DISCLAIMER:

This document does not meet the
current format guidelines of
the Graduate School at
The University of Texas at Austin.

It has been published for
informational use only.

The Dissertation Committee for Michael Brendan Sullivan
certifies that this is the approved version of the following dissertation:

LOW-COST DUPLICATION
FOR SEPARABLE ERROR DETECTION
IN COMPUTER ARITHMETIC

Committee:

_____
Mattan Erez, Supervisor

_____
Earl E. Swartzlander, Jr., Co-Supervisor

_____
Nur A. Touba

_____
Aloysius K. Mok

_____
Michael J. Schulte

LOW-COST DUPLICATION FOR SEPARABLE ERROR DETECTION IN COMPUTER ARITHMETIC

by

Michael Brendan Sullivan, BA; BS; MS; MSE

DISSERTATION

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August 2015

LOW-COST DUPLICATION
FOR SEPARABLE ERROR DETECTION
IN COMPUTER ARITHMETIC

Michael Brendan Sullivan, PhD
The University of Texas at Austin, 2015

Supervisors: Mattan Erez
Earl E. Swartzlander, Jr.

Low-cost arithmetic error detection will be necessary in the future to ensure correct and safe system operation. However, current error detection mechanisms for arithmetic either have high area and energy overheads or are complex and offer incomplete protection against errors. Full duplication is simple, strong, and separable, but often is prohibitively costly. Alternative techniques such as arithmetic error coding require lower hardware and energy overheads than full duplication, but they do so at the expense of high design effort and error coverage holes. The goal of this research is to mitigate the deficiencies of duplication and arithmetic error coding to form an error detection scheme that may be readily employed in future systems. The techniques described by this work use a general duplication technique that employs an alternate number system in the duplicate arithmetic unit. These novel dual modular redundancy organizations are referred to as *low-cost duplication*, and they provide compelling efficiency and coverage advantages over prior arithmetic error detection mechanisms.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION *

Rising levels of integration and decreasing component reliabilities make error protection increasingly important in computer systems. At the same time, the crucial need for energy efficiency necessitates low-cost reliability techniques. The error protection of arithmetic circuitry is typically more expensive than that of memory or data movement; protection of arithmetic has correspondingly been reserved for critical or high-availability applications. Current trends, however, indicate that error protection of the arithmetic pipeline will be necessary in the future for more diverse application areas.

This dissertation investigates a novel class of arithmetic error detection schemes based on a general technique called *low-cost duplication*. These low-cost duplicate units are able to provide substantive strength and efficiency advantages over existing arithmetic error detection schemes. Section 1.1 describes the four design goals that motivate low-cost duplication. Section 1.2 describes the general error detection techniques that are currently in use; these techniques are unable to satisfy all of the aforementioned design goals. Section 1.3 introduces low-cost duplication; a set of error detection schemes based on low-cost duplication are able to simultaneously satisfy all four design goals, providing complete and efficient reliability for the arithmetic pipeline.

1.1 ARITHMETIC ERROR DETECTION DESIGN GOALS

A comprehensive and implementable arithmetic error detection scheme should improve the reliability of the arithmetic pipeline without impacting the design or efficiency of the arithmetic units themselves. Also, the arithmetic error detection circuitry should have low hardware overheads and should readily interface with higher level error recovery systems. More formally, these requirements lead to four design goals for effective arithmetic error detection. This research describes a set of error detection mechanisms that simultaneously satisfy all of these goals.

* Parts of this chapter appear in [1], with Earl E. Swartzlander, Jr. serving as supervisor.

1.1.1 *Strong, Fault-Agnostic Error Detection*

Error detection mechanisms often limit their coverage to restricted classes of faults or errors, such as permanent gate oxide faults [2, 3, 4, 5] or those errors that originate from one gate or cell [6, 7, 8, 9, 10]. There are numerous faults that can affect the datapath of a system, some of which produce error patterns that are not well captured by synthetic models. Due to the uncertainty of fault rates and error manifestation in the arithmetic pipeline it is best for an error detection mechanism to have high error coverage across faults and to not rely on a restrictive error model. A beneficial side effect of this strong error detection is that it simplifies system-level reliability analysis. Without complete error coverage, proper analysis of the system-level effect of undetected errors is complicated and requires intimate knowledge of the expected workloads and arithmetic circuitry.

1.1.2 *Separable Checking*

An error detection mechanism is separable if the checking procedure can operate without any communication from the protected unit. Separability is critical for efficient design development, as it allows error detection to proceed independent of arithmetic. This independence enables the reuse of tested and optimized arithmetic units and also facilitates the modification of arithmetic circuitry without concern for its impact on reliability. Separability also enables the fine-grained gating of error detection, allowing the use of alternate error detection mechanisms where appropriate and avoiding the checking overheads for naturally resilient applications [11, 12, 13].

1.1.3 *Low-Latency, Concurrent Detection*

Arithmetic error detection mechanisms rely on higher-level recovery mechanisms to halt error propagation and restore proper machine function. Low, fixed latency error detection simplifies the design of this error reporting and handling system. It also allows designers to keep all datapath error reporting synchronous, which is important for efficient common-case error recovery at higher system levels [13, 14, 15].

1.1.4 *Low Area and Energy Overheads*

Faults, by their nature, are typically infrequent and unpredictable. While the expected fault rate for the arithmetic pipeline may be high enough to warrant error detection circuitry, this circuitry should not needlessly waste chip area or energy during error-free operation. The importance of efficient error detection is exacerbated by the current trend towards energy-constrained operation.

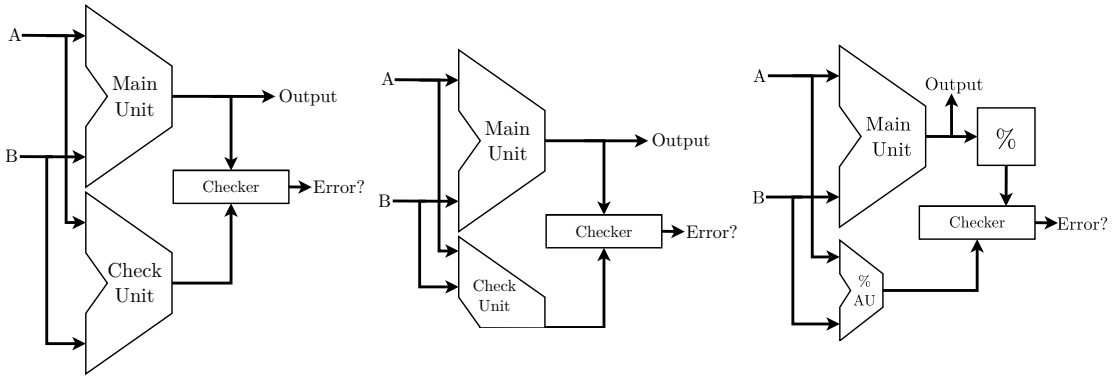
1.2 CURRENT SEPARABLE ARITHMETIC ERROR DETECTION APPROACHES

There are two broad techniques that find wide applicability to separable error detection across different arithmetic operations: duplication and residue checking. Duplication and residue checking each have advantages, but neither approach is able to satisfy all of the aforementioned design goals.

1.2.1 *Duplication, Partial Duplication, and Lazy Duplication*

Coarse-grained spatial duplication, also known as dual modular redundancy (DMR), is simple, intuitive, strong, separable, and general, and it may be applied to any arithmetic operation (as shown in Figure 1.1a). The area and power costs of duplication, however, are often prohibitive. Duplication has only been employed in specialized processors where reliability is of paramount importance, surpassing the need for efficiency [16, 17, 18, 19, 20]; even IBM uses alternative arithmetic error detection mechanisms in some of its notoriously reliable mainframe computers for increased efficiency [21, 22, 23]. Also, full duplication can also have incomplete coverage against certain faults. Timing violations, design faults, and fabrication faults can possibly affect both the main arithmetic unit and duplicate check unit, escaping error detection and possibly resulting in silent data corruption or a system failure.

Partial duplication (Figure 1.1b) is an ad-hoc method that only replicates *some* sub-components to reduce the overheads of full duplication. Two prevailing partial duplication strategies seem to exist: (1) focusing on those components (such as control circuitry) where an error tends to have a massive effect on the output [24], or (2) duplicating just enough circuitry to cover a narrow range of errors, such as those that affect a single bit of the output [25]. Because partial



(a) Duplication (b) Partial Duplication (c) Residue Checking

Figure 1.1: Arithmetic error detection through duplication and residue checking. The % component represents residue generation, and the % arithmetic unit (AU) performs residue generation and modular arithmetic.

duplication is applied in an ad-hoc manner, it can potentially have large and unbounded coverage holes.

Lazy duplication decreases the cost of strict duplication by employing a simplified duplicate checker with a long checking latency [26, 27]. While lazy duplication retains the strength and separability of the DMR organization, it suffers from a long detection latency and can potentially introduce hidden data movement and synchronization costs in higher-level error reporting and recovery mechanisms.

1.2.2 *Residue Checking*

Addition and multiplication can be checked by testing the equality of Equation 1.1, where $|N|_A = N \bmod A$ and $\oplus$ denotes a modular operation [28]. If both sides of Equation 1.1 are equal, it is likely that no error has occurred. If they are not equal, then some error *has* occurred.

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

This scheme is called *residue checking*, and it is depicted by Figure 1.1c. In order to simplify operations, residue checking often relies on a restricted class of residues in the form $A = [2^a - 1; a \in \mathbb{N}]$ [29]. The error coverage of a residue code depends on the width, a, of its checking modulus. In general, large checking

moduli are prohibitively expensive. As a result, residue checking has lower error coverage than duplication [12, 30].

1.3 A NEW APPROACH: LOW-COST DUPLICATION

Full duplication satisfies most of the design goals from Section 1.1, offering strong, separable, low-latency error detection. However, its high area and energy overheads preclude full duplication from being a realistic error detection scheme for most processors. Also, some faults (such as design bugs or environmental timing faults) may lead to correlated errors in both the main and duplicate arithmetic unit, lessening the fault-agnosticism of traditional DMR organizations [31, 32]. Residue checking offers the ability to lower the hardware and energy overheads of arithmetic error protection, but it does so at the expense of high design-effort or holes in error coverage. Partial duplication maintains the simplicity of duplication at lower costs but it has correspondingly low error coverage, meaning that undetected errors can significantly corrupt program state.

This dissertation investigates novel organizations of dual modular redundancy that provide compelling strength and efficiency advantages over simple duplication. Specifically, this work is based off of the intuition that a *low-cost duplicate checker* may employ a non-standard number system with superior speed and efficiency. Also, because the output of the duplicate unit is discarded after error detection, many of the costs traditionally associated with these non-standard number systems (such as the data movement and storage costs of redundant arithmetic or the imprecision accumulation of approximate arithmetic) have little to no impact. Figure 1.2 shows a block diagram of the low-cost duplication process.

Low-cost duplication differs from lazy duplication (as described above in Section 1.2.1) in two respects. First, lazy duplicate checkers perform arithmetic in the same number representation as the main arithmetic unit, and do not fit the definition of a specialized low-cost duplicate unit considered in this dissertation. Also, unlike lazy duplicate checking, low-cost duplication maintains the low-latency error detection of strict duplication in order to simplify implementation, cheapen higher-level recovery, and avoid a dependence on aggressive latency-tolerant microarchitectural features to lessen performance loss.

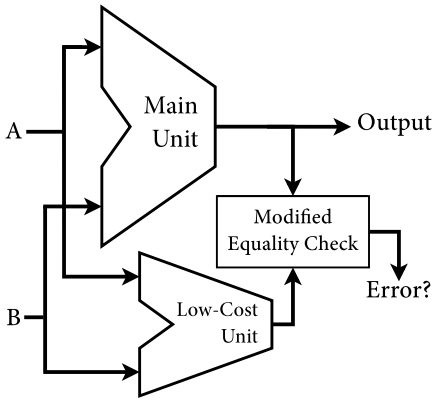


Figure 1.2: A block diagram of low-cost duplication. An operation is performed both with the main arithmetic unit and with a low-cost duplicate unit. This low-cost unit converts the inputs into an alternate number system, using redundant arithmetic and application-specific number representations to increase the speed and efficiency of the duplicate operation. Following the completion of the operation, a modified checker tests the equality of the main arithmetic result with the duplicate result in its alternative format.

Low-cost duplicate units can be thought of in two ways. First, low-cost duplication can be considered as a more efficient replacement for DMR in systems demanding high reliability and availability—in fact, its substantive cost savings may even make low-cost duplication a viable approach in systems where DMR is prohibitively costly. Alternatively, low-cost duplication can be thought of as a more optimized baseline for the evaluation of specialized or best-effort error detection mechanisms. Currently, the assumption is often made that concurrent, strong, low-latency duplication requires >100% implementation overheads. This dissertation suggests, however, that with careful design this may not always be the case.

Table 1.1 gives an overview of the low-cost duplication organizations considered in this research. The schemes use operation-appropriate low-cost number systems to check the results of fixed-point addition, fixed-point multiplication, and floating-point multiplication.

1.4 THE STRUCTURE OF THIS DISSERTATION

The dissertation proceeds as follows. Chapter 2 reviews the basic concepts related to arithmetic error detection and defines the scope of this research. Chap-

Table 1.1: An overview of the low-cost duplication schemes in this dissertation.

Arithmetic Operation	Chapter	Number System
Fixed-Point Addition	3	Carry-Save Arithmetic
Fixed-Point Multiplication	4	Carry-Save or Residue Number System Arithmetic
Floating-Point Multiplication	5	Truncated Fractional Arithmetic

ter 3 develops a parameterized family of low-cost duplicate checkers for addition. Chapter 4 investigates an alternative form of low-cost duplication to protect multiplication. Finally, Chapter 5 examines approximate duplication for the low-cost, precision-proportional protection of floating-point multiplication and Chapter 6 describes how low-cost duplication may have broader applicability to other problem domains.

2 BACKGROUND CONCEPTS †

Before describing the main contribution of this dissertation, some error detection concepts and terminology are reviewed. Section 2.1 introduces some relevant terminology and describes the faults and errors that can affect arithmetic. Sections 2.2–2.5 give further motivation and conceptual background for the design goals (from Section 1.1) of strong, separable, low-latency, and low-cost error detection. Section 2.6 describes the methodology and assumptions used by this research.

2.1 FAULTS AND ERRORS IN COMPUTER ARITHMETIC

This research adopts the established terminology that a *fault* is a physical phenomenon or defect that may cause an error or failure, an *error* is a discrepancy between the intended and actual data in a system, and a *failure* is an instance in time when a system displays a behavior that is contrary to its specification [31, 33]. The aim of this dissertation is to use strong and efficient error detection to prevent arithmetic pipeline errors from developing into a system failure. Correspondingly, an error indicates the corruption of data in an arithmetic unit or its pipeline registers and a failure indicates that erroneous results are propagated to subsequent instructions or to memory.

2.1.1 Fault Mechanisms

A plethora of faults can cause arithmetic errors; a brief description follows of five of the most frequently studied fault mechanisms. These five faults differ in the way that they tend to manifest and whether they persist—some faults tend to be transient, others are active intermittently (only in certain conditions or for some inputs), and others tend to permanently affect a system until the affected component is removed. The rate of these faults is largely unknown in current technologies and this uncertainty is exacerbated by the unpredictability of future technology challenges and limitations [34, 35, 36, 37, 38].

† Parts of this chapter appear in [1], with Earl E. Swartzlander, Jr. serving as supervisor.

2.1.1.1 TRANSIENT FAULTS

Mechanism: Transient (or soft) faults are those that corrupt a single computation. Such faults are typically caused by a chain of events following the strike of an energetic particle from the environment. The fundamental mechanism behind a transient fault is as follows: (1) an energetic ion[1] strikes the active region of a transistor, freeing electron-hole pairs as it passes through the device; (2) freed charge collects at the source and drain of the device, generating current; (3) this generated current propagates through the transistor, regardless of its state; (4) a fault occurs if the current passing through an off-state transistor is sufficient to overcome its load capacitance and erroneously switch devices connected to its output [41].

Properties and Error Activation: Transient faults are rare and unpredictable and can occur any time during system operation. The manifestation of a transient fault is an erroneous current or voltage spike at the output of one or more neighboring transistors [42, 43, 44]. A particle-induced spike may or may not cause an error, and there are different effects that can prevent a transient fault in a pipeline register or logic from becoming microarchitecturally visible [34, 41, 45, 46]. The rate and duration of transient faults is highly variable and it depends on situational factors (altitude, latitude), environmental factors (temperature, position in the solar cycle), supply voltage, and the specifics of fabrication technology [47, 48, 49, 50, 51, 52, 53].

2.1.1.2 PERMANENT FAULTS

Mechanism: There are a variety of mechanisms that can cause a device to become permanently faulty. Extrinsic gate oxide breakdown can occur during the infancy of a weak chip [54, 55], and end-of-life faults may develop over time due to electromigration [56], hot carrier degradation [57], or other time-dependent breakdown mechanisms [36, 58].

1 Such an energetic ion can be produced by multiple events. Two likely candidates are alpha particles that are produced directly by the decay of solder material and package impurities [39] or secondary ionized particles that follow the nuclear spallation of an environmental neutron passing through the processor substrate [40].

Properties and Error Activation: The manifestation of a permanent fault is generally the same as an erroneous open or short-circuited wire. Once permanent faults manifest, their effects tend to persist. Permanent faults generally occur either during the infancy of a chip or after years of continuous operation, leading to the so-called *bathtub curve* for fault rates over time [54].

