RADECS)*, Cannes, France, Sep. 1997, pp. 311–318.

[6] *International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design*. Austin, TX: SIA, 2005, pp. 6–7.

[7] P. Shivakumar, M. Kistle, S. W. Keckle, D. Burger, L. Alvis, "Modeling the effect of technology trends on the soft error rate of combinational logic," *International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[8] J. R. Schwank, M. R. Shaneyfelt, D. M. Fleetwood, J. A. Felix, P. E. Dodd, P. Paillet, V. Ferlet-Cavrois, "Radiation effects in MOS oxides," *IEEE Trans. on Nucl. Sci.*, vol. 55, no. 4, pp. 1833-1853, Nov. 2008.

[9] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronic," *IEEE Trans. on Nucl. Sci.*, vol. 50, no. 3, pp. 583-602, Jun. 2003.

[10] S. Sarkar, A. Adak, V. Singh, K. Saluja, and M. Fujita, "SEU tolerant SRAM for FPGA applications," *International Conf. on Field-Programmable Tech.*, pp. 491-494, Dec. 2010.

[11] R. Naseer, Y. Boulghassoul, J. Draper, S. DasGupta, and A. Witulski, "Critical charge characterization for soft error rate modeling in 90nm SRAM," *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1879-1882, May 2007.

[12] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration," *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 47-54, 2010.

[13] S. Roy, and V. Beiu, "Majority Multiplexing—Economical redundant fault-tolerant designs for nanoarchitectures," *IEEE Transactions on Nanotechnology,* vol. 4, no. 4, pp. 441-451, Jul. 2005.

[14] B. Pratt, M. Caffrey, J. F. Carroll, P. Graham, K. Morgan, and M. Wirthlin, "Fine-grain SEU mitigation for FPGAs using partial TMR," *IEEE Transactions on Nuclear Science,* vol. 55, no. 4, pp. 2274-2280, Aug. 2008.

[15] J. Vial, A. Virazel, A. Bosio, L. Dilillo, P. Girard, C. Landrault, and S. Pravossoudovitch, "Using TMR architectures for SoC yield improvement," *First International Conference on Advances in System Testing and Validation Lifecycle,* pp. 155-160, Sep. 2009.

[16] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft error resilience," *IEEE Computer,* vol. 38, no. 2, pp. 43–52, Feb. 2005.

[17] D. G. Mavis and P. H. Eaton, "Soft error rate mitigation techniques for modern microcircuits," in *Proc. Intl. Reliability Physics Symposium*, pp. 216–225, 2002.

[18] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, Mar. 1984.

[19] W. C. Huffman and V. Pless, *Fundamentals of Error Correcting Codes*. Cambridge, UK: Cambridge University Press, 2003, pp. 168–208.

[20] S. Buchner and M. Baze, "Single-event transients in fast electronic circuits," in *Proc. IEEE Nucl. Space Radiation Effects Conf. Short Course Text*, 2001.

[21] M. C. Casey, B. L. Bhuva, J. D. Black, and L. W. Massengill, "HBD using cascode-Voltage switch logic gates for SET tolerant digital designs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2510–2515, Dec. 2005.

[22] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," *IEEE Trans. on Nucl. Sci.,* vol. 44, no. 6, pp. 2209-2216, Dec. 1997.

[23] Big Bang: http://en.wikipedia.org/wiki/Big_Bang.

[24] Supernova: http://en.wikipedia.org/wiki/Supernova.

[25] S. Perlmuttes et al., "Measurements of Ω and Λ from 42 high-redshift supernovae," *The Astrophysical Journal*, vol. 517, no. 2, pp. 565–586, Jun. 1999.

[26] Accelerating Universe: http://en.wikipedia.org/wiki/Accelerating_universe.

[27] Cosmic Ray: http://en.wikipedia.org/wiki/Cosmic_ray.

[28] J. F. Ziegler, "Terrestrial cosmic rays intensities," *IBM Journal of R & D*, vol. 42, no. 1, pp. 117-139.

[29] J. F. Ziegler et al., "IBM experiments in soft fails in computer electronics," *IBM Journal of R & D*, vol. 40, no. 1, pp. 3-18, Jan. 1996.

