Application of Residue Codes for Error Detection in Modern Computers

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____
Mattan Erez

_____
Earl E. Swartzlander, Jr.

The Report Committee for Michael Brendan Sullivan
Certifies that this is the approved version of the following report:

Application of Residue Codes for Error Detection in Modern Computers

by

Michael Brendan Sullivan, B.A., B.S., M.S.

REPORT

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN
December 2010

Application of Residue Codes for Error Detection in Modern Computers

Michael Brendan Sullivan, M.S.E.
The University of Texas at Austin, 2010

Supervisor: **Mattan Erez**

Residue codes have successfully been used for decades as a low overhead method of arithmetic error detection. This work explores the design space of residue checking for error detection in processors with modern word sizes and technology nodes. The area overheads of detecting arithmetic errors are considered for a variety of processor configurations, ranging from those best suited for embedded processors to those best for high-performance computers. The ultimate goal of this work is to enable the study of low overhead arithmetic error protection and correction in a wider variety of computer architectures than has previously been attempted in a systematic manner.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This study focuses on the detection of transient errors in the arithmetic datapath of modern microprocessors through the application of residue error detecting codes. Errors in a processor can occur due to many sources, including neutron strikes, electromagnetic interference, inconsistent supply voltage, and degradation of materials over time [1]. The proper application of residue checking can provide strong, comprehensive error protection of a processor datapath irrespective of the duration or cause of an error—both permanent faults and temporary errors can be reliably detected, as well as intermittent errors due to design flaws or conditional variation.

The use of residue codes for error detection is relevant mainly to protect the arithmetic computation within a processor. Error protection in the compute logic of a processor is becoming more important; soft-error rates are rising with the number of transistors on chip, and the error vulnerability of combinational logic may be increasing faster than that of memory or sequential elements [1, 2, 3]. Furthermore, soft errors in the execution path are the most likely to produce fail-silent data violations, which are more insidious and may be more dangerous than visible software malfunctions [4].

The rest of the report is organized as follows. Section 1.1 gives a brief introduction of error detection through residue codes. Section 1.1.1 provides a more in-depth introduction to the field of error correcting codes, while emphasizing the relationship between the assumed error model with the overheads and error coverage of residue checking. Section 1.1.2 gives an overview of the costs and design parameters associated with error protection via residue codes, and begins to transition over to the discussion of implementation overheads and details. In Chapter 2, Sections2.1 and 2.2 explain the methodology and describe the experimental platform, as well as the design of the circuit used for the analyses. Section 2.3 reports the overheads found from applying a residue code for error detection in the datapaths of three processors. Finally, some future work is described and conclusions are summarized in Chapter3.

1.1 Residue Codes

Figure 1.1 shows an overview of the error detection process using residue codes. Most arithmetic operations can be checked by testing the equality of Equation 1.1, where $|N|_A = N \bmod A$ and $\oplus$ is the operation of interest. If both sides of Equation 1.1 are equal, it is likely that no error has occurred. If both sides are not equal, then some error *has* occurred. The coverage of a given residue code (chance for a false negative) is discussed in Section 1.1.1.

$$|a \oplus b|_A \stackrel{?}{=} \big||a|_A \oplus |b|_A\big|_A \tag{1.1}$$

The self-checking checker circuit shown in Figure 1.1 exists to both

check the result of computation, and to check its own conclusions—it is designed to operate without fail even when the induced error happens within the checker itself [5]. Specialized self-checking checkers exist for certain classes of residue codes, as well as for the general case [6, 7, 8].

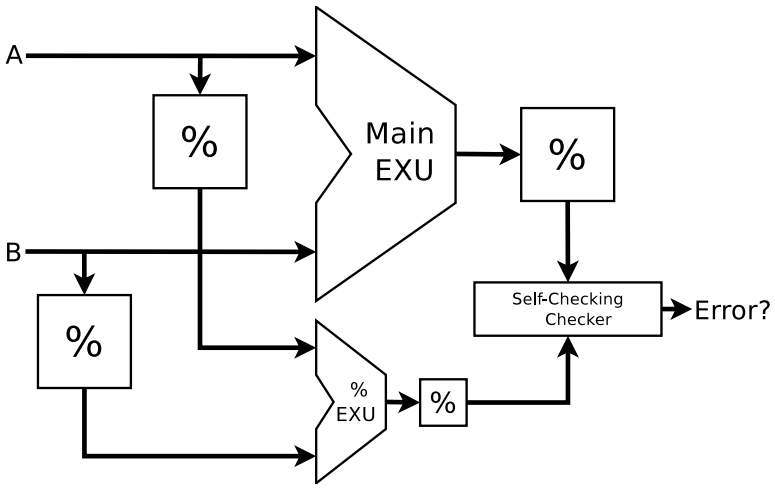


Figure 1.1: An overview of the residue code error detection process.

Residue codes are popular for their simplicity, for their low implementation overhead, for providing separability between the residue code and the result of regular arithmetic, and for having provable properties under unidirectional error models. We intend to investigate their novel application for low overhead error detection and correction in modern processor architectures; this report begins this work by applying residue checking to several processor datapaths using modern word lengths, tools, and technology libraries.

1.1.1 Choosing an Appropriate Residue Code

In general, any number greater than one can serve as a residue code, and any odd number will provide decent error coverage. However, this study is restricted to a subset of *low-cost residue codes*, which are defined as $[rc \in 2^n - 1, 2^n; n \in \mathbb{N}]$. This form is chosen in order to allow the residue to be calculated in an inexpensive manner [9, 10].

Choosing the best residue code for a given design is a non-trivial problem. It is a decision which depends on the error model adopted by the engineer, and the amount of error detection ability sought. This section briefly discusses the most popular error models, and provides a preliminarily investigation into the cost of complete protection with residue codes, taking minimal assumptions about the errors themselves.

The arithmetic weight of an error, aw, is defined to be the number of digits in a minimal binary signed-digit representation of the number. In other words, the arithmetic weight of a number N is the minimum number of non-zero digits in the following equation, given that $[a_m \in \{-1, 0, 1\}; \forall m \in \mathbb{N}]$.

$$N = a_n r^n + a_{n-1} r^{n-1} + a_{n-2} r^{n-2} + ... + a_1 r^1 + a_0 \tag{1.2}$$

1.1.1.1 Concurrency of Errors

The first distinction to be made when classifying and categorizing errors has to do with the concurrency of faults—whether a single or multiple components in the circuit are expected to malfunction at the same time.

Single Event Upsets (SEUs):

The Single Event Upset (SEU) error model allows up to one component in the fault resistant circuit to malfunction at a time. Given that the circuit is protected against the incidence of SEUs, the circuit should guarantee correct behavior with a high probability despite the faulty component. The single event upset error model does not make any assumptions about the number of perturbed bits at the output of a logic block, given a single disruptive event within the logic itself. While specific fault injection research is needed for the field of computer arithmetic, an analysis of an Alpha-like processor shows that roughly 17% of SEUs in arbitrary logic result in multiple bit perturbations later in the pipeline [4]. Algirdas Avizienis uses the terms *local fault* and *distributed fault* to describe SEUs which propagate and are latched into one and multiple bits, respectively [11]. The same convention is used for the remainder of this report.