2.1.1.3 DESIGN FAULTS

Mechanism: Despite aggressive pre-silicon testing, design faults can make their way into a computer system. The most famous arithmetic design fault is probably the Intel Pentium FDIV bug [59], where a lookup table flaw in floating-point division caused an incorrect result for certain inputs. Because of the design flaw, Intel was responsible for replacing faulty processors to astute customers at a reported cost of $475 million USD (approximately $693.5 million USD in today's currency) [60, 61].

Properties and Error Activation: It is likely that any straightforward design fault will be caught during testing. Therefore, uncaught design faults are likely to manifest as errors intermittently for specific inputs. So long as these problematic inputs persist, the design error will manifest. Unlike permanent or fabrication faults, design bugs are completely intrinsic and affect every fabricated chip—in a large system, every node is affected. Also, the manifistation of a design fault is not confined to a single device or wire, and its effect on the output of a component is not well characterized.

2.1.1.4 ENVIRONMENTAL TIMING FAULTS

Mechanism: Environmental timing faults can result when insufficient timing and voltage margins are provided for the operating conditions of a chip. This can occur due to voltage droop [62, 63] age and temperature-related slowdown [37, 64, 65], and on-chip variability [32, 65]. As an example, floating-point design errors have caused a recall of AMD Opteron chips due to a heat density issue where demanding workloads cause timing faults [66].

Properties and Error Activation: Similar to design faults, environmental timing faults are typically intermittent and will persist so long as the operating conditions or supply voltages are outside of the safe range. The manifestation of

a timing fault depends on the detailed timing information of a circuit; variability in devices, the environment, and voltage add uncertanty to the characterization of such faults [32, 63, 65].

2.1.1.5 FABRICATION FAULTS

Mechanism: Deep sub-micron technologies suffer from some fabrication-related fault mechanisms that can lead to manufacturing defects. For instance, step coverage problems during the metalization process can lead to faulty open circuits in a fabricated design [38]. Complete testing of fabrication faults is expensive, and corresponding post-synthesis testing coverage targets of less than 100% fail to catch all faults [67].

Properties and Error Activation: Fabrication faults will typically manifest as short-circuited or open-circuited bridging defects. Their effects will persist throughout the lifetime of an affected chip.

2.1.2 *Error Models*

An arithmetic error occurs when a fault propagates through logic and pipeline state to produce an incorrect result. The direct observation or simulation of faults is difficult and time-consuming and it is highly design and technology-dependent. As such, many error detection mechanisms employ synthetic error models for analysis. Some of the most common and important error models are described below.

2.1.2.1 BITS AND BURSTS

A common error model for memory is to consider a single erroneous bit or a contiguous burst of bits in error. While some studies of application error propagation attempt to use single-bit-flip errors for arithmetic [68, 69, 70], none of the faults from Section 2.1.1 are constrained to produce single-bit errors. Such studies may therefore underestimate the severity of arithmetic errors and correspondingly overestimate the potential for weak error protection techniques to mitigate system failures.

2.1.2.2 ARITHMETIC WEIGHT

The severity of an arithmetic error is sometimes expressed in terms its *arithmetic weight*. This can be thought of as the Hamming weight of the non-adjacent normal form of the distance between the intended and corrupted output value [71, 72]. While the concept of arithmetic weight is important to the theory of arithmetic error detecting (and correcting) codes, it is not especially suited for use as a synthetic error model as faults do not typically manifest in a way that constrained to certain arithmetic weights.[2]

2.1.2.3 SINGLE DEVICE ERRORS

A common error model is to assume a single faulty gate, cell, or node in the arithmetic unit or checker. This model is referred to as a single device (SD) error,[3] and the erroneous output can propagate from the faulty node to affect one or more connected devices in the component. The SD error model captures the effects of most transient, permanent, or fabrication faults, as it is unlikely that such faults affect many simultaneous devices.

2.1.2.4 SINGLE COMPONENT ERRORS

A more general approach than the SD error model is to assume an arbitrary error in any single component. This model, the single component (SC) error model, can capture multiple faults within the arithmetic unit or checker. An example of such a severe fault is a powerful transient error that affects a burst of pipeline latches [43, 73, 74] or neighboring logic cells [44, 75]. Design faults and environmental timing violations can produce arbitrary errors in an arithmetic unit, and as such they may be encompassed by the SC error model so long as it is known that the fault will only affect a single system component. Section 2.2.1 describes how to ensure this property holds in the context of duplication, such

2 Careful arithmetic unit design can constrain the arithmetic weight of errors [10]. Certain co-dependant checkers (as defined later in Section 2.3.1.3) can therefore use arithmetic weight as an effective error model, but this is not true for more separable error detection schemes.

3 In the case of transient faults, this model is often referred to as a single event upset [41]. Confusingly, due to its ability to affect an entire connected component, this error model is also called a *multiple error* [22] or a *single distributed fault* [29].

that a duplicate checker can provide complete coverage against the SC error model.

2.2 BACKGROUND FOR GOAL #1: STRONG AND FAULT-AGNOSTIC ERROR DETECTION

The typical approach to arithmetic error detection is either to attempt a best-effort mechanism with incomplete error coverage, such as residue checking [12, 28, 29, 30, 76, 77], or to try and select the most prevalent faults or errors and to deal with them with specialized mechanisms. Examples of such specialized mechanisms include detectors for transient faults [78, 79], gate oxide faults [2, 3, 4, 5], timing violations [80, 81], fabrication-related faults [82], and design faults [26]. Alternatively, an error detector can focus on a narrow error model, such as the single-device errors [6, 7, 8, 9, 10]; depending on the protected circuit, such an error model may capture the effects of some faults (such as single-event-transient faults, gate oxide faults, and some fabrication-related faults) but may miss others (single-event-multiple-transient faults, timing, violations, and design faults).

While specialized error detection mechanisms boast low area and energy overheads, their use in a comprehensive arithmetic error detection scheme is fraught with difficulty. Choosing the most prevalent fault mode can be costly and difficult, especially given the uncertainty of future technologies, use-cases, environmental conditions and design constraints. Also, whether and how a fault manifests as an error is strongly design dependent. This design dependence introduces additional development complexity—error detection mechanisms that protect against a narrow error model must often dictate or react to the arithmetic unit design to have high coverage, and there can be a tradeoff between arithmetic unit optimization and error coverage [6, 10, 83, 84].

In lieu of using such detection mechanisms, this work proposes the use of low-cost duplication for holistic, fault-agnostic arithmetic error detection. The high error coverage of low-cost duplication avoids the need to tailor error detection towards the most severe and prevalent errors, lessening the onus of fault rate analysis, error modeling, and system-level error propagation and failure modeling on the chip designer.

2.2.1 *Ensuring that the Duplicate Checker is Fault-Agnostic*

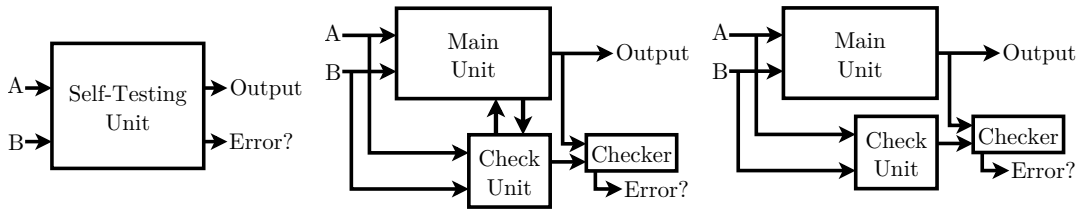
Strict duplication achieves complete single component error coverage, which can be considered a relatively strong level of protection. The SC error model, however, is generally not applicable to correlated faults that can simultaneously affect both the arithmetic unit and checker. In the case of strict duplication, such potentially-correlated faults include design faults and environmental timing violations, both of which can affect both the main and duplicate arithmetic unit.

There are well-known and well-established methods of ensuring that correlated design and environmental timing faults do not simultaneously affect the arithmetic unit and checker. Design faults can be correctly diagnosed by ensuring design diversity between the main arithmetic unit and checker; this approach has been recognized since the beginnings of mechanical computation [26, 31, 85, 86]. Timing violations can be correctly diagnosed by manipulating designs or supply voltages to keep all checking circuitry well off of the circuit's critical path [2].

Ensuring design diversity and keeping the duplicate checker off of the critical path has an added cost for duplication. Since the most efficient main arithmetic unit is normally employed, diversified duplication implies some inefficiency [86]. Also, keeping the duplicate checker off of the critical path requires some additional checking latency. In contrast, low-cost duplication makes use of redundancy and alternate number representations in the checker and is naturally diversified by construction. Thorough these alternate number systems, low-cost duplication can be more efficient than the main arithmetic unit despite this diversification. Also, by avoiding lengthy carry-propagation, low-cost duplication is able to operate faster than the main arithmetic unit and can be non-critical without additional checking latency.

2.3 BACKGROUND FOR GOAL #2: SEPARABLE CHECKING

Systematicity and separability are important classifying properties of error detecting mechanisms. A mechanism is *systematic* if its data bits and check bits are distinct and the data can be extracted from the codeword without first passing through a decoder. A code is *separable* if checking can proceed without any communication from the checked unit. By construction, all separable codes must be



(a) Non-Systematic (b) Inseparable (c) Separable

Figure 2.1: The separability classification of arithmetic error detecting codes and the design of different error checkers.

systematic. A simple high-level illustration of non-systematic, systematic (but inseparable) and separable checkers is shown in Figure 2.1.

It is highly desirable for an error detection scheme to be separable for several reasons. First and foremost, separable checkers make for modular designs, as shown by Figure 2.1c, and allow chip designers to implement arithmetic units and checkers independently. This modularity lessens the design burden of reliable execution. Also, separable designs can operate completely off of the critical path (during error-free execution) and allow pre-existing and highly optimized arithmetic units to be used without introducing new design constraints and unwanted timing effects. For these reasons, this research focuses exclusively on separable error detection mechanisms.

The full meaning of the term *separability* is not always clear from prior research, and there exists a gradient in the degree of separability provided by error detection mechanisms. To be clear and precise, this research uses a complete taxonomy of separable error detection checkers that is presented below.

2.3.1 *A Taxonomy of Separable Error Detectors*

The definition of separability is normally limited to the physical communication required by a checker, as described above. This leaves potential for separable designs to require intimate knowledge of the main arithmetic unit, or even to constrain the main unit design in order to satisfy an error model. Such design interaction greatly reduces the modularity advantages of separability, complicating the design of a reliable arithmetic pipeline. To eliminate this grey area in classification and to be more explicit about the modularity of competing designs, this research uses a more complete separability classification that is

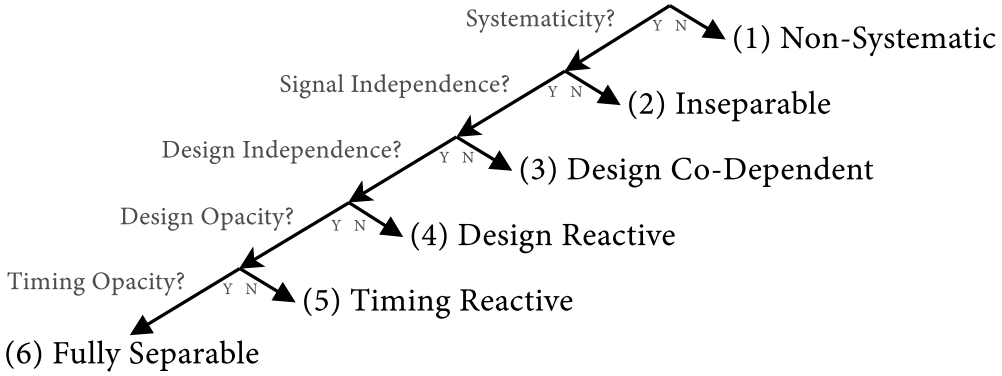


Figure 2.2: A taxonomy of separability. Designs are arranged in order of increasing separability going from the upper right to the bottom left.

shown in Figure 2.2. This classification separates physically separable designs into four classes that vary in their design modularity and independence. When combined with non-systematic and inseparable checkers, the full classification ends up with six classes; each of these classes is briefly described below.

2.3.1.1 NON-SYSTEMATIC CHECKERS

A non-systematic checker uses a single encoding for both the data and redundant check information such that the data bits are not preserved. Such checkers either demand the system-wide adoption of this specialized encoding or require that the data be encoded and decoded before entering the protected arithmetic pipeline. The most studied example of non-systematic checkers are the AN codes, a non-systematic representation of residue coding [71, 72].

2.3.1.2 INSEPARABLE CHECKERS

A systematic-but-inseparable checker separates the information and check bits, but uses both as inputs to a single unit to generate an arithmetic result and error indicator. An example of such a checker is optimized parity prediction, where the parity prediction circuitry and main arithmetic unit are carefully co-designed to allow sharing of circuitry between the two without compromising the single device error coverage of the error detection scheme [7, 8, 9].

2.3.1.3 DESIGN CO-DEPENDENT CHECKERS

If a separable checker relies on the main arithmetic unit to be designed or synthesized in a certain fashion in order to provide coverage guarantees, it is referred to as a design co-dependent separable checker. An example of a design co-dependent separable checker is parity prediction where the main arithmetic unit is synthesized in such a way that any SD error will result in a unidirectional (and therefore detectable) error [6].

2.3.1.4 DESIGN-REACTIVE CHECKERS

Design-reactive separable checkers are designed in response to a specific arithmetic unit. While such checkers do not dictate the arithmetic unit design itself, they require intimate knowledge of the arithmetic unit in order to provide coverage guarantees. Also, appropriate design-reactive separable checkers may be unknown for some main arithmetic unit designs. An example of a design-reactive separable checker is the auto-creation of residue checking circuitry to provide complete SD error coverage for simple multipliers [10].

2.3.1.5 TIMING-REACTIVE CHECKERS

Timing-reactive separable checkers treat the main arithmetic unit is a black box, and do not require intimate knowledge of the arithmetic unit design. Such checkers, however, reactively utilize detailed timing information from the main unit to overlap some computation with error detection and increase efficiency. This research develops some timing-reactive error detection organizations, and garners modest efficiency gains using detailed timing information from the main unit [87].

2.3.1.6 FULLY SEPARABLE CHECKERS

Finally, fully separable checkers require no knowledge of the main arithmetic unit apart from its output interface and the arithmetic operation it supports. No checking is overlapped with computation. In addition to timing-reactive designs, this research develops fully separable checkers. There is a tradeoff between the

timing-reactive and fully separable checkers, with full separability providing superior design modularity at some associated cost.

2.4 BACKGROUND FOR GOAL #3: LOW-LATENCY, CONCURRENT ERROR DETECTION

Low-latency, concurrent error detection simplifies the higher level replay and state restoration mechanisms that are necessary for arithmetic error correction. Long-latency error reporting can require higher level mechanisms to stall in order to guarantee that errors do not propagate outside of a correctable region [15]. Alternatively, higher level mechanisms may need to handle long-latency detected errors with more expensive recovery mechanisms, such as transactional rewind for data rematerialization or the use of a distant global checkpoint, impacting system efficiency [13, 88].

2.5 BACKGROUND FOR GOAL #4: LOW AREA AND ENERGY OVERHEADS

Arithmetic errors tend to be rare, unpredictable events, and due to the complexity of logic they can be difficult and costly to detect. Therefore, the high and fixed overheads of conventional strong error detection are generally wasted during fault-free operation. Meanwhile, general-purpose computers continue to be applied to a more diverse range of applications, including those with high reliability demands [89]. This motivates the development of low-cost error detection techniques such as those considered in this dissertation.

Lower error detection overhead often comes at the expense of separability, strength, or detection latency. Many existing mechanisms attempt to lower the overheads of error detection by violating separability, by restricting error coverage to a narrow set of fault or error models, or by asynchronously reporting detected errors with a long and unpredictable latency. In contrast, the goal of this research is to lower the area and energy costs of duplication without sacrificing greatly in these areas—this dissertation shows that careful design can reduce the overheads of duplication without losing its strength and implementation benefits.

2.6 THE METHODOLOGY OF THIS RESEARCH

This research focuses on timing-reactive and fully separable checkers that can provide complete detection of single-component errors. Such error detection mechanisms are able to satisfy all of the design goals listed in Section 1.1.

2.6.1 Hardware Synthesis and Analysis

Unless otherwise noted, gate-level design space exploration is used to examine the area and energy required for hardware components. The Synopsys toolchain is used for synthesis, targeting the 40nm TSMC standard cell library [90, 91]. All circuits are compiled using the Synopsys Design Compiler with high mapping effort and optimization options consistent with an area-optimized implementation. Structural Verilog descriptions of each circuit are used throughout, with compressor-based multipliers and minimum-depth parallel prefix adders. The Synopsys Design Compiler provides area (in μm^2) and power (in μW) estimates for each design. Dynamic power is estimated at the gate level using random test vectors.A TSMC wire model is used for timing estimates, and it is assumed that each circuit is driven by, and drives, a pipeline register. Wherever energy estimates are given, it is assumed that the protected arithmetic unit determines the clock frequency of its parent chip; energy consumption is derived accordingly using the timing and power estimates. Dual-rail encoded equality checkers are used at the output of error detection to create totally self-checking designs.

2.6.2 Encoding of Input Operands and End-to-End Protection

This research focuses on arithmetic error detection in a generative context, where operands arrive to the arithmetic unit in an unencoded format (or using a non-arithmetic, systematic ECC code). An alternative organization is to protect an operation in an end-to-end fashion, where local memory and data movement are protected by the same error code as the operation [76, 92, 93, 94]. An example of a cohesive end-to-end protection scheme is the STAR computer, an experimental processor that uses residue checking to protect memory, data movement, and arithmetic [76].

