AN ANALYSIS OF ERROR DETECTION TECHNIQUES FOR ARITHMETIC LOGIC UNITS

By

Ryan Bickham

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, 2010

Nashville, Tennessee

Approved:

Professor Bharat L. Bhuva

Professor William H. Robinson

ACKNOWLEDGEMENTS

I want to thank Dr. Bharat L. Bhuva and Dr. William H. Robinson for their patience and guidance throughout the attainment of this Master's degree. They challenged me academically and provided beyond helpful suggestions. Also, I want to thank Daniel Limbrick for sharing his time and knowledge. His insight and goodwill is greatly appreciated.

In addition, I want to thank my mother, Donnis Ringstaff. Her unconditional love and support is uplifting emotionally and spiritually. Lastly, I want to thank all my loved ones for their thoughts and prayers.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS……………………………………………………………….ii

LIST OF TABLES…………………………………………………...………….....vii

LIST OF FIGURES...……………………………………………………………...viii

LIST OF ACRONYMS……..……………………………………….….................x

Chapter

I. Introduction………………………………………………...………………..…1

II. Overview of Error Detection Methods …………………………………...…4

Redundancy Codes ……………………..…………………….……….4
Arithmetic Codes..………….…………...……………………….….…..6
Arithmetic Codes..………….…………..……………………….….…..7
Parity Codes…………………………..……………………..……..…7

III. Selected Error Detection Techniques for ALUs…..……………………….…8
Redundancy
RESO………………………………………………….………..8
RERO……………………………………………………….......9
Arithmetic Codes
Residue Codes……………………….…………………...……………11
Berger Codes
Reduced Berger Check Prediction …...…………..……………...12
Parity Codes
Parity Prediction Code...……….………………..………………14
Parity and Logic Circuit...……….………………...……………………...15

IV. Error Detection Capabilities and Limitations ...……….……....……………....17

RESO……………………………………………………………………17
RESO for Logical Operations....……………………………………17
RESO for Arithmetic Operations....…………………………….…18

RERO……………………………………………………………………….21

High-speed Modulo-3 Generator……………………………………..…22

Berger Codes……………………..…………………………….……..……23
Berger Check Prediction for Addition………………………..…….23
Berger Check Prediction for 2's Complement Subtraction…...…25
Berger Check Prediction for Logical Operations ……………….26
Parity Prediction…………………………………………………………….…26
Parity and Logic Unit ……………………………………………….....27

V. VHDL Implementation of Error Detection Techniques…………………………28

RESO/RERO……………………..………...……………….……..……28
High-speed Modulo-3 Generator……………………………….....….31
Berger………………………………………………………………….34
Parity/Parity and Logic Unit…………………………………….……36
Synthesis……………………………………..……………………36

VI. Results ….………………………..………………...……………………38

Area, Time, and Power Comparisons for All Techniques
Area Comparison…………………………………………….……38
Time Comparison……….………………………...………..….39
Power Comparison..………………………………..………………41

Area, Time, and Power Comparisons for Each Technique
DMR..……...…………………………………………….……43
High-speed Modulo-3 Generator……………………………...44
Berger…...……………………………………..…………...…46
RESO……………………………………………………………47
RERO……...…………………………………...…………....49
Parity…………………………………………………………….…50
Parity and Logic……………………………..………………….…52

Implications of the Error Detection Techniques on Processing Systems…53

VII. Conclusion and Future Exploration……………………………………....55

Appendix

A. DMR VHDL DESCRIPTION……………………………………………...56

B. MODULO-3 VHDL DESCRIPTION ……………………………………...…59

C. BERGER CHECK PREDICTION VHDL DESCRIPTION…………………..…66

D. RESO VHDL DESCRIPTION……………………………………………..73

E. RERO VHDL DESCRIPTION……………………………………………..80

F. PARITY DESCRIPTION…………………………………………………..87

G. PARITY AND LOGIC DESCRIPTION………………………………...90

LIST OF TABLES

Page

1. Area penalty for an *n*-bit ALU with error detection.........…………………………39

2. Timing results for an *n*-bit ALU with error detection ………………………...…40

3. Power results for an *n*-bit ALU with error detection ……………………………42

LIST OF FIGURES

Page

1. DMR technique which duplicates the ALU and compares outputs…..…………...5

2. Concurrent error detection in an ALU using RESO ……………………......…9

3. Concurrent error detection in an ALU using RERO …………………….....10

4. Residue code adder (or any arithmetic operation)……………………………….12

5. Proposed BCP for ALU………………………………………………………13

6. Parity Prediction Circuit………………………………………………………14

7. Parity and Logic Circuit………………………………………………………16

8. RESO implementation that ensures computation and recomputation steps…….29

9. RERO implementation that ensures computation and recomputation steps……..30

10. Module 1 of high-speed modulo-3 generator ……………………………………32

11. Module 2 of high-speed modulo-3 generator ……………………………………33

12. High-speed modulo-3 generator…………………………………………………34

13. Control PLA of BCP circuit……………………………………………………...35

14. Multioperand Carry Save Adder…………………………………………………35

15. Chain of Logic XOR gates that generate even parity……………………………36

16. Area overhead with an ALU as the baseline…………………………….……….39

17. Timing overhead with an ALU as the baseline………………………………….41

18. Power overhead with an ALU as the baseline……………………………….....42

19. Timing, area, and power results for an *n*-bit ALU using DMR error detection…43

20. Timing, area, and power overhead for an *n*-bit ALU using DMR error detection………………………………………………………………………..44

21. Timing, area, and power results for an *n*-bit ALU using Mod-3 error detection...45

22. Timing, area, and power overhead for an *n*-bit ALU using Mod-3 error detection……………………………………………………………..45

23. Timing, area, and power results for an *n*-bit ALU using Berger error detection...46

24. Timing, area, and power overhead for an *n*-bit ALU using Berger error detection…………………...…………………………………………………47

25. Timing, area, and power results for an *n*-bit ALU using RESO error detection...48

26. Timing, area, and power overhead for an *n*-bit ALU using RESO error detection……………………………………………………………...48

27. Timing, area, and power results for an *n*-bit ALU using RERO error detection...49

28. Timing, area, and power overhead for an *n*-bit ALU using RERO error detection……………………………………………………………...50

29. Timing, area, and power results for an *n*-bit ALU using Parity error detection....51

30. Timing, area, and power overhead for an *n*-bit ALU using Parity error detection……………………………………………………………...51

31. Timing, area, and power results for an *n*-bit ALU using Parity and Logic error detection...52

32. Timing, area, and power overhead for an *n*-bit ALU using Parity and Logic error detection……………………………………………………………...53

LIST OF ACRONYMS

1. ALU……………………………………………………….Arithmetic and Logic Unit
2. BCP…………………………………………………………Berger Check Prediction
3. DMR……………………………………………………...Dual Modular Redundancy
4. MCSA……………………………………………….Multioperand Carry Save Adder
5. MOS……………………………………………………Metal-Oxide-Semiconductor
6. MSB………………………………………………………..Most Significant Bit
7. Mux…………………………………………………………………Multiplexer
8. PLA…………………………………………………………Programmable Logic Array
9. RERO………………………………………Recomputing with Rotated Operands
10. RESO………………………………………...Recomputing with Shifted Operands
11. RTL……………………………………………………..Register Trasnfer Level
12. SFS……………………………………………………………..Strongly Fault Secure
13. VHDL……Very-High-Speed Integrated Circuits Hardware Description Language
14. VLSI…………………………………………………...Very-Large-Scale Integration

CHAPTER I

INTRODUCTION

Advanced microelectronic technologies are becoming increasingly susceptible to faults and errors due to radiation particles. Scaling in very-large-scale integration (VLSI) systems leads to higher packing densities for transistors [1]. As a result, they are more likely to be hit by an incident particle, such as neutrons or alpha particles. The interaction of neutron and alpha particles with semiconductor devices may lead to permanent, intermittent, or transient faults that result in an error [2, 3]. Thus, error detection becomes a greater concern [4] for system reliability as transistor size decreases.

When a metal-oxide-semiconductor (MOS) transistor is exposed to high-energy ionizing irradiation, electron-hole pairs are created in the transistor [5]. Transistor source and diffusion nodes accumulate charge that may invert the logic state of the transistor [2]. The minimum charge necessary to invert a logic state, or cause a fault, is called critical charge. Critical charge differs from circuit to circuit and node to node [6].

When technology scales, the probability of collecting the critical charge decreases, yet critical charge decreases even faster because of lower supply voltages [3]. Thus, faults will increase as transistor sizing decreases. Permanent faults remain for indefinite periods until corrective action is taken [2]. Intermittent faults occur repeatedly at the same location and can be removed by replacing the circuit. Transient faults are induced by neutron and alpha particles. When a fault is made visible to a user, it is then called an error. Although faults are necessary to cause an error, not all faults manifest as errors

because of the functionality within the circuit [7, 8]. Errors can be classified as either soft errors or hard errors. Soft errors are caused by transient or intermittent faults, while hard errors are caused by permanent faults [2].The majority of errors are caused by transient and intermittent faults [3].

Computer architects investigate new techniques to detect and correct soft errors caused by transient faults. Usually, a tradeoff is made between the performance of a processor and the area and power required for error detection. Several error detection methods exist: redundancy codes [4], arithmetic codes [9], Berger codes [10], and parity codes [2]. Although these methods can detect radiation-induced errors, they were first developed to deal with errors induced by harsh environments, novice users, or component obsolescence [11].

The arithmetic logic unit (ALU) is considered the heart of a processor [4]. An ALU is a circuit that performs arithmetic and logic operations. A soft error originating from an ALU can propagate to multiple stages in a processor [12]. An investigation of area, power, and speed for different error detection techniques will be provided for ALUs in this thesis. Surprisingly, power consumption penalties for some commonly used techniques can exceed 300%.

The organization of this thesis is as follows. Chapter II gives an overview of error detection methods. Specific techniques for each method are provided in Chapter III. Chapter IV discusses the capabilities of each error detection technique. VHDL code for each technique is provided in the appendices, and their implementation and synthesis are provided in detail in Chapter V. Chapter VI presents the area overhead, maximum timing

delay, and power consumption of techniques. Chapter VII provides concluding remarks and future research opportunities.

CHAPTER II

OVERVIEW OF ERROR DETECTION METHODS

Soft errors can lead to corrupted data, incorrect program execution, or a complete disruption of a running program. Error detection methods attempt to ensure system reliability by identifying errors and producing correct data or results. Advantages and disadvantages are associated with different techniques within each method. Redundancy codes, arithmetic codes, parity codes, and Berger codes are types of error detection methods. Increasing speed, minimizing area, and minimizing power consumption of an error detection method are a computer architect's goals.

Self-checking ALU

An ALU is used as a baseline when comparing error detection techniques in this thesis. These techniques are considered self-checking circuits. Self-checking circuits are encoded in some error-detecting code so that faults may be detected by a checker [13]. For this thesis, the baseline ALU includes addition, subtraction, logical AND, logical OR, and logical XOR. These particular operations are chosen to provide a fair comparison to the detection limitations for the Berger Check Prediction circuit (BCP).

Redundancy

Redundancy implies multiple computations of the same inputs for a given circuit. If a fault occurs in any of the computations, a comparison step of the results will identify the

presence of an error. Redundancy may be achieved spatially or temporally. Spatial redundancy (also termed as hardware redundancy) duplicates hardware for simultaneous computations, while temporal redundancy (also termed as time redundancy) is by repeating computations using the same hardware.

Hardware redundancy is the most common form of redundancy [14]. Increased timing for hardware redundancy is not an issue due to concurrent error detection. Concurrent error detection is the process of detecting and reporting errors while, at the same time, performing normal operations of the system [14]. The simplest hardware redundancy scheme is dual modular redundancy (DMR). DMR, shown in Figure 1, duplicates the ALU and compares the outputs of the two ALUs. A fault propagating through one of the ALUs will flag an error when the two ALU outputs are compared [15]. DMR provides 100% error detection, yet it requires 100% overhead (plus the comparator).

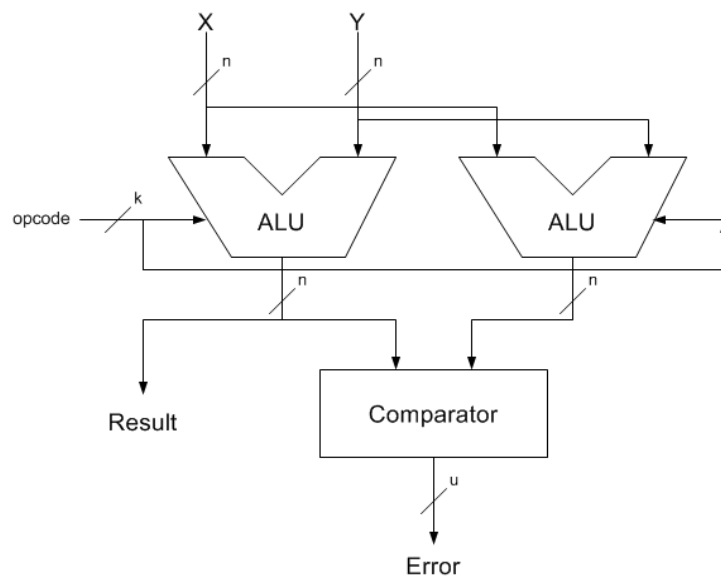


Figure 1: DMR technique which duplicates the ALU and compares outputs.

Time redundancy is an error detection scheme that reduces additional hardware at the expense of using extra time [16]. Depending on the application of the processor, time redundancy may be more affordable than extra hardware. The basic concept of time redundancy is the repetition of computations in ways that allow errors to be detected. The leading techniques that use time redundancy are recomputing with shifted operands (RESO), recomputing with rotated operands (RERO), and Alternating logic [17]. RESO and RERO are discussed in detail in Chapter III. Alternating logic is not discussed because it may require 100% area overhead (in addition to time redundancy) to detect error for some circuit functions [14].

Arithmetic Codes

Arithmetic codes are efficient for providing detection for only arithmetic operators within an ALU [18]. The information parts of an operand are processed through a typical arithmetic operator, while a check symbol is concurrently generated (based on the information bits). Arithmetic codes can ensure fault detection for most arithmetic operators [19]. AN codes are the simplest form of arithmetic codes [2]. They are formed by multiplying each data word N and ALU result by a constant A. The following equation gives an example of an AN code:

$$A(N_1 + N_2) = A(N_1) + A(N_2) \tag{1}$$

AN codes can be derived by left shift operations [2]. Thus, this thesis does not investigate AN codes because it incurs both hardware (DMR) and timing (RESO) penalties. Yet, residue codes are a commonly used arithmetic code and are explored in this thesis.

Berger Codes

Berger codes provide error detection for arithmetic and logic operations. The strongly fault secure (SFS) BCP is the only known technique for self-checking ALUs other than hardware duplication and two-rail encoded ALUs. A SFS BCP is more efficient than a two-rail encoded ALU. Berger codes are valid for all unidirectional errors, for which both 1 → 0 and 0 → 1 errors may occur but they do not occur simultaneously in a single data word [19, 20]. The encoding scheme uses the binary representation of the number of 0's in information bits as the check symbol [21].

Parity Prediction Codes

Parity prediction circuits only provide detection for arithmetic operations. The term "prediction" suggests parity is "predicted". However, parity "prediction" is not a speculative process, but it computes the parity of the operands and result for comparison [2]. Parity prediction adders require the lowest hardware overhead among all known self-checking adder schemes [22].

CHAPTER III

SELECTED ERROR DETECTION TECHNIQUES FOR ALUS

This thesis investigates DMR, RESO, RERO, residue codes, Berger codes, and parity codes for error detection. These techniques are explored because they represent a variety of error detection methods. DMR was previously discussed in Chapter II.

RESO

RESO-k refers to shifting by k bits. Assume RESO uses an n-bit ALU and an n-bit shifter. During the first computation, operands undergo an ALU operation. The result of the ALU is shifted left and stored into a register, as shown in Figure 2. During the recomputation step, the operands are shifted left upon entering the ALU. This result is compared to the previously stored result in the register [23].

Usually, when an n-bit operand is shifted left by k-bit(s), the leftmost k-bit(s) are moved out of the operand and the right most k-bit(s) are zero. This presents the possibility of essential bits being removed whenever shifted left, which would lead to an incorrect result. In order to preserve essential k-bit(s), an $(n + k)$ shifter and $(n + k)$ ALU is needed. For example, assume the recomputation step uses an n-bit shifter. Let the operand X be equal to 01010 in binary. If the operand is shifted left by two bits, then the shifted operand is equal to 01000. The MSBs are shifted out. Now assume the recomputation step uses an $(n + k)$-bit shifter. Then the shifted operand is equal to 01010XX. The most significant bits (MSBs) remain in the operand to ensure correct

results. The rightmost k-bit(s), XX, should always be zero [23]. Moreover, during the first computation, k-zeros are added as MSBs for each of the operands.

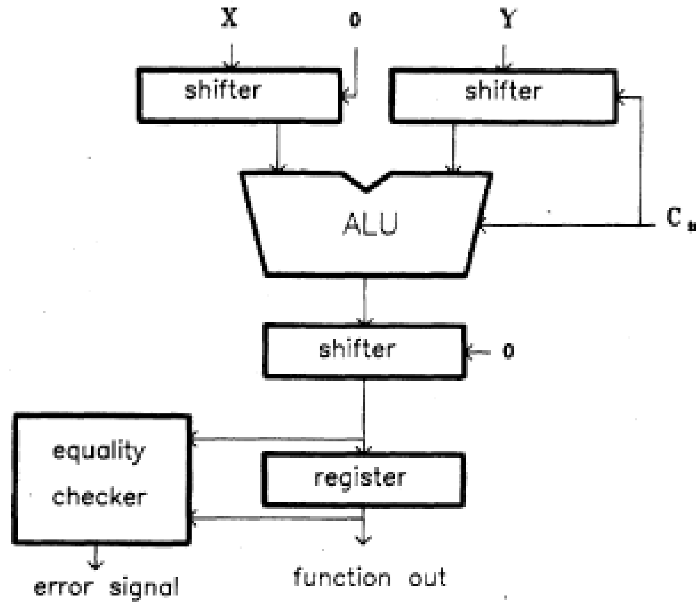


Figure 2: Concurrent error detection in an ALU using RESO [23].

RERO

RESO and RERO have similar time redundancy characteristics, yet they differ in ALU size. RESO uses an $(n + k)$-bit shifter and an $(n + k)$-bit ALU; whereas Figure 3 shows RERO uses an $(n + 1)$-bit rotator and an $(n + 1)$-bit ALU. RERO-k refers to rotating by k bits. Assume RERO uses an n-bit rotator and an n-bit ALU for two sequential computations. During the first computation, two operands undergo an ALU operation and the result is stored in a register. During the recomputation, operands are rotated right before entering the ALU. Next, the result of the ALU is rotated left and then compared to the previous result stored in the register from the first computation [4].

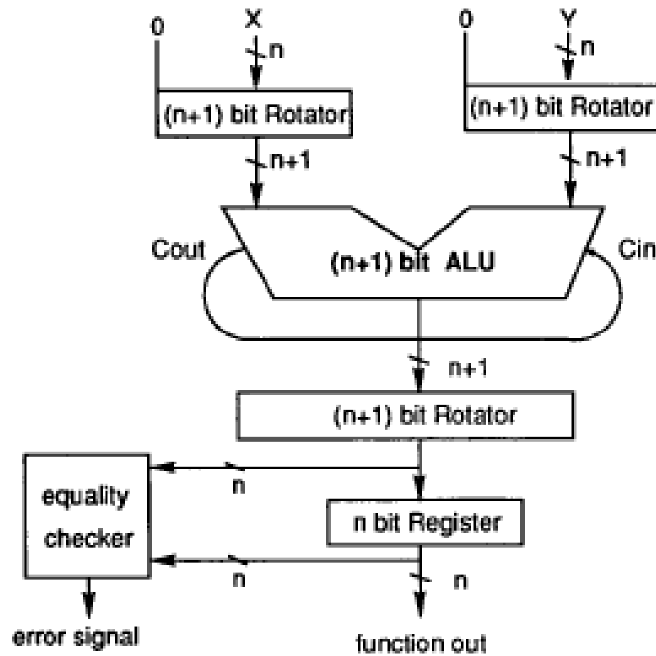


Figure 3: Concurrent error detection in an ALU using RERO [4].

RERO designers discovered a serious problem with the carry-out/carry-in bit for an n-bit ALU. The physical and logical patterns in the ALU are shown as follows:

Physical pattern of bits:	$(n-1), (n-2), \ldots, (i+1), i, \ldots, 2, 1, 0$
Logical pattern of bits before rotation:	$(n-1), (n-2), \ldots, (i+1), i, \ldots, 2, 1, 0$
Logical pattern of bits after rotation:	$i, \ldots, 2, 1, 0, (n-1), (n-2), \ldots, (i+1)$.