Multiple Event Upsets:

Multiple event upsets are particularly difficult to protect against because they relax the assumption that only one component of the fault-resistant circuit may fail at a time. This leads to a greater amount of redundancy needed to make any guarantees about error coverage [12]. While classical error protection techniques, including residue checking, can deal with multiple faults in the common case, there is further evidence that classical techniques may reduce the incidence of multiple errors from ever occurring [11, 13]. For simplicity, multiple event upsets are not considered in this study.

1.1.1.2 Error Model

In addition to the concurrency of expected errors, it is helpful to fit a model to the nature of the errors themselves. There are some situations where only errors of a certain type may occur, and a lower overhead error detection or correction scheme may be applied as a result. There are three general forms of errors which are of importance to arithmetic error detecting codes [14]. They are shown below, in the order of the most restrictive model to the least restrictive model being applied to the form of the errors.

Asymmetric Errors:

Asymmetric errors may ever only induce $0 \rightarrow 1$ or $1 \rightarrow 0$ errors in the output word. The type of error to expect is known beforehand. We do not consider this type of error; it has been exhaustively covered and is of little importance to the detection of transient errors in general.

Unidirectional Errors:

Unidirectional errors may induce either induce $0 \rightarrow 1$ or $1 \rightarrow 0$ errors in the output word, but not both at the same time. In other words, a malfunctioning unit under the unidirectional error model is guaranteed only to corrupt 0 bits or 1 bits in the valid data at its output.

While it seems unintuitive, this error model is popular because it has been shown that certain classes of errors, such as errors due to voltage droop and errors induced in LSI memories, fall into this category [15]. Also, circuit-level techniques exist which can make observing a unidirectional error more likely [13].

Residue codes have been successfully modified to be unidirectional error detecting codes, by using the check $|N|_{A'} = A - (N \bmod A)$ instead of $|N|_A = N \bmod A$ [11]. Under an inverse residue code, Equation 1.1 changes to resemble Equation 1.3. This type of error code is desirable for unidirectional errors because an inverse residue code of width n (modulo 2^{n-1}) is guaranteed to protect against *all* unidirectional errors with an absolute arithmetic weight of less than or equal to $(n-1)$ [16].

$$|a \oplus b|_{A'} \stackrel{?}{=} \left||a|_{A'} \oplus |b|_{A'}\right|_{A} \tag{1.3}$$

Symmetric Errors:

The model of symmetric errors makes no assumption about the direction of induced errors. The output word of a faulty component may have suffered $0 \rightarrow 1$, $1 \rightarrow 0$, or both $0 \rightarrow 1$ and $1 \rightarrow 0$ errors simultaneously. Due to the lack of studies targeting our technology and the degree of automated circuit design used during the synthesis process, the symmetric error model is most appropriate for this research.

1.1.1.3 Other Error Models

Some studies find it is useful to further divide the error landscape according to the period of time that a fault manifests itself. However, these classifications are not relevant to this study, which should be able to detect errors of all time periods with equal probability.

1.1.1.4 Error Magnitude

The severity of an error is often expressed in literature in terms of the arithmetic weight of the difference between the proper value, which would have resulted from non-faulty execution, and the corrupted value. This can be thought of as the Hamming weight of the non-adjacent normal form of the distance. Given this analysis, it is evident that the expected arithmetic weight of a completely random error $E[aw] = \frac{m}{3}$ for an m bit number, and $0 \leq$

$aw \leq \frac{m}{2}$. Figure 1.2 verifies these intuitions though simulation, by empirically examining the distribution of the arithmetic weight of a randomly altered 64-bit number.

In practice, errors induced by particle strikes in an m bit circuit rarely approach the upper limit of $\frac{m}{2}$, or even the expected value of $\frac{m}{3}$. This is because SEU errors do not often severely perturb the data, but rather propagate to the outputs in an unpredictable, yet biased, way. Figure 1.3 shows, through simulation, the arithmetic weights induced in a simple single-cycle 64-bit ripple-carry adder under the influence of an SEU. It can be seen that $|aw| \leq 1$, meaning this circuit always produces local faults in the event of a single error. As such, the circuit is guaranteed to be protected by any odd residue code. In general, the severity of errors induced at the output of a component are the product of that component's design, and where the error strike occurs. Furthermore, the impact of SEUs may increase for any component which operates over multiple cycles in a serial manner [11, 17]. Analysis of component error susceptibility must be done on a circuit by circuit basis; this has been done both analytically as well as through simulation in the past [11, 18, 19, 20, 3].

Because transient errors in circuits rarely approach their theoretical intensity bound, it is difficult to predict the strength of error protection necessary for an arbitrary circuit. Without making any assumption as to the magnitude or sign of errors induced by an SEU, it is possible to provide complete coverage against *any* error, though to do so requires a worst-case assumption about

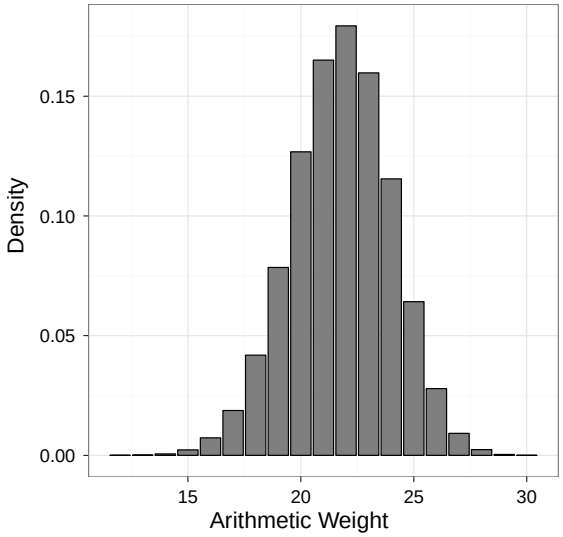


Figure 1.2: The arithmetic weight, aw, of a randomly altered 64-bit number. $E[aw] = 21.333$, $0 \leq aw \leq 32$.

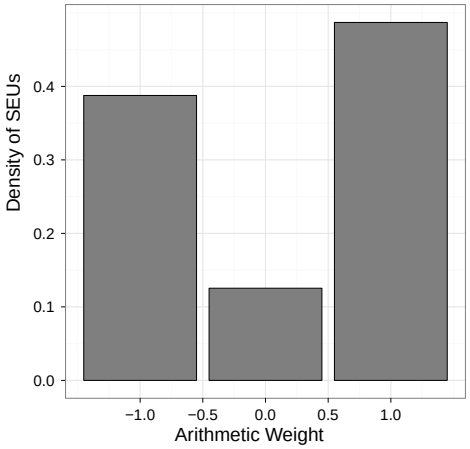


Figure 1.3: The arithmetic weight of an SEU in a simple ripple carry adder.

both the error model and magnitude of errors.

Errors are missed with a single residue code when the difference between the faulty value and the sought value is equal to a multiple of the check modulo. The most practical way to completely protect a number with a residue code is to use multiple relatively prime residues, which increases the effective period of the residue check by the product of the residues. In order to completely protect an m-bit number, one must choose k relatively prime residues such that $r_{k-1} * rk - 2 * ... * r_1 * r_0 \geq 2^m$. This has obvious parallels to the field of residue arithmetic, where multiple coprime residues are sought to perform arithmetic with numbers of a certain dynamic range [21].