An end-to-end strategy is not considered in this study, as it pushes complexity to the memory and data movement subsystems (the costs of which

often surpass that of arithmetic in control-intensive architectures). Also, operations that do not preserve the error code suffer an increase in complexity under end-to-end arithmetic error detection. This diffusion of implementation costs makes the evaluation of end-to-end schemes difficult without fixed knowledge of the microarchitecture in which a protected arithmetic unit operates. It is assumed that the low-cost duplicate checkers from this research will be incorporated with other protection mechanisms for data storage and movement to form an end-to-end protection scheme (similar to most existing solutions [16, 17, 18, 19, 20, 21, 22, 23, 77]).

2.6.3 *Arithmetic Error Correction*

There has been a wealth of work that focuses on the correction of arithmetic errors [71, 95, 96, 97, 98, 99]. Such efforts either violate separability [98, 99], add costly hardware and correction latency [71, 96], or their correction capabilities are limited to simple, synthetic errors [95].

Furthermore, unlike memory errors (where an uncorrected error can result in data loss) arithmetic errors can typically be corrected by replaying the erroneous instruction.[4] Such replay is especially effective for low-latency error detectors, since microarchitectural replay mechanisms can handle an error before it is allowed to overwrite any of the input data [15]. For these reasons, this research focuses solely on arithmetic error detection, leaving the role of error correction to higher system levels.

4 In the case of a permanently faulty arithmetic unit, correction-through-replay may have to occur on a different, fault-free arithmetic unit or using a different set of instructions. Higher-level recovery mechanisms exist to deal with such situations [22].

3 LOW-COST DUPLICATION FOR FIXED-POINT ADDITION ‡

Adders are fundamentally important to computer systems and their protection against errors is correspondingly imperative in high availability and mission-critical systems. Addition is utilized for different purposes including data manipulation, memory addressing, and control flow—this means that an error in addition can manifest in many ways, ranging from silent data corruption to catastrophic system failure. The wide range of maladies that can result from an adder error potentially makes the strong protection of addition desirable across many problem domains, including scientific computing and system software.

This chapter describes a low-cost duplicate organization for adders based on the redundant carry-save number representation. Section 3.1 describes carry-save addition and presents a modified equality checker with a carry-save input. Section 3.2 describes and evaluates a fully-separable carry-save duplication scheme for fast adders and Section 3.3 extends it to form a timing-reactive separable checker for any speed of adder.[5]

3.1 THE CARRY-SAVE NUMBER SYSTEM AND A MODIFIED CHECKER

Addition using a non-redundant positional number system (such as two's complement) requires a lengthy carry propagation for addition; even the fastest possible adders require a logarithmic number of stages to perform this carry propagation [101, 102]. There are redundant number representations, however, where addition can be performed with superior efficiency in constant time. The following subsections describe constant-time addition using the *carry-save* number system[6] [101, 102] and present a modified equality checker that evaluates the equality of a positional number with a carry-save input.

‡ Parts of this chapter appear in [87, 100], with Earl E. Swartzlander, Jr. serving as supervisor.

5 Refer back to Section 2.3.1 for the distinction between a fully separable and a timing-reactive checker.

6 Carry-save numbers are sometimes also referred to as *stored-carry* numbers [101].

3.1.1 *Carry-Save Addition*

Binary carry save addition performs arithmetic with the redundant digit set $d \in \{0, 1, 2, 3\}$ [102]. Intuitively, carry-save addition saves the intermediary carry signals during addition instead of adding them to more significant bit positions. This results in the need for $2N$ bits to represent an N-bit dynamic range, but it allows addition to proceed quickly and efficiently in a carry-free fashion. [7]

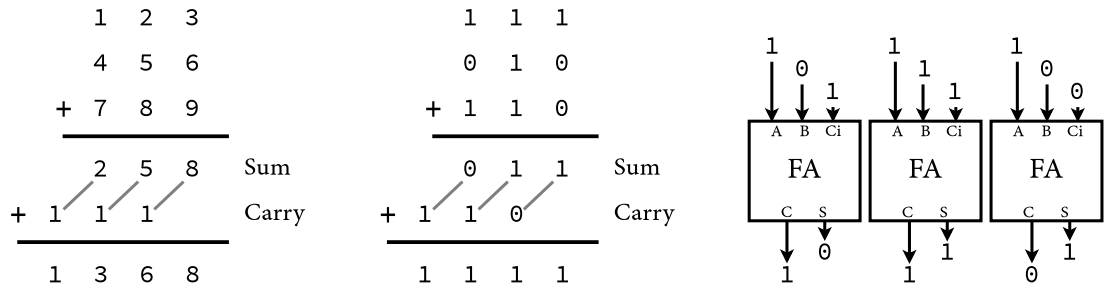
Figure 3.1 illustrates carry-save addition through an example. Figure 3.1a demonstrates the multi-operand addition of three decimal numbers. Each positional column of digits is independently added, with the result being a sum digit and one carry digit to be added to the next most significant position. Full non-redundant multi-operand addition performs this carry propagation and addition, adding the labeled intermediary result called the sum to the carry. Carry-save addition keeps the number in this intermediary representation as both a sum and a carry, meaning that conversion back to a non-redundant form entails the addition of the two terms and $2N$ bits are required to represent an N-bit result. It can be seen, however, that the carry-save addition of each positional digit proceeds independently, leading to constant-time addition that does not depend on the input operand width.

Figure 3.1b shows another illustrative example of carry-save arithmetic using a binary representation, and Figure 3.1c gives its hardware implementation. It can be seen that N-bit three-operand binary addition can be performed using N full adders following a (constant) single full-adder delay.

3.1.2 *The Carry-Save Equality Checker*

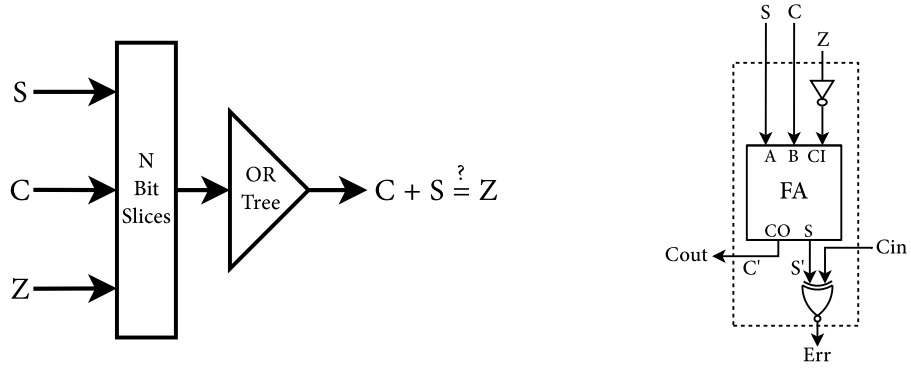
Given a carry-save input represented by a sum and carry term, $\{S, C\}$ and a weighted positional fixed-point input, Z, the equality of $S+C \stackrel{?}{=} Z$ can be evaluated by detecting whether $S + C - Z \stackrel{?}{=} 0$ through cancellation. This cancellation can be detected without carry-propagation by reducing the three-input addition $S + C +$ $(-Z)$ down to a carry-save intermediary term $\{S', C'\}$ and detecting whether S' and C' complement each other. Such a modified equality checker with one carry-save and one weighted input is shown in Figure 3.2. While this work (published

7 Carry-save addition is carry-free *between* digits. A limited amount of carry propagation is needed to form each digit.



(a) Decimal Example (b) Binary Example (c) Binary CS Addition

Figure 3.1: An example depicting decimal and binary three-operand addition. Carry-save arithmetic uses the intermediary sum and carry results directly and can perform binary arithmetic within a constant full-adder delay.



(a) The carry-save equality checker (b) A checker bit slice

Figure 3.2: A depiction of the carry-save equality checker. Cancellation is detected between the carry-save and positional inputs through a bit-sliced, constant-delay structure. Any non-complementary bits are detected through an OR reduction tree.

in [100]) is the first to use this modified equality checker for arithmetic error detection, a similar equality comparison has been employed elsewhere [102, 103, 104] for carry-save arithmetic.

3.1.3 *Carry-Save Equality for One's/Two's Complement Numbers*

The carry-save equality checker is easily adapted to work for either one's or two's complement numbers. One's complement arithmetic involves an end-around-carry signal in the main adder. Correspondingly, the one's complement equality checker carries around the carry of the final bit-slice, as shown by the

(a) One's Complement (b) Two's Complement

Figure 3.3: The Dadda dot diagram [105] of the bits and carry dependencies in the one's and two's complement checkers. Dots represent bits of data, and directional arrows indicate a carry-dependence between bits. Therefore, the top and bottom row of dots respectively represent the sum and carry bits output from the full adders in the carry-save equality checker. An 8-bit checker is shown; no carry-dependence exists between digits due to the carry-save representation. The carry-out of the final bit-slice differs between the one's complement and two's complement equality checker; it is accordingly highlighted in grey.

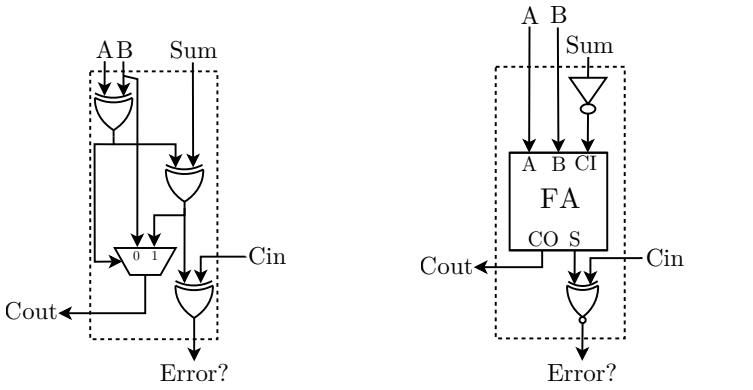
Dadda dot diagram [105] in Figure 3.3a. For two's complement arithmetic, the final check slice carry-out is checked against the complemented carry-out of the main adder and the main adder carry-in is propagated into the first checking stage, as depicted in Figure 3.3b. While two's complement checking requires an additional stage, the first and last check-slice can be significantly simplified such that the complexity and power efficiency of the checker are only marginally affected.

3.2 CARRY-SAVE DUPLICATION

Carry-save duplication makes use of the carry-save equality checker[8] to verify addition in constant time with low hardware overheads.[9] Whereas full duplication checks for the equality of $Z = Z'$ (assuming Z is the result following the addition of input operands A and B and Z' is the result of the duplicate adder), carry-save duplication tests for $A + B = Z$ using the carry-save equality checker. The input operands A and B assume the role of the sum and carry terms of the carry-save input to the checker, and no redundant representation is propagated outside of the modified equality check.

8 Carry-save duplication uses a two-rail encoded equality output (unlike the depiction from Figure 3.2a) in order to be a self-checking design.

9 Constituent work also calls the carry-save duplicate checker the *long residue checker* [100] for reasons later explained in Section 3.2.1.2.



(a) Lazy Adder Checker (b) Carry-Save Duplicate Checker

Figure 3.4: A 1-bit slice of the lazy adder checker [106] and of the carry-save duplicate checker.

3.2.1 *Relationship to Existing Work*

Apart from its obvious relation to full duplication, carry-save duplication has close ties to alternate adder error detection schemes including lazy error detection and residue checking. The nature of the relationship between carry-save duplication and this existing work is investigated below.

3.2.1.1 LAZY ERROR DETECTION

Figure 3.4a shows one bit-slice of the lazy error checker[10] proposed by Yilmaz et al. [106]. Each slice checks one output bit using a modified full adder and an XOR gate; the error signal from each bit is ORed together to determine if an error has occurred. The checker has no carry dependence, such that the checker delay (apart from the OR tree) does not directly depend on the word width. Careful inspection shows that the functionality of the carry-save and lazy checkers are essentially equivalent;[11] each is a bit-sliced design, and the two have similar error coverage. Figure 3.4(b) replicates a slice of the carry-save duplicate checker for a side-to-side comparison.

10 Despite the name similarity, the lazy error checker is unrelated to lazy duplication (described previously in Section 1.2.1).

11 The carry-save and lazy checker bit-slices have a different carry interface and therefore cannot be mixed.

While the carry-save duplicate checker is functionally similar to the lazy checker, the carry-save scheme has the potential to significantly reduce error detection overheads using standard cell synthesis. An important advantage of carry-save duplication is that it uses a full adder as a fundamental unit. There exist efficient full-adder cells [107] in many standard cell libraries, increasing the efficiency of carry-save duplication without resorting to custom cell design.

To demonstrate the efficiency advantages of the carry-save duplicate organization, this research makes use of the mirror adder cell found in the Nangate 45nm standard cell library [108].[12] Isolated analysis[13] of the checkers in Figure 3.4 shows that a carry-save duplicate bit-slice consumes 10.73% less area and 19.66% less power than its lazy-checker counterpart. The following section evaluates all separable adder protection approaches, and finds that the full carry-save duplicate scheme has efficiency advantages equal to or greater than these initial estimates.

3.2.1.2 RESIDUE CHECKING

Residue checking (described briefly in Section 1.2.2) can be applied to addition. In general, the strength of a residue check is proportional to its checking modulus and its latency is inversely proportional to the modulus, such that the strongest and lowest-latency organization is the residue check whose modulus is as large as the input operand width. Such a strong and fast residue checker is equivalent to full duplication.

The above observation relating residue checking to full duplication provides an insight into the relationship between carry-save duplication and residue checking. Carry-save duplication is equivalent to a limited-carry variant of residue checking, described below, in the case of a maximal checking modulus.

Carry-Save Residue Checking:

An improved residue checker is presented that uses residue generation circuitry with carry-save outputs along with a carry-save equality checker. The

12 Unlike the overall experimental methodology described in Section 2.6, area and power estimates in this section (Section 3.2) utilize the 45nm Nangate Open Cell Library [108]. This is done to be consistent with published work [100]. Section 3.3 and the rest of this dissertation uses the TSMC 40nm library and is consistent with the previously described methodology from Section 2.6.

13 This isolated analysis characterizes and analyzes a single checker bit-slice in the context of its application as an error detection for addition.

checker relies on a restricted class of residues in the form $A = [2^a - 1; a \in \mathbb{N}]$, for which $|X|_A$ (X being an N-bit fixed-point number) can be generated with an $^N/_a$-input, a-bit *multi-operand modular adder* (MOMA) followed by an a-bit end-around-carry adder [109]. A traditional residue checker performs this full residue generation on each input, adds the two resulting residues using an end-around-carry adder, and compares the result for equality with a residue generated at the output of the main adder [12, 92]. The modified residue checker eliminates the end-around-carry adder from the output of each residue generator, instead keeping the output from each *carry-save multi-operand-modular-adder* (CS-MOMA) in its redundant format. This modification eliminates four carry-propagations from the checking procedure—two for the input residue generators, one for the modular check adder, and one for the output residue generator—while replacing the modular check adder with a 4-input CS-MOMA and replacing the equality checker with a carry-save equality checker.[14]

Due to the fact that the carry-save residue checking algorithm has no carry dependence, its efficiency does not strongly depend on the checking modulus. This makes the modified residue checker more efficient than traditional residue checking, especially for larger (and stronger) checking moduli. Further simplifications are possible when the checking modulus is equal to the input size, in which case the scheme becomes completely equivalent to carry-save duplication. Later, in Section 3.2.2, it is shown that the efficiency gained by eliminating residue generation outweighs any added checking costs, making carry-save duplication the most efficient modified residue checking organization and eliminating traditional residue checking as a viable contender.

3.2.2 *Carry-Save Duplication Evaluation*

The overheads of carry-save duplication and other competing error detectors are evaluated relative to an efficient two's complement adder design. This efficient baseline is chosen through a search of the design space to minimize the ED^2 metric [110]. Figure 3.6 shows a simplified Pareto frontier of the possible 16-bit adder designs, highlighting the reference adder used in this dissertation.

14 This carry-save equality checker is similar to the one described in Section 3.1.2 except that it accepts *two* carry-save numbers as inputs. This replaces the full-adder in the bit-sliced checker with a (4:2) compressor, slightly increasing hardware costs.

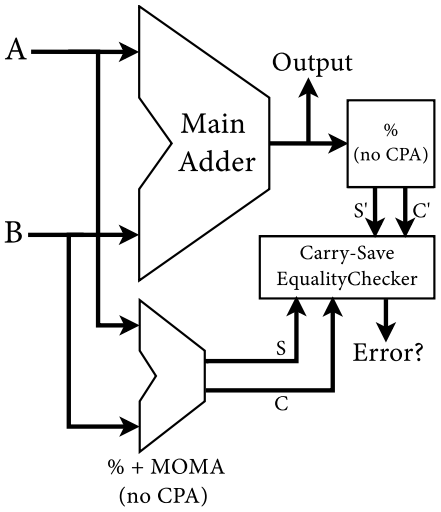


Figure 3.5: A block diagram of the carry-save residue checker. All carry propagation during residue generation and modular addition is avoided by using carry-save modular multi-operand adders (CS-MOMAs); a carry-save equality checker is used at the output to verify the congruence of the output with the result of modular arithmetic. The % component represents carry-save residue generation without a final carry-propagate modular adder.