Now, assume two previously rotated operands undergo an arithmetic operation. One concern of RERO is to ensure the correct carry-in for logic bit (i + 1) and to extract the carry-out from logic bit (n – 1). There could be cases where the carry-out from logic bit i to (i + 1) propagates pass logic bit (n – 1) to 0. This technique described could potentially produce an incorrect result [4].

In order to ensure correct results, Li et al. propose two features to prevent a case of the carry-bit continually propagating through the operand [4]. The first feature

involves inserting an additional bit in the rotators. The additional bit becomes the MSB and an (n + 1)-bit ALU is used during computations. The new physical and logical patterns in the ALU are shown as follows:

Physical pattern with additional bit:	@, (n – 1), (n – 2),. . ., (i + 1), i, . . ., 2, 1, 0
Logical pattern of bits before rotation:	@, (n – 1), (n – 2), . . ., (i + 1), i, . . ., 2, 1, 0
Logical pattern of bits after rotation:	i, . . . , 2, 1, 0, @, (n – 1), (n – 2), . . ., (i + 1)

The @-bit represents the additional bit, which is initially set to '0.' A carry out cannot be generated from logical bit @ to 0 with this modification. The second feature avoids the propagating carry bit by connecting the carry-out of physical bit n with the carry-in of physical bit 0. During the first computation, the carry-out of the extra bit @ will always be 0 for arithmetic operations, because the @-bits are always set to '0'. After rotation, during the recomputation step, the physical index n contains bit i and the physical index 0 contains bit (i + 1). The physical connection during this step allows the correct carry-out from logic bit i to be applied to logic bit (i + 1) [4].

Residue Codes

Residue codes are a type of separate arithmetic code, in which the information to be used in checking is called the residue. The residue, r, of an operand, A, is equal to the remainder of A divided by the modulo m [24]. For notation, $r = A \bmod m = |A|_m$. For example, if $m = 3$, the residue of A could be any number 00_2 to 10_2. Thus, two bits are needed for checking. If $m = 15$, the residue of A could be any number 0000_2 to 1110_2. Thus, four bits are needed for checking.

Again, residue codes can only be used for arithmetic operations. Two computations are occurring concurrently in Figure 4 [25]. For the first computation step, two operands, *A* and *B*, undergo an addition operation in the ALU. A residue generator then produces a residue code from the ALU result. For the recomputation step, each operand concurrently enters a residue generator. These residues then undergo the same ALU operation as in the first computation (addition in this case). A modulo-3 residue code would use a 2-bit ALU and a modulo-15 residue code would use a 4-bit ALU during the recomputation step. Decreasing n in an n-bit ALU decreases hardware. Thus, a modulo of $m = 3$ is used for residue codes in this thesis.

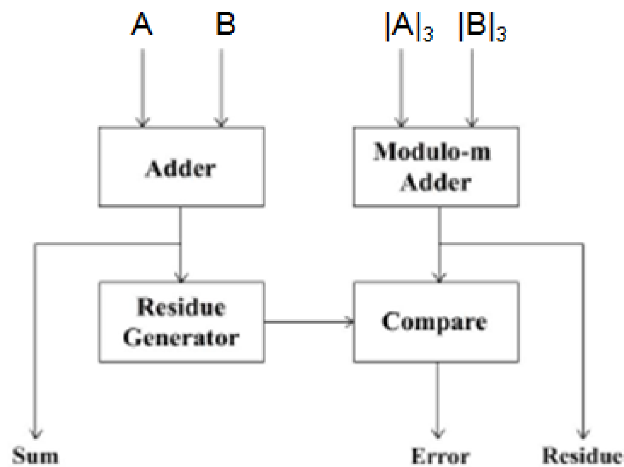


Figure 4: Residue code adder (or any arithmetic operation) [25].

Reduced Berger Check Prediction

Berger codes provide concurrent error detection in arithmetic and logical operations. The proposed BCP design proved to be fault-secure and self-testing with

respect to any single fault in the ALU [26]. Lo et al. suggest that the scheme will provide considerable savings in hardware logic (or chip area). It assumes the BCP circuit is implemented instead of a second ALU (for DMR).

During BCP two computations are occurring concurrently in Figure 5. Operands, for the first computation, undergo simple ALU operations. A Berger check code is then created based upon the result. The second computation uses BCP to generate a Berger check code based on the length of the operands. For this step, the Berger check code is formulated via equations associated with a particular ALU operation [26]. Calculating the Berger check code will be discussed further in Chapter IV.

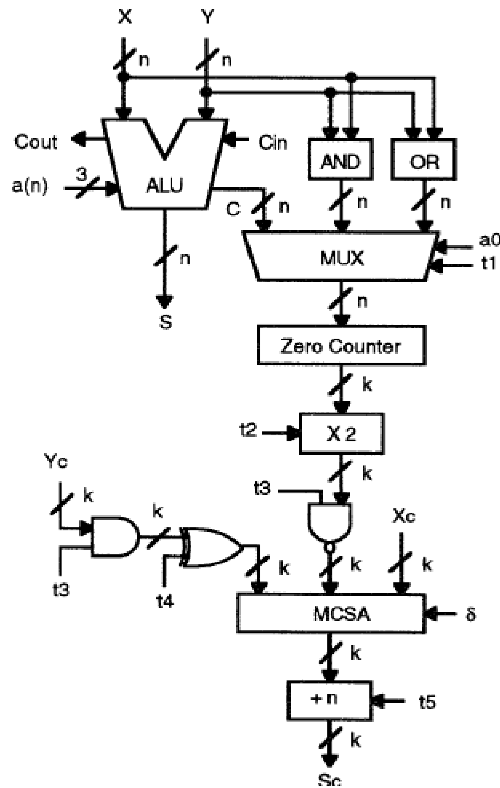


Figure 5: Proposed BCP for ALU [26].

Parity Prediction

Parity prediction circuits generate parity bits for operands and the result, as shown in Figure 6. Remember, this technique can only be used for arithmetic operations. For the first concurrent computation, two operands undergo an arithmetic operation, where the parity of the result is generated. During the second concurrent computation, parity bits for each operand are inputs to a logical XOR gate. This result is compared to the result of the first computation. Moreover, parity prediction circuits are currently used in commercial microprocessor, such as the Fujitsu SPARC V microprocessor [2].

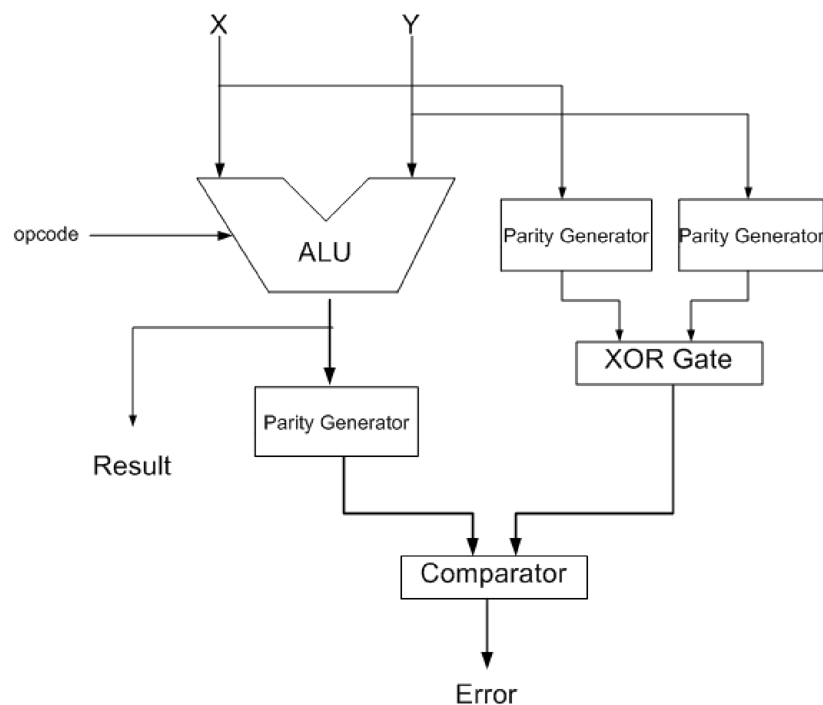


Figure 6: Parity Prediction Circuit.

Parity and Logic Circuit

This thesis investigates combining the parity prediction circuit with a logic circuit. This design allows error detection for not only arithmetic, but logic operations. Figure 7 shows the parity prediction circuit along with a duplicated logic unit of an ALU (i.e., excluding the original arithmetic hardware). During the first computation, two operands undergo an ALU operation. This result propagates to the comparator and a parity generator circuit. Concurrently, for recomputation, the parity bit for each operand is generated. These parity bits are sent through a logical XOR gate with the parity bit of the result from the first computation. This stage checks for errors between the computations. Also, the operands are sent to a logic unit. This result is compared with that of the first computation. A multiplexer (Mux) selects the error signal of the comparator (for logical operations) or the parity prediction circuit (for arithmetic operations). An example of this technique is provided in Chapter IV.

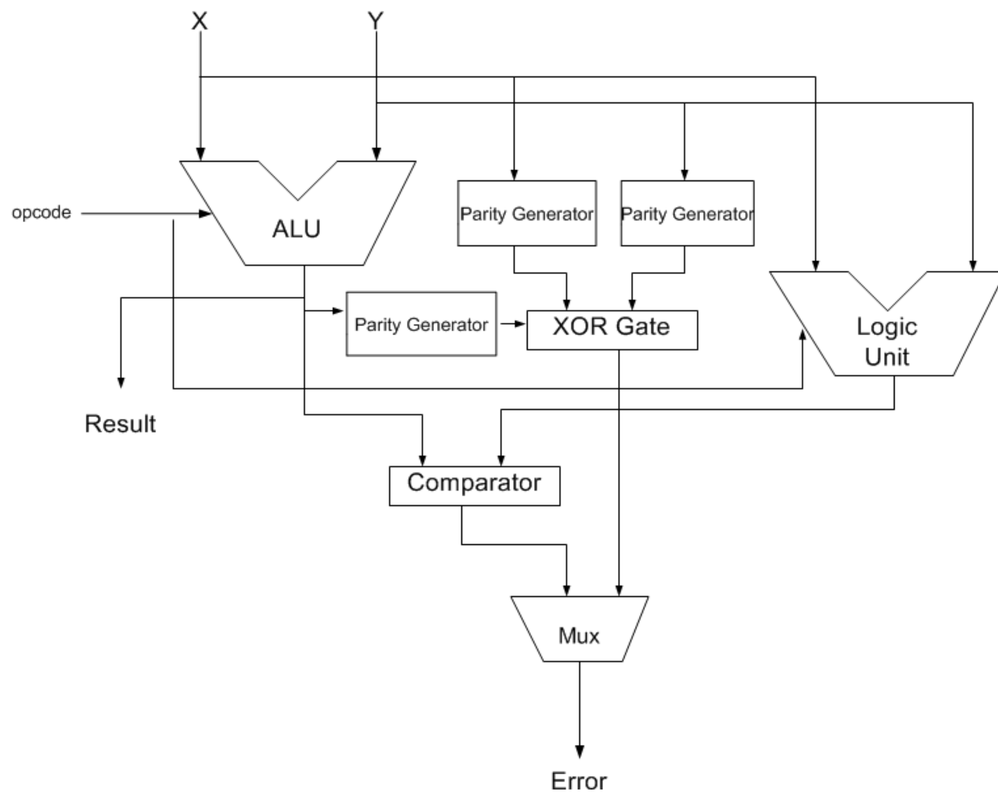


Figure 7: Parity and Logic Unit Circuit

CHAPTER IV

ERROR DETECTION CAPABILITIES AND LIMITATIONS

One may assume the proposed error detection techniques detect erroneous results for any case. However, techniques have certain limitations depending on their use. This section explains special cases when a fault causes an incorrect result, but does not flag an error. In addition, this section shows the error detection capabilities of other techniques.

RESO

RESO for Logical Operations

For logical operations (AND, OR, NOT, XOR, etc.), RESO-k (for any k) detects all errors for bit-wise operations when the fault is confined to a single bit-slice. Let bit-slice i be faulty. If a fault produces an error in the result then the bit i of the first computation step is incorrect. During the second computation, after the operand is shifted left by one bit, the bit i is computed by the non-faulty bit-slice bit $(i + 1)$. Bit i of the recomputation step will be correct and the error would appear in the $(i - 1)$ bit-slice. The result will not match that of the first computation step, thus signaling an error message [23]. Below is an example (faulty bits are underlined):

First computation step with a faulty bit-slice:

Operand X = 1011 and operand Y = 1100

Faulty bit slice: 1

Affected operand X = 10$\underline{0}$1

X_fault OR Y = 1101

Shift results = 11010

Recomputation step with a faulty bit-slice:

$k = 1$

$X = 10110$ and $Y = 11000$

Affected operand $X = 10100$

X_fault OR $Y = 11100$

An error is flagged when the results of each computation step is compared. Thus, RERO-k, for any k, detects all bit-wise logic operations.

RESO for Arithmetic Operations

RESO-1 for arithmetic operations needs to be analyzed further for arithmetic operations. RESO-1 can be applied to ALUs that use disjoint sum and carry networks and a carry look-ahead network. Instead, a ripple-carry adder is used for the ALU implementation. Consider a bit-sliced ripple-carry adder with a faulty bit slice i. During the first computation step, the sum bit of bit-slice i carries a weight of 2^i and the carry-out bit carries a weight of 2^{i+1} [23]. A fault in the sum and/or carry-out i bit-slice could produce the following results:

Fault in the sum bit:	Result is off by $\pm 2^i$
Fault in the carry-out bit:	Result is off by $\pm 2^{i+1}$
Fault in the sum and carry-out bit:	Result is off by $\pm 2^i \pm 2^{i+1}$ ($= \pm 3 \text{ X } 2^i$).

Thus, the result of the first computation step could be off by one of $\{0, \pm 2^i, \pm 2^{i+1}, \pm 3 \text{ X } 2^i\}$.

For the second computation step, the operand is shifted left by one bit. Now, the sum bit of bit-slice i carries a weight of 2^{i-1} and the carry-out bit carries a weight of 2^i. A fault in the sum and/or carry-out i bit-slice could produce the following results:

Fault in the sum bit: Result is off by $\pm 2^{i-1}$

Fault in the carry-out bit: Result is off by $\pm 2^i$

Fault in the sum and carry-out bit: Result is off by $\pm 2^{i-1} \pm 2^i$ (= ± 3 X 2^i).

Thus, the result of the second computation step could be off by one of {0, $\pm 2^{i-1}$, $\pm 2^i$, ± 3 X 2^{i-1}}. From this analysis, a no-error message could be reported when not only when there is no error, but when a fault occurs in the sum bit of the first computation and the carry bit of the second computation ($\pm 2^i$). An example of faults that does not flag an error is shown below (faulty bits are underlined):

$X = 1$ and $Y = 0$

Faulty bit slice = 1

<u>First computation step</u>:

$X = 01$ and $Y = 00$

$X + Y = 01$

Faulty sum = <u>1</u>1

Shift left = <u>1</u>10

<u>Recomputation step:</u>

Shift X and Y left: $X = 10$ and $Y = 00$

Carry bit = 000; Faulty carry bit = <u>1</u>00

Faulty sum = <u>1</u>10

For the recomputation step, the operand must be shifted by more than one bit [23], so a fault in each computation will flag an error message.

RESO-2 is used for the recomputation step. Results of the first computation step will be the same as in RESO-1 {0, $\pm 2^i$, $\pm 2^{i+1}$, ± 3 X 2^i}. For the recomputation step, the operand is shifted left by two bits. Now, the sum bit of bit-slice i carries a weight of 2^{i--2} and the carry-out bit carries a weight of 2^{i-1}. A fault in the sum and/or carry-out i bit-slice could produce the following results:

Fault in the sum bit: Result is off by $\pm 2^{i-2}$

Fault in the carry-out bit: Result is off by $\pm 2^{i-1}$

Fault in the sum and carry-out bit: Result is off by $\pm 2^{i-2} \pm 2^{i-1}$ (= ± 3 X 2^{i-2}).

Now, the result of the second computation step could be off by one of {0, $\pm 2^{i-2}$, $\pm 2^{i-1}$, ± 3 X 2^{i-2}}. No single error appears in the first computation step and the second computation step for RESO-2 [23]. Apply the example from RESO-1 to RESO-2 (faulty bits are underlined):

X = 1 and Y = 0

Faulty bit slice = 1

First computation step:

X = 001 and Y = 000

$X + Y$ = 001

Faulty sum = 011

Shift left = 0110

Recomputation step:

Shift X and Y left: X = 100 and Y = 000

Carry bit = 0000; Faulty carry bit = 0100

Faulty sum = 1000

A fault in each computation step would flag an error message ($0110 \neq 1000$) if using RESO-2.

RERO

RERO-k refers to operands rotating by k-bit(s). An error caused by a faulty bit-slice can be detected depending on the number of rotations. Li et al. discuss error detection capabilities for k faulty bit-slices [4]. However, this thesis focuses on a single faulty bit-slice. RERO-k for a single faulty bit-slice has the same constraints for arithmetic and logic operations. A single error in each computation step cannot be detected if $k = (n + 1)$ during the recomputation step. An example is below (faulty bits are underlined):

$X = 101$ and $Y = 010$

First computation step:

$X + Y = 111$

First computation step with faulty bit slice i:

Faulty bit slice = 1

$X + Y = 1\underline{0}1$

Recomputation step:

The @-bit is included in the operands: $X = 0101$ and $Y = 0010$

$k = (n+1) = 3 + 1 = 4$

After left rotation: $X = 0101$ and $Y = 0010$

Faulty bit slice = 1

$X + Y = 01\underline{0}1$

After being rotated by $k = (n + 1)$, the operands remain in the same bit position. Thus, a faulty bit-slice has the potential of inverting the same bit during the recomputation step.

This thesis uses RERO-2 to be consistent with RESO-2. The smallest bit width for an operand is 8 bits, so k will never equal $(n + 1)$.

High-speed Modulo-3 Generator

The high-speed modulo-3 generator has the capabilities of producing a modulus 3 remainder. The sum feature of the module will not be used. Every operand, A, has a certain codeword or in this case associated residue, r [24]. Residue codes for error detection have two concurrent computation steps. During the first concurrent computation step, two operands, A and B, undergo an arithmetic operation. Then the high-speed modulo-3 generator will produce a residue, $r_{A[]B}$ ([] refers to any arithmetic operation). During the second concurrent computation step, the residues, r_A and r_B, are generated by the high-speed modulo-3 generator with respect to A and B. Then r_A and r_B undergo the same arithmetic operation as in the first computation step. The outcome of this method should lead to $r_{1[]2}$ being equal to r_A [] r_B if there were no faults.

Mathematically speaking, for addition

$$((A + B) \text{ Mod } m) = ((A \text{ Mod } m) + (B \text{ Mod } m)) \text{ Mod } m. \quad (2)$$

The addition operation can be substituted for other arithmetic operators. An example of residue codes for an addition operation is provided below. $A = 10$, $B = 9$ and $m = 3$.

First concurrent computation: $A + B = 10 + 9 = 19$

Residue of first computation: 19 Mod 3 = 1

Residue of A during second concurrent computation: 10 Mod 3 = 1

Residue of B during second concurrent computation: 9 Mod 3 = 0

Addition of r_A and r_B: = 1 + 0 = 1.

Thus, the residue of the first computation is equal to modulus of r_A + r_B of the second computation.

Berger

The mathematical foundation for arithmetic operations and (addition and subtraction) then for logical operations (AND, OR, XOR) are provided. Each Berger check result, S_c, of an ALU operation is a function of X, Y, X_c, and Y_c, where X and Y are operands and X_c and Y_c are encoded Berger checks [26]. Given an operation, $S = X$ op Y, then:

$$S_c = F(X, Y, X_c, Y_c) \quad (3)$$

Berger Check Prediction for Addition

We are given the two n-bit operands X (x_n, . . ., x_2, x_1) and Y (y_n, . . ., y_2, y_1) to obtain a sum S (s_n, . . ., s_2, s_1) with internal carries C(c_n, . . ., c_2, c_1). Every x_i, y_i, s_i, and c_i are either 0 or 1. The formula for the ith bit of the operation is:

$$x_i + y_i, + c_{i-1} = s_i, + 2c_i = (s_i + c_i) + c_i \ \ [26]. \quad (4)$$

Let N(X) stand for the number of 1s in the binary representation of X (i.e. $N(x_i) = x_i$). Equation (4) shows a relationship between the number of 1's in the operand and in the sum. The carry output c_{out} is accounted for as one of the internal carries and the MSB of the sum. The formula for the n-bit case is:

$$N(X) + N(Y) + N(C) - c_{in} = N(S) + c_{out} + N(C) \quad (5)$$

C_{in} (c_{i-1}) is the carry input and $c_{out} = c_n$. The Berger check code is the inversion of (4) and (4) because it develops a relationship between the operands and sum by calculating the number of 0's. For example the Berger check symbol for the number of 0's in the X operand is X_c.

$$X_c = n - N(X). \quad (6)$$

We can arrive at the Berger check symbol, S_c, by using (4) and (5):

$$S_c = X_c + Y_c - C_c - c_{in} + c_{out}. \quad (7)$$

C_c is denoted as the number of 0s in the internal carry. By (4) we know that (5) = n – N(S) [26].