1.1.2 The Cost of Error Protection

The overall cost of implementing error detection with residue codes depends on the amount of error detection coverage required, the circuits being protected, and the types of errors to protect against. For a fixed detection rate, error detecting codes can have a differing cost of implementation based upon the context, form, and severity of faults that are expected. In addition, the circuits that are protected influence the cost of error protection, both by defining the appropriate error model as well as by requiring different operations to be performed on the residue coded operands.

While residue codes are well-behaved under a unidirectional error model, the error detection coverage of a given error code is less predictable for symmetric errors. Figure 1.4(a) demonstrates, through an analytical model, the

varied effectiveness of small low-cost residue codes at protecting against even relatively minor symmetric errors. No assumption is made about the distribution of the numbers that make up the erroneous bits, only the arithmetic weights are defined. The erroneous bits are selected uniformly via sampling without replacement, and their signs are randomly determined. All residue codes can detect errors with $aw = 1$ unconditionally. In contrast, silent errors with arithmetic weights of 2 are epidemic for single low-cost residue codes, owning to the fact that every low-cost modulo, itself, has an arithmetic weight of 2 [11]. In order to make any sort of reasonable expectation of error detection rates, a fairly large residue code must be used.

Biresidues or multiresidue codes are a possible solution to both decrease the failure rate for susceptible errors (those with $aw = 2$), as well as increase the overall detection coverage of the residue code. Table 1.1.2 shows the error coverage for a set of small low-cost residues, as well as the improved coverage gained from using biresidue and multiresidue combinations of these numbers. It can be seen that combinations of residues perform approximately as well as the residue nearest the product of their parts. Exhaustive simulation verifies that, for any m-bit number, any combination of residues whose product is greater or equal to the dynamic range of 2^m will provide complete error coverage. It should be noted that complete error coverage for detection may be wasteful in the common case where some orthogonal mechanism is used for recovery, and a weaker error model may be assumed.

For comparison, Figure 1.4(b) demonstrates the use of inverse residue

Table 1.1: Combinations of residues can be used to improve detection accuracy for small low-cost residue codes. The detection accuracy of several small residues, combinations of residues, and larger residues is shown.

Residue Code(s)	E.D. Rate, $aw = 2$	Average E.D. Rate, $2 \leq aw \leq 6$
2	0.49956	0.64381
3	0.83333	0.85303
5	0.90315	0.95490
7	0.93315	0.97726
15	0.97314	0.99364
31	0.99111	0.99638
255	0.99999	0.99999
(3,5) $\approx$ 15	0.98356	0.99540
(2,3,5) $\approx$ 31	0.99230	0.99846
(2,3,5,7) $\approx$ 255	0.99952	0.99998

codes, which are often applied for their ability to strengthen error detection. It can be seen that the inverse residue code successfully detects all error intensities, but still requires a large residue code to detect all errors. However, our results confirm the previously derived findings that the inverse residue codes provide unambiguously better coverage than do normal residue codes, though they do not provide any better guarantees for symmetric errors [11]. While it is not directly tied to this study, Appendix A verifies the (guaranteed) behavior of inverse residue codes under a unidirectional error model.

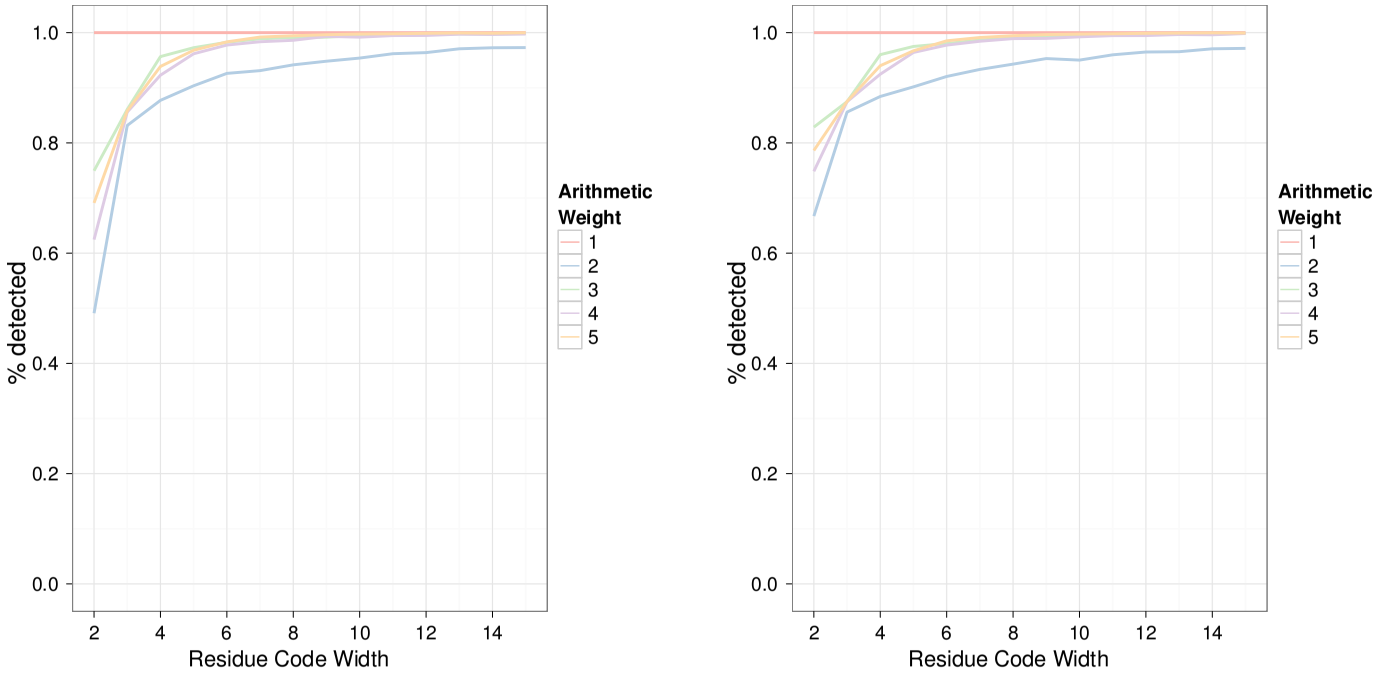
The implementation cost of different moduli is an orthogonal area of concern when designing an error detecting system. Figure 1.5 shows the relative area and delay of different 64-bit low-cost residue circuits. In order to generate a spectrum of results, the flexible delay-optimized parallel prefix

adder from the Synopsys DesignWare library is used, and all possible design points are identified by the compiler while optimizing for area under different delay constraints.

There are several notable results from the area/delay analysis. It is apparent that given no delay constraint, any width of modulus costs the same area—in general, the tradeoff exists between spending area on logic (for large moduli), or interconnect (for small moduli). As expected, a power-of-two modulus typically is computed more cheaply than a low-cost (odd) modulus, since normal twos compliment adder trees may be used. Also, larger moduli typically require a moderate amount more area than a smaller modulus to achieve the same delay. However, at least for the DesignWare component libraries, some adders get more efficiently compiled than others, such that for a strict delay budget, a larger modulus (typically $\frac{m}{2}$) may be the best choice. The added complexity and power due to propagating a larger modulus throughout the residue arithmetic and checking circuitry is not included in this analysis, however.