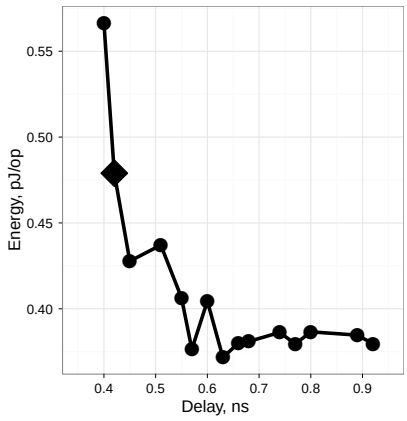


Figure 3.6: Pareto-efficient 16-bit adder designs. The highlighted adder minimizes the ED^2 metric and is used as a baseline. A similar selection procedure is used to choose the 32, 64, and 128-bit baseline adders.

Table 3.1 gives the area, power, and delay of each selected design. Some results are normalized relative to the baseline adder or the carry-save duplicate checker; absolute area and power consumption estimates can be derived accordingly.

Figure 3.7 shows the area and power consumption of traditional and modified residue checking for a 32-bit adder across different residue widths. The

Table 3.1: Baseline adder properties.

Adder Width	Delay (ns)	Area (μm^2)	Energy (pJ)
16	0.42	381.35	0.479
32	0.55	848.79	1.108
64	0.67	1546.06	1.898
128	0.7	3247.85	4.028

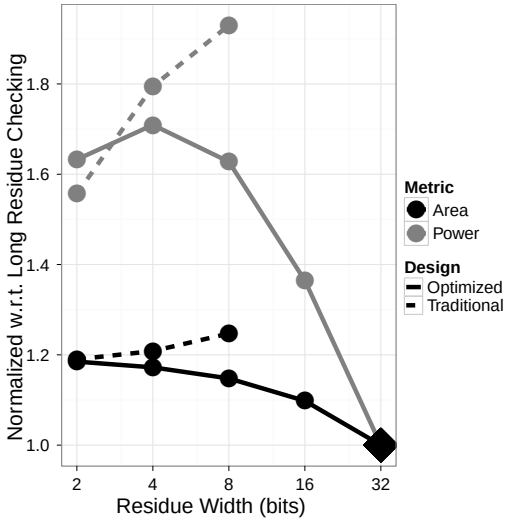


Figure 3.7: The area and power of traditional and modified residue checking, normalized with respect to carry-save duplication (denoted "long residue checking"). The carry-save duplicate checker is marked by a rhombus.

carry-save duplicate organization (denoted "long residue checking") is marked by a rhombus. The CS-MOMA for modified residue checking is implemented using a tree of carry-save adders with an end-around-carry at each level. Any such CS-MOMA requires a constant number of full adders regardless of its modulus width; wiring complexity increases drastically at low residue widths, however, making the carry-save duplicate configuration ($a = 32$) more area and power efficient than any alternative. Increasing CS-MOMA depths conspire to make residue checking at short residue widths ($a \ll 32$) a relatively long-latency operation. As such, all designs in Figure 3.7 use a fixed detection latency of three cycles, despite the fact that carry-save duplication easily completes earlier.

From Figure 3.7 it is apparent that that: (1) the limited-carry version is more efficient than unmodified residue checking across all checking moduli; (2) carry-save duplication is the design point with the lowest cost for this modified

Table 3.2: The overheads of carry-save duplication and lazy error detection.

Carry-Save Duplication		
Width	Area [μm^2] (+%)	Energy [pJ] (+%)
16	144.9 (+38)	0.33 (+69)
32	280.1 (+33)	0.78 (+70)
64	556.6 (+36)	1.59 (+84)
128	1104.3 (+34)	3.46 (+86)
Lazy Checker		
Width	Area [μm^2] (+%)	Energy [pJ] (+%)
16	197.1 (+48.3)	0.49 (+102.1)
32	305.3 (+64.0)	0.96 (+86.8)
64	612.2 (+60.4)	1.96 (+103.3)
128	1214.7 (+62.6)	4.33 (+107.5)

checker, despite the fact that it also has the highest coverage and lowest checking latency. These two observations demonstrate the clear superiority of carry-free duplication over residue checking for generative error detection of fixed-point addition. These results are robust across all tested input widths, though only the 32-bit results are shown.

Table 3.2 gives the area and energy needed for carry-save duplication and the lazy error checker relative to the baseline adders. The relative energy overheads of carry-save duplication are higher than its area costs; this trend is consistent with prior work [106]. Due to the continual utilization of the simulated adder, dynamic power dominates.

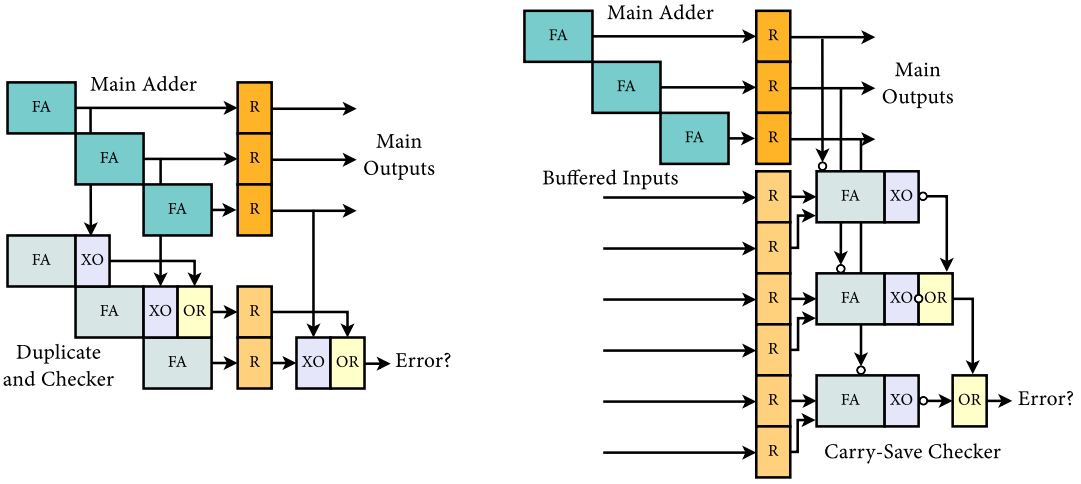
Carry-save duplication has significant efficiency gains relative to the lazy error detector while requiring less checking latency. The lazy checker consumes about 10% more area and 25% more energy than does carry-save duplication. The energy advantages of the carry-save duplicate organization slightly outweigh the savings of a single bit-slice (see Section 3.2.1.1) due to reduced switching activity at its error checking tree. The longer latency of lazy checking causes a significant loss in its relative efficiency at 16-bits—in order to satisfy timing, the lazy checker must increase the strength and power consumption of its cells.

3.3 CARRY-FREE/CARRY-PROPAGATE DUPLICATION: A TIMING-REACTIVE VARIANT OF CARRY-SAVE DUPLICATION

A wide and diverse assortment of fixed-point adder designs exist that vary in their speed, area, and power dissipation [111]. Carry-save duplication provides strong, fully separable, low-cost error detection for fast, minimum-depth fixed-point addition. However, carry-save duplication defers all checking until after the results of the main addition, and there are main adder organizations—such as a slow, pipelined adder—where the pipelining costs of this deferred evaluation may outweigh the costs of strict duplication. This section presents a flexible family of separable error detection techniques called *carry-propagate/carry-free (CP/CF) duplication* that provide superior error detection efficiency across a wider variety of adders. Carry-propagate/carry-free duplication reactively utilizes detailed timing information from the main adder to check result bits as they are produced, and therefore it is considered a timing-reactive checker.

As a motivating example, it is noted that the careful application of duplication may be less costly than carry-save duplication when protecting a pipelined ripple-carry adder. This is because both ripple-carry duplication and carry-save duplication use the same number of logic cells, but ripple-carry duplication can overlap some checking with computation as the least-significant result bits are produced first. This overlapped execution can reduce the number of pipeline registers that are required for error detection precipitously. Figure 3.8 illustrates these savings through the example of a 3-bit ripple-carry main adder (two-rail checking logic is not shown).

Based on the above observations, this chapter investigates a parameterized family of error detectors that combines the advantages of the carry-save duplication with those of strict duplication. The family is characterized by three parameters: N (the input width), lw (the *logic width* of strict duplication), and cw (the *checking width* of strict duplication). Functionally, the lw least-significant bits of addition are computed using partial duplication, with cw of these bits being fully checked before the main addition completes. In the pipeline stage following the main addition, the remaining ($lw-cw$) *duplicated-but-unchecked* result bits are equality checked, and the ($N-cw$) *unduplicated* bits are checked via carry-save duplication. This scheme is referred to as carry-propagate/carry-free (CP/CF) duplication owing to the fact that a carry-propagate adder is used



(a) Ripple-Carry Duplication

(b) Carry-Save Duplication

Figure 3.8: Ripple-carry duplication and carry-save duplication for a 3-bit ripple-carry adder. Cells labeled FA denote full adders, R denote registers, XO denote XOR gates, and OR denote OR gates. While both ripple-carry duplication and carry-save duplication use the same number of logic cells, ripple-carry duplication uses fewer pipeline registers due to some overlapped execution. For simplicity, two-rail checking logic is not shown.

to check the least significant bits of the result while the carry-free carry-save equality checker is used to check the most-significant bits.

Figure 3.9 shows a block diagram of CP/CF duplication. It should be noted that the parameters of CP/CF duplication must be chosen in a reactive manner in order to stay off of the critical path and maintain separability. Because of this timing-reactive separability, carry-propagate/carry-free duplication degenerates to carry-save duplication for protecting a fast adder with perfectly balanced outputs but can operate as ripple-carry duplication for protecting a very slow adder. Figure 3.8 can be used to demonstrate this—Figure 3.8a can be thought of as CP/CF duplication with (lw=3, cw=2) and Figure 3.8b as a design with (lw=0, cw=0). Table 3.3 qualitatively summarizes the properties of carry-propagate/carry-free duplication relative to other separable error detectors. Strict and lazy duplication require few registers to pass duplicated data, but their logic overheads can be costly when protecting a fast adder. Carry-save duplication greatly reduces this logic overhead, especially when protecting fast adders, but it requires a constant number of extra registers to buffer the inputs. Carry-propagate/carry-free duplication combines the best features of lazy and

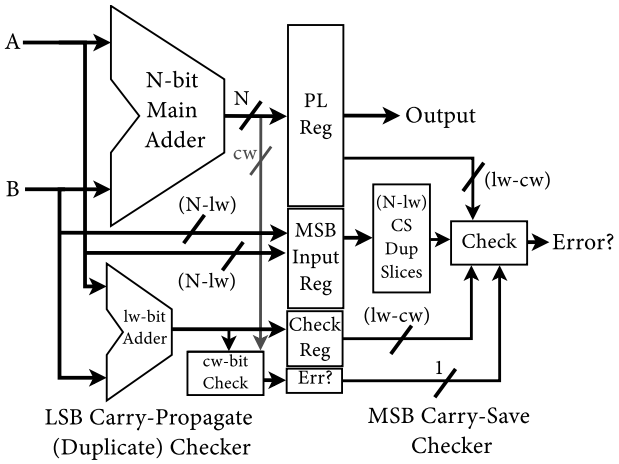


Figure 3.9: A block diagram of carry-propagate/carry-free duplication. A partial adder duplicates the *lw* least-significant bits of addition, *cw* of which are checked in the first pipeline stage (in parallel with the main addition). In the following pipeline stage, the (*lw*–*cw*) duplicated-but-unchecked bits are equality checked, and the (*N*–*cw*) unduplicated bits are checked via carry-save duplication. The carry-out of the strict duplicate adder (which is passed through a register to the carry-in of the carry-save duplicate checker) is not shown.

Table 3.3: A summary of the separable adder error detectors with complete single-component error coverage.

Scheme	FAs, XOs, ORs	Registers	Checking Latency
Strict or Lazy Duplication	$\{>N, N, N\}$	Few	Short
Carry-Save Duplication	N	Many	Short
CP/CF Duplication	N	Few	Short

carry-save duplication. Designers can tailor the behavior of CP/CF duplication to match the delay of an adder, achieving low-cost separable error detection regardless of the speed and design of the protected circuit.

3.3.1 Carry-Free/Carry-Propagate Evaluation

The efficiency and flexibility of CP/CF duplication is demonstrated by protecting a range of adders with different speeds. Following the mixed serial/parallel prefix methodology used for flexible parallel prefix addition [111], five different 32-bit

Table 3.4: Baseline adders with varying delay budgets.

Design	Delay (ns)	Area (μm^2)	Energy (pJ/op)
BK-32	0.29	1008	0.64
{BK-24, RC-8}	0.45	842	0.59
{BK-16, RC-16}	0.75	792	0.58
{BK-8, RC-24}	1.01	737	0.56
RC-32	1.21	719	0.56

Table 3.5: The area and energy allocation of the baseline adders.

Design	Area (%)		Energy (%)	
	Registers	Logic	Registers	Logic
BK-32	39	61	65	35
{BK-24, RC-8}	47	53	71	29
{BK-16, RC-16}	50	50	73	27
{BK-8, RC-24}	54	46	75	25
RC-32	55	45	75	25

baseline adders are considered. The adders use slow, ripple-carry (RC) propagation for the least-significant bits of addition and Brent-Kung (BK) parallel prefix addition [112] for all remaining bits. Table 3.4 gives the delay of each selected baseline, along with its area and energy demands. It can be seen that adder costs go down with decreasing speeds, such that an error detection mechanism must change proportionally in order to be competitive across all time scales. Table 3.5 gives the relative cost of pipeline registers and logic for each baseline design. It is apparent that the relative cost of pipeline registers is inversely proportional to the speed of addition, which indicates that a pipelining-cognizant approach is needed at slow speeds.

Table 3.6 shows the area and energy required for strict DMR (with the checker in the pipeline stage following addition), carry-save duplication (with the checker in the following pipeline stage[15]), and carry-propagate/carry-free duplication (as shown in Figure 3.9). The area and energy cost of registers, logic, and the total cost (including both registers and logic) are shown. The design with the lowest total cost is highlighted in bold. In addition to the absolute hardware

15 Prior work describes carry-save duplication for an unpipelined adder [100]. As such, this pipelining scheme is based upon that used by the lazy adder checker [106].

Table 3.6: The absolute cost of separable error detection for adders of varying speeds. Strict, carry-save, and CP/CF duplication are shown.

Duplication						
Design	Area (μm^2)			Energy (pJ/op)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	134	749	883	0.174	0.318	0.492
{BK-24, RC-8}	134	584	718	0.176	0.264	0.440
{BK-16, RC-16}	134	534	668	0.177	0.249	0.426
{BK-8, RC-24}	134	479	613	0.178	0.233	0.410
RC-32	134	461	594	0.180	0.231	0.411

Carry-Save Duplication						
Design	Area (μm^2)			Energy (pJ/op)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	264	254	518	0.263	0.207	0.471
{BK-24, RC-8}	264	254	518	0.262	0.208	0.470
{BK-16, RC-16}	264	254	518	0.262	0.208	0.470
{BK-8, RC-24}	264	254	518	0.263	0.208	0.471
RC-32	264	254	518	0.262	0.208	0.470

Carry-Propagate/Carry-Free Duplication						
Design	Area (μm^2)			Energy (pJ/op)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	247	260	**508**	0.246	0.160	**0.407**
{BK-24, RC-8}	213	269	**482**	0.206	0.150	**0.356**
{BK-16, RC-16}	138	302	**440**	0.136	0.130	**0.266**
{BK-8, RC-24}	113	281	**394**	0.110	0.116	**0.226**
RC-32	49	323	**372**	0.041	0.097	**0.138**

costs of separable error detection for addition, Table 3.7 shows the area and energy overheads required relative to the cost of each baseline adder. These data represent the percent area and energy costs required to protect each baseline design from error.

Strict duplication uses a constant number of registers, such that its register costs remain invariant across designs. Meanwhile, the absolute logic costs of full duplication scale with the speed (and cost) of the checked adder. As expected, duplication requires >100% overhead for logic, because a duplicate main adder and a two-rail checker are used to detect an error. The two duplicate adders share the input registers, and duplication naturally compresses the amount of state that

Table 3.7: The overheads of separable error detection relative to adders of varying speeds. Strict, carry-save, and CP/CF duplication are shown.

Duplication						
Design	Area (%)			Energy (%)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	13.3	74.4	87.7	27.1	49.6	76.7
{BK-24, RC-8}	15.9	69.3	85.2	29.7	44.6	74.3
{BK-16, RC-16}	16.9	67.4	84.3	30.8	43.2	74.0
{BK-8, RC-24}	18.2	65.0	83.1	31.7	41.5	73.2
RC-32	18.6	64.1	82.7	32.0	40.9	73.0

Carry-Save Duplication						
Design	Area (%)			Energy (%)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	26.2	25.2	51.4	41.0	32.4	73.4
{BK-24, RC-8}	31.3	30.2	61.5	44.4	35.2	79.5
{BK-16, RC-16}	33.3	32.1	65.4	45.5	36.0	81.5
{BK-8, RC-24}	35.8	34.5	70.2	46.9	37.1	84.1
RC-32	36.7	35.3	72.0	46.5	36.9	83.4

Carry-Propagate/Carry-Free Duplication						
Design	Area (%)			Energy (%)		
	Reg.	Log.	Tot.	Reg.	Log.	Tot.
BK-32	24.6	25.8	**50.4**	38.4	25.0	**63.4**
{BK-24, RC-8}	25.4	31.9	**57.3**	34.9	25.4	**60.2**
{BK-16, RC-16}	17.4	38.1	**55.5**	23.6	22.5	**46.1**
{BK-8, RC-24}	15.4	38.1	**53.5**	19.6	20.7	**40.2**
RC-32	6.82	44.9	**51.7**	7.31	17.2	**24.5**

needs to be propagated to the checker. These two factors conspire to give duplication modest register costs, resulting in~82–88% area and~73–77% total energy overheads. Duplication is slightly more efficient for slower adders, because the relative cost of data movement dominates at slow speeds (see Table 3.5).