Below is an example of (6):

X = 101011 and Y = 101101 and $c_{in} = 0$

S = 011000, C = 10111, and $c_{out} = 1$

$X_c = 2$, $Y_c = 2$ and $C_c = 1$.

From (7) we know that:

$S_c = 2 + 2 - 1 + 1 = 4$.

We also know that S_c must equal n- N(S). N(S) = N(011000) = 2. Thus, $S_c = n - N(S) = 6 - 2 = 2$ [26].

Berger Check Prediction for 2's Complement Subtraction

The subtraction operation is $S = X - Y$. We know that subtraction can be calculated by using addition. Thus, we complement Y bit-wise and add 1. Now, $S = X + \bar{Y} + 1$. We must take into account the carry input to the adder. In order to obtain plus 1, c_{in} is also complemented. $S = X + \bar{Y} + \bar{c}_{in}$. The formula for n-bit subtraction is:

$$\mathrm{N}(X) + \mathrm{N}(\bar{Y}) + \mathrm{N}(C) + \bar{c}_{in} = \mathrm{N}(S) + c_{out} + \mathrm{N}(C) \quad (8)$$

In thus equation, $\mathrm{N}(\bar{Y}) = Y_c$. The Berger check symbol equation is:

$$S_c = X_c - Y_c + C_c - \bar{c}_{in}. + c_{out} \text{ (14).} \quad (9)$$

Below is an example of (6):

$X = 101011$ and $Y = 101101$ and $c_{in} = 0$

$S = 111110$, $C = 111100$, and $c_{out} = 0$

$X_c = 2$, $Y_c = 2$ and $C_c = 2$.

From (8) we know that:

$S_c = 2 - 2 + 2 - 1 = 1.$

We also know that S_c must equal n- N(S). N(S) = N(111110) = 5. Thus,

$S_c = n - \mathrm{N}(S) = 6 - 5 = 1$ [26].

Berger Check Prediction for Logical Operations

The equations for the three basic logic operations And ($\wedge$), OR($\vee$), and XOR($\oplus$) are listed below:

AND equation: $x_i \wedge y_i \equiv x_i + y_i - (x_i \vee y_i)$ (10)

OR equation: $x_i \vee y_i \equiv x_i + y_i - (x_i \wedge y_i)$ (11)

XOR equation: $x_i \oplus y_i \equiv x_i + y_i - 2(x_i \wedge y_i)$. (12)

Now, we determine a relationship between the numbers of 1's for (10)

$$N(X \wedge Y) = N(X) + N(Y) - N(X \vee Y) \quad (13)$$

A Berger check code for the AND operation can be derived from (13):

$$S_c = (X \wedge Y)_c = N(X)_c + N(Y)_c - N(X \vee Y)_c. \quad (14)$$

Berger check codes can be derived for OR and XOR operations similarly to the AND derivation.

Parity Prediction

Even and odd parity are two types of parity prediction. This thesis uses even parity to generate parity bits. When using even parity, the parity bit is set to 1 if there is an odd number of 1's in the operand or result. Since parity prediction only detects arithmetic operations, an addition example is provided below:

$X = 101$ and $Y = 010$

First computation step:

$X + Y = 111$

$P_S = 1$ (P is the parity bit).

Recomputation step:

$P_X = 0$, $P_Y = 1$

P_X XOR $P_Y = 0$ XOR $1 = 1 = P_C$

$P_S = P_C = 1$

If a fault occurs during the first computation, one of the bits is inverted. This changes the parity bit, P_S, to 0, which would flag an error when compared to the parity bit of the concurrent recomputation step.

Parity and Logic

The parity and logic error detection technique operates the same as the parity prediction circuit. The only difference occurs for a logic operation, in which the technique uses the duplicated logic unit to compare results with the ALU.

CHAPTER V

VHDL IMPLEMENTATION OF ERROR DETECTION TECHNIQUES

All error detection techniques discussed in Chapters III and IV were implemented with VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language, or VHDL, using Altera's Quartus II [27] software. VHDL is a hardware description language (HDL) that describes the behavior and structure of digital designs. It is used for a variety of digital systems ranging from a few gates to an interconnection of complex integrated circuits [28]. ModelSim-Altera [29] was used for simulation and debugging to verify correct behavior of the VHDL models.

RESO/RERO

RESO and RERO use two computation steps with each taking a cycle to complete. For both techniques, the first computation step involves the conventional operand undergoing ALU operations. The second computation either shifts or rotates the operands respectively. In order for the methods to be implemented correctly, a Mux is placed after the shifter or rotator, as shown in Figure 8 and Figure 9, to ensure the correct operand enters the ALU. A clock signal is fed to the select input for a 2-to-1 Mux, so the first computation runs when clock is low and the recomputation runs when clock is high.

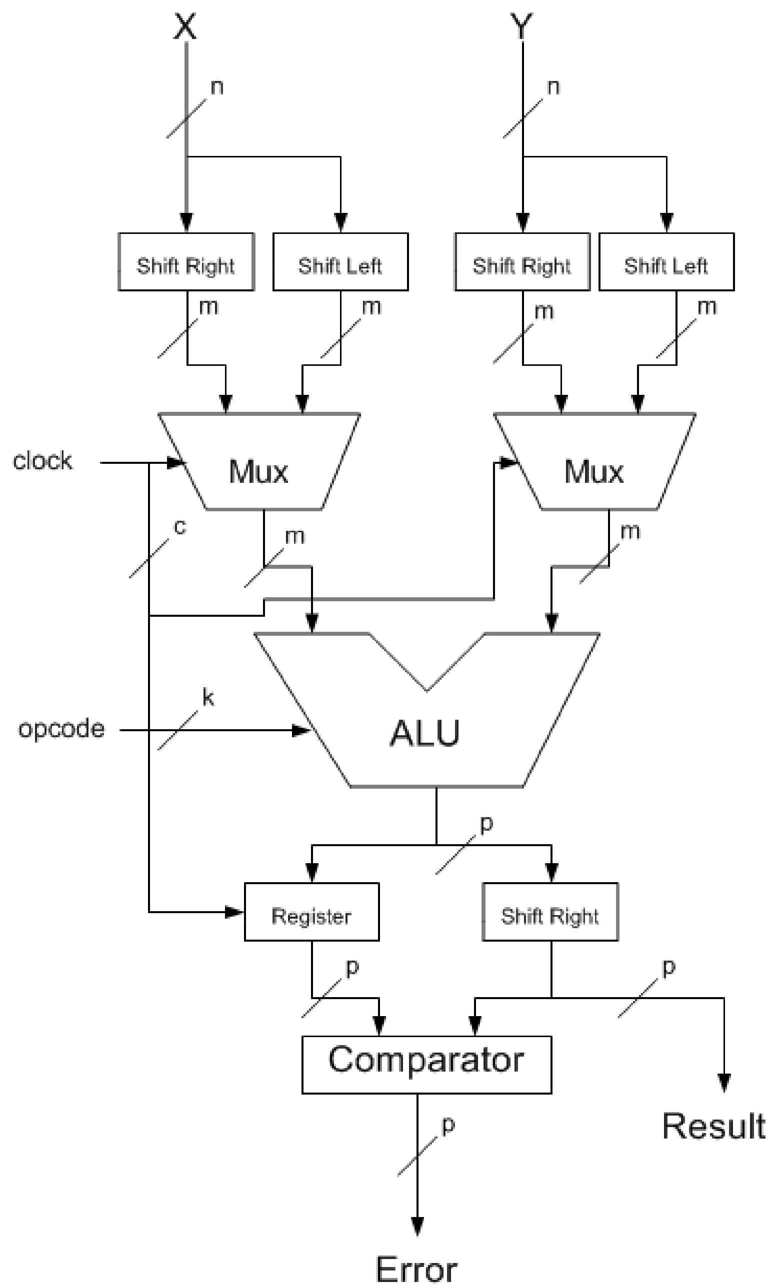


Figure 8: RESO implementation that ensures computation and recomputation steps.

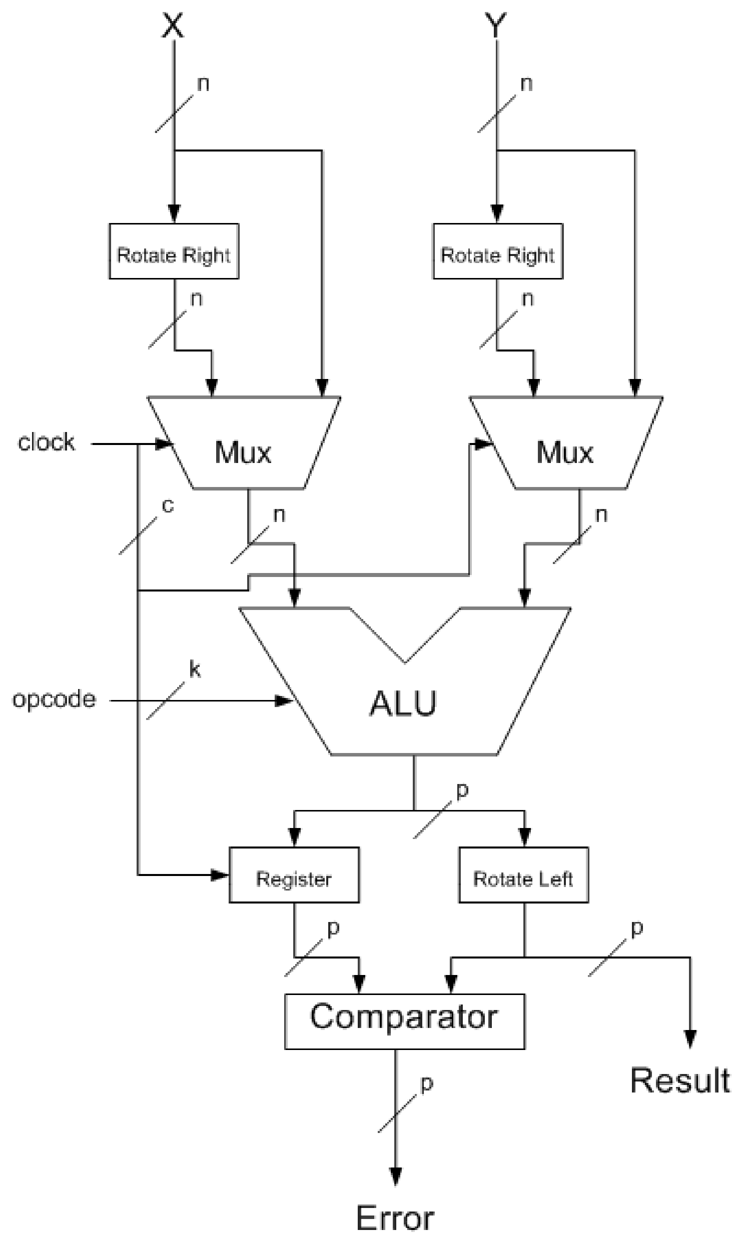


Figure 9: RERO implementation that ensures computation and recomputation steps.

High-speed Modulo-3 Generator

The high-speed modulo-3 generator produces residue codes through two modules for implementation. The general process of this technique is shown in Figure 10. An operand X (x_n, x_{n-1}, . . ., x_1, x_0) is partitioned into multiple 2-bit inputs for Module 1. Module 1 consists of two AND logic gates and four logic inverters. Module 1 is designed so that the first input receives a binary variable x_1 and the second input receives a binary input x_1 (mod3). For example, x_0 and x_1 are inputs for Module 1 and x_{n-1} and x_n or inputs for the last Module 1 if there is an even number of bits in X. If there is an odd number of bits in X, then the inputs to the last Module 1 are x_n and 0. Module 1, shown in Figure 10, has four outputs Y_0 to Y_3:

$$Y_0 = x_1 \bar{x}_0$$

$$Y_1 = \bar{Y}_0$$

$$Y_2 = \bar{x}_1 x_0$$

$$Y_3 = \bar{Y}_2.$$

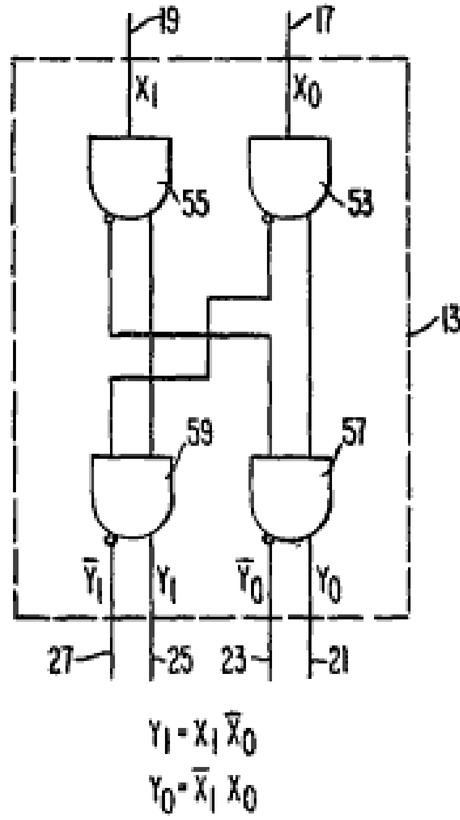


Figure 10: Module 1 of the high-speed modulo-3 generator [30].

Module 2, shown in Figure 11, consist of six AND and two OR logic gates. It has four outputs Z_0 to Z_3 and eight inputs from the outputs of two Module 1s (the first Module 1's outputs: Y_0 to Y_3 and the second Module 1's outputs: Y_4 to Y_7):

$$Z_0 = x_0\bar{x}_2\bar{x}_3 + x_2\bar{x}_0\bar{x}_1 + x_1x_3$$

$$Z_1 = \bar{Z}_0$$

$$Z_2 = x_3\bar{x}_1\bar{x}_0 + x_1\bar{x}_3\bar{x}_2 + x_2x_0$$

$$Z_3 = \bar{Z}_2.$$

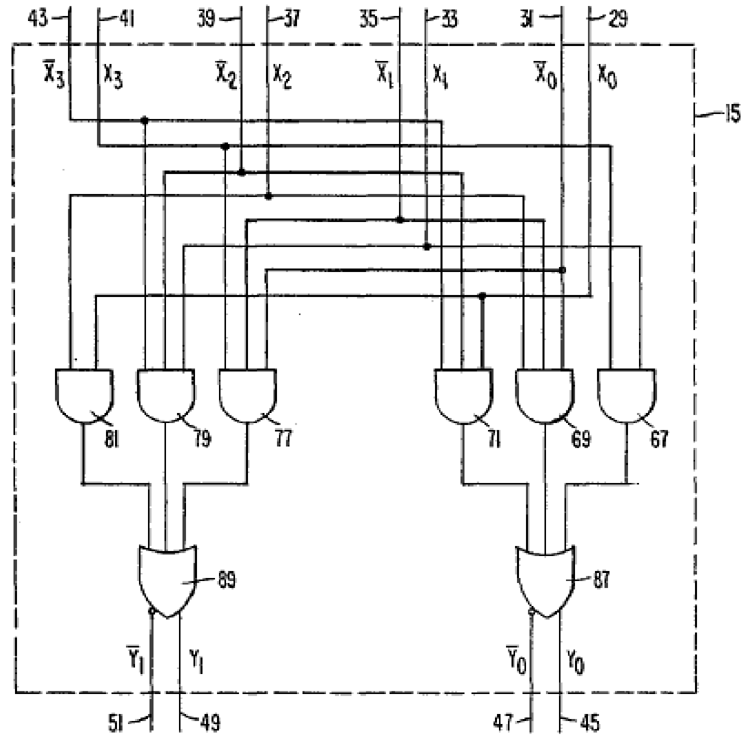


Figure 11: Module 2 of the high-speed modulo-3 generator [30].

A complete high-speed modulo-3 generator is constructed from a plurality of Module 1s followed by a logarithmic array of Module 2s [30]. The configuration of Module 1 and 2 in Figure 12 can provide modulo-3 generation for binary of any size.

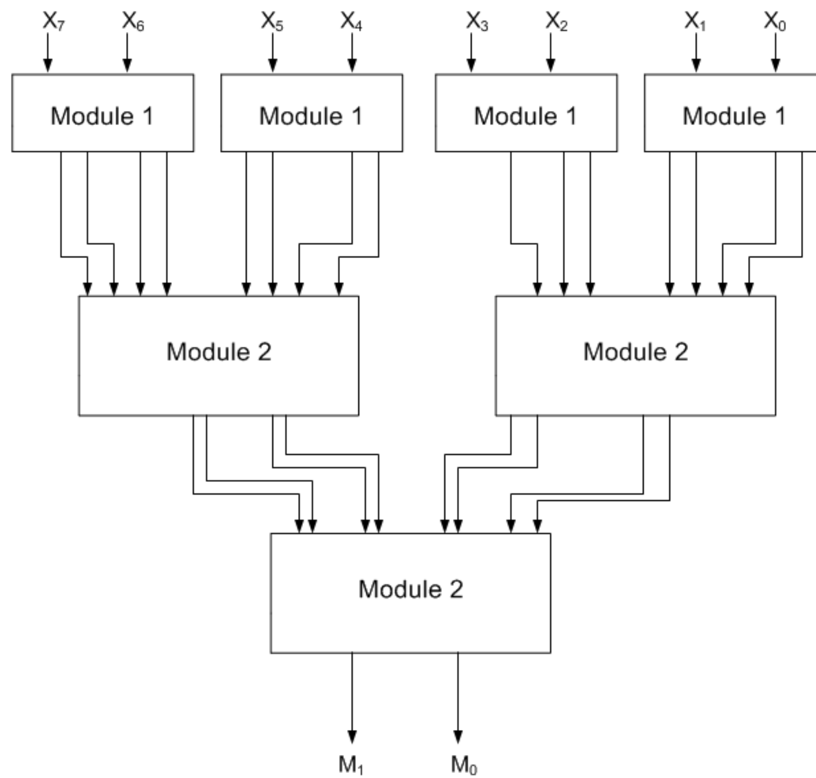


Figure 12: High-speed modulo-3 generator [30].

Berger

The first computation involves two operands entering an ALU and calculating a result. The number of zeros is then counted in the result. The second computation, which is concurrent with the first, is more complicated. Two operands are sent through Logical OR and Logical AND gates, where the outputs enter a Mux. The other input for the 3-to-1 Mux is the carry-out bits from the ALU. The Mux is controlled by select inputs which are dependent on the control programmable logic array (PLA), shown in Figure 13. The Mux's output is then sent to a zeros counter. This result and zeros count of the two operands ultimately enter the Multioperand Carry Save Adder (MCSA) in Figure 14. The

MCSA determines the Berger check code [31], which is compare to the zeros count of the ALU's result.

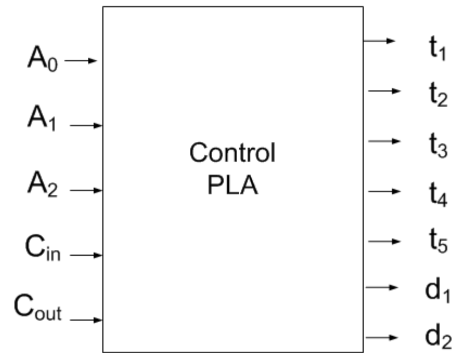


Figure 13. Control PLA of BCP circuit.