This study, focuses on single event upsets, which induce symmetric errors somewhere at their output after propagating through logic. For simplicity, and to mimic prior studies, a modulo 3 generator and checker is implemented. This should provide decent, yet not complete, coverage for SEUs that strike the circuit [22, 18]. The overheads associated with a stronger residue check modulus, 255, which has a much higher coverage than modulo 3, are also provided. Finally, the added cost to provide complete error protection using a

combination of 3 moduli is investigated. The moduli chosen to achieve our desired dynamic range ($2^{\frac{m}{2}}$, $2^{\frac{m}{2}} - 1$, 7 for an m-bit word length) are selected using the systematic method proposed in prior literature [10].



(a) Low-Cost Residue Codes

(b) Inverse Low-Cost Residue Codes

Figure 1.4: The effectiveness of low-cost residue codes and inverse residue codes against arithmetic errors with differing weights. Data for a 64-bit number are shown.

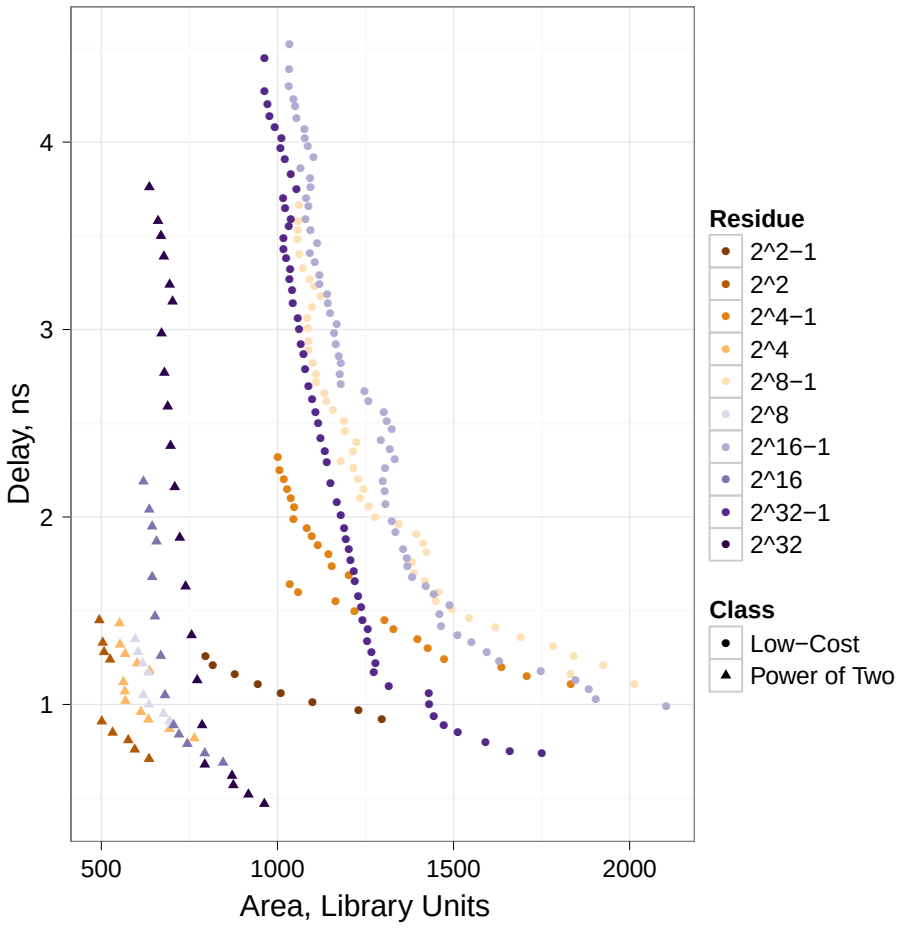


Figure 1.5: The relative area and delay of different residue codes.

Chapter 2

Implementation

2.1 Framework and Methodology

Part of this research is to develop the infrastructure to conduct meaningful circuit-level architectural studies of resiliency techniques in the future. Towards that end, we have begun developing a platform for early design exploration, functional verification, and automatic timing and power analysis. This platform is composed of a parametrized library of residue components, which support a variety of word and input widths. This platform also has a series of scripts for automatic testbench creation (with exhaustive or random inputs), and the exploration of the timing and delay characteristics of a design. Automatic generation of user-annotated switching activity for power analysis is also supported, though power analysis is absent from this study due to time restrictions.

Empirically, this study focuses on the implementation of residue checking in three modern processor datapaths, which are described in Table 2.1. The Synopsys DesignWare IP library was used for the designs, and synthesis was performed using the Synopsys toolchain, targeting the 45nm Nangate Open Cell Library [23, 24]. All designs are compiled using the Synopsys Design

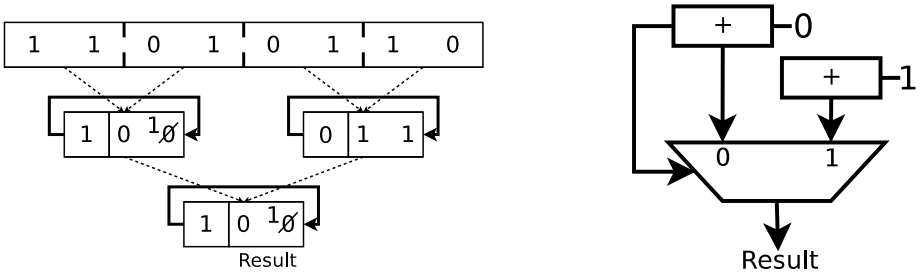
Table 2.1: The datapaths which were implemented and analyzed.

Datapath Width	Target Domain
16	Embedded
32	General Purpose
64	Scientific

Compiler with the `-map_effort medium` and `-area_effort high` optimization flags.

2.2 Implementation Details

Our implementation of residue checking focuses in part on the moduli of the form $2^n - 1$, In order to calculate this class of low-cost residue in a single cycle, a parallel tree of wrap-around-carry adders was used, as can be seen in Figure 2.1(a). A simple general implementation of a wrap-around carry adder is found in Figure 2.1(b).



(a) $1 \equiv 214 \bmod 3$ (b) Wrap-Around Adder

Figure 2.1: Taking the residue of a number with a low-cost residue code, and a possible implementation of a wrap-around carry adder.

In order to simplify the error detection process, the ALU, shifter, and a multiply-accumulate unit (MACC) are logically grouped into a higher-level entity called the *execution unit* (EXU). A circuit-level diagram depicting the residue code checking process is given in Figure 2.2. In lieu of having a specialized EXU which operates under modular arithmetic, the residue of the output is taken from a reduced normal EXU. This design differs from Figure 2.2 only in that it is pipelined, to represent the flow down a normal datapath. A cycle time of $2ns = 500MHz$ is targeted for all designs, with a single-cycle MACC unit. This represents a moderately fast design point, after which large penalties are incurred to gain further speed. An error is detected at the beginning of the second stage following the output of the EXU, which is one full cycle after the completion of the multiplier, and two full cycles after the completion of the shifter and adder.

The EXU design is based upon the interface of the OpenSPARC T1 and a residue hardened ALU by Kinniment, Sayers & Chester [25, 26]. The ALU, shifter, and MACC units are fully parametrized using the Synopsys DesignWare library.