Because of its structure and efficient implementation, carry-save duplication uses significantly less logic area than duplication across all adder designs. However, due to the need to pass both inputs (and the carry-in bit) to the checker, the register overhead of carry-save duplication is roughly double that of full duplication. The low logic and substantial register costs of the carry-save organization add up to a~51–72% total area overhead. The relative cost of pipeline registers in-

Table 3.8: The CP/CF parameters used in Table 3.6 and Table 3.7.

Design	CP/CF Parameters	
	Logic Width (lw)	Check Width (cw)
BK-32	4	2
{BK-24, RC-8}	8	7
{BK-16, RC-16}	18	16
{BK-8, RC-24}	21	19
RC-32	29	28

creases with the delay budget, making carry-save duplication less energy efficient than duplication at protecting slow adders. Note that carry-save duplication requires <100% area overheads to protect ripple-carry addition, despite the fact that an carry-save equality checker bit-slice contains a full-adder cell. This is because a DesignWare ripple-carry adder [113] was used as the baseline, which (at the delay budget used) alternates full-adder cells with more expensive logic in order to increase speed. Carry-save duplication requires a constant number of registers and check slices, such that the absolute cost of the carry-save checker never changes. Due to the scaling behavior of the baseline adders, carry-save duplication is relatively more efficient at protecting faster designs.

Carry-propagate/carry-free duplication overlaps some checking with execution, significantly decreasing the amount of pipelined state (and therefore register costs) relative to carry-save duplication. Meanwhile, the second-stage carry-free checker decreases the logic costs relative to duplication. When the costs of both logic and data movement are taken into account, CP/CF duplication achieves superior area and energy efficiency across all designs and time budgets. Carry-propagate/carry-free duplication offers a~2–28% decrease in total checking area and a~14–66% reduction in total checking energy consumption relative to the next most efficient separable mechanism. Table 3.8 shows the logic depth and checking depth parameters chosen for the carry-propagate/carry-free checkers used in Table 3.6 and Table 3.7. These parameters were determined through an automated computer search; due to the simplicity and parameterization of the error detection scheme, the time required for this search is not prohibitive.

3.4 DISCUSSION AND FUTURE WORK

Some discussion of experimental caveats and future research opportunities related to carry-save and CP/CF duplication follow.

3.4.1 Carry-Save Duplication Discussion

The majority of efficiency gains from carry-save duplication come from its ability to leverage an efficient carry-free checker while using efficient standard library cells. Custom cell design may be used to improve the efficiency of the lazy checker, likely making it roughly as efficient as carry-save duplication. One of the main strengths of carry-save duplication is that it is able to avoid the complexity and cost of custom cell design while providing superior implementation efficiency.

Careful inspection shows that the relative overheads presented for carry-save duplication (Table 3.2) are higher than those claimed in prior work for lazy checking [106], despite the fact that this dissertation finds carry-save duplication to be more efficient. It is likely that this difference is mainly due to differences in the baseline adders. However, some methodological decisions could also have an impact. All error detectors in this dissertation employ a dual-rail encoded checker (unlike the prior work). This is consistent with totally self-testing circuit implementations and is intended to correctly diagnose permanent checker errors. Experiments indicate that employing a single-rail checker (or a checker with a testable input) would reduce the area and power of carry-save duplication by ~8% and ~13–15%, respectively.

3.4.2 CP/CF Duplication Discussion and Future Work

Carry-propagate/carry-free duplication relies on the least-significant sum bits to be produced first in order to overlap some duplication and checking with the main addition. This early production of the lower result bits is consistent with 2's complement addition. However, there are some designs (such as end-around-carry adders used for 1's complement addition [114]) that produce perfectly balanced outputs. For these designs, CP/CF duplication will degenerate to carry-save duplication.

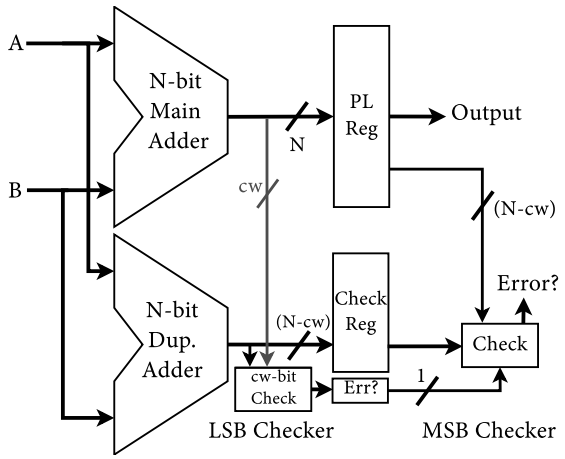


Figure 3.10: An alternate error detection scheme that employs a split two-rail checker along with duplication.

For simplicity, this dissertation assumes that the CP/CF carry-propagate (duplicate) adder is implemented using a ripple-carry adder. While preliminary results indicate that this duplicate adder is sufficient for many designs, using a more general form of carry-propagate duplication (such as one based on flexible parallel-prefix addition [111]) could give modest efficiency increases, especially for main adders with arbitrarily unbalanced outputs. Analysis of CP/CF duplication with a more flexible and aggressive duplicate adder is left for future work.

Some of the benefits of CP/CF duplication are due to its split two-rail checker, which opportunistically checks the least-significant bits of the duplicate result in parallel with the main addition. This split checking can sometimes decrease the amount of registered state that needs to be propagated to the next pipeline stage. In addition to its use for CP/CF duplication, such a split checker scheme could also be employed with full duplication, as shown in Figure 3.10. Such a design offers superior register costs relative to strict duplication, but does not reduce the significant costs of fully duplicated logic (as does CP/CF duplication). Initial experiments indicate that duplication with split checking operates as expected, providing a modest improvement over full duplication but failing to achieve the superior efficiency of CP/CF duplication. A full analysis of duplication with a split checker is left for future work.

4 LOW-COST DUPLICATION FOR FIXED-POINT MULTIPLICATION §

This chapter investigates the costs and benefits of low-cost duplication for the strong, holistic error detection of fixed-point multiplication. Two low-cost duplicate schemes are described and evaluated; it is shown that specialized carry-save or residue number system checking can be used to increase the efficiency of duplicated multiplication.

This chapter is structured as follows. Section 4.1 presents the baseline multipliers and the baseline duplicate designs considered in this work and Section 4.2 investigates the design and overheads of using specialized carry-save checkers in low-cost duplicate multipliers. It is shown that carry-save duplication lowers the costs of duplication in a straightforward manner without sacrificing error coverage, checking latency, or separability. Section 4.3 investigates an alternate low-cost duplicate multiplier that uses the residue number system to good effect.

4.1 LOW-COST DUPLICATE MULTIPLICATION METHODOLOGY

Throughout this chapter, the description and evaluation of each low-cost duplicate multiplier is interleaved. For this reason, the chosen baseline multipliers and prior approaches are described up front. Apart from these chosen baselines and prior approaches, the prevailing experimental methodology described in Section 2.6 applies; it is not reproduced here in the interest of brevity.

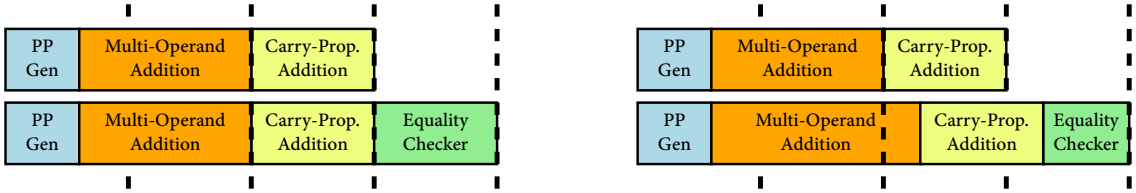
4.1.1 The Baseline Multipliers

Three efficient unsigned binary multipliers are selected to serve as the main arithmetic unit baselines at 16, 32, and 64 bits. The Pareto-optimal (over area and time) post-synthesis design which minimizes the AT metric is chosen at each word length through a search of the design space. Table 4.1 gives the properties of the selected baselines.

§ Parts of this chapter appear in [1], with Earl E. Swartzlander, Jr. serving as supervisor.

Table 4.1: The baseline multipliers selected for this work.

Width (N)	Critical Latency [ns]	AT-Efficient Latency [ns]	AT-Efficient Area [μm^2]
16	1.21	1.25	3547.8
32	1.70	1.73	12937.9
64	2.30	2.34	38179.8



(a) Strict Duplication Visualization

(b) Lazy Duplication Visualization

Figure 4.1: The stages of strict and lazy duplication. "PP gen" denotes partial product generation. Strict duplication uses a mirrored multiplier for checking, whereas lazy duplication utilizes any extra detection latency to reduce the complexity of the duplicate multiplier. Simple timing diagrams are shown for a multiplier; the relative time spent in each step of computation is not accurate and is for visualization purposes.

4.1.2 *Strict and Lazy Duplication*

Two baseline DMR organizations are considered: strict duplication, which uses a mirrored multiplier for checking, and lazy duplication, which uses any extra detection latency to reduce the cost of the duplicate multiplier. Figure 4.1 shows both strict and lazy duplication. A typical parallel fixed-point multiplier goes through three steps of computation: partial product generation, the multi-operand addition of the partial products, and the carry-propagate addition of the redundant carry-save product. The steps of a multiplier are shown over time along with the steps of a duplicated checker. Strict duplication proceeds in lockstep with the main multiplier; lazy duplication utilizes the extra checking latency for modest cost savings.

Strict duplication requires a 30–20% detection latency and 102.8%, 102.3%, and 101.2% area overheads to protect the 16, 32, and 64-bit multipliers, respectively. This reduction in relative overheads with increasing word length is due to the quadratic scaling behavior of parallel multipliers—the dual-rail equal-

ity checker scales linearly, such that its contribution to the total area becomes smaller.

The >100% implementation overheads of strict duplication are consistent with many prior assumptions about DMR organizations, but the overheads of duplication can be easily decreased by allowing extra detection latency. Later, Table 4.2 gives the estimated overheads for lazy duplication. Lazy duplication considerably lowers the overheads of strong error detection by utilizing any available slack, but it may still require a prohibitive amount of overhead for many applications. Conversely, lazy duplication may demand too much detection latency to garner sufficient efficiency; later evaluation shows that low-cost duplication mitigates this deficiency.

4.1.3 Residue Checking

The cost and error coverage of residue checking depends on the width of its checking modulus. The area of a fast multiplier (both modular and otherwise) scales quadratically with its input width such that a small checking modulus (e.g. 3 or 15) provides error detection with low overheads relative to a large 32+-bit multiplier. Because of its ability to efficiently protect large multipliers, residue checking with small moduli has been widely employed in reliable systems—a modulus of 3 and 15 are the most popular choices. [21, 22, 23, 31].

In general, large checking moduli are prohibitively expensive, such that residue checking suffers from some coverage holes, even for relatively weak single-device errors [12, 10, 30]. For example, a simulation-based error injection study finds that on average, 7% of unmasked single-device errors in a fast 32-bit multiplier remain undetected by a modulus-3 code [12]. Low-cost duplicate multiplication offers complete single-component error coverage, and as such has much higher error detection coverage than residue checking—this difference is expected to be especially significant for severe errors. Also, it is likely that low-cost duplication will have a lower latency than residue checking,[16] at the expense of more checking hardware.

16 The higher latency of residue checking relative to low-cost duplication comes from the need to generate the residue of the result before checking. The latency of residue generation is inversely proportional to the modulus width, such that the checking latency is greatest for the smallest residue codes.

It is possible to use multiple co-prime moduli in a multi-residue code for higher error coverage [115, 116]. Low-cost residue number system duplication (Section 4.3) can be thought of as an extreme organization of multi-residue coding where multiple large moduli are used and the product of the chosen moduli is larger than the dynamic range of the 2N-bit result of multiplication—such a multi-residue code is strong enough to have complete single-component error coverage. In this context, it is obvious that single or multi-residue checking can be less expensive than residue number system duplication, but with incomplete error coverage against severe errors.

4.2 CARRY-SAVE LOW-COST DUPLICATION

A simple and straightforward scheme for low-cost duplicate multiplication is to eliminate the final carry-propagate adder in the duplicate multiplier, checking its result directly in the redundant carry-save representation. This scheme is referred to as *lazy carry-save duplication*.

Lazy carry-save duplication can increase efficiency over traditional lazy duplication in two ways. First, the cost of a carry-propagate adder is replaced with a slight increase in the complexity of the modified checker; the cost increase of the checker is strictly equal to or less than that of the carry-propagate adder, leading to some savings [100]. Second, and more notably, the latency of carry-propagate addition is avoided; this additional checking slack may be used to reclaim checking efficiency in a manner similar to lazy duplication. The latency of this carry-propagation is significant, taking roughly a third of the time for a logarithmic-time multiplication.[17] A visualization of the carry-save duplication process is shown in Figure 4.2. The carry-save equality checker (Section 3.1.2) is used to check the results of multiplication, similar to its application for fixed-point addition.

Table 4.2 reports the cost of both lazy and lazy carry-save duplication. The benefits of lazy carry-save duplication are significant and robust; several trends are of note. Carry-save duplication adds some additional latency relative to the fastest strict or lazy duplicate designs due to the need to perform modified

17 The 16, 32, and 64-bit baselines spend roughly 28%, 37%, and 29% of their time performing the carry-propagate addition, respectively.

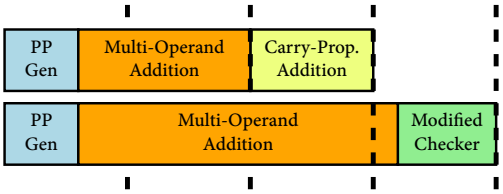


Figure 4.2: Carry-propagate addition can be eliminated in the duplicate multiplier by employing a carry-save equality checker.

checking following the main multiplication. For any achievable detection latency, carry-save duplication shows superior efficiency to lazy duplication, achieving roughly the same area efficiency as a lazy duplicate design with 30% additional detection slack.

The cost savings of carry-save duplication saturate quickly, such that it is neither necessary nor profitable to increase the checking latency past 30–40%. This is because there is sufficient slack in the absence of duplicate carry-propagate addition to use the least expensive standard cells at this point. For this reason, lazy duplication and carry-save duplication asymptote to the same efficiency; carry-save duplication just gets there more quickly. The use of a slower and more area-efficient design (or the use of a flexible delay-proportional multiplier, such as the DesignWare PPArch multiplier [117]) would almost certainly allow for carry-save duplication to provide detection latency-proportional savings.

4.2.1 *Carry-Save Karatsuba Duplication*

The implementation of carry-save multiplication is straightforward for a fully parallel, tree-based multiplier like the considered baselines. There are alternative multiplier architectures, however, where the adoption of carry-save checking is slightly more nuanced. With careful design, carry-save checking can apply to a wide class of parallel multipliers. This section demonstrates the flexibility of carry-save duplication through its application to a Karatsuba multiplier.

Karatsuba multiplication (originally attributed to [118]) is a divide-and-conquer scheme that is able to perform N-bit fixed-point multiplication using three $^{N}/_{2}$-bit multipliers by exploiting Identity 4.1 (where a_H, a_L, b_H, and b_L rep-

Table 4.2: The overheads of lazy and lazy carry-save duplication. Lazy carry-save duplication avoids the need for carry-propagate addition in the duplicate unit, increasing efficiency.