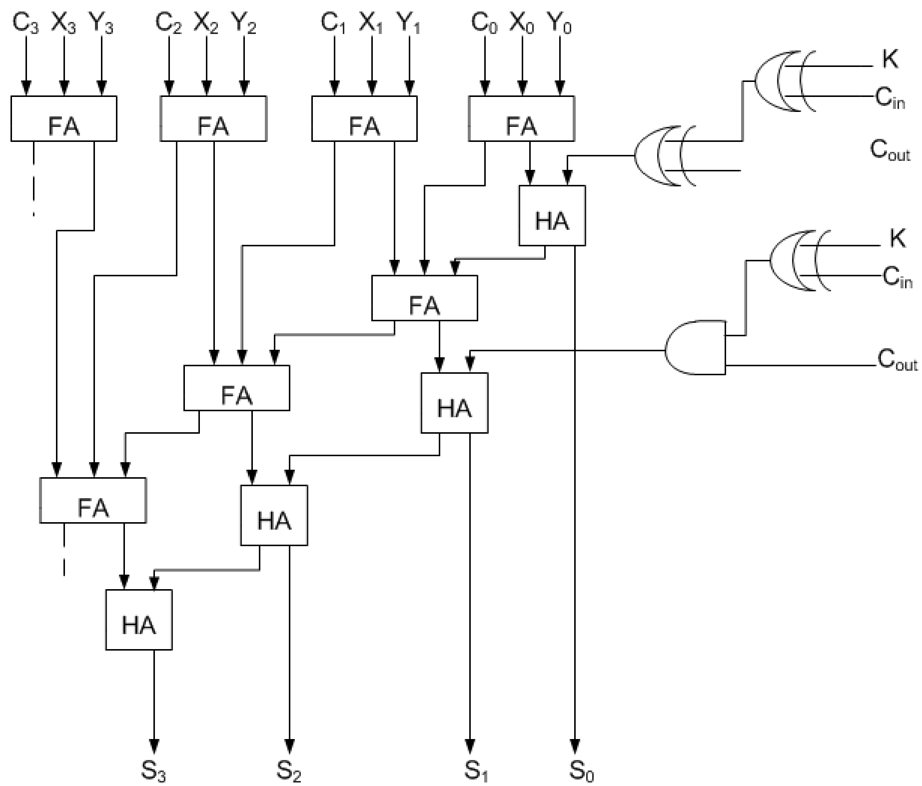


Figure 14: Multioperand Carry Save Adder [31].

Parity Prediction Circuit/ Parity and Logic Circuit

The parity generator in the parity prediction circuit and the parity and logic circuit use a chain of logic XOR gates to generate a parity bit. For the parity and logic circuit, the logic unit is a duplication of only the logic unit in the ALU. Figure 15 conveys the circuitry for generating an even parity bit for an operand *X*.

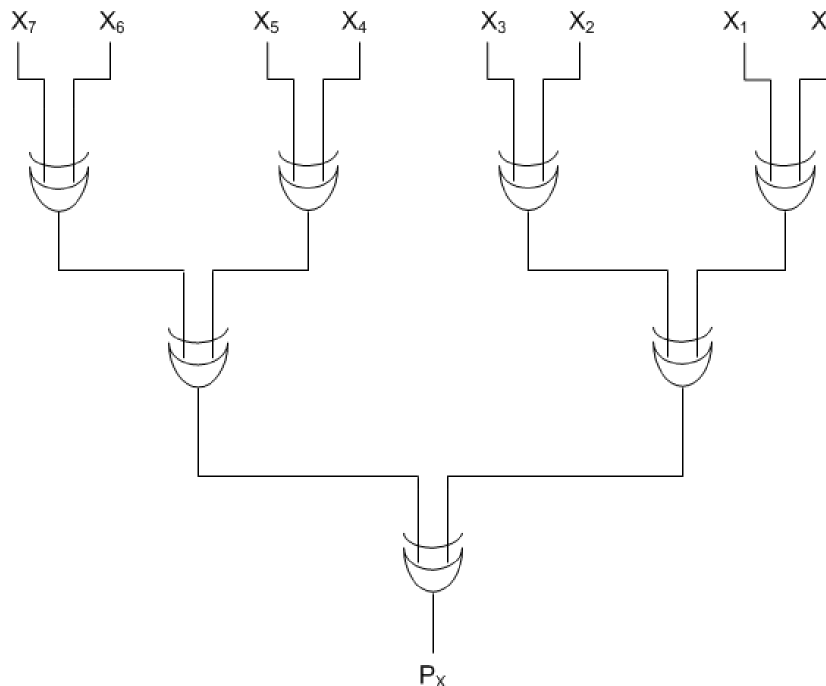


Figure 15. Chain of Logic XOR gates that generate even parity.

Synthesis

VHDL models were synthesized using the FreePDK45 cell library [32] and Cadence Encounter Register Trasnfer Level (RTL) complier. The FreePDK45 cell library was developed by the Oklahoma State University VLSI Computer Architecture Group. It consists of 33 cells with a 45-nm transistor size. The FreePDK45 library was chosen because it was an open-source implementation of a current fabrication technology. Area,

power consumption, and maximum time delay reports are generated via the RTL compiler, cell library, and VHDL models. The area of each gate (cell) and wiring between gates produce a total area report for each error detection technique. The timing report uses the longest path in the VHDL model to generate the maximum timing delay. The RTL compiler sums the inertial (gate) delay of each cell and transport (wire) delay along the longest path. The power report uses voltage drops associated with a particular gate to calculate power consumption. Simultaneous switching logic can cause high transients of dynamic voltage drops for power rails [33], which may drastically increase power consumption.

CHAPTER VI

RESULTS

The focus of this thesis is to compare the area, timing, and power penalties of error detection techniques for an *n*-bit ALU. First, the penalties for each technique are compared amongst each other. Then, penalties are compared for a particular technique for different ALU sizes.

Area, Timing, and Power Comparisons for All Techniques

Area Comparison

Error detection techniques are compared to a baseline ALU in Table 1 and Figure 16. Table 1 shows all of the error detection techniques are around 2X that of the baseline ALU, with the exception of RESO, RERO, and Parity. The area penalties for RESO and RERO are less than 2X of the baseline ALU since they use the same hardware for recomputation. Parity error detection incurs the smallest area penalty, since it does not allow error detection for logic operators. Including logic operators would increase the area. Pargic is an abbreviation for parity and logic error detection technique. Pargic experiences a larger area penalty than Parity because it includes error detection for logic operators. However, the high-speed modulo-3 generator area penalty is much larger than Parity, yet it only detects arithmetic errors. Figure 16 shows the area percent overhead. DMR percent overhead for all ALU sizes is consistently 120% more than the baseline ALU. The duplicated ALU uses 100% more area while the comparator uses 20% additional area.

	Area (mm^2)			
	8-bit ALU	**16-bit ALU**	**32-bit ALU**	**64-bit ALU**
ALU	477	914	1778	3564
DMR	1037	2002	3899	7816
Mod3	987	1959	3898	7793
Berger	894	1927	3552	7332
RESO	858	1549	2826	5617
RERO	786	1449	2858	5687
Parity	592	1163	2292	4585
Pargic	858	1689	3354	6670

Table 1: Area penalty for an *n*-bit ALU with error detection.

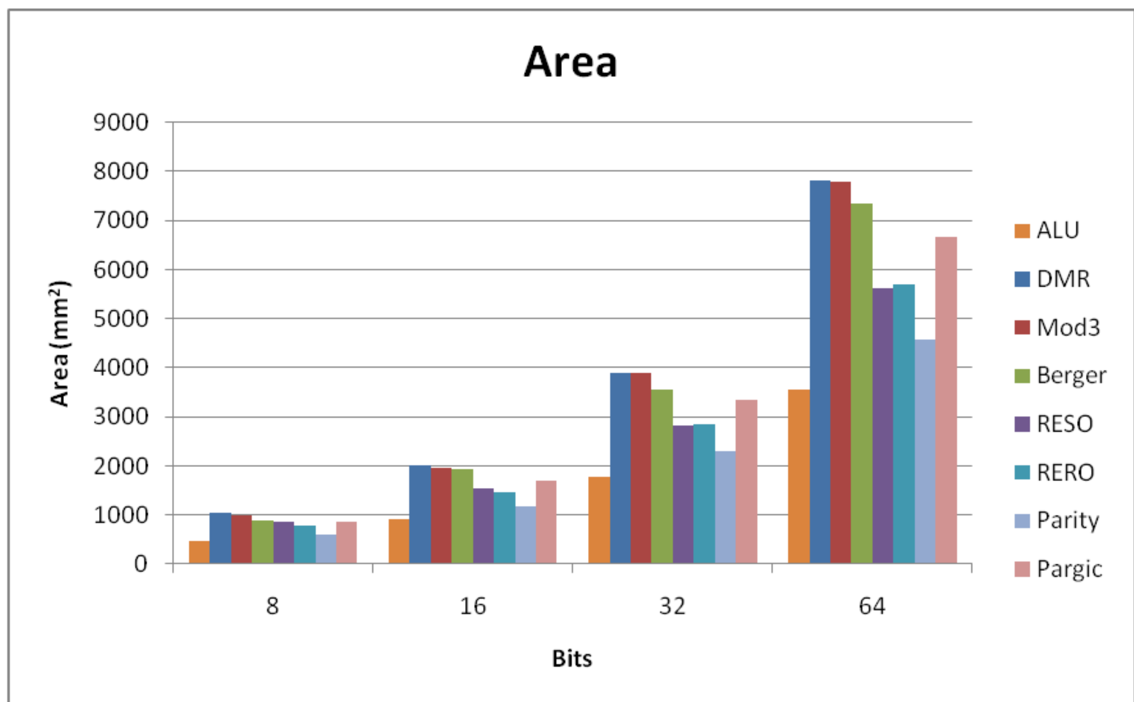


Figure 16: Area overhead with an ALU as the baseline.

Timing Comparison

Table 2 show the raw timing results for each design. DMR is consistently the fastest error detection technique since it uses two concurrent computations. ALU timing

results differ from that of DMR because of the DMR's comparison stage. Figure 17 shows the percentage in additional timing for all techniques when compared to the ALU baseline. RESO and RERO are at least 2X slower than the baseline ALU due to sequential computations. Parity results are similar to DMR. The difference in timing results for Parity and DMR occurs when the parity bit of the ALU's result is generated.

	Time (ps)			
	8-bit ALU	**16-bit ALU**	**32-bit ALU**	**64-bit ALU**
ALU	898	1727	3398	6716
DMR	1157	2046	3693	7052
Mod3	1611	2695	4514	7990
Berger	1284	2282	4178	7651
RESO	2194	3926	7387	14005
RERO	2156	3840	7144	13711
Parity	1163	2050	3780	7100
Pargic	1232	2095	3850	7169

Table 2: Timing results for an n-bit ALU with error detection.

.

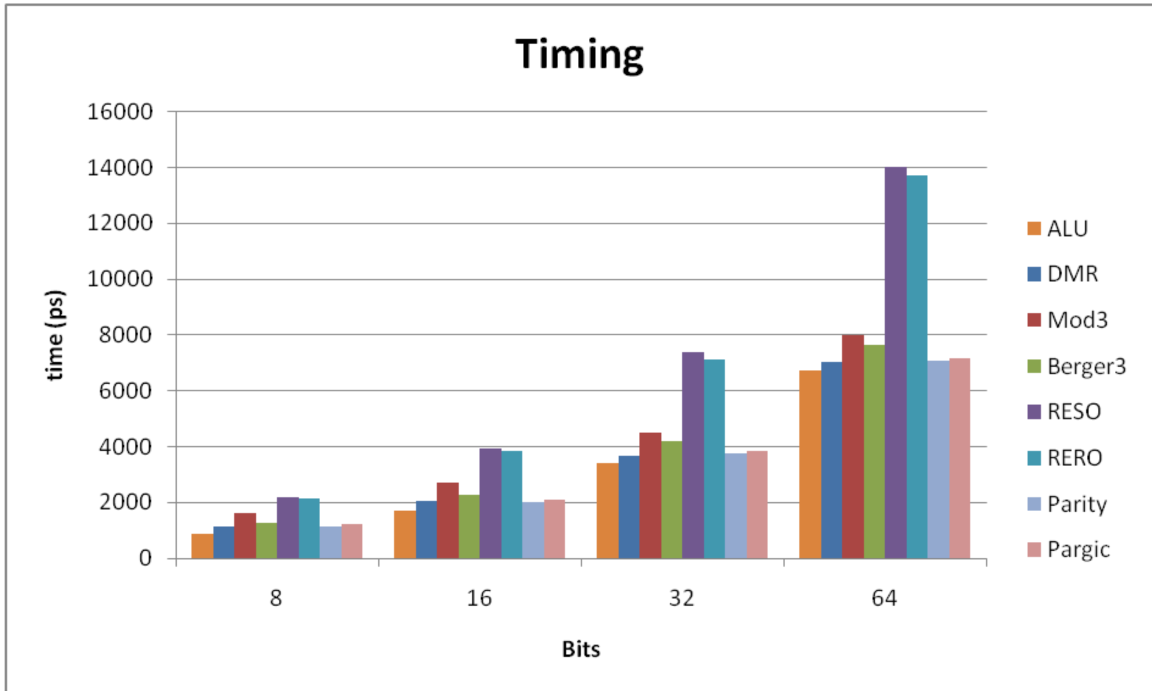


Figure 17: Timing overhead with an ALU as the baseline.

Power Comparison

Power consumption results are shown in Table 3. DMR power consumption overhead is very consistent for all ALU sizes. RESO and RERO consume less power than all error detection techniques, except for Parity, for all *n*-bit ALUs. One could assume that Parity should use far less power than RESO and RERO because of low area penalty. However, the additional power is due to many concurrent computations. As stated previously, simultaneous switching increases power consumption. Figure 18 conveys that BCP consumes more power as an ALU increases. Power consumption for BCP ranges from 186% to 327% more power than the baseline ALU. BCP experiences a drastic

power penalty for the 64-bit ALU because of several concurrent computations. The larger zeros counter causes the large increase in power consumption.

	Power (μW)			
	8-bit ALU	16-bit ALU	32-bit ALU	64-bit ALU
ALU	39	87	170	339
DMR	100	200	403	821
Mod3	116	249	546	1025
Berger	142	295	568	1450
RESO	85	171	324	669
RERO	86	167	346	664
Parity	72	152	313	660
Pargic	115	242	502	1036

Table 3: Power results for an *n*-bit ALU with error detection.

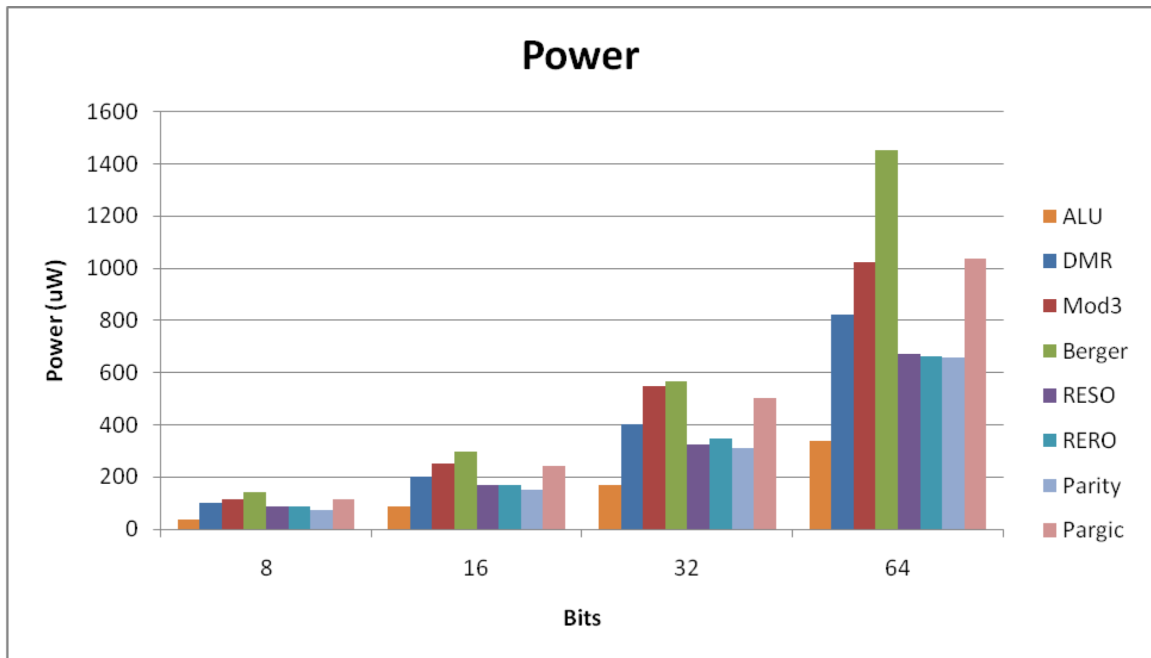
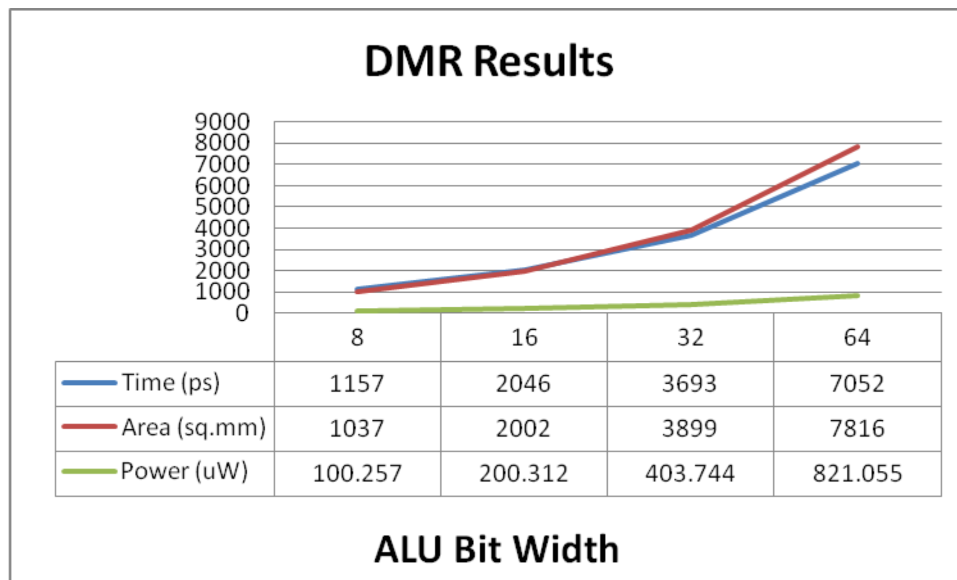


Figure 18: Power overhead with an ALU as the baseline.

Area, Time, and Power Comparisons for All Techniques

DMR Results

Figure 18 shows the timing, area, and power penalties for DMR error detection. DMR synthesis was provided for 8-, 16-, 32-, and 64-bit ALUs. Power consumption and area doubles as the ALU size doubles. Yet, DMR incurs less than a 2X timing penalty as the ALU size doubles. Since computations are bit-wise and concurrent, the timing discrepancies manifest for different comparator sizes. Figure 20 provides the overhead penalty for DMR error detection when compared to a baseline ALU. For understanding, DMR error detection for a 16-bit ALU requires 119.04% more area than a 16-bit ALU. The graph shows that area overhead is constant for an *n*-bit ALU since the ALU and comparator doubles. However, the maximum timing delay decreases as the ALU size increases.



	8	16	32	64
Time (ps)	1157	2046	3693	7052
Area (sq.mm)	1037	2002	3899	7816
Power (uW)	100.257	200.312	403.744	821.055

Figure 19: Timing, area, and power results for an *n*-bit ALU using DMR error detection.