The implementation of Figure 2.2 presented in this study does not have a self-checking checker to test for equality between the separable halves of our error code word. A simple XOR-based equality check is substituted, due to time constraints. A small amount of miscellaneous overhead is added to approximate missing features of the design, including the self-checking checker. Future implementations of this datapath will have a self-checking checker;

Table 2.2: The supported EXU interface and operations.

EXU	ADD	SFT/ROT	MACC	Functional Unit	Operation
000	X	X	X	Control	Pass A
001	X	X	X	Control	Pass B
010	X	X	X	Control	Pass 1
011	X	X	X	Control	Pass 0
100	00 001 10 11	X	X	ADD	Add Add with Carry Subtract Subtract with Carry
101	X	00 01 10 11	X	Shifter Rotater	Left Shift (Logical) Right Shift (Logical) Left Rotate Right Rotate
110	X	X	00 01 10 11	MACC Unit	Unsigned Multiply Signed Multiply Unsigned MACC Signed MACC

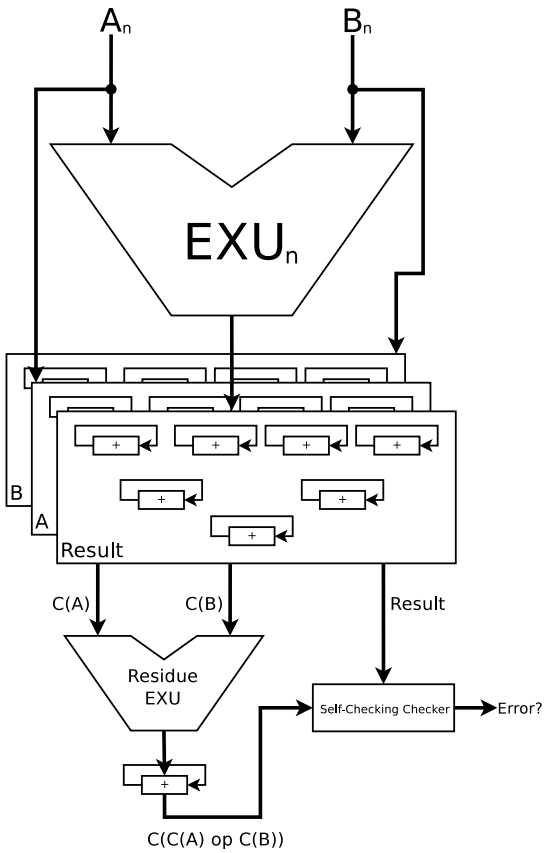


Figure 2.2: An unpipelined, circuit-level diagram of the residue checking error detecting scheme.

the results presented here should be considered as a rough estimate of the overheads for a tuned, totally self-checking residue code circuit.

The overheads of complete error protection are approximated through the use of the multiresidues found in Table 2.2. These moduli are chosen in order to minimize the area of our targeted design, by using the insights that the 2^n modulus should be as large as possible, and that a large low-cost

Table 2.3: The moduli chosen to implement complete error detection.

Word Length	Modulo 1	Modulo 2	Modulo 3
16	7	$2^8 - 1$	2^8
32	7	$2^{16} - 1$	2^{16}
64	7	$2^{32} - 1$	2^{32}

modulus of the form $2^{\frac{m}{2}} - 1$ has an efficient implementation in our design. In addition, these designs are optimized for moduli that evenly divide the machine word length, and odd moduli suffer a corresponding area penalty. Further optimizations for odd-width low-cost moduli may change the choice of coprime bases in the future.

2.3 The Overheads of Error Detection

Table 2.4 shows the estimated area overheads of our residue checking error detecting implementation. The selected moduli can be thought of as providing an approximate lower and upper bound on the possible spectrum of implementation overheads, due to their all-encompassing natures. It is apparent that the residue generation units dominate the area cost for embedded processors, whereas residue arithmetic and assorted other checking costs become more prevalent with larger machines. In embedded processors, the high cost of generating the residues at each functional unit can be mitigated by propagating coded outputs between multiple protected units in the datapath. However, in this case, parity checking is needed at the inputs to ensure the absence of transmission errors [18, 27]. Assuming a reasonable overhead for

parity checking at the inputs, this reduces the overhead of protecting a 16-bit EXU with a modulus of 3 from about 35% to about 25% of the original area.

In general, the relative cost of residue checking is much higher on the embedded processor than on the machine modeling a high performance scientific processor. This is mainly because the area of the EXU increases non-linearly with word size, as can be seen from Figure 2.3. This is *not* due to the aggressive targeting of a $2ns$ cycle time—it can be seen that this frequency is right at the area of diminishing returns for all processors. Rather, it is symptomatic of the necessary scaling of the logic itself.

The high overhead associated with complete error detection indicates that it would not be competitive against other architectural techniques, such as duplication, which are able to detect errors regardless of the error model. However, the information contained in the moduli for complete error detection is sufficient to provide an architectural mechanism for concurrent error correction [28]. The error correcting aspect of multiresidue codes, while unexplored in this paper, may justify the large overheads required for complete detection. In addition, any small relaxation of the requirement for complete error coverage greatly reduces the required overheads—being able to tolerate just one possible error over the entire dynamic range of a number halves the amount of information that the residue code must contain. Due to the nature of errors and the rate at which error detection increases with error code widths, it is possible to provide a high degree of error protection for most machines with a moderately sized residue code with a reasonably low overhead. This is

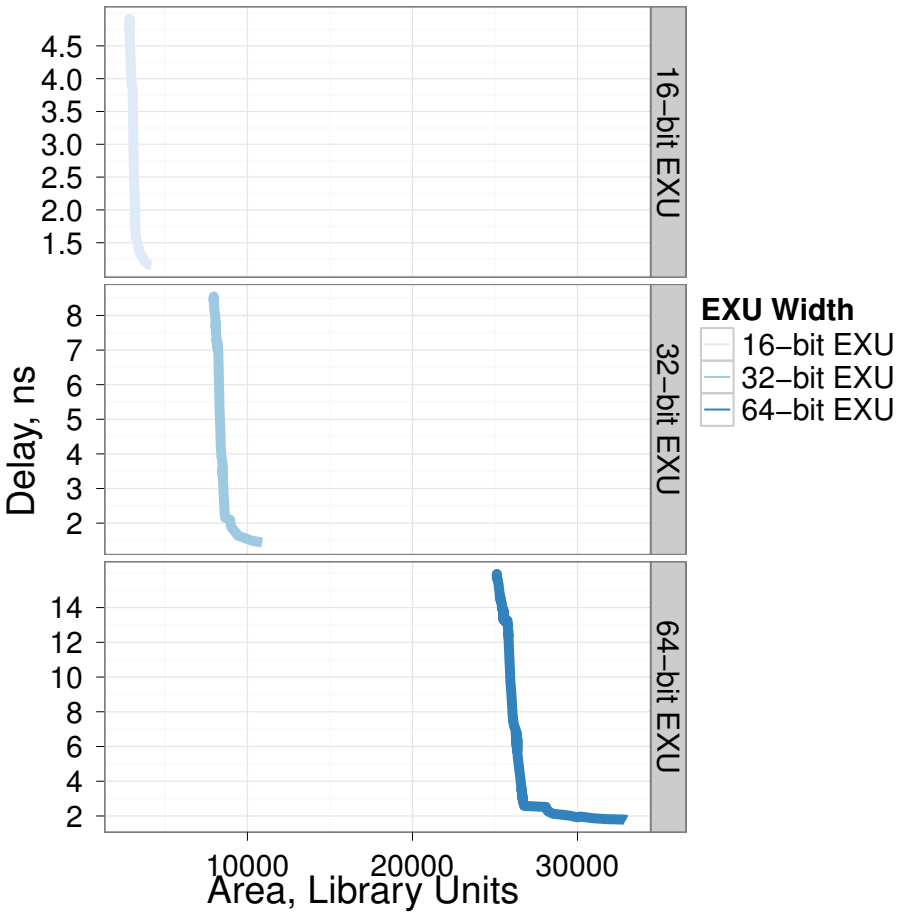


Figure 2.3: The relative area and delay of different EXU designs.