[30] Solar Flare: http://en.wikipedia.org/wiki/Solar_flare.

[31] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Trans. on Nuclear Sci.*, vol. NS-22, no. 6, pp. 2675-2680, Dec. 1997.

[32] J. Velamala, R. LiVolsi, M. Torres, and Yu Cao, "Design sensitivity of single event transients in scaled logic circuits," *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 694-699, June 2011.

[33] Y. H. Lho and K. Y. Kim, "Radiation effects on proton particles in bipolar memory device," *International Joint Conference SICE-ICASE*, pp. 4427-4430, Oct. 2006.

[34] B. Mossawir et al., "A TID and SEE radiation-hardened, wideband, low-noise amplifier," *IEEE Trans. on Nuclear Sci.*, vol. 53, no. 6, pp. 3439-3448, Dec. 2006.

[35] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Trans. on Electron Devices*, vol. 26, no. 1, pp. 2-9, Jan. 1979.

[36] J. Barak, J. Levinson, M. Victoria, and W. Hajdas, "Direct processes in the energy deposition of protons in silicon," *IEEE Trans. on Nucl. Sci.*, vol.43, no. 6, pp. 2820–2826, Dec. 1996.

[37] S. Duzellier, R. Ecoffet, D. Falguère, T. Nuns, L. Guibert, W. Hajdas, and M. C. Calvet, "Low energy proton induced SEE in memories," *IEEE Trans. on Nucl. Sci.*, vol. 44, no. 6, pp. 2306–2310, Dec. 1997.

[38] P. E. Dodd, "Physics-based simulation of single-event effects," *IEEE Trans. on Dev. and Mater. Reliab.*, vol. 5, no. 3, pp. 343–357, Sep. 2005.

[39] Static random-access memory: http://en.wikipedia.org/wiki/Static_random-access_memory.

[40] D-Type Flip-flop Diagram: http://upload.wikimedia.org/wikipedia/commons/3/37/D-Type_Flip-flop_Diagram.svg.

[41] S. M. Jahinuzzaman PhD Thesis, University of Waterloo, May 2008: https://ece.uwaterloo.ca/~cdr/pubs/phd_thesis_jahinuzzaman.pdf.

[42] R. Schneiderwind, D. Krening, S. Buchner, K. Kang, and T. R. Weatherford, "Laser confirmation of SEU experiments in GaAs MESFET combinational logic," *IEEE Trans. on Nucl. Sci.*, vol. 39, no. 6, pp. 1665–1670, Dec. 1992.

[43] File formats for SIS package: http://www2.engr.arizona.edu/~veda/cadtools/sis/tutorials/sis_fileformats.pdf.

[44] Compiler vs. interpreter: http://web.cs.wpi.edu/~gpollice/cs544-f05/CourseNotes/maps/Class1/Compilervs.Interpreter.html.

[45] Lexical Analysis: http://en.wikipedia.org/wiki/Lexical_analysis.

[46] Lex & YACC howto: http://ds9a.nl/lex-yacc/.

[47] Lex and YACC primer/howto: http://tldp.org/HOWTO/pdf/Lex-YACC-HOWTO.pdf.

[48] Lex & YACC tutorial: http://epaperpress.com/lexandyacc/.

[49] Steve Litt's perls of wisdom: http://www.troubleshooters.com/codecorn/littperl/perlreg.htm.

[50] Parsing: http://en.wikipedia.org/wiki/Parsing.

[51] Binary Decision Diagram: http://en.wikipedia.org/wiki/Binary_decision_diagram.

[52] F. Somenzi, CUDD: CU Decision Diagram Package, Univ. of Colorado. Boulder [Online]. Available: ftp://vlsi.colorado.edu/pub/.

[53] Corey Toomey, MS Thesis, Vanderbilt University, 2010.

[54] Verilog Procedural Interface: http://en.wikipedia.org/wiki/Verilog_Procedural_Interface.

[55] S. Sutherland, The Verilog PLI Handbook, Second Edition. Kluwer Academic Publishers, 2002.

[56] All test circuits are available at: http://www-lab13.kuee.kyoto-u.ac.jp/eda/benchmark/mcnc/benchmarks/LGSynth93/.

[57] Parse Tree: http://en.wikipedia.org/wiki/Parse_tree