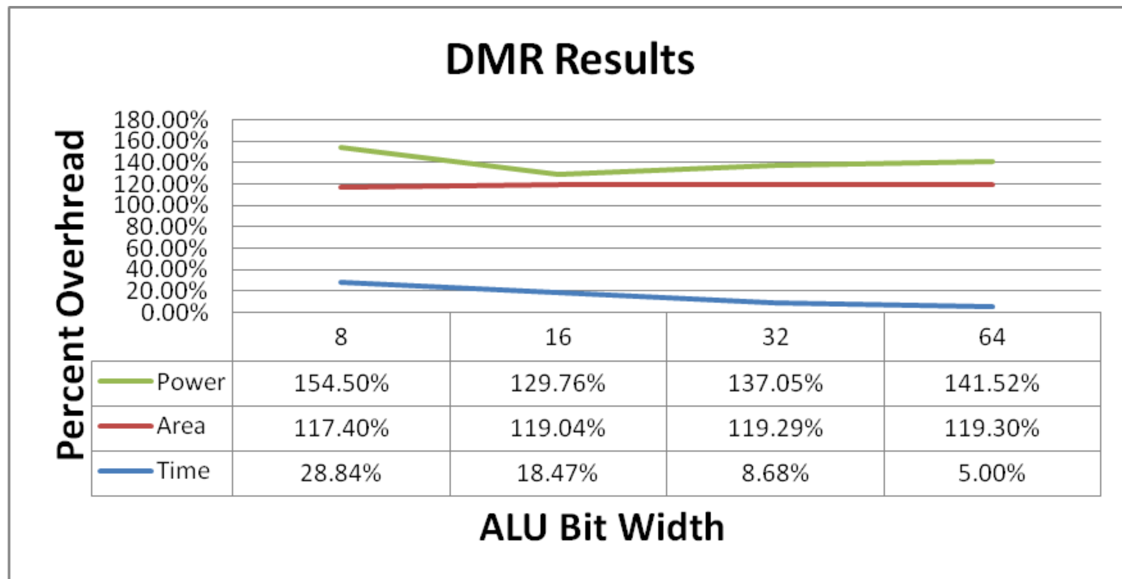
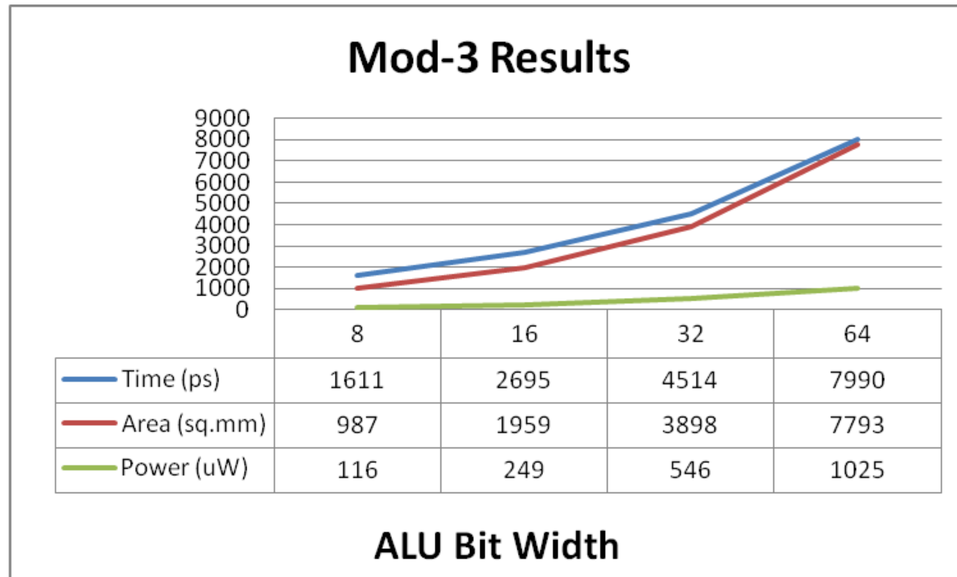


Figure 20: Timing, area, and power overhead for an *n*-bit ALU using DMR error detection.

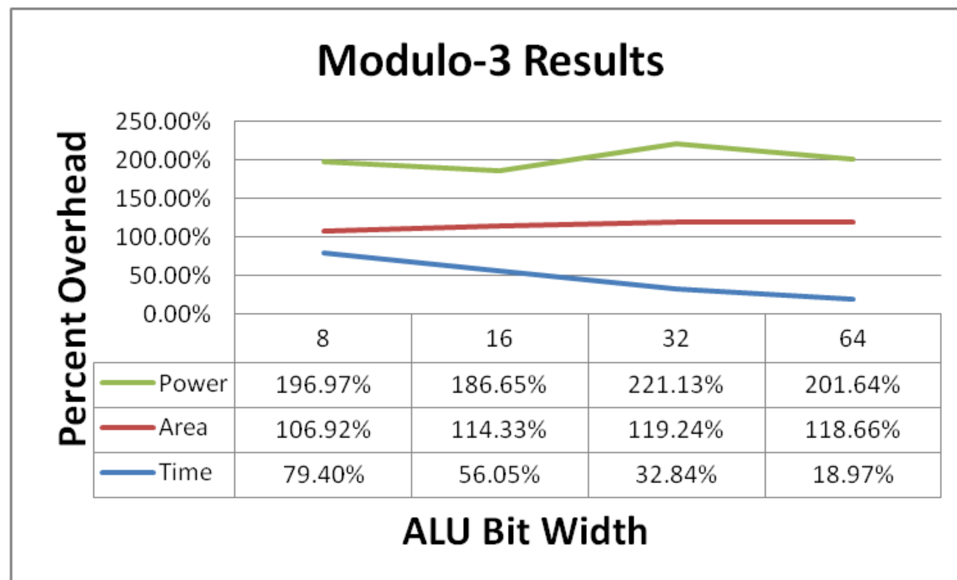
High-speed Modulo-3 Generator Comparison

Figure 21 shows the raw Modulo-3 results and Figure 19 shows the percent overhead for time, area, and power for an *n*-bit ALU using Modulo-3 error detection. Figure 22 shows that area overhead is almost constant for an *n*-bit ALU and the time overhead decreases as ALU size increases. The power overhead averages around 200%.



	8	16	32	64
Time (ps)	1611	2695	4514	7990
Area (sq.mm)	987	1959	3898	7793
Power (uW)	116	249	546	1025

Figure 21: Timing, area, and power results for an *n*-bit ALU using Mod-3 error detection.

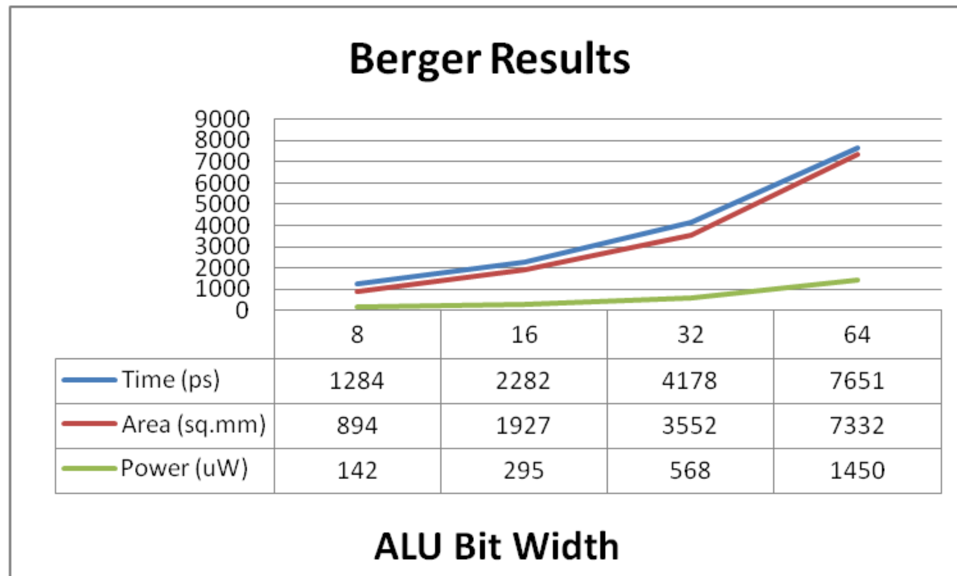


	8	16	32	64
Power	196.97%	186.65%	221.13%	201.64%
Area	106.92%	114.33%	119.24%	118.66%
Time	79.40%	56.05%	32.84%	18.97%

Figure 22: Timing, area, and power overhead for an *n*-bit ALU using Modulo-3 error detection.

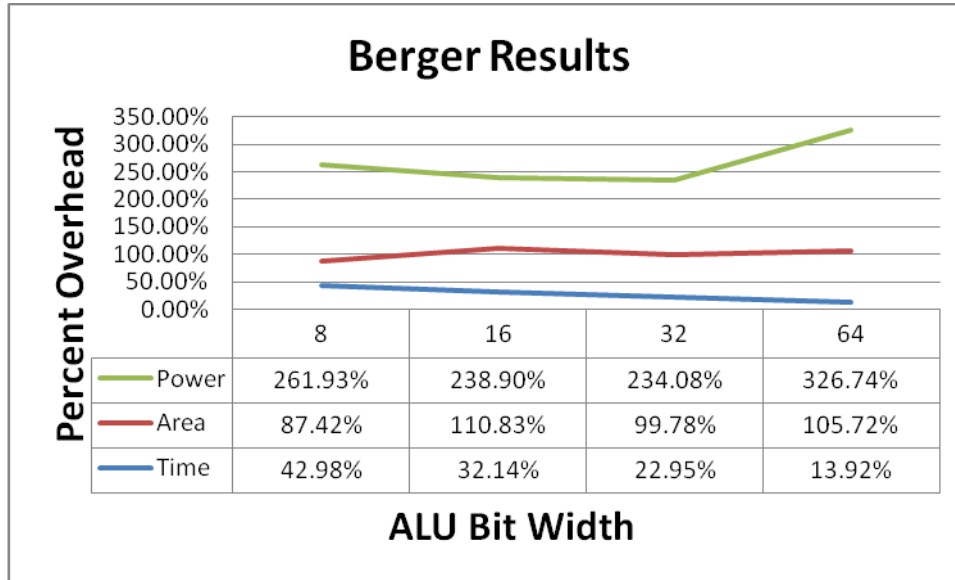
Berger Comparison

Figure 23 shows the raw data for Berger error detection and Figure 24 shows the percent overhead for time, area, and power for an *n*-bit ALU. The graph shows that area overhead is almost constant for all ALU sizes. The maximum time delay decreases as the ALU increases. The power overhead increases as the ALU size increases. The large timing penalty for the BCP error detection for a 64-bit ALU is due to the increase in power consumption of the zeroes counter. One zeros counter for a 32-bit ALU consumes 73 μW, and a zeros counter for a 64-bit ALU consumes 227 μW. The increase in simultaneous switching logic from the 32-bit ALU to the 64-bit ALU causes extreme power penalty.



	8	16	32	64
Time (ps)	1284	2282	4178	7651
Area (sq.mm)	894	1927	3552	7332
Power (uW)	142	295	568	1450

Figure 23: Timing, area, and power results for an *n*-bit ALU using Berger error detection.



	8	16	32	64
Power	261.93%	238.90%	234.08%	326.74%
Area	87.42%	110.83%	99.78%	105.72%
Time	42.98%	32.14%	22.95%	13.92%

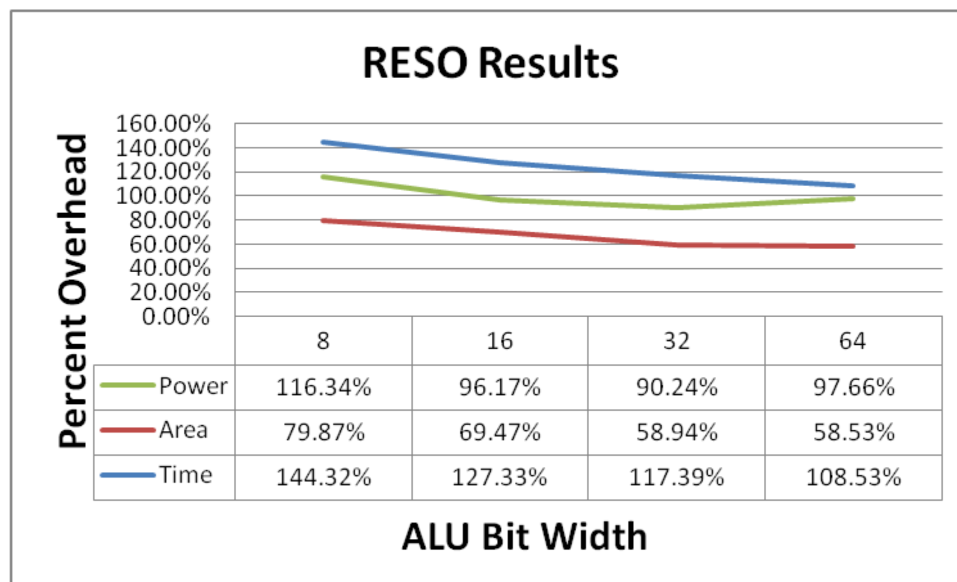
Figure 24: Timing, area, and power overhead for an *n*-bit ALU using Berger error detection.

RESO Comparison

Figure 25 shows the raw data for RESO error detection and Figure 26 shows the percent overhead for time, area, and power for an *n*-bit ALU. The graph shows that area overhead, time delay, and power consumption decrease as ALU size increases.

	8	16	32	64
Time (ps)	2194	3926	7387	14005
Area (sq.mm)	858	1549	2826	5617
Power (uW)	85	171	324	669

Figure 25: Timing, area, and power results for an n-bit ALU using RESO error detection.



	8	16	32	64
Power	116.34%	96.17%	90.24%	97.66%
Area	79.87%	69.47%	58.94%	58.53%
Time	144.32%	127.33%	117.39%	108.53%

Figure 26: Timing, area, and power overhead for an n-bit ALU using RESO error detection.

RERO Comparison

Figure 27 shows the raw data for RERO error detection and Figure 28 shows the percent overhead for time, area, and power for an *n*-bit ALU. The graph shows that area overhead and power consumption are almost constant for all ALU sizes. Yet, time delay decreases as the ALU increases.

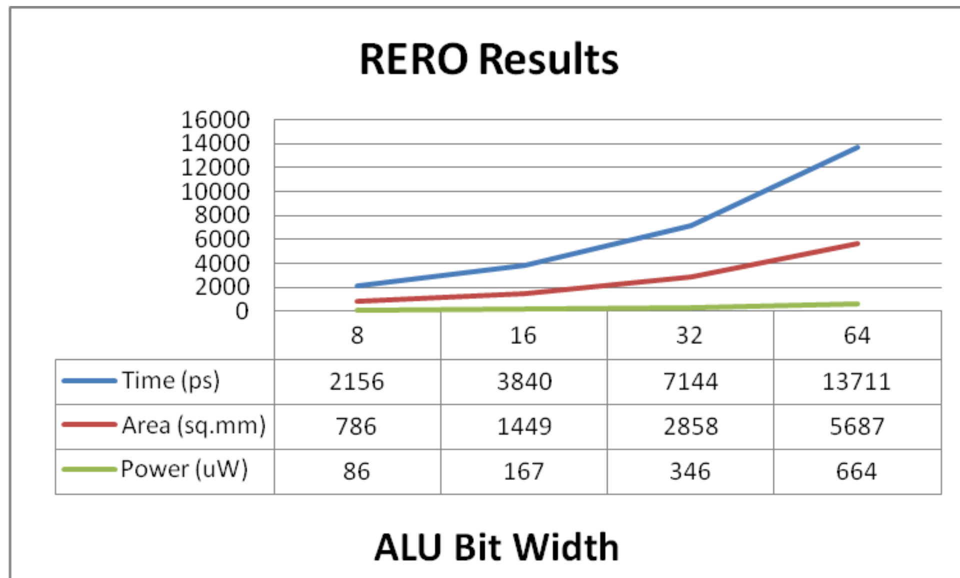
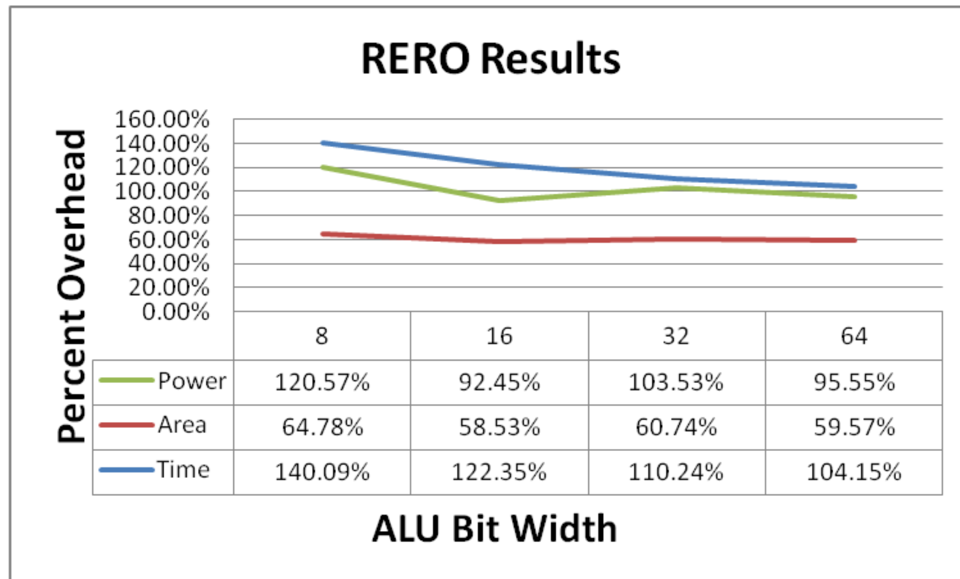


Figure 27: Timing, area, and power results for an *n*-bit ALU using RERO error detection.



	8	16	32	64
Power	120.57%	92.45%	103.53%	95.55%
Area	64.78%	58.53%	60.74%	59.57%
Time	140.09%	122.35%	110.24%	104.15%

Figure 28: Timing, area, and power overhead for an *n*-bit ALU using RERO error detection.

Parity Comparison

Figure 29 shows the raw data for Parity error detection and Figure 30 shows the percent overhead for time, area, and power for an *n*-bit ALU. The graph shows that area and power consumption almost doubles for all ALU sizes. Also, time delay overhead decreases as the ALU increases.

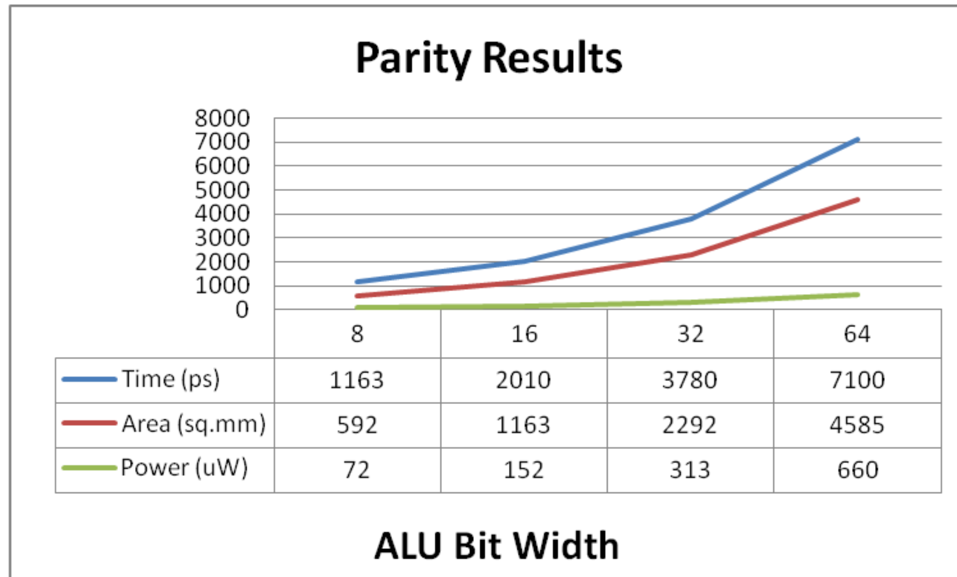


Figure 29: Timing, area, and power results for an *n*-bit ALU using Parity error detection.

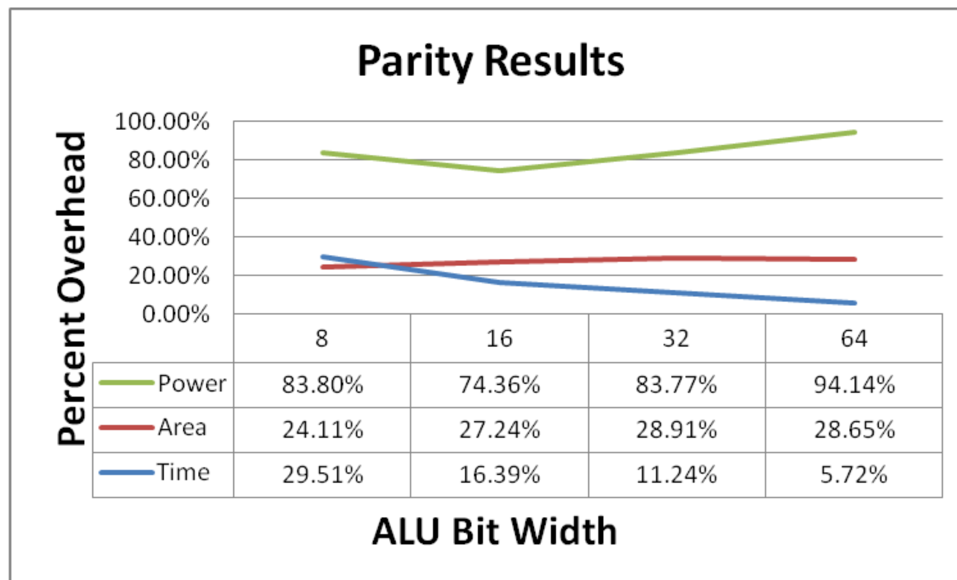
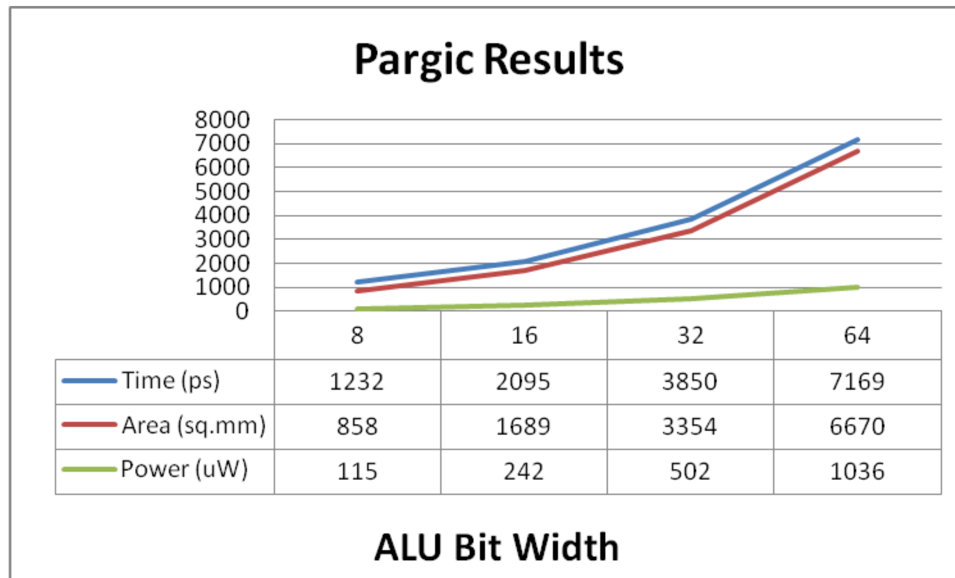


Figure 30: Timing, area, and power overhead for an *n*-bit ALU using Parity error detection.