demonstrated by the use of modulo $255 = 2^8 - 1$ in the following data.

Table 2.4: The area overheads from minimal and complete error coverage via residue checking.

	16-bit Machine			
	% Total Overhead	% Residue Generation	% Residue Arithmetic	% Misc
Mod 3	35.31	52.29	25.54	22.18
Mod 255	75.41	29.61	54.81	15.58
Mod $\{7, 2^8 - 1, 2^8\}$	204.7	34.00	46.86	19.14

	32-bit Machine			
	Total Overhead	% Residue Generation	% Residue Arithmetic	% Misc
Mod 3	22.70	58.30	14.11	27.58
Mod 255	43.18	43.43	34.01	22.56
Mod $\{7, 2^{16} - 1, 2^{16}\}$	158.0	32.20	47.97	19.83

	64-bit Machine			
	Total Overhead	% Residue Generation	% Residue Arithmetic	% Misc
Mod 3	8.37	32.04	17.01	50.94
Mod 255	14.64	29.39	30.35	40.26
Mod $\{7, 2^{32} - 1, 2^{32}\}$	117.8	29.32	52.58	18.10

Chapter 3

Future Work and Conclusion

3.1 Prior Work

Residue codes are part of a class of arithmetic error control codes, which are preserved under arithmetic operations. Residue codes were chosen over other arithmetic codes, such as AN codes, for the simplicity and separability of their design [29]. Nonarithmetic error codes may also be applied to integer operations through a process called check prediction. Parity codes [29], checksum codes [30], parity-based linear codes [31], and Berger codes [32] have been successfully applied to arithmetic. These techniques were eschewed for residue checking due to its low cost of application. In addition, there are many different architectural, software and circuit mechanisms which can detect and correct transient errors in a processor; it is not within the scope of this report to attempt to describe any of them.

Because of their relatively low overhead and good separability, residue codes have enjoyed popularity in the commercial sector. Multipliers, in particular, are often protected by residue checking because of the efficient implementation it allows [33, 18]. Some notable error detecting implementations include the application of parity prediction and residue codes in the STAR computer

and a residue checking processor by Kinniment, Sayers & Chester [34, 26]. In addition, IBM has a long history of putting both parity prediction as well as residue checking into their chips, including the System/9000 processor and vector co-processor, the IBM eServer z900, the IBM eServer z10, and the Power6 [17, 35, 36, 37].

3.2 Future Work

Future work will analyze the power overheads of residue checking, similar to the area analysis presented in this report. The experimental infrastructure which aids the area exploration can mostly be shared with power studies; however, the added work required to simulate workloads and gather traces of switching activity for dynamic power estimation prevented us from presenting any meaningful power results in this report.

There is still remaining work to be done in the basic design of the residue checking datapath. The self-checking checker needs to be implemented, and the totally self-checking nature of the circuit should be verified. In addition, there are many unexplored areas which can be tuned to reduce the area, and most likely power, of our design. Some possibilities include:

1. Pipelining the datapath logic, or splitting the residue logic, to allow a better balancing of the residue pipeline stages.
2. Further relaxing the semantics for error detection latency in order to allow a simplified and more deeply pipelined residue checking design.

3. Optimizing our low-cost residue code generation for the case when base widths do not divide evenly into the machine word length.
4. Coupling the control of multiresidue EXUs in a SIMD-style manner, to reduce control overheads.

From previous literature and the overheads observed in the initial results, it is obvious that (1) error detection overheads scale favorably with the machine word size, (2) logic for modular arithmetic and result checking dominates the cost of residue checking in systems with strong protection. An area of future study which will be explored is the targeted application of residue checking to SIMD computer architectures. All execution units in a SIMD machine, by construction, perform the same operation, such that the inputs and outputs of each may be treated as a very long word. If overflow between the packed operands is handled in the same manner as with normal residue checking, then the resulting residue codes are still preserved under arithmetic operations. Figure 3.1(a) and Figure 3.1(b) illustrate this concept. Breaking traditional EXU boundaries has the effect of increasing the cost of residue creation; the modified circuit has double the number of adders spent on residue generation and probably roughly doubles the area spent in this domain. However, treating a SIMD word as one long input greatly reduces the number of check EXUs and self-checking checkers which are required. An n wide SIMD circuit only requires *one* of each (instead of n), which becomes especially valuable for large input moduli or large (packed) machine word lengths. In addition, the reduction of error detection costs may make this approach practical for smaller word

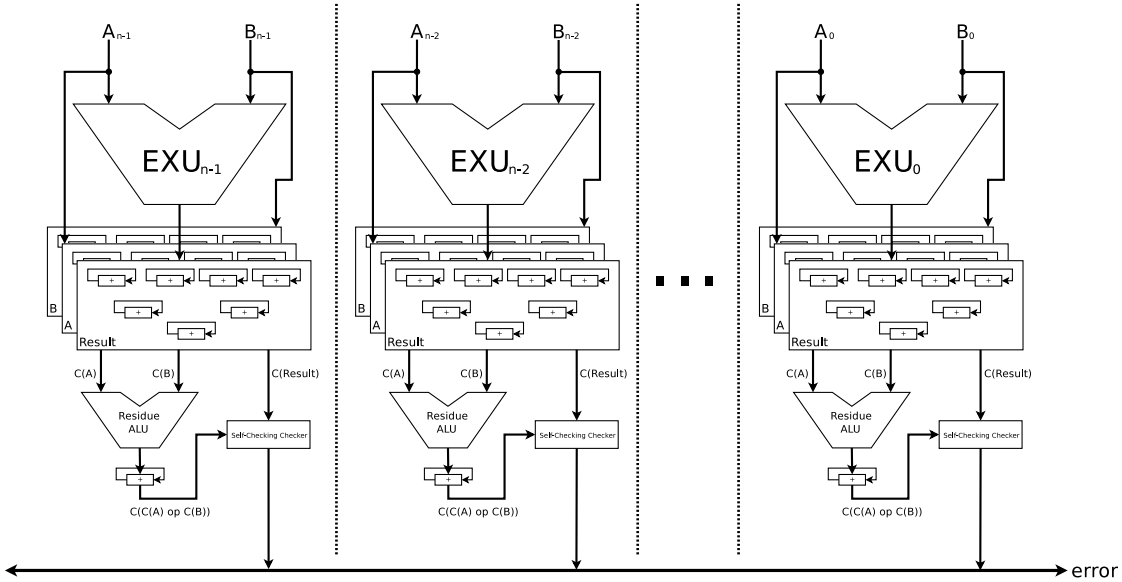
length machines, which are typically expensive to protect with residue codes. Also, this scheme may make very strong error protection practical, whereas it is prohibitively expensive to protect a single EXU completely via residue codes. Two things seem to make strong error protection more practical in the SIMD scheme. First, as this study has shown, the cost of strong error protection is dominated by the modular arithmetic and checkers, whose aggregate overhead may be greatly reduced. Also, an SEU, even in the general case, is guaranteed not to transcend the fixed SIMD packed unit boundaries. Therefore, in an intelligently applied residue checking scheme, one may be able to detect (and perhaps correct) any single digit error in a manner which scales favorably with the number of SIMD lanes.