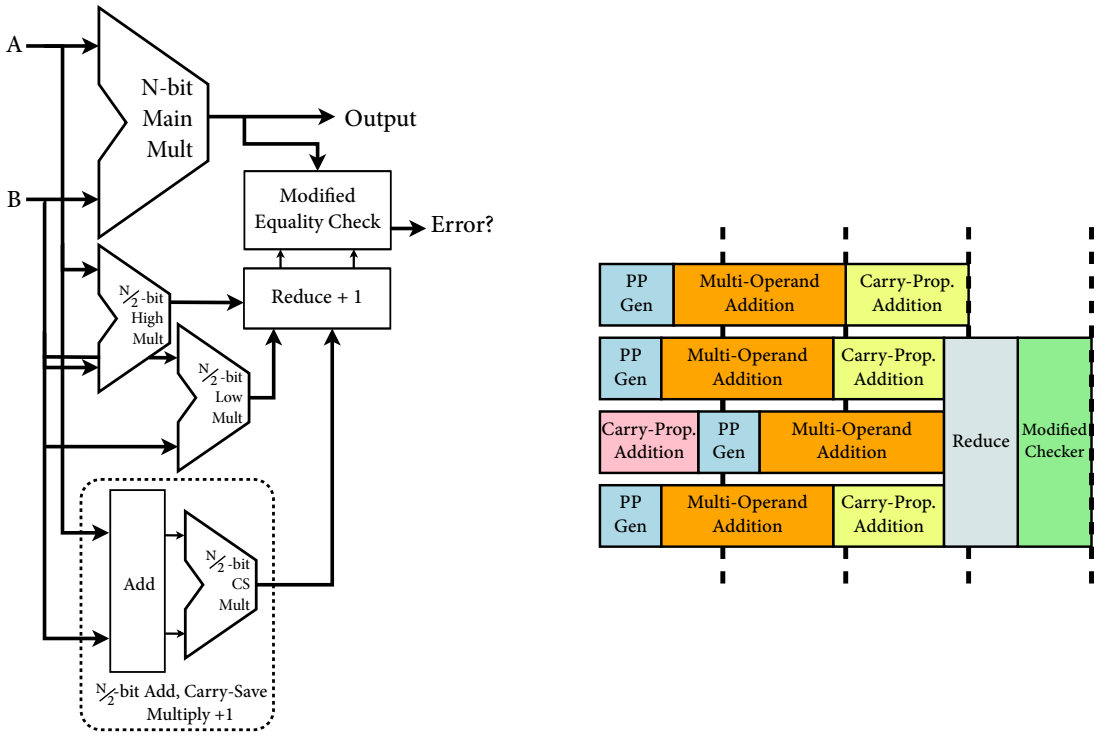
Width (N)	Detection Latency (%)	Lazy Area [μm^2] (+%)	Lazy CS Area [μm^2] (+%)
16	40	2448.4 (69.0)	1869.8 (52.7)
	50	2273.8 (64.1)	1665.2 (46.9)
	60	2093.2 (59.0)	1670.9 (47.1)
32	30	9672.9 (74.8)	7306.8 (56.5)
	40	8380.9 (64.8)	6989.8 (54.0)
	50	7855.8 (60.7)	6899.5 (53.3)
	60	7299.6 (56.4)	6899.5 (53.3)
64	30	33760.2 (88.4)	28118.6 (73.6)
	40	30698.8 (80.4)	27625.9 (72.4)
	50	29834.2 (78.1)	27625.9 (72.4)
	60	28153.1 (73.7)	27625.9 (72.4)

resent the high and low halves of the input operands a and b, respectively) [101].

$$\left(2^{\frac{N}{2}}a_H + a_L\right)\left(2^{\frac{N}{2}}b_H + b_L\right) = 2^N a_H b_H + a_L b_L + 2^{\frac{N}{2}}\left((a_H + a_L)(b_H + b_L) - a_H b_H - a_L b_L\right) \tag{4.1}$$

Karatsuba multiplication can be somewhat area efficient at large word lengths, since the area of parallel multiplication tends to scale quadratically and it replaces a full N-width multiplier with just three smaller $N/2$ multipliers [101]. However, hidden in the $2^{\frac{N}{2}}\left((a_H + a_L)(b_H + b_L) - a_H b_H - a_L b_L\right)$ term of Karatsuba multiplication is a lengthy adder carry propagation before the results of $(a_H + a_L)(b_H + b_L)$ can be determined. This additional latency negatively impacts the overall efficiency of the scheme.

An optimized Karatsuba baseline is used that avoids many of the latency issues with the $(a_H + a_L)(b_H + b_L)$ term. By slightly modifying the multiplier, the inner subtractions $(-a_H b_H - a_L b_L)$ can be performed without any carry propagation. This is done as follows: a regular carry-save adder (CSA) is used with complemented inputs from the $a_H b_H$ and $a_L b_L$ multipliers. The two incrementations necessary for two's complement arithmetic are achieved in a carry-free manner by (1) setting the empty carry bit in the least-significant position of the CSA and (2) placing an extra bit in the partial product generation of the



(a) Carry-Save Karatsuba Duplication Block Diagram (b) Carry-Save Karatsuba Duplication Visualization

Figure 4.3: Karatsuba multiplication can perform an N-bit fixed-point multiplication using three $N/2$-bit multipliers. Carry-free duplication uses a modified checker to eliminate the carry-propagate addition of the sub-components.

$(a_H + a_L)(b_H + b_L)$ multiplier. Neither incrementation has any impact on the latency or complexity of the resultant duplicate multiplier.

Carry-save Karatsuba duplication uses a carry-save checker for the final addition of the constituent subcomponents, as shown in Figure 4.3. Table 4.3 reports the overheads of both lazy and carry-free Karatsuba duplication relative to 16, 32, and 64-bit baseline Karatsuba multipliers. Again, carry-save duplication shows consistent, robust efficiency improvements. By avoiding the latency of the final carry-propagate addition, carry-save duplication is able to reach the efficiency of the asymptotic lazy checker with about a 20% detection latency; it takes lazy duplication 20–30% more slack to compete.

Table 4.3: The overheads of lazy and carry-save Karatsuba duplication (relative to a baseline Karatsuba multiplier).

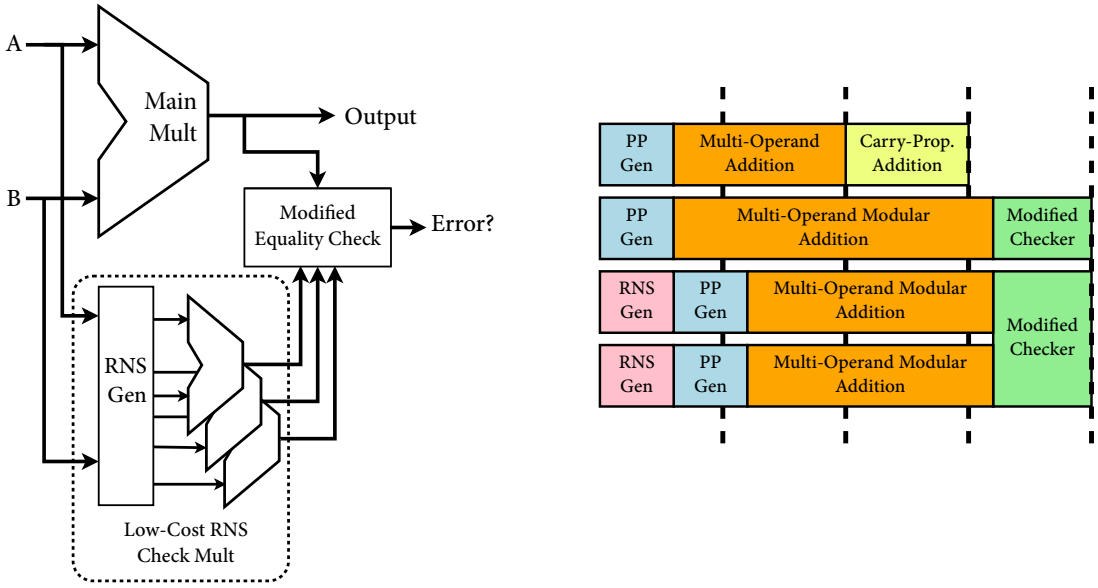
Width (N)	Detection Latency (%)	Lazy Area [μm^2] (+%)	Lazy Carry-Save Area [μm^2] (+%)
16	20	2651.1 (66.7)	1988.9 (50.0)
	30	2234.1 (56.2)	1799.1 (45.3)
	40	1834.9 (46.2)	1697.5 (42.7)
	50	1752.9 (44.1)	1693.6 (42.6)
	60	1712.0 (43.1)	1693.6 (42.6)
32	10	9888.4 (89.8)	N/A
	20	9652.1 (87.7)	6800.7 (61.8)
	30	7531.7 (68.4)	6183.3 (56.2)
	40	6532.1 (59.3)	6044.7 (54.9)
	50	6221.2 (56.5)	5908.5 (53.7)
	60	6088.2 (55.3)	5888.2 (53.5)
64	10	32499.5 (91.0)	N/A
	20	30662.8 (85.9)	24355.4 (68.2)
	30	27025.9 (75.7)	23073.2 (64.6)
	40	24932.4 (69.8)	22947.2 (64.3)
	50	23534.6 (65.9)	22827.1 (63.9)
	60	23146.7 (64.8)	22812.8 (63.9)

4.3 RESIDUE NUMBER SYSTEM DUPLICATION

A compelling low-cost duplication alternative using the residue number system (RNS) is described and evaluated. Before delving into the details of RNS duplication, the basics of the residue number system are reviewed.[18]

The residue number system represents integer values using a small number of non-weighted digits. An RNS number is formed from a weighted number, X with respect to a set of n co-prime bases, $\left(m_0|m_1|...|m_{n-1}\right)$. To convert to the RNS representation, the residue of X is formed with respect to each base. In general, the formation of an arbitrary residue $\left(|X|_m ; m \in \mathbb{N}\right)$ is expensive; for efficiency, designs often restrict themselves to specialized moduli in the form $m = 2^a \pm 1$, $a \in \mathbb{N}$. Common arithmetic operations can be performed without carry propagation between the digits of RNS numbers, significantly increasing the speed. Operations within each digit are carried out in a modular manner

18 This short summary is felt to be sufficient in the context of low-cost duplication. For a more formal introduction, the reader is referred to [119]. For a comprehensive treatment of RNS arithmetic, see [109, 120, 121].



(a) RNS Duplication Block Diagram

(b) RNS Duplication Visualization

Figure 4.4: A block diagram and visualization of RNS duplication. "RNS Gen" represents residue generation. Partial product generation and multi-operand addition are modified specific to each modulus. Multiplication of an RNS-encoded number can proceed more quickly than its fixed-point counterpart; this extra slack can lead to overall cost savings.

such that the arithmetic result for the RNS digit corresponding to a modulus m is computed as $|X \oplus Y|_m = ||X|_m \oplus |Y|_m|_m$, where $\oplus \in \{+, -, *\}$.

Despite their fast arithmetic speed, the general-purpose usefulness of RNS numbers is greatly limited by practical concerns: bit-wise logical operations, truncation, division, sign detection and magnitude comparison are all expensive in this representation, as is the conversion back to a fixed-point format. Low-cost duplicate RNS multiplication is able to exploit the superior speed and diversified design of the residue number system while avoiding the aforementioned limitations. Because the output of the duplicate RNS unit is discarded after error detection, no expensive operations are performed nor is backwards conversion necessary. Figure 4.4 shows an organization of RNS duplication—the input operands are converted to the RNS representation, and the modular arithmetic for each RNS modulus proceeds in parallel.

To evaluate the idea of RNS duplication, an RNS duplicate multiplier is formed. The residue generator circuitry from [122] is used along with the parallel modular multipliers from [114] (mod 2^a-1) and [123] (mod 2^a+1). Following

Table 4.4: The overheads of residue number system duplication (relative to the compressor-based fully parallel multiplier).

Width (N)	Checking Latency (%)	Modulus 1 ($2^a \pm 1$)	Modulus 2 (2^b)	Area [μm^2] (+%)
16	30	5	23	1982.2 (55.9)
	40	5	23	1974.4 (55.7)
	50	9	15	1959.6 (55.2)
	60	9	15	1913.9 (53.9)
32	30	18	29	8119.9 (62.8)
	40	18	29	7323.4 (56.6)
	50	17	31	7065.0 (54.6)
	60	17	31	6957.2 (53.8)
64	30	41	47	33170.1 (86.9)
	40	41	47	31410.9 (82.3)
	50	41	47	29938.6 (78.4)
	60	41	47	29040.2 (76.1)

the recommendations of [124], a moduli set in the form $\{2^a - 1,\ 2^a + 1,\ 2^b\}$ is employed. Table 4.4 shows the overheads of RNS duplication, along with the moduli sets used.[19] The experimental results indicate that the speed advantages of RNS arithmetic provide modest cost savings in the duplicate multiplier through increased design slack. Also, because RNS arithmetic is faster than the main multiplication, there is sufficient slack to saturate these benefits at 30–40% detection latencies and further detection latency does not significantly lessen the overheads of detection.

These experimental results demonstrate that RNS duplication can provide diversified, low-latency duplication that is completely off of the critical path. This allows RNS duplication to provide strong, fault-agnostic error detection. The efficiency of RNS duplication is on par with that of lazy carry-save duplication (Section 4.2). Furthermore, it is possible that the RNS organization and experimental methodology used in this dissertation could under-represent the potential efficiency of RNS duplication; Section 4.4.1 describes some of the future research that these initial results warrant.

19 These moduli sets were chosen through a brief computer-guided search and are expected to be aggressive but not optimal.

4.4 DISCUSSION AND FUTURE WORK

Low-cost duplicate multiplication presents the opportunity for many exciting avenues of future research. Some discussion of this potential future work follows.

4.4.1 Further RNS Duplication Evaluation

There are several future experiments that may better represent and analyze the potential advantages of RNS duplication. First, it has been noted that some of the efficiency advantages of RNS arithmetic come from its increased circuit regularity relative to two's complement arithmetic units [101]; these layout advantages are not taken into account in this work for methodological reasons. Also, initial experiments indicate that an RNS duplicate multiplier can be faster than the main arithmetic unit, garnering modest area savings. This high speed could be better exploited for increased efficiency by more flexible modular multiplier designs or by using a multi-Vth design flow where small and low power (but slow) high Vth cells are mixed with their standard Vth equivalents. The use of a lower supply voltage for the RNS duplicate unit could also lead to substantive power savings; it has been noted in the past that the speed of RNS arithmetic allows for reduced-voltage operation [125].

4.4.2 Alternate Number Systems and Organizations

The low-cost duplicate checkers described by this work are based on carry-save and RNS arithmetic and by no means exhaust the search space. Alternate low-cost duplicate checkers and different organizations of the described checkers undoubtedly exist. Some avenues of future research include other RNS organizations, including the use of different moduli sets [126, 127, 128], pseudo-residues [129] or redundancy through non-coprime moduli [130, 131]. Also promising is the investigation of other number systems where multiplication is inexpensive, such as the logarithmic number system [132] (or approximate binary logarithms [133]) and the use of index calculus for inexpensive multiplication [134].

4.4.3 *Signed Arithmetic and Multiply-Accumulate*

The described low-cost duplication scheme works for signed multiplication and multiply-accumulate with trivial modifications. Support for signed multiplication can be achieved by using Booth-recoded parallel prefix generation or with Baugh-Wooley's method for fixed-point multipliers [101]. Multiply-accumulate can be supported by adding an additional level to the multi-operand adder that accumulates the partial products of multiplication [101]. The modifications for signed arithmetic and multiply-accumulate are minimal and add roughly the same complexity to both the main arithmetic unit and duplicate unit. It is therefore expected that carry-save duplication will provide efficient error detection for these operations as well.

5 FLOATING-POINT MULTIPLICATION

Floating-point numbers are used extensively to represent real and continuous values in computer systems. The characteristics of floating-point arithmetic tends to make their protection difficult and costly—because of their piecewise and complicated nature, there is no arithmetic error code that is closed under floating-point arithmetic and strict duplication doubles the (already significant) area and energy costs of floating-point operations. This chapter investigates the application of low-cost duplication using approximate number systems for floating-point multiplication.[20] This new paradigm is referred to as *approximate duplication*, and it is able to provide low-cost *precision-proportional* error detection, offering error coverage whose cost is commensurate with the maximum relative magnitude of any undetected errors.

Figure 5.1 shows a block diagram of approximate duplication. A reduced-precision duplicate unit uses an approximate number system and a specialized checker to detect any error that exceeds a known tolerance. Unlike most prior approaches for floating-point error detection, separability is not compromised—no communication with the main floating-point unit (FPU) is necessary and the approximate duplicate unit neither dictates nor reacts to the design of the protected multiplier.

The use of approximate number representations for the low-cost duplication of floating-point arithmetic makes intuitive sense taking the specifics of floating-point numbers into account. Because they seek to represent the infinitely-dense real number system in a finite number of bits, floating point values are inherently approximate. Concepts such as rounding and imprecision accumulation[21] exist, and potentially-unbounded relative imprecision can occur even during the course of error-free floating-point arithmetic [137]. It is therefore felt that

20 Due to its importance and standardization, this dissertation restricts itself to the IEEE 754-2008 floating point format [135]. It should also be noted that these analyses are focused on the common-case uses of IEEE 754-2008 floating-point numbers. Corner cases such as exceptions, denormalized numbers, or design decisions such as partial software support [136] must be accounted for and protected with other mechanisms.

21 While many sources refer to the accumulation of rounding as "error" [135, 137], here the term "imprecision" is used in order to differentiate it from arithmetic errors that are the manifestation of underlying faults.

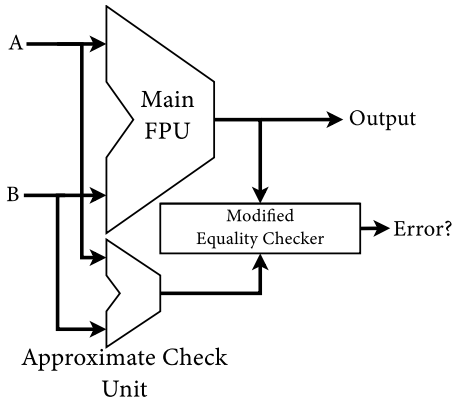


Figure 5.1: Block diagram of approximate duplication. A reduced-precision unit checks the results of computation (to within a known tolerance).

the fastidious reproduction of inherently-imprecise floating-point arithmetic is unnecessary and wasteful.

This chapter proceeds as follows. Section 5.1 gives a brief overview of floating-point multiplication. Section 5.2 describes the few and limited prior approaches taken towards floating-point error detection. Section 5.3 describes the concept of approximate duplication for downward-rounded floating point values. The organization is general enough to apply to a range of approximate multiplication schemes; one that is based off of truncated multiplication is described in depth. To investigate the fidelity that is appropriate for approximate duplication, Section 5.4 performs application-level error propagation and sensitivity experiments using floating-point-intensive programs. Finally, Section 5.5 gives some closing thoughts, caveats, and exciting areas of future research.