Parity and Logic Comparison

Figure 31 shows the raw data for Parity and Logic error detection and Figure 32 shows the percent overhead for time, area, and power for an *n*-bit ALU. The graph shows that area overhead and power consumption are almost constant for all ALU sizes. Time delay decreases as the ALU increases.



	8	16	32	64
Time (ps)	1232	2095	3850	7169
Area (sq.mm)	858	1689	3354	6670
Power (uW)	115	242	502	1036

Figure 31: Timing, area, and power results for an *n*-bit ALU using Parity error detection.

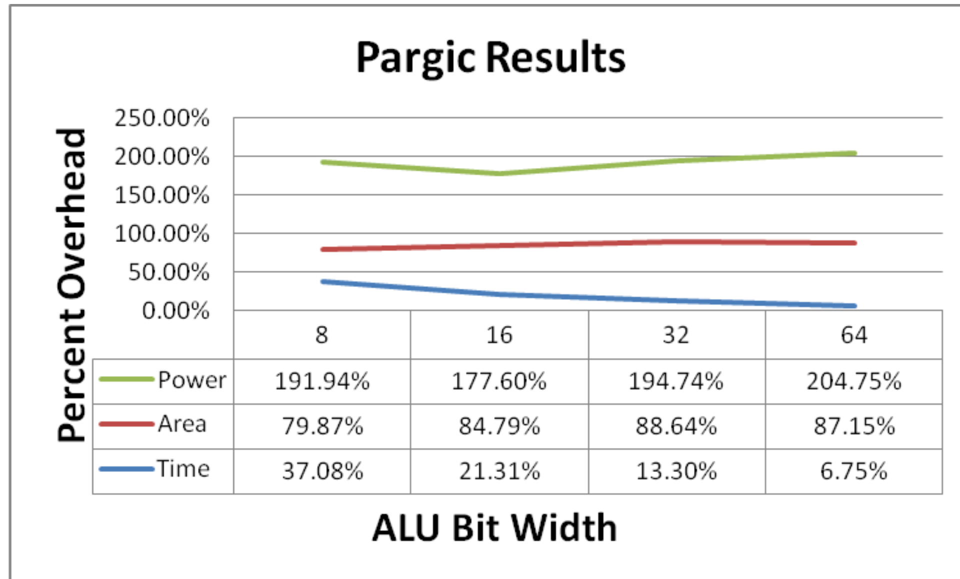


Figure 32: Timing, area, and power overhead for an *n*-bit ALU using Parity error detection.

Implications of the Error Detection Techniques on Processing Systems

DMR is the simplest error detection technique to implement. Computer architects search for techniques that consume less power and use less area than DMR. No technique will be faster than DMR, yet the goal is to approach its timing capabilities. Most importantly, an ALU with additional functionality may lead to different results than those presented in this thesis. Remember the operations for the ALU in this thesis are the same as those protected by BCP. If area and power are of concern for a computer architect and both were weighted equally, then RERO is a sufficient error detection technique. RERO detects error for both arithmetic and logic operations, unlike Parity error detection. Parity area penalty may be smaller than that of RERO, but it does not protect logic operations. If area and time are of concern for a computer architect and both are weighted equally, then Parity and Logic error detection is a sufficient error detection method because the time

penalty of RERO exceeds the power penalty of Parity and Logic. Lastly, if power and time are of concern and both are weighted equally, then DMR is a sufficient error detection technique because the time penalty of RERO exceeds the power penalty of DMR.

Chapter VII

Conclusion and Future Exploration

This thesis provides timing, area, and power reports for five error detection techniques from three different error detection categories: redundancy codes, arithmetic codes, and parity codes. Using the 45-nm cell library and Encounter RTL Compiler enabled a study to accurately synthesize the VHDL models and compare their results. When comparing each technique individually for an *n*-bit ALU, area overhead is almost consistent and the maximum timing delay decreases as the ALU's size increases. Yet, power consumption for all techniques did not show a particular trend. Additional dynamic power analysis would need to be conducted to account for the switching activity factor. Moreover, there exist many more error detection techniques than the ones analyzed in this thesis. The goal was to synthesize techniques (or ones from the same category) being used today. Future exploration of this thesis could involve pipelining the time redundancy techniques in order to improve timing delays.

APPENDIX A

DMR VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;



entity DMR is
	port
	(
		-- Input ports
		A				: in std_logic_vector(7 downto 0);
		B				: in std_logic_vector(7 downto 0);
		--input A with injected fault
		A_f				: in std_logic_vector(7 downto 0);
		--input B with injected fault
		B_f				: in std_logic_vector(7 downto 0);
		opcode			: in std_logic_vector(2 downto 0);

		-- Output ports
		S				: out std_logic_vector(8 downto 0);
		error			: out std_logic );
end DMR;


architecture structure of DMR is

		--instance alu
component alu8bit
port
	(
		-- alu inputs
		A				: in std_logic_vector(7 downto 0);
		B				: in std_logic_vector(7 downto 0);
		opcode			: in std_logic_vector(2 downto 0);

		-- alu outputs
		C				: out std_logic_vector(8 downto 0)
			);
end component;
```

```
                --internal wires

signal  S1                          : std_logic_vector(8 downto 0);
signal  S_f                         : std_logic_vector(8 downto 0);
signal errorbus                     : std_logic_vector(8 downto 0);

                --connect entities

begin

alu: alu8bit                        port map (A=>A, B=>B, opcode=>opcode, C=>S1);
alu_f: alu8bit                      port map (A=>A_f, B=>B_f, opcode=>opcode, C=>S_f);


S <= S1; --DMR output

                -- Comparator
Errorbus        <= s1 xor s_f;
error           <= errorbus(8) or errorbus(7) or  errorbus(6) or  errorbus(5) or  errorbus(4)
                        or  errorbus(3) or  errorbus(2) or  errorbus(1) or  errorbus(0);

end structure;


LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


ENTITY alu8bit is
        port(

            -- alu inputs
                A                       : in std_logic_vector(7 downto 0);
                B                       : in std_logic_vector(7 downto 0);
                opcode                  : in std_logic_vector(2 downto 0);

                -- alu outputs
                C                       : out std_logic_vector(8 downto 0)
                );
END alu8bit;
```

```
architecture behavior of alu8bit is
begin
	process(opcode, a, b)
	begin
		IF opcode = "000" THEN
			C <= ('0'& A) + ('0' & B);   -- add
		ELSIF opcode = "001" THEN
			C <= ('0' & A) - ('0' & B);  -- subtract
		ELSIF opcode = "010" THEN
			C <= ('0' & A) and ('0' & B);  -- and
		ELSIF opcode = "011" THEN
			C <= ('0' & A) or ('0' & B);   -- or
		ELSIF opcode = "100" THEN
			C <= ('0' & A) xor ('0' & B);   -- xor
		ELSE
			C <= ('0' & A) xor ('0' & B);
		END IF;
	end process;
end behavior;
```

APPENDIX B

MODULO-3 VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity Mod3 is
	port
	(
		-- Input ports
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		A_f			: in std_logic_vector(7 downto 0);
		B_f			: in std_logic_vector(7 downto 0);
		opcode			: in std_logic_vector(2 downto 0);
		-- Output ports
		C			: out std_logic_vector(8 downto 0);
		error			: out std_logic);
end Mod3;


architecture structure of Mod3 is

		--instance alu
component alu8bit
port
	(
		-- alu inputs
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		Opcode			: in std_logic_vector(2 downto 0);

		-- alu outputs
		C			: out std_logic_vector(8 downto 0));
end component;
```

```
        --instance modulo-3 residue generator for alu result

component modulo3alu
port(

         -- modulo3 inputs
        C                               : in std_logic_vector(8 downto 0);
         -- modulo3 outputs
        modulo3aluout                   : out std_logic_vector(1 downto 0)

        );
end component;


                --instance modulo-3 residue generator for operands
component modulo3input
        port(

                -- modulo3 inputs
                A                       : in std_logic_vector(7 downto 0);
                -- modulo3 outputs
                modulo3out              : out std_logic_vector(1 downto 0)
                );
end component;


                --instance residue alu

component alu4bit
port
        (
                -- alu inputs
                A               : in std_logic_vector(1 downto 0);
                B               : in std_logic_vector(1 downto 0);
                opcode          : in std_logic_vector(2 downto 0);
                AluModOut       : in std_logic_vector(1 downto 0);

                -- alu outputs
                C               : out std_logic_vector(1 downto 0)
                        );
end component;

                -- internal wires

signal alu8out                  :  std_logic_vector(8 downto 0);
signal alu4wire1                :  std_logic_vector(1 downto 0);
```

```
signal alu4wire2                    :  std_logic_vector(1 downto 0);
signal comp2                        :  std_logic_vector(1 downto 0);
signal comp1                        :  std_logic_vector(1 downto 0);

                -- connect entities

begin

alu8: alu8bit                 port map (A=>A, B=>B, opcode=>opcode, C=>alu8out);
m3alu: modulo3alu             port map (C=>alu8out, modulo3aluout=>comp1);
m31: modulo3input             port map (A=>A_f, modulo3out=>alu4wire1);
m32: modulo3input             port map (A=>B_f, modulo3out=>alu4wire2);
alu4: alu4bit                 port map (A=>alu4wire1, B=>alu4wire2, opcode=>opcode,

AluModOut=>comp1, C=>comp2);

                -- comparator

error <= (comp1(1) xor comp1(0)) or (comp2(1) xor comp2(0));


C <= alu8out;

end structure;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


ENTITY alu8bit is
        port(

            -- alu8bit inputs
                A                       : in std_logic_vector(7 downto 0);
                B                       : in std_logic_vector(7 downto 0);
                opcode                  : in std_logic_vector(2 downto 0);

                -- alu8bit outputs
                C                       : out std_logic_vector(8 downto 0)
                );
END alu8bit;

architecture behavior of alu8bit is
begin
```

```
        process(opcode, a, b)
        begin
                IF opcode = "000" THEN
                        C <= ('0'& A) + ('0' & B);   -- add
                ELSIF opcode = "001" THEN
                        C <= ('0' & A) - ('0' & B);  -- subtract
                ELSIF opcode = "010" THEN
                        C <= ('0' & A) and ('0' & B);  -- and
                ELSIF opcode = "011" THEN
                        C <= ('0' & A) or ('0' & B);   -- or
                ELSIF opcode = "100" THEN
                        C <= ('0' & A) xor ('0' & B);   -- xor
                ELSE
                        C <= ('0' & A) xor ('0' & B);
                END IF;

        end process;
end behavior;

ENTITY alu4bit is
        port(

           -- alu4bit inputs
                A                       : in std_logic_vector(1 downto 0);
                B                       : in std_logic_vector(1 downto 0);
                opcode                  : in std_logic_vector(2 downto 0);
                AluModOut               : in std_logic_vector(1 downto 0);

                -- alu4bit outputs
                C                       : out std_logic_vector(1 downto 0)
                );
END alu4bit;

architecture behavior of alu4bit is
begin
        process(opcode, a, b)
        begin
        if opcode = "000" then
                C <= a + b;
        elsif opcode = "001" then
                C <= a - b;
        else
                C <= AluModOut;
        end if;
        end process;
end behavior;
```

```
ENTITY modulo3input is
	port(

	-- alu8bit inputs
	A						: in std_logic_vector(7 downto 0);
	-- alu8bit outputs
	modulo3out				: out std_logic_vector(1 downto 0)
		);
END modulo3input;


architecture behavior of modulo3input is
signal
	x	:  std_logic_vector(15 downto 0);
signal
	z	:  std_logic_vector(7 downto 0);
begin

	-- module1 outputs
	x(0)	<= a(0) and not a(1);
	x(1)	<= not x(0);
	x(2)	<= a(1) and not a(0);
	x(3)	<= not x(2);
	-- module1 outputs
	x(4)	<= a(2) and not a(3);
	x(5)	<= not x(4);
	x(6)	<= a(3) and not a(2);
	x(7)	<= not x(6);

	--module1 outputs
	x(8)	<= a(4) and not a(5);
	x(9)	<= not x(8);
	x(10)	<= a(5) and not a(4);
	x(11)	<= not x(10);
	-- module1 outputs
	x(12)	<= a(6) and not a(7);
	x(13)	<= not x(12);
	x(14)	<= a(7) and not a(6);
	x(15)	<= not x(14);

	--module2 outputs
	z(0)	<= (x(2) and x(6)) or (x(1) and x(3) and x(4)) or (x(0) and x(5) and x(7));
	z(1)	<= not z(0);
	z(2)	<= (x(4) and x(0)) or (x(7) and x(5) and x(2)) or (x(6) and x(3) and x(1));
	z(3)	<= not z(2);
```

```
z(4)    <= (x(10) and x(14)) or (x(9) and x(11) and x(12)) or (x(8) and x(13) and x(15));
z(5)    <= not z(4);
z(6)    <= (x(12) and x(8)) or (x(15) and x(13) and x(10)) or (x(14) and x(11) and x(9));
z(7)    <= not z(6);

--module2 outputs
modulo3out(0) <= (z(2) and z(6)) or (z(1) and z(3) and z(4)) or (z(0) and z(5) and z(7));
modulo3out(1)<= (z(4) and z(0)) or (z(7) and z(5) and z(2)) or (z(6) and z(3) and z(1));

end behavior;

ENTITY modulo3alu is
        port(

         -- alu8bit inputs
        C                               : in std_logic_vector(8 downto 0);
        -- alu8bit outputs
        modulo3aluout                   : out std_logic_vector(1 downto 0)
                );
END modulo3alu;


architecture behavior of modulo3alu is
signal
        x       :  std_logic_vector(15 downto 0);
signal
        z       :  std_logic_vector(7 downto 0);
signal
        w       :  std_logic_vector(7 downto 0);
begin

        -- module1 outputs
        x(0)    <= c(0) and not c(1);
        x(1)    <= not x(0);
        x(2)    <= c(1) and not c(0);
        x(3)    <= not x(2);
        -- module1 outputs
        x(4)    <= c(2) and not c(3);
        x(5)    <= not x(4);
        x(6)    <= c(3) and not c(2);
        x(7)    <= not x(6);

        -- module1 outputs
        x(8)    <= c(4) and not c(5);
        x(9)    <= not x(8);
```

```
        x(10)   <= c(5) and not c(4);
        x(11)   <= not x(10);
        -- module1 outputs
        x(12)   <= c(6) and not c(7);
        x(13)   <= not x(12);
        x(14)   <= c(7) and not c(6);
        x(15)   <= not x(14);

        --module2 outputs
        z(0)    <= (x(2) and x(6)) or (x(1) and x(3) and x(4)) or (x(0) and x(5) and x(7));
        z(1)    <= not z(0);
        z(2)    <= (x(4) and x(0)) or (x(7) and x(5) and x(2)) or (x(6) and x(3) and x(1));
        z(3)    <= not z(2);


z(4)    <= (x(10) and x(14)) or (x(9) and x(11) and x(12)) or (x(8) and x(13) and x(15));
z(5)    <= not z(4);
z(6)    <= (x(12) and x(8)) or (x(15) and x(13) and x(10)) or (x(14) and x(11) and x(9));
z(7)    <= not z(6);

        --module2 outputs
        w(0)    <= (z(2) and z(6)) or (z(1) and z(3) and z(4)) or (z(0) and z(5) and z(7));
        w(1)    <= not w(0);
        w(2)    <= (z(4) and z(0)) or (z(7) and z(5) and z(2)) or (z(6) and z(3) and z(1));
        w(3)    <= not w(2);

        w(4)    <= c(8) and not '0';
        w(5)    <= not w(4);
        w(6)    <= '0' and not c(8);
        w(7)    <= not w(6);


        --module2 outputs
modulo3aluout(0)    <= (w(2) and w(6)) or (w(1) and w(3) and w(4)) or (w(0) and w(5)
                    and w(7));
modulo3aluout(1)    <= (w(4) and w(0)) or (w(7) and w(5) and w(2)) or (w(6) and w(3)
                    and w(1));
end behavior;
```

APPENDIX C

BERGER CHECK PREDICTION VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

        -- main module contains alu, berger check predictionn and counter entity

ENTITY bcp8 IS
PORT(

                A                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                B                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                Af                      :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                Bf                      :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                opcode                  :IN STD_LOGIC_VECTOR(0 TO 2);
                result                  :OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
                error                   :OUT STD_LOGIC
);

END bcp8;

ARCHITECTURE structure OF bcp8 IS

        --internal wires
SIGNAL carryin                          :STD_LOGIC;
SIGNAL carry_out                        :STD_LOGIC;
SIGNAL carry                            :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL comp1                            :STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL comp2                            :STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL cout_res                         :STD_LOGIC_VECTOR(7 DOWNTO 0);
signal errorbus                         :STD_LOGIC_VECTOR(3 DOWNTO 0);

        --instance alu

COMPONENT alu8 IS
PORT(
                A                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                B                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                opcode                  :IN STD_LOGIC_VECTOR(0 TO 2);
                result                  :OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                carry_out               :OUT STD_LOGIC;
                carry                   :OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
```

```
);
END COMPONENT alu8;

        --instance BCP(included several entities)

COMPONENT berg IS
PORT(
                A                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                B                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                C                       :IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                carry_in                :IN STD_LOGIC;
                carry_out               :IN STD_LOGIC;
                opcode                  :IN STD_LOGIC_VECTOR(0 TO 2);
                result_c                :OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END COMPONENT;

        -- 0's counter
COMPONENT counter0 is
        port(

        -- inputs
        A                               : in std_logic_vector(7 downto 0);
        -- outputs
        counterout                      : out std_logic_vector(3 downto 0)
                );
END COMPONENT;

        --instance entities
BEGIN

alu1    :alu8           PORT MAP (A => A, B => B, opcode => opcode, result =>
                        cout_res, carry_out => carry_out, carry => carry);
bcp1    :berg           PORT MAP (A => Af, B => Bf, C => carry, carry_in => carryin,
                        carry_out => carry_out, opcode => opcode, result_c => comp2);
count1 :counter0        PORT MAP (A => cout_res, counterout => comp1);

        carryin <= '0';
        result <= carry_out & cout_res; --carry out added to result

        --comparator
ErrorBus        <= comp1 xor comp2;
Error           <= errorbus(3) or errorbus(2) or errorbus(1) or errorbus(0);


END structure;
```

```
ENTITY counter0 is
	port(

	-- inputs
	A					: in std_logic_vector(7 downto 0);

	-- outputs
	counterout				: out std_logic_vector(3 downto 0)
		);
END counter0;


architecture behavior of counter0 is
signal
	x		:  std_logic_vector(7 downto 0);
signal
	z,w,y		:  std_logic_vector(3 downto 0);
signal
	v		:  std_logic_vector(3 downto 0);
begin

	x <= not A;

	--full adder1
	y(0) <= x(0) xor x(1) xor x(2);
	y(1) <= (x(0) and x(1)) or (x(2) and (x(0) xor x(1)));

	--full adder2
	z(0) <= x(3) xor x(4) xor x(5);
	z(1) <= (x(3) and x(4)) or (x(5) and (x(3) xor x(4)));

	--full adder3
	w(0) <= x(6) xor x(7);
	w(1) <= x(6) and x(7);

	--undefined bits
	y(2) <= '0';
	y(3) <= '0';
	z(2) <= '0';
	z(3) <= '0';
	w(2) <= '0';
	w(3) <= '0';