3.3 Conclusion

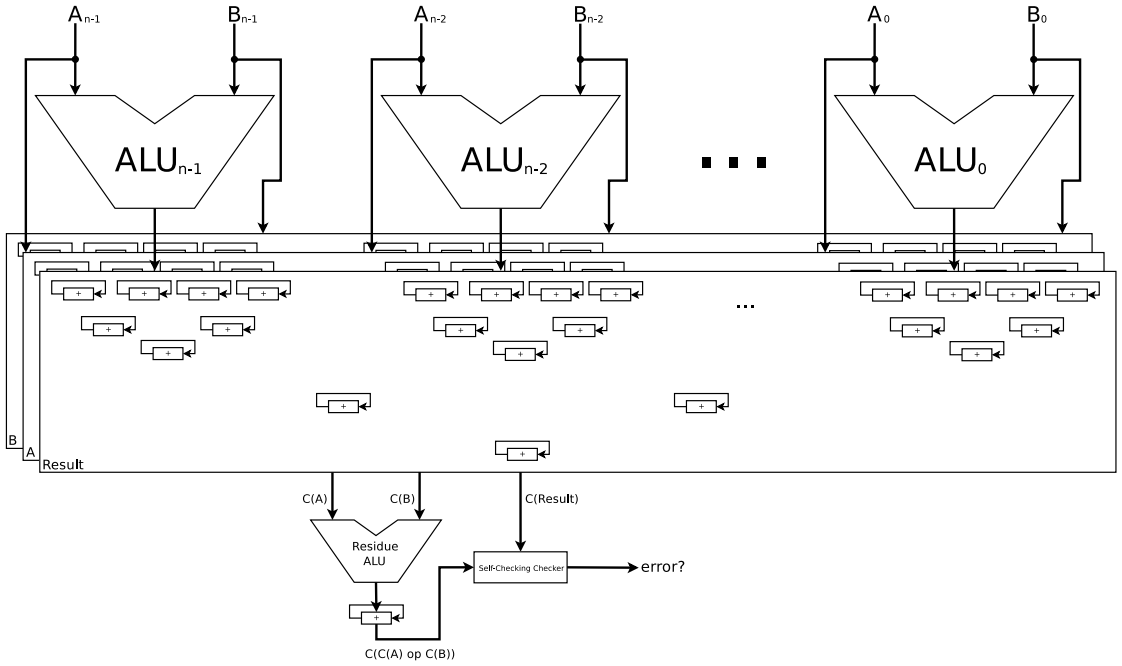
This work is *not* meant to represent a novel development in the field of residue arithmetic or error control codes; also, this is far from the first study to implement an error detection system using residue checkers. The goal of this work is to develop a platform and body of knowledge which will enable the study of low overhead arithmetic error protection and correction in a wider variety of computer architectures. This report represents the first step towards developing such infrastructure. A broad overview of residue codes is given, while stressing the importance of explicitly recognizing the different error models. The nature of expected errors is, by far, the most important parameter when designing an efficient residue checked circuit. As has been shown, the

difference is huge between the area overhead required for the weakest low-cost residue code $(2^3 - 1)$ and an error code providing complete coverage of any error.

An orthogonal contribution of this work is the description of an experimental platform for the empirical study of residue checking in modern processors. A parametrized model of a residue checker has been developed, a basic tools have been created to aid in design space exploration and fault injection. Ultimately, we hope to use these tools in order to apply low overhead arithmetic error protection and correction to a wide variety of different architectures.



(a) Naïve Application of Residue Codes



(b) Breaking Traditional EXU Boundaries

Figure 3.1: A possible future area of exploration: the efficient application of residue codes to SIMD architectures.

Appendices

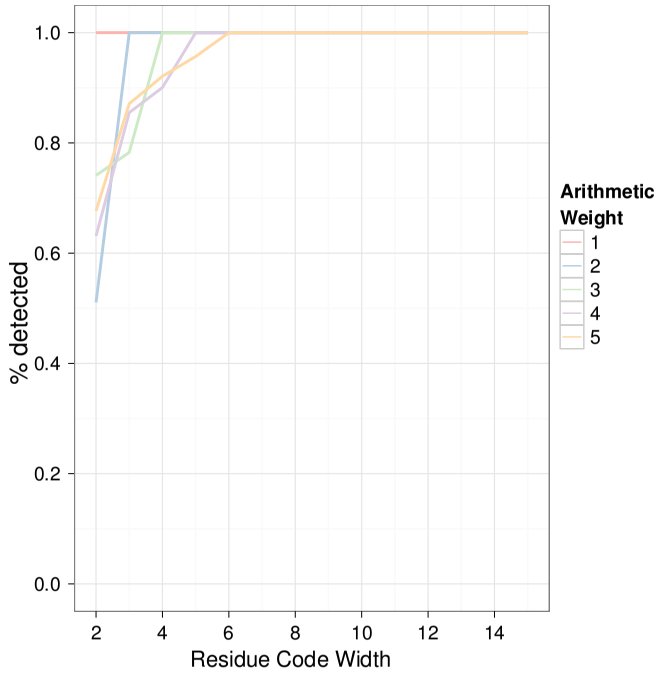
Appendix A

Residue Checking in a Unidirectional Channel

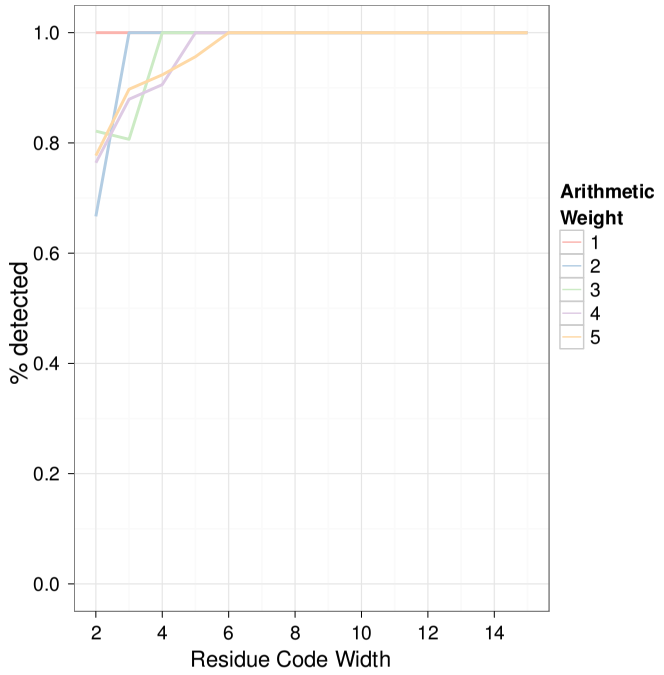
As is mentioned in 1.1.1.2, an inverse residue code of width n (modulo 2^{n-1}) is guaranteed to protect against all unidirectional errors with an absolute arithmetic weight of less than or equal to $(n-1)$. For completeness, and to verify the functionality of our error injection framework, the behavior of residue codes and inverse residue codes is simulated in a unidirectional channel under a uniform error assumption. This experiment is analogous to that which produced Figure A.2(a) and Figure A.2(b), only under a unidirectional instead of symmetric error model.