5.1 A BRIEF INTRODUCTION TO FLOATING-POINT MULTIPLICATION

A floating-point number represents a real value, RX, using Equation 5.1. An exponent, e, is used primarily to capture the magnitude of RX in a storage-efficient manner. The term $(1+sf)$ is typically referred to as the *significand*. The value sf is fractional ($0 \leq sf < 1$), and in this dissertation the term sf is called the *significand fraction*. Whereas the exponent is used to capture the magnitude of a real number, the significand is used to provide precision.

$$RX = \pm 2^{e} * (1 + sf) \tag{5.1}$$

The multiplication of two floating-point numbers involves multiplying the significands and adding the exponents, as shown by Equation 5.2. In terms the evaluation of computational complexity, circuit area, and latency, the significand multiplication $(1+sf_{RX})*(1+sf_{RY})$ dominates [101].

$$\begin{aligned} RX * RY &= \left(\pm 2^{e_{RX}} * (1 + sf_{RX})\right) * \left(\pm 2^{e_{RY}} * (1 + sf_{RY})\right) \\ &= \pm 2^{e_{RX}+e_{RY}} * (1 + sf_{RX}) * (1 + sf_{RY}) \end{aligned} \tag{5.2}$$

An IEEE 754 floating-point number is composed of three parts. (1) a sign bit, used to indicate whether RX is positive or negative, (2) an integer exponent value (stored in a biased format), and (3) a normalized significand, for which only the significand fraction is stored (the "hidden" 1-bit is used for calculations but need not be stored).

The range of each IEEE 754 floating-point significand falls within [1, 2), such that the product the unmodified significand multiplication is in the range [1, 4). In order for the output of floating-point multiplication to be a valid floating-point value itself, a significand and exponent adjustment is applied to any significand product equal or greater than two. This adjustment may be performed by incrementing the output exponent and scaling the significand product by $1/2.0$ through shifting it one position to the right. Furthermore, the significand multiplication must be rounded to fit within the same storage footprint as other floating-point values; if this rounding again pushes the significand above two, then a second significand and exponent adjustment is required. Figure 5.2 depicts this floating-point procedure. The sign and exponent calculations (shown in purple and green) are simple and straightforward; the cost of the significand multiplication (shown in pink) dominates, and the rounding and adjustment circuitry (shown in blue) consumes a modest amount of area but adds a significant amount of latency and complexity to the multiplication procedure.

Approximate duplication offers compelling efficiency advantages when the details of floating-point arithmetic implementation are considered. Floating-point multipliers spend the most area and energy calculating the bits of the result that impact the magnitude of the result the least, as illustrated in Figure 5.3. The relative cost of each floating-point arithmetic sub-component is inversely proportional to its impact on the magnitude of the result. Approximate duplication estimates the expensive significand calculation and does not require rounding

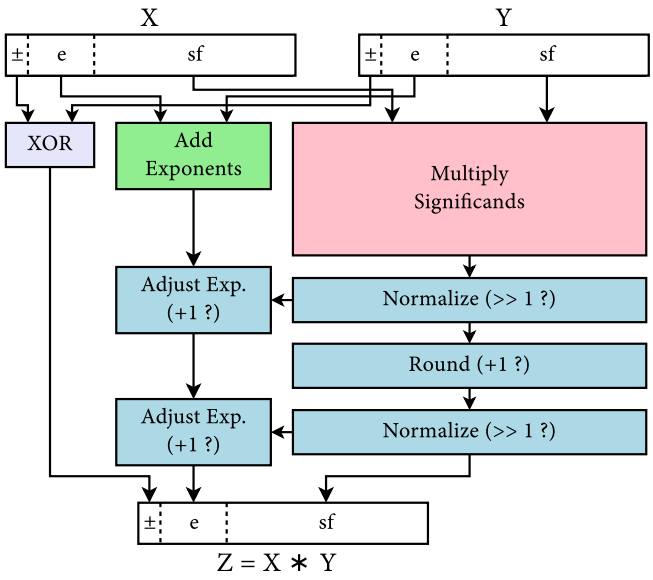


Figure 5.2: A block diagram of floating-point multiplication. The exponents of the inputs are summed, the significands multiplied, and the significand of the result normalized and rounded to fit within the footprint of the output number. Sign logic is shown in purple, exponent addition logic is shown in green, significand multiplication is shown in pink, and normalization and rounding logic is shown in blue.

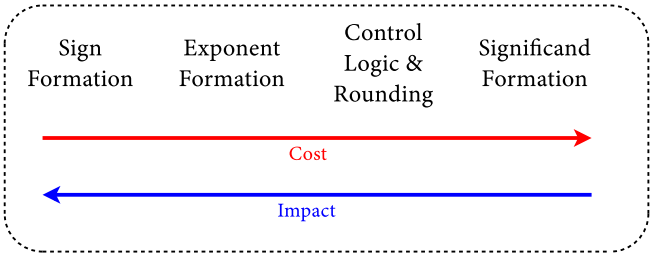


Figure 5.3: A depiction of the relative hardware cost of the components in a floating-point multiplier and their impact on the final magnitude of the result.

and output adjustment circuitry. As a result, approximate duplication is able to provide efficient floating-point error detection in a *precision proportional* manner—the cost of error detection is proportional to the maximum error introduced by any undetected error.

While there is no known arithmetic error code that is closed under floating-point arithmetic, a non-separable checker has been applied to floating-point operations [138, 139]. This approach can be thought of as applying residue or Berger-code-based checking component-by-component in the FPU, with some sharing of control signals. It is inseparable, assumes an end-to-end protection scheme, and its error coverage will be limited by the strength of the error codes it uses.

The intuition that floating-point intensive programs have inherent imprecision (and therefore may be tolerant to error) has been used to justify a partially-duplicated checker [24, 140]. However, it has been previously observed that while many scientific codes can tolerate limited floating-point errors, they can catastrophically fail if the error impacts the most significant bits [141]—the incomplete coverage of partial duplication leaves a significant chance of failure or silent data corruption. The error coverage semantics of approximate duplication (complete error coverage within a known tolerance) allow for the error detection capabilities to match the error tolerance of scientific applications. In Section 5.4, application-level error propagation and sensitivity experiments corroborate this observation. It is shown that many floating-point programs are able to tolerate a small relative amount of undetected error, but that undetected large errors often lead to program failure or data corruption.

There is some prior research that serves as a relevant precursor to the concept of approximate duplication. Eibl, Cook, and Sorin [142] present a brief-yet-significant design that could be considered approximate duplication for floating-point addition; this dissertation extends the concept of approximate duplication to the more costly operation of floating-point multiplication, and also performs application-level error injection to analyze the effects of bounded error semantics. The use of approximate binary logarithms for approximate duplication has also been described for fixed-point multiplication. Davis describes the concept of using Mitchell's approximation [133], to check fixed-point multiplication [143], as do Sellers, Hsiao, and Bearnson [25].[22] More recently, two

22 Sellers, Hsiao, and Bearnson also describe use of a more precise approximation [144] for checking fixed-point multiplication.

publications have utilized similar schemes for the concurrent checking of fixed-point digital signal processing hardware [145, 146]. This dissertation represents the first application of approximate duplication (with or without approximate binary logarithms) to floating-point multiplication.

5.3 APPROXIMATE DUPLICATION BASED ON TRUNCATED MULTIPLICATION

Truncated multiplication[23] is often used to control the width of fractional arithmetic. The output of an N-bit truncated multiplier is the t most-significant-bits of the full $2N$-bit product. This section describes a straightforward approximate duplicate organization that employs a truncated fractional multiplier to check the results of floating-point multiplication. For simplicity, downward-directed floating-point arithmetic is assumed throughout Sections 5.3–5.3.2; the changes necessary to support other IEEE 754-2008 rounding modes are investigated later in Section 5.3.3.

Various schemes exist to minimize the error of truncation without resorting to a fully-rounded result [147, 148, 149, 150, 151, 152]. In lieu of such approaches, this section employs a simple downward-rounded truncated multiplier—the semantics of approximate duplication focus on the maximum undetected error, such that minimizing the average undetected error is of secondary importance.

A downward-rounded truncated multiplier is very simple: it forms and adds the most-significant positions in the partial product matrix, implicitly forcing all lesser entries to zero. Table 5.1 illustrates such a multiplier by way of an example: the results of a 3-bit significand multiplication (before normalization or rounding) normally would result in a 6-bit full-width product; by using just the first 3 bits of the partial product matrix, an approximate significand is formed with a maximum error of $100 * 2^{-3} = 12.5\%$. Notice in Table 5.1 that while the relative error of the result is bounded, the bits of the approximate result can differ from those of the exact result—for example, $1.75 * 1.75$ differs in multiple bit positions, despite only having a -10.204% relative error.

23 Truncated multiplication is sometimes also called *fixed-width* multiplication [147].

Table 5.1: An example of truncated multiplication. Two 3-bit significands are multiplied. DecE and BinE denote the exact result (before normalization or rounding), and DecA and BinA give the approximate result from a truncated multiplier using the first three bits of the partial product matrix.

S_A		S_B		Product (S_Z)				Rel. Err. (%)
Dec.	Bin.	Dec.	Bin.	DecE	BinE	DecA	BinA	
1.00	1.00	1.00	1.00	1.0000	01.0000	1.00	01.00	0.000
1.00	1.00	1.25	1.01	1.2500	01.0100	1.25	01.01	0.000
1.00	1.00	1.50	1.10	1.5000	01.1000	1.50	01.10	0.000
1.00	1.00	1.75	1.11	1.7500	01.1100	1.75	01.11	0.000
1.25	1.01	1.00	1.00	1.2500	01.0100	1.25	01.01	0.000
1.25	1.01	1.25	1.01	1.5625	01.1001	1.50	01.10	-4.000
1.25	1.01	1.50	1.10	1.8750	01.1110	1.75	01.11	-6.670
1.25	1.01	1.75	1.11	2.1875	10.0011	2.00	10.00	-8.570
1.50	1.10	1.00	1.00	1.5000	01.1000	1.50	01.10	0.000
1.50	1.10	1.25	1.01	1.8750	01.1110	1.75	01.11	-6.670
1.50	1.10	1.50	1.10	2.2500	10.0100	2.25	10.01	0.000
1.50	1.10	1.75	1.11	2.6250	10.1010	2.50	10.10	-4.760
1.75	1.11	1.00	1.00	1.7500	01.1100	1.75	01.11	0.000
1.75	1.11	1.25	1.01	2.1875	10.0011	2.00	10.00	-8.570
1.75	1.11	1.50	1.10	2.6250	10.1010	2.50	10.10	-4.760
1.75	1.11	1.75	1.11	3.0625	11.0001	2.75	10.11	-10.204

5.3.1 *An Imprecision Threshold Checker for Truncated Significand Multiplication*

The use of a truncated significand multiplier to cheapen duplication is fairly simple—such a scheme has been used in its own right to perform low-power approximate arithmetic (albeit with correction circuitry to reduce the average-case imprecision) [153]. The design of the threshold checker necessary to enforce the semantics of approximate duplication is a bit more nuanced, however.

Ideally, an imprecision threshold checker for approximate duplication could check whether the relative difference between the exact (yet possibly erroneous) result and the approximate duplicate result differ by more than the maximum imprecision of approximation. In general, however, the determination of relative imprecision between two values requires division, making it prohibitively lengthy and expensive. To simplify the relative imprecision check, this dissertation makes the observation that the output significand is a normalized value—that is, its

magnitude falls between [1, 2). Therefore, if the difference between the output of floating-point multiplication, S_{RZ} and approximate duplication, S'_{RZ}, is given by $S_{RZ} - S'_{RZ}$, then a conservative bound on the relative imprecision of the result is $(S_{RZ} - S'_{RZ})/1.0$. The looseness of this bound is maximized when S_{RZ} approaches 2.0, resulting in an effective loss of one bit of precision; therefore, this imprecision threshold checker can use a t-bit truncated multiplier to check the results of a floating-point significand multiplication to within a tolerance of $100 * 2^{-(t+1)}$%, a modest penalty given the simplicity of the checker.

An implementation of the imprecision threshold checker for truncated significand multiplication is straightforward. The normalized output of floating-point multiplication, S_{RZ}, and the output of a t-bit truncated multiplier, S'_{RZ}, are fed into the checker. A full carry-propagate subtraction is performed on the two values.[24] At the end of the subtraction, if any of the $(t-1)$ most significant bits are set, then the values differ by more than $100 * 2^{-(t+1)}$%.

5.3.2 *Using Truncated Significand Multiplication for Approximate Duplication*

A truncated significand multiplier and its imprecision threshold checker can be employed to form a separable approximate duplicate floating-point multiplier. Figure 5.4 shows an organization of this error detector—only the input floating-point values (RX and RY) and the output of the floating-point multiplier, RZ, are required and the scheme is fully separable. (This represents the first known fully-separable checker floating-point multiplication apart from full duplication.) The checker proceeds as follows: the sign logic is duplicated and equality checked, the exponent addition is duplicated with a dual incremented output, and the significand is approximately duplicated using a truncated multiplier. Well-established and simple techniques can produce the dual output of an addition and its increment [114, 154, 155, 156, 157]; these techniques can be used to produce the two exponent addition outputs required for the checker with negligible overhead. The dual outputs from the exponent addition and the approximate significand are passed to a specialized approximate duplication equality checker whose design is described below.

24 This carry-propagate subtraction can be significantly simplified because of the reduced width of the truncated approximate multiplier. However, it still requires a full carry-propagation and a logarithmic number of logic stages.

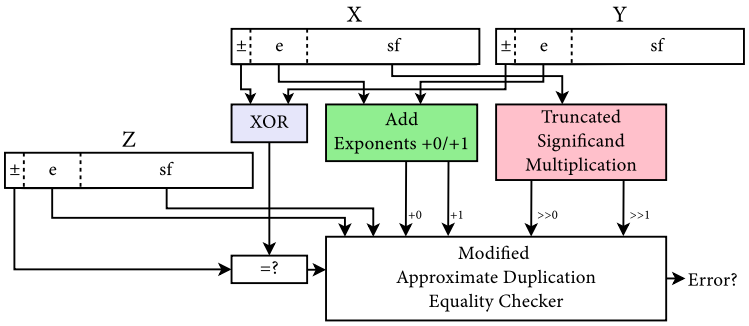


Figure 5.4: A block diagram of approximate duplication using a truncated significand multiplier. The scheme is general enough to use any form of approximation, so long as it always underestimates the exact result and has a well-characterized maximum relative imprecision.

The modified approximate equality checker checks for the conditions in Equation 5.3 and Equation 5.4, where E_{RZ} and S_{RZ} are the exponent and normalized significand of the floating-point product, E' is the exact output of the duplicated exponent adder ($E' = RX_e + RY_e$), and S' is the unnormalized approximate significand ($S' = (1 + RX_{sf}) \tilde{*} (1 + RY_{sf})$). The notation $\wedge$ and $\vee$ represent a conditional AND and OR, respectively. Approximate equality (denoted "$\stackrel{?}{\approx}$") is evaluated using the imprecision threshold checker from Section 5.3.1. Approximate magnitude comparison (denoted "$\tilde{\geq}$") is formed in a similar manner—its purpose is to check for errors in the floating-point normalization logic, and if it is properly designed then the fact that it compares magnitude approximately will not have an adverse affect on the error coverage of approximate duplication. The reasoning behind this statement is explained below.

$$\left(E_{RZ} \stackrel{?}{=} E'\right) \vee \left(\left(E_{RZ} \stackrel{?}{=} E' + 1\right) \wedge \left(S' \tilde{\geq} 2.0\right)\right) \tag{5.3}$$

$$\left(S_{RZ} \stackrel{?}{\approx} S'\right) \vee \left(\left(S_{RZ} \stackrel{?}{\approx} \frac{S'}{2.0}\right) \wedge \left(S' \tilde{\geq} 2.0\right)\right) \tag{5.4}$$

Approximate magnitude comparison is used in Equation 5.3 and Equation 5.4 to check for errors in the floating-point normalization logic. Given that the approximate significand is always underestimated using truncated multiplication and has a maximum relative imprecision of $MI\%$, $MI \leq 50\%$,[25] this approximate

25 This expression means that the weakest possible truncated multiplier has less-than-50% relative maximum error. Larger truncated multipliers have accordingly smaller maximum error.

magnitude comparison checks for $S' \geq \frac{MI}{100} * 2.0$. Depending on whether there is an error in the normalization logic and the value of the approximate significand, the following cases may occur.[26] The notation $\tilde{<}$ is used to denote the complement of the approximate magnitude comparator.

(1) **No error in normalization; No normalization;** $S' \tilde{<} 2.0$**.**

There is no error and no magnitude comparison imprecision. $E_{RZ} = E'$ and $S_{RZ} \approx S'$.

$\Longrightarrow$ **No error detected.**

(2) **No error in normalization; No normalization;** $S' \tilde{\geq} 2.0$**.**

There is no error and no normalization but the approximate comparison indicates that there is normalization due to imprecision. The equality tests $E_{RZ} = E'$ and $S_{RZ} \approx S'$ resolve despite this comparison approximation.

$\Longrightarrow$ **No error detected.**

(3) **No error in normalization; Normalization occurs;** $S' \tilde{\geq} 2.0$**.**

There is no error. Whenever normalization occurs the approximate comparison is precise due to the comparison threshold; therefore, equality tests $E_{RZ} = E' + 1$ and $S_{RZ} \approx \frac{S'}{2.0}$ resolve along with $S' \tilde{\geq} 2.0$.

$\Longrightarrow$ **No error detected.**

(4) **Erroneous normalization performed;** $S' \tilde{<} 2.0$**.**

An erroneous normalization corrupts the exponent and significand of the FPU product. The equality tests for $E_{RZ} = E' + 1$ and $S_{RZ} \approx \frac{S'}{2.0}$ resolve, but $S' \tilde{\geq} 2.0$ fails.

$\Longrightarrow$ **Error detected.**

(5) **Missing normalization;** $S' \tilde{\geq} 2.0$**.**

Normalization does not occur despite the fact that it should. In this

26 These six cases exhaust the possibilities given a single-component error. In the case of simultaneous errors in both the checker and FPU, other situations may occur and error coverage will be impacted.

case, $E_{RZ} = E'$ and the produced significand will be truncated to form a fractional significand, corrupting its value. The magnitude of this significand corruption necessarily causes the test $S_{RZ} \stackrel{?}{\approx} S'$ to fail.