	--2bit adder
	v <= y + z;
```

```
        counterout <= v + w;
end behavior;

ENTITY mcsa IS
PORT(
                x_c:            IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                y_c:            IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                c_c:            IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                d:              IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                result:         OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
);
END mcsa;

ARCHITECTURE structure OF mcsa IS
SIGNAL partial_sum:             STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL shift_carry:             STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL ps_sc_sum:               STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL ps:                      STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL sc:                      STD_LOGIC_VECTOR (4 DOWNTO 0);

BEGIN

        PROCESS(x_c, y_c, c_c, d, partial_sum, shift_carry, ps, sc, ps_sc_sum)
        BEGIN

                partial_sum     <= x_c XOR y_c XOR c_c;
                shift_carry     <=(x_c AND y_c) OR (x_c AND c_c) OR (y_c AND c_c);
                ps              <= "0" & partial_sum;
                sc              <= shift_carry & "0";
                ps_sc_sum       <= ps + sc + ("00" & d);
                result          <= ps_sc_sum(3 DOWNTO 0);
        END PROCESS;
END structure;

entity mux is
        port
        (
                -- Input ports
                A               : in std_logic_vector(7 downto 0);
                B               : in std_logic_vector(7 downto 0);
                carries         : in std_logic_vector(7 downto 0);
                opcode0         : in std_logic;
                t1              : in std_logic;

                -- Output ports
                output          : out std_logic_vector(7 downto 0)
```

```
                              );
end mux;


architecture behavior of mux is

begin
        process(opcode0, a, b, carries, t1)
        variable temp: std_logic_vector(7 downto 0);
        begin
        if opcode0 = '0' and t1 = '0' then
                temp := carries;
        elsif opcode0 = '0' and t1 = '1' then
                temp := carries;
        elsif opcode0 = '1' and t1 = '0' then
                temp := a and b;
        else
                temp := a or b;
        end if;
        end process;
end behavior;

ENTITY pla IS
PORT(
                carry_in                : IN STD_LOGIC;
                carry_out               : IN STD_LOGIC;
                opcode                  :       IN STD_LOGIC_VECTOR(0 TO 2);
                t                               :       OUT STD_LOGIC_VECTOR(1 TO
5);
                d                               :       OUT STD_LOGIC_VECTOR(1
DOWNTO 0)
);
END pla;

ARCHITECTURE structure OF pla IS

SIGNAL c_in, c_out: STD_LOGIC_VECTOR (1 DOWNTO 0);

BEGIN

        PROCESS(opcode, carry_in, carry_out, c_in, c_out)
        BEGIN
                t <= "00000";
                d <= "00";
                c_in <= "0" & carry_in;
                c_out <= "0" & carry_out;
```

```
		CASE(opcode) IS
			WHEN "000" =>
				t <= "00100";
				d <= c_out - c_in + 1;
			WHEN "001" =>
				t <= "00100";
				d <= c_out - c_in + 1;
			WHEN "010" =>
				t <= "00111";
				d <= c_out - c_in + 2;
			WHEN "011" =>
				t <= "00111";
				d <= c_out - c_in + 2;
			WHEN "100" =>
				t <= "10100";
				d <= "01";
			WHEN "101" =>
				t <= "01101";
				d <= "01";
			WHEN "110" =>
				t <= "00100";
				d <= "01";
			WHEN "111" =>
				t <= "00000";
				d <= "00";
			WHEN OTHERS =>
				t <= "00000";
				d <= "00";
		END CASE;
	END PROCESS;
END structure;

ENTITY alu8 IS
PORT(
		A:		IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		B:		IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		opcode:		IN STD_LOGIC_VECTOR(0 TO 2);
		result: `	OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
		carry_out:	OUT STD_LOGIC;
		carry:		OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END alu8;
```

```
ARCHITECTURE structure OF alu8 IS

BEGIN

        PROCESS(opcode, A, B)
        BEGIN
         --carry is hardcoded to arbitrary #s (did not use FAs to obtain carry)
                carry <= "00000000"; carry
                carry_out <= '0';
                IF opcode = "000" THEN
                        result <= A + B;   -- add
                ELSIF opcode = "001" THEN
                        result <= A - B; --subtract
                ELSIF opcode = "100" THEN
                        result <= A and B;  -- and
                ELSIF opcode = "110" THEN
                        result <= A or B;   -- or
                ELSIF opcode = "101" THEN
                        result <= A xor B;   -- xor
                ELSE
                        result <= "00000000";
                END IF;
        END PROCESS;
END structure;
```

APPENDIX D

RESO VHDL DECSRIPTION

```
entity RESO is
	port
	(
		-- Input ports
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		A_f			: in std_logic_vector(7 downto 0);
		B_f			: in std_logic_vector(7 downto 0);
		opcode			: in std_logic_vector(2 downto 0);
		clr			: in std_logic;
		clk			: in std_logic;

				-- Output ports

		S			: out std_logic_vector(10 downto 0);
		error			: out std_logic

	);
end RESO;


architecture structure of RESO is

		--instance alu
component alu12bit
	port(

	   -- alu8bit inputs
		A		: in std_logic_vector(9 downto 0);
		 B		: in std_logic_vector(9 downto 0);
		opcode		: in std_logic_vector(2 downto 0);
		clk		: in std_logic;

		-- alu8bit outputs
		C		: out std_logic_vector(10 downto 0)
		);
END component;

	--instance register

component Reg
	port(
```

```
		-- DFF inputs
		D			: in std_logic_vector(10 downto 0);
		clk			: in std_logic;
		clr			: in std_logic;

			-- DFF outputs
			Q		: out std_logic_vector(10 downto 0)
			);
END component;

	--instance mux

component mux2_to_1
	port
	(
		-- Input ports
		d0	: in  std_logic_vector(9 downto 0);
		d1	: in  std_logic_vector(9 downto 0);
		Sel	: in  std_logic;


		-- Output ports
		f	: out std_logic_vector(9 downto 0)
			);
end component;

	--instance shift left shifter
component ShiftL
	port(

		 -- rotator inputs
		 A	: in std_logic_vector(7 downto 0);


		-- rotator outputs
		C	: out std_logic_vector(9 downto 0)
		);
END component;

	--instance shift right shifter
component ShiftR
	port(

		-- inputs
		A	: in std_logic_vector(7 downto 0);
```

```
		-- outputs
		C	: out std_logic_vector(9 downto 0)
		);
END component;

	--instance alu result shifter
component ShiftL_lsb
	port(

		-- inputs
		A	: in std_logic_vector(10 downto 0);


		-- outputs
		C	: out std_logic_vector(10 downto 0)
		);
End component;

	--internal wires

signal aluout		:  std_logic_vector(10 downto 0);
signal MuxOut1		:  std_logic_vector(9 downto 0);
signal MuxOut2		:  std_logic_vector(9 downto 0);
signal ROut1		:  std_logic_vector(9 downto 0);
signal ROut2		:  std_logic_vector(9 downto 0);
signal LOut1		:  std_logic_vector(9 downto 0);
signal LOut2		:  std_logic_vector(9 downto 0);
signal lsbout		:  std_logic_vector(10 downto 0);
signal S1		:  std_logic_vector(10 downto 0);
signal errorbus		:  std_logic_vector(10 downto 0);

	--instance entities

begin


SL1: ShiftL		port map (A=>A, C=>Lout1);
SL2: ShiftL		port map (A=>B, C=>Lout2);
SR1: ShiftR		port map (A=>A_f, C=>Rout1);
SR2: ShiftR		port map (A=>B_f, C=>Rout2);
Mux1: mux2_to_1	port map (d0=>Lout1, d1=>Rout1, f=>MuxOut1, sel=>clk);
Mux2: mux2_to_1	port map (d0=>Lout2, d1=>Rout2, f=>MuxOut2, sel=>clk);
alu: alu12bit		port map (A=>MuxOut1, B=>MuxOut2, opcode=>opcode,
				C=>aluout, clk=>clk);
SL_lsb: ShiftL_lsb	port map (A=>aluout, C=> lsbout);
```

```
Reg1: Reg               port map (D=>aluout, Q=>S1, clk=>clk, clr=>clr);

S <= lsbout; --result

--comparator
errorbus        <= lsbout xor s1;
error           <= errorbus(10) or errorbus(9) or errorbus(8) or errorbus(7) or  errorbus(6)
                        or  errorbus(5) or  errorbus(4) or  errorbus(3) or  errorbus(2) or
                        errorbus(1) or  errorbus(0);

end structure;

--really an 11-bit alu
ENTITY alu12bit is
        port(

           -- alu8bit inputs
                A               : in std_logic_vector(9 downto 0);
                B               : in std_logic_vector(9 downto 0);
                opcode          : in std_logic_vector(2 downto 0);
                clk             : in std_logic;

                -- alu8bit outputs
                C               : out std_logic_vector(10 downto 0)
                );
END alu12bit;

--really an 11-bit alu
architecture behavior of alu12bit is
begin
        process(opcode, a, b)

        variable temp: std_logic_vector(10 downto 0);
        begin
        if opcode = "000" then
                temp := ('0' & a) +  ('0' & b);
                C <= temp;
        elsif opcode = "001" then
                temp := ('0' & a) -  ('0' & b);
                C <= temp;
        elsif opcode = "010" then
                temp := ('0' & a) and  ('0' & b);
                C <= temp;
        elsif opcode = "011" then
                temp := ('0' & a) or  ('0' & b);
                C <= temp;
```

```
        else
                temp := ('0' & a) xor  ('0' & b);
                C <= temp;
        end if;
                end process;
end behavior;

entity mux2_to_1 is
        port
        (
                -- Input ports
                d0      : in  std_logic_vector(9 downto 0);
                d1      : in  std_logic_vector(9 downto 0);
                Sel     : in  std_logic;


                -- Output ports
                f       : out std_logic_vector(9 downto 0)
                        );
end mux2_to_1;


architecture behavior of mux2_to_1 is

        begin
--              f <= (d0 and  not S) or (d1 and S); -- 2 to 1 mux boolean equation
                with Sel select
                                f <= d0 when '0',
                                        d1 when others;
end behavior;

ENTITY Reg is
        port(

        -- DFF inputs
        D                       : in std_logic_vector(10 downto 0);
        clk                     : in std_logic;
        clr                     : in std_logic;

        -- DFF outputs
        Q                       : out std_logic_vector(10 downto 0)

                );
END Reg;

architecture behavior of Reg is
```

```
begin

	process(d, clk, clr)
	begin
		if clr = '1' then Q <= "00000000000";
			elsif clk'event and clk = '1'
				then Q <= D;
		end if;
	end process;


end behavior;

ENTITY ShiftL is
	port(

	-- rotator inputs
	A				: in std_logic_vector(7 downto 0);


	-- rotator outputs
	C				: out std_logic_vector(9 downto 0)
		);
END ShiftL;

architecture behavior of ShiftL is
begin

	C <= A & "00";

end behavior;


ENTITY ShiftL_lsb is
	port(

	-- rotator inputs
	A				: in std_logic_vector(10 downto 0);


	-- rotator outputs
	C				: out std_logic_vector(10 downto 0)
		);
END ShiftL_lsb;

architecture behavior of ShiftL_lsb is
```

```
begin

	C <= A(8 downto 0)& "00";

end behavior;

ENTITY ShiftR is
	port(

	-- rotator inputs
	A				: in std_logic_vector(7 downto 0);


	-- rotator outputs
	C				: out std_logic_vector(9 downto 0)
		);
END ShiftR;

architecture behavior of ShiftR is
begin

	C <= "00" & A;

end behavior;
```

APPENDIX D

RESO VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


entity RERO is
	port
	(
		-- Input ports
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		A_f			: in std_logic_vector(7 downto 0);
		B_f			: in std_logic_vector(7 downto 0);
		Opcode			: in std_logic_vector(2 downto 0);
		clr			: in std_logic;
		clk			: in std_logic;





		-- Output ports
		S			: out std_logic_vector(8 downto 0);
		error			: out std_logic

	);
end RERO;


architecture structure of RERO is

		--instance alu
component alu9bit
port
	(
		-- alu inputs
		A			: in std_logic_vector(8 downto 0);
		B			: in std_logic_vector(8 downto 0);
		opcode			: in std_logic_vector(2 downto 0);

		-- alu outputs
		C			: out std_logic_vector(8 downto 0)
```

```
                        );
end component;

        --instance register
component FFR
        port(

           -- DFF inputs
           D                    : in std_logic_vector(8 downto 0);
           clk                  : in std_logic;
           clr                  : in std_logic;

                -- DFF outputs
                Q                       : out std_logic_vector(8 downto 0)
                );
END component;

        --instance mux
component mux_2to1
        port
        (
                -- Input ports
                d0      : in  std_logic_vector(8 downto 0);
                d1      : in  std_logic_vector(8 downto 0);
                Sel     : in  std_logic;


                -- Output ports
                f       : out std_logic_vector(8 downto 0)
                        );
end component;

        --instance left rotator
component RotatorL
        port(

        -- rotator inputs
         A                      : in std_logic_vector(8 downto 0);


        -- rotator outputs
        C                       : out std_logic_vector(8 downto 0)
                );
END component;

        --instance right rotator
```

```
component RotatorR
          port(

           -- rotator inputs
             A                          : in std_logic_vector(7 downto 0);


          -- rotator outputs
          C                             : out std_logic_vector(8 downto 0)
                    );
END component;


          --instance unrotated operands to enter ALU for first computation

Component Unrotated is
          port(

           -- rotator inputs
           A                            : in std_logic_vector(7 downto 0);

          -- rotator outputs
          C                             : out std_logic_vector(8 downto 0)
                    );
END Component;

          --internal wires
signal alu8out                  :  std_logic_vector(8 downto 0);
signal MuxOut1                  :  std_logic_vector(8 downto 0);
signal MuxOut2                  :  std_logic_vector(8 downto 0);
signal ROut1                    :  std_logic_vector(8 downto 0);
signal ROut2                    :  std_logic_vector(8 downto 0);
signal uROut1                   :  std_logic_vector(8 downto 0);
signal uROut2                   :  std_logic_vector(8 downto 0);
signal LOut                     :  std_logic_vector(8 downto 0);
signal S1                       :  std_logic_vector(8 downto 0);
signal S_f                      :  std_logic_vector(8 downto 0);
signal errorbus                 :  std_logic_vector(8 downto 0);

          --instance entities

begin


RR1: rotatorR          port map (A=>A, C=>Rout1);
RR2: rotatorR          port map (A=>B, C=>Rout2);
```

```
UR1: unrotated          port map (A=>A_f, C=>uRout1);
UR2: unrotated          port map (A=>B_f, C=>uRout2);
Mux1: Mux_2to1          port map (d0=>Rout1, d1=>uRout1, f=>MuxOut1, sel=>clk);
Mux2: Mux_2to1          port map (d0=>Rout2, d1=>uRout2, f=>MuxOut2, sel=>clk);
alu: alu9bit            port map (A=>MuxOut1, B=>MuxOut2, opcode=>opcode,
                                C=>alu8out);
RL: rotatorL            port map (A=>alu8out, C=>lout);
FF:     FFR             port map (D=>alu8out, Q=>S_f, clk=>clk, clr=>clr);

S <= lout; --resul

--comparator

errorbus        <= lout xor s_f;
error           <= errorbus(8) or errorbus(7) or  errorbus(6) or  errorbus(5) or  errorbus(4)
                        or  errorbus(3) or  errorbus(2) or  errorbus(1) or  errorbus(0);

end structure;


ENTITY alu9bit is
        port(

            -- alu8bit inputs
                 A              : in std_logic_vector(8 downto 0);
                B               : in std_logic_vector(8 downto 0);
                opcode          : in std_logic_vector(2 downto 0);

                -- alu8bit outputs
                C               : out std_logic_vector(8 downto 0)
                );
END alu9bit;

architecture behavior of alu9bit is
begin
        process(opcode, a, b)
        variable temp: std_logic_vector(9 downto 0);
        begin

        if opcode = "000" then
                temp := ('0'&a) + ('0'&b) + '1';
                C <= temp(8 downto 0);
        elsif opcode = "001" then
                temp := ('0'&a) - ('0'&b) + '1';
                C <= temp(8 downto 0);
        elsif opcode = "010" then
```

```
                temp := ('0'&a) and ('0'&b);
                C <= temp(8 downto 0);
        elsif opcode = "011" then
                temp := ('0'&a) or ('0'&b);
                C <= temp(8 downto 0);
        else
                temp := ('0'&a) xor ('0'&b);
                C <= temp(8 downto 0);
        end if;
        end process;
end behavior;

ENTITY FFR is
        port(

                -- DFF inputs
                D                       : in std_logic_vector(8 downto 0);
                clk                     : in std_logic;
                clr                     : in std_logic;

                -- DFF outputs
                Q                       : out std_logic_vector(8 downto 0)

                );
END FFR;

architecture behavior of FFR is
begin

        process(d, clk, clr)
        begin
                if clr = '1' then Q <= "000000000";
                        elsif clk'event and clk = '1'
                                then Q <= D;
                end if;
        end process;


end behavior;

entity mux_2to1 is
        port
        (
                -- Input ports
                d0      : in  std_logic_vector(8 downto 0);
                d1      : in  std_logic_vector(8 downto 0);
```

```
		Sel	: in  std_logic;


		-- Output ports
		f	: out std_logic_vector(8 downto 0)
			);
end mux_2to1;


architecture behavior of mux_2to1 is

	begin
--		f <= (d0 and  not S) or (d1 and S); -- 2 to 1 mux boolean equation
		with Sel select
				f <= d1 when '0',
					d0 when others;
end behavior;

ENTITY RotatorL is
	port(

	 -- rotator inputs
	A				: in std_logic_vector(8 downto 0);


	-- rotator outputs
	C				: out std_logic_vector(8 downto 0)
		);
END RotatorL;

architecture behavior of RotatorL is
begin

	C <= A(6 downto 0) & A(8 downto 7);

end behavior;

ENTITY RotatorR is
	port(

	 -- rotator inputs
	A				: in std_logic_vector(7 downto 0);

	-- rotator outputs
	C				: out std_logic_vector(8 downto 0)
		);
```

```
END RotatorR;

architecture behavior of RotatorR is
begin
                        c <= A(1 downto 0) & '0' & A(7 downto 2);
end behavior;


ENTITY Unrotated is
        port(

           -- rotator inputs
        A                       : in std_logic_vector(7 downto 0);

        -- rotator outputs
        C                       : out std_logic_vector(8 downto 0)
                );
END Unrotated;

architecture behavior of Unrotated is
begin
                        c <= '0' & A;
end behavior;
```

APPENDIX E

RERO VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;


entity RERO is
	port
	(
		-- Input ports
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		A_f			: in std_logic_vector(7 downto 0);
		B_f			: in std_logic_vector(7 downto 0);
		opcode			: in std_logic_vector(2 downto 0);
		clr			: in std_logic;
		clk			: in std_logic;

		-- Output ports
		S			: out std_logic_vector(8 downto 0);
		error		: out std_logic
		);
end RERO;


architecture structure of RERO is

		--instance alu
component alu9bit
port
	(
		-- alu inputs
	A			: in std_logic_vector(8 downto 0);
	B			: in std_logic_vector(8 downto 0);
		opcode		: in std_logic_vector(2 downto 0);

		-- alu outputs
	C			: out std_logic_vector(8 downto 0)
			);
end component;
```

```
    --instance register
component FFR
    port(

      -- DFF inputs
      D                : in std_logic_vector(8 downto 0);
      clk              : in std_logic;
      clr              : in std_logic;

        -- DFF outputs
       Q               : out std_logic_vector(8 downto 0)
        );
END component;

    --instance mux
component mux_2to1
    port
    (
        -- Input ports
        d0      : in  std_logic_vector(8 downto 0);
        d1      : in  std_logic_vector(8 downto 0);
        Sel     : in  std_logic;


        -- Output ports
        f       : out std_logic_vector(8 downto 0)
                );
end component;

    --instance left rotator
component RotatorL
    port(

      -- rotator inputs
    A                  : in std_logic_vector(8 downto 0);


        -- rotator outputs
    C                  : out std_logic_vector(8 downto 0)
        );
END component;
```

```
--instance right rotator
component RotatorR
        port(

           -- rotator inputs
         A                          : in std_logic_vector(7 downto 0);


                -- rotator outputs
        C                           : out std_logic_vector(8 downto 0)
                );
END component;

        --instance unrotated operands to enter ALU for first computation
Component Unrotated is
        port(

           -- rotator inputs
         A                          : in std_logic_vector(7 downto 0);

                -- rotator outputs
        C                           : out std_logic_vector(8 downto 0)
                );
END Component;