It can be seen that the correctness guarantee of inverse residue codes is maintained, and that inverse residue codes provide higher overall error detection rates. There are, theoretically, some errors which will fail for normal residue codes such that they do not hold the same bounded correctness guarantee as the inverse residue codes. However, not enough of such errors occured in order for them to be noticeable by inspection.

Figure A.1: The effectiveness of low-cost residue codes and inverse residue codes against unidirectional errors of differing weights. Data for a 64-bit number are shown.



(a) Low-Cost Residue Codes



(b) Inverse Low-Cost Residue Codes

Bibliography

[1] T. Karnik and P. Hazucha. "Characterization of soft errors caused by single event upsets in CMOS processes". In: *IEEE Transactions on Dependable and Secure Computing* 1.2 (2004), pp. 128 –143.

[2] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings of the International Conference on Dependable Systems and Networks* (2002), pp. 389–398.

[3] G.P. Saggese et al. "Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic". In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2005, pp. 760 –769.

[4] G.P. Saggese et al. "An Experimental Study of Soft Errors in Microprocessors". In: *IEEE Micro* 25.6 (2005).

[5] M.J. Ashjaee and S.M. Reddy. "On Totally Self-Checking Checkers for Separable Codes". In: *IEEE Transactions on Computers* C-26.8 (1977), pp. 737 –744.

[6] N. Gaitanis. "Totally Self-Checking Checkers for Low-Cost Arithmetic Codes". In: *IEEE Transactions on Computers* C-34.7 (1985), pp. 596 –601.

[7] D. Nikolos, A.M. Paschalis, and G. Philokyprou. "Efficient design of totally self-checking checkers for all low-cost arithmetic codes". In: *IEEE Transactions on Computers* 37.7 (1988), pp. 807 –814.

[8] S.J. Piestrak. "Self-testing checkers for arithmetic codes with any check base A". In: *1991. Proceedings of the Pacific Rim International Symposium on Fault Tolerant Systems*. 1991, pp. 162 –167.

[9] Nicholas S. Szabó and Richard I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. New York: McGraw-Hill, Inc., 1967.

[10] B. Parhami. "On Equivalences and Fair Comparisons Among Residue Number Systems with Special Moduli". In: *44th Asilomar Conference on Signals, Systems and Computers*. 2010.

[11] A. Avizienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design". In: *IEEE Transactions on Computers* C-20.11 (1971), pp. 1322–1331.

[12] C.A.L. Lisbôa, E. Schüler, and L. Carro. "Going beyond TMR for protection against multiple faults". In: *18th Symposium on Integrated Circuits and Systems Design* (2005).

[13] D. Rossi et al. "Multiple Transient Faults in Logic:An Issue for Next Generation ICs". In: *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance* (2005), pp. 352–360.

[14] B. Bose and T.R.N. Rao. "Theory of Unidirectional Error Correcting/Detecting Codes". In: *Computers, IEEE Transactions on* C-31.6 (1982), pp. 521 –530.

[15] R.W. Cook et al. "Design of a Self-Checking Microprogram Control". In: *IEEE Transactions on Computers* 22 (3 1973), pp. 255–262.

[16] B. Parhami and A. Avizienis. "Detection of Storage Errors in Mass Memories Using Low-Cost Arithmetic Error Codes". In: *IEEE Transactions on Computers* C-27.4 (1978), pp. 302 –308.

[17] C.L. Chen et al. "Fault-tolerance design of the IBM Enterprise System/9000 Type 9021 processors". In: *IBM Journal of Research and Development* 36.4 (1992), pp. 765–779.

[18] U. Sparmann and S.M. Reddy. "On the effectiveness of residue code checking for parallel two's complement multipliers". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.2 (1996), pp. 227 –239.

[19] S.S. Mukherjee et al. "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor". In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 2003, pp. 29 –40.

[20] H.H.K. Tang. "SEMM-2: A new generation of single-event-effect modeling tools". In: *IBM Journal of Research and Development* 52.3 (2008), pp. 233–244.

[21] B. Parhami. "Computer Arithmetic: Algorithms and Hardware Designs". In: (1999).

[22] W.H. Debany et al. "Effective concurrent test for a parallel-input multiplier using modulo 3". In: *Proceedings of the IEEE VLSI Test Symposium*. 1992, pp. 280 –285.

[23] Synopsys Inc. *Design Compiler*. 2010.

[24] Nangate. *Open Cell Library v1.3*. 2009.

[25] Sun Microsystems. *OpenSPARC T1*. 2005.

[26] D.J. Kinniment, I.L. Sayers, and E.G. Chester. "Design of a reliable and self-testing VLSI datapath using residue coding techniques". In: *IEE Proceedings-E in Computers and Digital Techniques* 133.3 (1986), pp. 169 –179.

[27] B. Parhami. "Approach to the Design of Parity-Checked Arithmetic Circuits". In: *36th Asilomar Conference on Signals, Systems, and Computers*. 2002, pp. 1084–1088.

[28] T. R. N. Rao. "Biresidue Error-Correcting Codes for Computer Arithmetic". In: *IEEE Transactions on Computers* 19.5 (1970), pp. 398–402.

[29] A. Avizienis. "Arithmetic Algorithms for Error-Coded Operands". In: *IEEE Transactions on Computers* C-22.6 (1973), pp. 567 –572.

[30] J.C. Lo. "Reliable floating-point arithmetic algorithms for error-coded operands". In: *Computers, IEEE Transactions on* 43.4 (1994), pp. 400–412.

[31] T.R.N. Rao. *Error Coding for Arithmetic Processors*. Orlando, FL, USA: Academic Press, Inc., 1974.

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

[33] A. Naini et al. "1 GHz HAL SPARC64R Dual Floating Point Unit with RAS features". In: *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*. 2001, pp. 173 –183.

[34] A. Avizienis et al. "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design". In: *IEEE Transactions on Computers* 20.11 (1971), pp. 1312–1321.

[35] L.C. Alves et al. "RAS design for the IBM eServer z900". In: *IBM Journal of Research and Development* 46.4.5 (2002), pp. 503–521.

[36] W.J. Clarke et al. "IBM System z10 design for RAS". In: *IBM Journal of Research and Development* 53.1 (2009), 11:1 –11:11.

[37] M.J. Mack et al. "IBM POWER6 reliability". In: *IBM Journal of Research and Development* 51.6 (2007), pp. 763–774.