        --internal wires
signal alu8out                      :  std_logic_vector(8 downto 0);
signal MuxOut1                      :  std_logic_vector(8 downto 0);
signal MuxOut2                      :  std_logic_vector(8 downto 0);
signal ROut1                        :  std_logic_vector(8 downto 0);
signal ROut2                        :  std_logic_vector(8 downto 0);
signal uROut1                       :  std_logic_vector(8 downto 0);
signal uROut2                       :  std_logic_vector(8 downto 0);
signal LOut                         :  std_logic_vector(8 downto 0);
signal S1                           :  std_logic_vector(8 downto 0);
signal S_f                          :  std_logic_vector(8 downto 0);
signal errorbus                     :  std_logic_vector(8 downto 0);

        --instance entities

begin


RR1: rotatorR                   port map (A=>A, C=>Rout1);
RR2: rotatorR                   port map (A=>B, C=>Rout2);
UR1: unrotated                  port map (A=>A_f, C=>uRout1);
```

```
UR2: unrotated                  port map (A=>B_f, C=>uRout2);
Mux1: Mux_2to1                  port map (d0=>Rout1, d1=>uRout1, f=>MuxOut1,
sel=>clk);
Mux2: Mux_2to1                  port map (d0=>Rout2, d1=>uRout2, f=>MuxOut2,
                                          sel=>clk);
alu: alu9bit                    port map (A=>MuxOut1, B=>MuxOut2, opcode=>opcode,
                                          C=>alu8out);
RL: rotatorL                    port map (A=>alu8out, C=>lout);
FF:FFR                          port map (D=>alu8out, Q=>S_f, clk=>clk, clr=>clr);

S <= lout; --resul

--comparator
errorbus <= lout xor s_f;
error <= errorbus(8) or errorbus(7) or  errorbus(6) or  errorbus(5) or  errorbus(4) or
errorbus(3) or  errorbus(2) or  errorbus(1) or  errorbus(0);

end structure;


ENTITY alu9bit is
        port(

           -- alu8bit inputs
         A                      : in std_logic_vector(8 downto 0);
         B                      : in std_logic_vector(8 downto 0);
                opcode          : in std_logic_vector(2 downto 0);

                -- alu8bit outputs
        C                       : out std_logic_vector(8 downto 0)
                );
END alu9bit;

architecture behavior of alu9bit is
begin
        process(opcode, a, b)
        variable temp: std_logic_vector(9 downto 0);
        begin

        if opcode = "000" then
                temp := ('0'&a) + ('0'&b) + '1';
                C <= temp(8 downto 0);
        elsif opcode = "001" then
                temp := ('0'&a) - ('0'&b) + '1';
                C <= temp(8 downto 0);
        elsif opcode = "010" then
```

```
                temp := ('0'&a) and ('0'&b);
                C <= temp(8 downto 0);
        elsif opcode = "011" then
                temp := ('0'&a) or ('0'&b);
                C <= temp(8 downto 0);
        else
                temp := ('0'&a) xor ('0'&b);
                C <= temp(8 downto 0);
        end if;
        end process;
end behavior;


ENTITY Unrotated is
        port(

           -- rotator inputs
        A                       : in std_logic_vector(7 downto 0);

                -- rotator outputs
        C                       : out std_logic_vector(8 downto 0)
                );
END Unrotated;

architecture behavior of Unrotated is
begin
                        c <= '0' & A;
end behavior;

ENTITY RotatorR is
        port(

           -- rotator inputs
         A                      : in std_logic_vector(7 downto 0);

                -- rotator outputs
        C                       : out std_logic_vector(8 downto 0)
                );
END RotatorR;

architecture behavior of RotatorR is
begin
                        c <= A(1 downto 0) & '0' & A(7 downto 2);
end behavior;

ENTITY RotatorL is
```

```
	port(

	   -- rotator inputs
	 A				: in std_logic_vector(8 downto 0);


		-- rotator outputs
	C				: out std_logic_vector(8 downto 0)
		);
END RotatorL;

architecture behavior of RotatorL is
begin

	C <= A(6 downto 0) & A(8 downto 7);

end behavior;

entity mux_2to1 is
	port
	(
		-- Input ports
		d0	: in  std_logic_vector(8 downto 0);
		d1	: in  std_logic_vector(8 downto 0);
		Sel	: in  std_logic;


		-- Output ports
		f	: out std_logic_vector(8 downto 0)
			);
end mux_2to1;


architecture behavior of mux_2to1 is

	begin
--		f <= (d0 and  not S) or (d1 and S); -- 2 to 1 mux boolean equation
		with Sel select
				f <= d1 when '0',
					d0 when others;
end behavior;


ENTITY FFR is
	port(
```

```
        -- DFF inputs
        D                   : in std_logic_vector(8 downto 0);
        clk                 : in std_logic;
        clr                 : in std_logic;

            -- DFF outputs
        Q                   : out std_logic_vector(8 downto 0)

            );
END FFR;

architecture behavior of FFR is
begin

    process(d, clk, clr)
    begin
        if clr = '1' then Q <= "000000000";
            elsif clk'event and clk = '1'
                then Q <= D;
        end if;
    end process;


end behavior;
```

APPENDIX F

PARITY VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;



entity ppc is
	port
	(
		-- Input ports
		A		: in std_logic_vector(7 downto 0);
		B		: in std_logic_vector(7 downto 0);
		A_f		: in std_logic_vector(7 downto 0);
		B_f		: in std_logic_vector(7 downto 0);
		Opcode		: in std_logic_vector(2 downto 0);



		-- Output ports
		S		: out std_logic_vector(8 downto 0);
		error		: out std_logic


	);
end ppc;


architecture structure of ppc is

		--instance alu8bit

component alu8bit
port
	(
		-- inputs
	   A		: in std_logic_vector(7 downto 0);
	   B		: in std_logic_vector(7 downto 0);
	   opcode		: in std_logic_vector(2 downto 0);

		-- outputs
```

```
		C			: out std_logic_vector(8 downto 0);
		aluout			: out std_logic_vector(7 downto 0)
				);
end component;

component parity is
	port(

		-- alu8bit inputs
		A			: in std_logic_vector(7 downto 0);

			-- alu8bit outputs
		C			: out std_logic
			);
END component;

			-- internal signals


signal ac				:  std_logic;
signal bc				:  std_logic;
signal pc				:  std_logic;
signal Sc				:  std_logic;
signal S1				:  std_logic_vector(7 downto 0);

begin
			-- Port connections

alu8: alu8bit			port map (A=>A_f, B=>B_f, opcode=>opcode, C=>S,
						aluout=>s1);
p1: parity			port map (A=>A, C=>ac);
p2: parity			port map (A=>b, C=>bc);
p3: parity			port map (A=>s1, C=>sc);

pc <= ac xor bc;
error <= pc xor Sc;

end structure;

ENTITY alu8bit is
	port(

		-- alu8bit inputs
			A				: in std_logic_vector(7 downto 0);
			B		: in std_logic_vector(7 downto 0);
			opcode		: in std_logic_vector(2 downto 0);
```

```
		-- alu8bit outputs
		aluout		: out std_logic_vector(7 downto 0);
		C		: out std_logic_vector(8 downto 0)
		);
END alu8bit;

architecture behavior of alu8bit is
signal temp		: std_logic_vector(8 downto 0);
	begin
	process(opcode, a, b)
	begin
		IF opcode = "000" THEN
			temp <= ('0'& A) + ('0' & B);   -- add
		ELSIF opcode = "001" THEN
			temp <= ('0' & A) - ('0' & B);  -- subtract
		ELSIF opcode = "010" THEN
			temp <= ('0' & A) and ('0' & B);  -- and
		ELSIF opcode = "011" THEN
			temp  <= ('0' & A) or ('0' & B);   -- or
		ELSIF opcode = "100" THEN
			temp <= ('0' & A) xor ('0' & B);   -- xor
		ELSE
			temp <= ('0' & A) xor ('0' & B);
		END IF;
	end process;
	aluout <= temp (7 downto 0);
	c <= temp;
end behavior;

ENTITY parity is
	port(

	   -- alu8bit inputs
	   A			: in std_logic_vector(7 downto 0);

		-- alu8bit outputs
	   C			: out std_logic
		);
END parity;

architecture behavior of parity is
begin

c <= a(7) xor a(6) xor a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);
end behavior;
```

APPENDIX G

PARITY AND LOGIC VHDL DECSRIPTION

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;



entity Pargic is
	port
	(
		-- Input ports
		A			: in std_logic_vector(7 downto 0);
		B			: in std_logic_vector(7 downto 0);
		A_f			: in std_logic_vector(7 downto 0);
		B_f			: in std_logic_vector(7 downto 0);
		opcode		: in std_logic_vector(2 downto 0);




		-- Output ports
		S			: out std_logic_vector(8 downto 0);
		error		: out std_logic


	);
end Pargic;


architecture structure of Pargic is

		--instance alu8bit

component alu8bit
port
	(
		-- inputs
		A		: in std_logic_vector(7 downto 0);
		B		: in std_logic_vector(7 downto 0);
		opcode		: in std_logic_vector(2 downto 0);

		-- outputs
```

```
		C			: out std_logic_vector(8 downto 0);
		aluout			: out std_logic_vector(7 downto 0)
			);
end component;

component parity is
	port(

	   -- alu8bit inputs
	A			: in std_logic_vector(7 downto 0);

		-- alu8bit outputs
	C			: out std_logic
		);
END component;

component logic is
	port(

	   -- alu8bit inputs
	A			: in std_logic_vector(7 downto 0);
	B			: in std_logic_vector(7 downto 0);
		opcode		: in std_logic_vector(2 downto 0);

		-- alu8bit outputs
	C			: out std_logic_vector(7 downto 0)
		);
END component;

component comparator is
	port(

	   -- alu8bit inputs
		A				: in std_logic_vector(7 downto 0);
		B				: in std_logic_vector(7 downto 0);
		-- alu8bit outputs
		C				: out std_logic
		);
END component;


component mux_2to1 is
	port
	(
		-- Input ports
		d0	: in  std_logic;
```

```
            d1      : in  std_logic;
            Sel     : in  std_logic_vector (2 downto 0);


            -- Output ports
            f       : out std_logic
                    );
end component;


            -- internal signals


signal ac                           :  std_logic;
signal bc                           :  std_logic;
signal pc                           :  std_logic;
signal Sc                           :  std_logic;
signal d1                           :  std_logic;
signal S1                           :  std_logic_vector(7 downto 0);
signal log                          :  std_logic_vector(7 downto 0);

begin

            -- Port connections

alu8: alu8bit                 port map (A=>A_f, B=>B_f, opcode=>opcode, C=>S,
aluout=>s1);
p1: parity                          port map (A=>A, C=>ac);
p2: parity                          port map (A=>b, C=>bc);
p3: parity                          port map (A=>s1, C=>sc);
l1: logic                           port map (A=>A, B=>B, opcode=>opcode,
C=>log);
c1: comparator                      port map (A=>log, B=>s1, C=>d1);
m1: mux_2to1                  port map (d0=>pc, d1=>d1, sel=>opcode, f=>error);



pc <= ac xor bc xor Sc;

end structure;

ENTITY alu8bit is
        port(

          -- alu8bit inputs
            A                 : in std_logic_vector(7 downto 0);
```

```
		B			: in std_logic_vector(7 downto 0);
		opcode		: in std_logic_vector(2 downto 0);

		-- alu8bit outputs
		aluout		: out std_logic_vector(7 downto 0);
		C				: out std_logic_vector(8 downto 0)
		);
END alu8bit;

architecture behavior of alu8bit is
signal temp			: std_logic_vector(8 downto 0);
	begin
	process(opcode, a, b)
	begin
		IF opcode = "000" THEN
			temp <= ('0'& A) + ('0' & B);   -- add
		ELSIF opcode = "001" THEN
			temp <= ('0' & A) - ('0' & B);  -- subtract
		ELSIF opcode = "010" THEN
			temp <= ('0' & A) and ('0' & B);  -- and
		ELSIF opcode = "011" THEN
			temp  <= ('0' & A) or ('0' & B);   -- or
		ELSIF opcode = "100" THEN
			temp <= ('0' & A) xor ('0' & B);   -- xor
		ELSE
			temp <= ('0' & A) xor ('0' & B);
		END IF;
	end process;
	aluout <= temp (7 downto 0);
	c <= temp;
end behavior;

ENTITY logic is
	port(

	  -- alu8bit inputs
	  A				: in std_logic_vector(7 downto 0);
	  B				: in std_logic_vector(7 downto 0);
		opcode		: in std_logic_vector(2 downto 0);

		-- alu8bit outputs
		C				: out std_logic_vector(7 downto 0)
		);
END logic;

architecture behavior of logic is
```

```
signal temp              : std_logic_vector(7 downto 0);
        begin
        process(opcode, a, b)
        begin
                IF opcode = "010" THEN
                        temp <= ( A) and ( B);  -- and
                ELSIF opcode = "011" THEN
                        temp  <= ( A) or ( B);   -- or
                ELSIF opcode = "100" THEN
                        temp <= (A) xor (B);
                ELSE
                        temp <= ( A) xor ( B);    -- xor
                END IF;
        end process;
        c <= temp;
end behavior;

ENTITY parity is
        port(

           -- alu8bit inputs
         A                      : in std_logic_vector(7 downto 0);

                -- alu8bit outputs
        C                       : out std_logic
                );
END parity;

architecture behavior of parity is
begin

c <= a(7) xor a(6) xor a(5) xor a(4) xor a(3) xor a(2) xor a(1) xor a(0);

end behavior;

entity mux_2to1 is
        port
        (
                -- Input ports
                d0      : in  std_logic;
                d1      : in  std_logic;
                Sel     : in  std_logic_vector (2 downto 0);


                -- Output ports
                f       : out std_logic
```

```
                        );
end mux_2to1;


architecture behavior of mux_2to1 is

        begin
--              f <= (d0 and  not S) or (d1 and S); -- 2 to 1 mux boolean equation
                with Sel select
                                f <= d0 when "000",
                                        d0 when "001",
                                        d1 when others;
end behavior;

ENTITY comparator is
        port(

          -- alu8bit inputs
          A                     : in std_logic_vector(7 downto 0);
                B                       : in std_logic_vector(7 downto 0);
                -- alu8bit outputs
                C                       : out std_logic
                );
END comparator;

architecture behavior of comparator is
signal d                        : std_logic_vector(7 downto 0);
begin
d <= a xor b;
c <= d(7) xor d(6) xor d(5) xor d(4) xor d(3) xor d(2) xor d(1) xor d(0);

end behavior;
```

REFERENCES

[1] K. Roy, T.M. Mak, K. Cheng. "Test considerations for nanometer scale CMOS circuits". *Proceedings of the 21st IEEE VLSI Test Symposium, pp.313-315*. 2003.

[2] S. Mukherjee. *Architecture Design for Soft Errors.* Burlington, MA: Morgan Kaufmann, 2008, pp. 2-8.

[3] C. Constantinescu. "Impact of Deep Submicron Technology on Dependability of VLSI Circuits", *Proceedings International Conference on Dependable Systems and Network, pp. 205- 209*, 2002.

[4] J. Li, E.E. Swartzlander Jr. "Concurrent error detection in ALUs by recomputing with rotated operands". *International Workshop on Defect and Fault Tolerance in VLSI Systems, pp. 109-116*, 1992.

[5] 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 Transactions on Nuclear Science, pp. 1833-1853*, 2008.

[6] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, "On latching probability of particle induced transients in combinational networks," in *24th IEEE International Symposium on Fault-Tolerant Computing*, Austin, TX, USA, 1994, pp. 340-9.

[7] L. W. Massengill, A. E. Baranski, D. O. Van Nort, J. Meng, and B. L. Bhuva, "Analysis of single-event effects in combinational logic-simulation of the AM2901 bitslice processor," *IEEE Transactions on Nuclear Science,* vol. 47, pp. 2609-15, 2000.

[8] P.E. Dodd, L.W. Massengill. "Basic mechanisms and modeling of single-event upset in digital microelectronics". *IEEE Transactions on Nuclear Science, vol.50, pp. 583-602*, 2003.

[9] S.J. Piestrak. "Design of residue generators and multioperand adders modulo 3 built of multioutput threshold circuits". *IEE Proceedings Computer and Digital Techniques, vol. 141, pp. 129-134,* 1994.

[10] J.C. Lo, S. Thanawastien, T.R.N. Rao, "Concurrent error detection in arithmetic and logical operations using Berger codes". *Proceedings of 9th Symposium on Computer Arithmetic, pp.* 233-240, 1989.

[11] D. P. Siemwiorek, "Architecture of fault-tolerant computers: an historical perspective," *Proceedings of the IEEE,* vol. 79, pp. 1710-34, 1991.

[12] V. Srinivasan, J. W. Farquharson, B.L. Bhuva, W.H. Robinson. ”Evaluation of Error Detection Strategies for an FPGA-Based Self-Checking Arithmetic and Logic Unit”. *MAPLD International Conference*, 2005.

[13] J.F. Wakerly. “Partially Self-Checking Circuits and Their Use in Performing Logical Operations*” IEEE Transactions on Computers, Vol. C-23 pp. 658-666,* 1974.

[14] B.W. Johnson, J.H. Aylor, H.H. Hana. “Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder.” *IEEE Journal of Solid-State Circuits, pp. 208-215*, 1988.

[15] A. Golander, S. Weiss, R. Ronen. “Synchronizing Redundant Cores in a Dynamic DMR Multicore Architecture”. *IEEE Transactions on Circuits and Systems II: Express Briefs, pp. 474-478*, 2009.

[16] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *17th IEEE VLSI Test Symposium*, Dana Point, CA, USA, 1999, pp. 86-94.

[17] D. A. Reynolds, G. Metze. “Fault Detection Capabilitiesof Alternating Logic*” IEEE Transactions on Computer Science, Vol. C-27, pp. 1093-1098,* 1978.

[18] M. Nicolaidis. “Carry Checking/Parity Prediction Adders and ALUs.” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.11, pp. 121-128*, 2003.

[19] M. Nicolaidis, R.O. Duarte, S. Manich, J. Figueras. “Fault-secure parity prediction arithmetic operators”. *IEEE Design & Test of Computers, vol. 14, pp. 60-71*, 1997. *IEEE Transaction on Computers,*1997.

[20] Y. Saitoh, H. Imai. “Multiple Unidirectional Byte Error-Correcting Codes”. *IEEE Transactions on Information Theory, vol. 37, pp. 903-908*, 1991.

[21] G. M. Koob, C. Lau. *Foundations of Dependable Computing: System Implementation*. Norwell, MA: Kluwer Academic Publishers, 1994, pp. 37, 49-53.

[22] M. Nicolaidis. “Efficient Implementation of Self-Checking Adders and ALUs*” The Twenty-Third International Symposium on Fault-Tolerant Computing, pp. 586-595,* 1993.

[23] J.H. Patel, L.Y. Fung. “Concurrent Error Detection in ALU's by Recomputing with Shifted Operands.” *IEEE Transactions on Computers, pp. 589-595*, 1982.

[24] D.D. Gajski, C. Vora. "High-speed modulo-3 generator". *Electronics Letters, Vol. 13,* 1977.

[25] R. Forsati, K. Faez, F. Moradi, A. Rahbar. "A Fault Tolerant Method for Residue Arithmetic Circuits". *International Conference on Information Management and Engineering, pp. 59-63*, 2009.

[26] J.C. Lo, S. Thanawastien, T.R.N. Rao, M. Nicolaidis. "An SFS Berger check prediction ALU and its application to self-checking processor designs". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol 11, pp. 525-540,* 1992.

[27] "Quartus II Subscription Edition Software". [Online] Available: http://www. altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html. [Accessed Mar. 12, 2010].

[28] C. Roth, L.K. John. *Digital Systems Design Using VHDL.* Toronto, Ontario: Thomson Learning, 2008.

[29] "ModelSim-Altera [Online] Available: http://www.altera.com/products/software/ quartus-ii/modelsim/qts-modelsim-index.html. [Accessed Mar. 12, 2010].

[30] Gajiski, D. *Modular Modulo 3 Module. 4,190,893.* United States, Feb 26, 1980.

[31] J.C. Lo, S. Thanawastien, and T.R.N. Rao. "Concurrent error detection in arithmetic and logical operations using Berger codes". *Proceedings of 9th Symposium on Computer Arithmetic, pp. 233-240*, 1989.

[32] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P.D.Franzon, M. Bucher, S. Basavarajaiah, J. Oh, R. Jenkal. "FreePDK: An Open-Source Variation-Aware Design Kit*" IEEE International Conference on Microelectronics Systems Education, pp. 173-174,* 2007.

[33] "Voltagestorm Power and Power Rail Verification". [Online] Available: http://www.cadence.com/rl/Resources/datasheets/voltage_storm_ds.pdf. [Accessed Mar. 23, 2010].