If the error-free unnormalized significand is 2.0, the missing normalization corrupts the output normalized significand to 1.0 such that $S_{RZ} \approx \frac{S'}{2.0}$. However, $E_{RZ} \neq E' + 1$ such that the approximate equality check still fails.

$\Longrightarrow$ **Error detected.**

(6) **Erroneous normalization performed;** $S' \tilde{\geq} 2.0$**.**

An erroneous normalization corrupts the exponent and significand of the generated product, and $S' \tilde{\geq} 2.0$ due to approximation imprecision. The equality tests for $E_{RZ} = E' + 1$ and $S_{RZ} \approx \frac{S'}{2.0}$ resolve along with $S' \tilde{\geq} 2.0$.

In this case, no error is detected. Due to the semantics of the approximate magnitude comparison, however, it is known that the error-free significand is at most the imprecision threshold checker's threshold less than 2.0. The corruption to the exponent and significand of the floating-point product due to an erroneous normalization therefore causes a worse-case magnitude change that is within the guaranteed bound of approximate duplication.

$\Longrightarrow$ **Error undetected, but the magnitude of the error is bounded and the semantics of approximate duplication are preserved.**

Approximate duplication provides fully-separable error detection for floating-point multiplication through four mechanisms. (1) The sign equality checker precisely checks the sign of the output with very little overhead; (2) the exponent addition equality checker precisely checks the exponent (apart from erroneous incrementation); (3) the imprecision threshold checker checks the significand to within a guaranteed bound; (4) the normalization logic dictated by Equation 5.3 and Equation 5.4 allows normalization to proceed without causing false-positives, checks for erroneous incrementation in the exponent, and checks for erroneous normalization.

5.3.3 *Separable Detection of Different Floating-Point Rounding Modes*

Having established how approximate duplication can provide separable error detection for downward-directed multiplication, this section extends the approximate duplicate checker to handle the added complexities of the other rounding modes in the IEEE 754-2008 standard.

The main intuition behind the separable checking of rounding modes with approximate duplication is that any single incorrectly-rounded value does not significantly affect the magnitude of the result—rounding occurs at the least-significant position of the significand, such that its effect is minuscule compared to the threshold of approximate duplication. Therefore, any corruption due to rounding imprecision can be subsumed into the guaranteed error bound and allowed without changing the behavior or error semantics of approximate duplication.

In order to satisfy the semantics of approximate duplication in the presence of rounding errors, a small additional amount of rounding imprecision must be incorporated into the threshold checker and the approximate magnitude comparators. In practice, this is not expected to change the circuits at all—the approximate checkers and comparators already take a loose bound on the maximum relative imprecision for simplicity (see Section 5.3.1).

5.3.4 *Use of Other Approximation Schemes and Number Systems*

The approximate-duplication organization is general enough to apply to a large class of approximate multiplication schemes and number systems. So long as the maximum relative imprecision of approximate significand multiplication is well-characterized, approximate duplication should work. A few design caveats of the specific organization presented here apply; these caveats are not fundamental to the concept of approximate duplication, but are rather due to simplifying assumptions made for conceptual clarity and implementation efficiency. (1) this chapter assumes that the approximate significand multiplier always under-estimates the true product; (2) the imprecision threshold checker from Section 5.3.1 uses a loose imprecision bound and will not benefit from improved precision that does

not cross a power-of-two threshold;[27] (3) the described imprecision threshold checker only reacts to the maximum relative imprecision of an approach, and average-case precision improvements will be squandered.

Multiplication through approximate binary logarithms ([133, 144, 158, 159, 160, 161] and others) is an attractive alternative to truncated significand multiplication for approximate duplication. Following approximate conversion to the logarithmic number system (LNS), multiplication can be performed with the superior speed and efficiency of addition. Approximate binary logarithms have been proposed for the checking of fixed-point multiplication [25, 143, 145, 146] and they should apply to approximate duplication for floating-point multiplication so long as the checker is modified to handle an LNS input. While approximate LNS duplication is worth investigating, its use has been forgone in favor of the conceptually-simpler truncated fractional multiplier until the conceptual foundations of approximate duplication are solidly established.

5.4 APPLICATION-LEVEL ERROR INJECTION AND SENSITIVITY STUDY

So far, it has been established that approximate duplication can offer separable error detection for floating-point arithmetic with full error coverage within a known tolerance. The use of truncated multiplication for approximate significand duplication has been described; truncated multiplication can be parameterized at design time to provide precision-proportional error detection, offering arbitrarily high error coverage with commensurate costs.

To investigate the fidelity that is appropriate for approximate duplication, an application-level error injection campaign is conducted. These error injection results focus on transient errors in high-performance computing (HPC) programs. Such transient errors are especially concerning for scientific applications, as they have a high associated risk of silent data corruption—other more permanent errors are expected to quickly lead to fail-stop scenarios which then no longer impact the correctness of the application.

27 Going from $12.5\% = 2^{-3}$ maximum relative imprecision to $7.5\% = 2^{-3.737}$ will not improve the bound of approximate duplication using the current imprecision threshold checker, for instance, but going past $6.25\% = 2^{-4}$ will.

5.4.1 *Binary Instrumentation-Based Error Injector*

This dissertation employs an error-injection methodology that simulates the effect of unmasked and undetected FPU faults (in logic or latches) using synthetic error models to produce representative errors. This is done using a fast, binary-instrumentation-based [162] error injector.

An important design component of the error injection methodology is that it only considers errors that go undetected by hardware error-detection mechanisms—the experiments in this dissertation investigate the coverage of partial duplication (similar to [24, 140]) and approximate duplication. Both partial duplication and approximate duplication offer incomplete single-component error coverage, potentially allowing some uncaught errors. The bounded error semantics of approximate duplication, however, are well-aligned with program needs and can be used by naturally-resilient algorithms to ensure reliable execution without the costs associated with a complete duplicate checker. By injecting undetected errors, the application-level implications of the error semantics of different checkers can be readily studied.

5.4.1.1 COMPARISON TO PRIOR ERROR INJECTORS

Error injection has long been performed at different system levels in order to test reliability mechanisms and to study error sensitivity and propagation in circuits, architectural organizations, or applications [22]. Error injectors based on RTL-level injection [163, 164, 165] or cycle-accurate-simulation [166, 167, 168] have the potential to be accurate, but they lack the speed to study application-level error propagation at-scale. Application-level error injectors exist through compiler-based instrumentation through LLVM [169], the program debugging interface [170], virtual machines [169, 171], field-programmable-gate-array based acceleration [172, 173] and binary instrumentation [174].

The error injection tool used in this dissertation is likely to be similar in speed to the LLVM and virtual-machine based approaches, but with the superior flexibility of binary instrumentation—a trait shared by the other published binary-instrumentation-based injector [174]. Due to its superior flexibility, binary instrumentation does not require special recompilation to inject errors into a program, and it can detach following error injection, roughly doubling the

average execution speed of error injection. The detection-mechanism-aware injection methodology employed by the error injector in this dissertation is unique; prior application-level error injectors either describe a narrow and synthetic error model for transient faults (such as a single bit-flip at the output of an instruction [169, 170, 171, 174]), inject into an unprotected circuit model [172, 173], or do not specify how an injected error is expected to manifest.

5.4.2 *Error Injection Results*

Figure 5.5 shows the results of the error injection campaign. Seven floating-point intensive NAS parallel benchmarks [28] [175] are evaluated using the non-trivial "A" input set, and the co-design mini-app CoMD [176] is run to completion. This experiment utilizes the NAS parallel benchmarks because of their relevance to HPC workloads, and also because they have well-defined acceptance procedures; these acceptance tests are used to judge whether an unacceptable level of silent data corruption occurs following an injected error. The CoMD miniapp is used because of its relevance to the emerging co-design paradigm; the fact that it also has programmer-specified assertions makes it interesting to study, as well. Because CoMD lacks a verification procedure, any human-visible output corruption is assumed to be significant.[29]

Six error detection schemes are evaluated with a synthetic error model that replaces any unprotected bits with a randomly generated value. The "random" scheme represents an unprotected floating-point multiplier—no bits are protected from potential corruption. The "mantrand" scheme represents partial duplication of the sign and exponent logic (with some non-separable communication from the rounding and normalization logic[30])—the effects of errors are localized to the normalized significand. The remaining error detection schemes, denoted "mantrand2" through "mantrand16", represent varying levels of approximate duplication using a truncated significand multiplier. These schemes provide error coverage for the 2–16 most significant bits of the fractional significand product, respectively, bounding any undetected error to the lesser bits.

28 To simplify error injection, this experiment uses a serial version of each benchmark.

29 The program-generated, human-readable output file that is generated by CoMD has truncated output values so small relative errors may not be human-visible.

30 This communication is not simulated, and it is assumed that the sign and exponent are protected with complete error coverage.

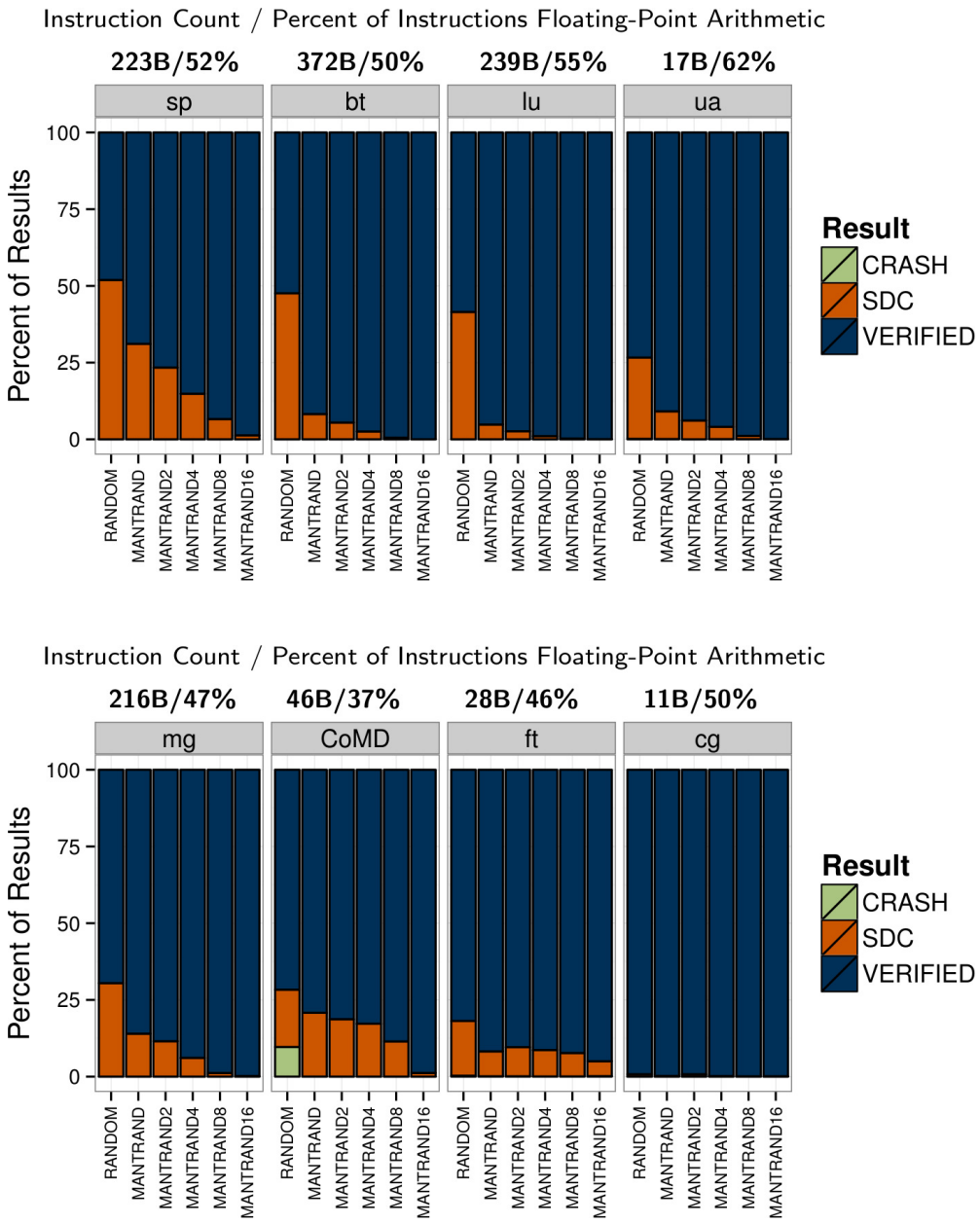


Figure 5.5: The results of an application-level error injection campaign for floating-point multiplication. Ten thousand undetected errors are injected into each of six error detection schemes per program; each error detection scheme is chosen to be representative of a partial duplication or approximate duplication organization. The results of every injected error are classified and tabulated based on their effect on the program output.

This is somewhat representative of approximate duplication, though it does not faithfully model the truncated significand multiplier.

Ten thousand undetected floating-point multiplier errors are injected per error detection scheme for every benchmark; the application-level manifestation of each undetected error is tracked using a Python-based support framework. Each injected undetected error is classified into one of three categories: a *verified* run passes the verification procedure without problem, a *crash* indicates that a segmentation-fault, arithmetic exception, or other critical failure has occurred, and *silent data corruption* (SDC) indicates that the undetected error causes the verification procedure to fail.

Several results are of note. First, almost every program is susceptible to significant levels of silent data corruption unless the floating-point multiplier is protected. Partial duplication reduces the silent data corruption rate across-the-board, but its relative rate of improvement is highly variable. Some programs (such as CoMD) see little improvement due to partial duplication, while others (bt and lu) see more drastic SDC rate reductions. Some programs such as sp and CoMD still see significant rates of silent data corruption despite partial duplication, showing the partial duplication organization to be insufficient for providing high levels of reliability.

In general, approximate duplication shows a clear and consistent trend—higher bounded error detection coverage leads to lower SDC rates,[31] demonstrating the effectiveness of its protection-proportional approach towards error detection. The amount of approximate fidelity required for different benchmarks differs, however—lu responds well to 4 bits of significand protection, many benchmarks respond well to 8 bits of protection, and sp and CoMD seem favorable to 16 bits of protection. Finally, ft seems to require more than 16 bits of error detection coverage to achieve low SDC rates, such that perhaps full duplication is a preferable solution. On the other end of the spectrum, cg is almost completely fault tolerant, and probably requires no FPU error detection. These two extreme benchmarks, ft and cg, motivate the fully-separable design of approximate duplication—due to its separability, the checker can be easily gated and disabled in the case that approximate duplication is unneeded or insufficient.

31 The benchmark ft shows a small deviation from this trend, perhaps due to experimental noise.

CoMD is the only benchmark that has programmer assertions; these are the source of its crashes. Two items of note can be said about the programmer assertions: first, they are insufficient to greatly lower the SDC rate of CoMD. Also, any level of hardware-based error detection does as well as the assertions, rendering them superfluous for the purpose arithmetic error detection.

5.5 DISCUSSION AND FUTURE WORK

This chapter demonstrates the ability of approximate duplication to provide the first known fully-separable error detector for floating-point multiplication (apart from full duplication). Several nuanced design decisions, higher-level support implications, and exciting avenues of future research follow.

5.5.1 The Use of Carry-Save Duplication for Exponent Checking

Carry-save duplication (Section 3) could be employed to perform the dual-output exponent equality check required for approximate duplication. Floating-point multipliers are typically pipelined, however, and it is likely that the cost savings from carry-save duplication would likely be outweighed by the data movement costs of its deferred checking.

5.5.2 The Use of Lazy Carry-Save Duplication for Significand Checking

It is attractive to try and keep the outputs of the truncated significand multiplier in a carry-save format, and to use lazy carry-save duplication (Section 4.2) for the approximate significand equality check. It is not possible, however, to perform magnitude comparison between carry-save inputs without a full carry-propagation [102], such that this representation is not suitable for the approximate equality checks used for the significand-checking logic.

5.5.3 The Use of RNS Duplication for Significand Checking

Another seemingly-attractive error detection scheme may be to apply RNS duplication (Section 4.3) for precise significand checking. RNS duplication, however, requires substantial modification to support truncated multiplication. The truncation operation, while trivial in a weighted fixed-point representation, is

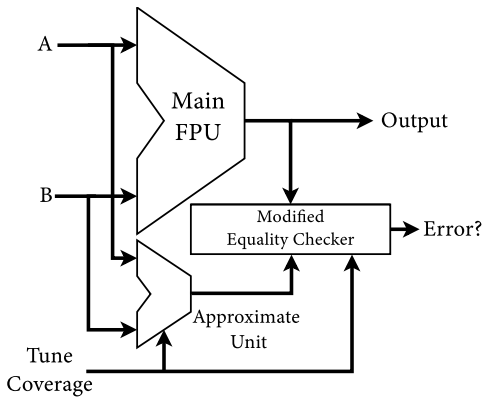


Figure 5.6: A block diagram of flexible approximate duplication. The maximum relative checking error is tunable and can be determined at runtime.

somewhat expensive for an RNS number [109, 120, 121]. Such modification, and an analysis of the best full-coverage low-cost duplicate scheme for floating-point multiplication, is left for future work.

5.5.4 *Correctly Diagnosing Permanent Failures*

The observation from Section 5.3.3 that an error in rounding logic is insignificant does *not* necessarily hold in the case that a permanent fault occurs in the rounding control logic. This is also true of permanent faults in the least-significant bit positions of the significand multiplier—both permanent errors in rounding logic and in the least-significant bits of multiplication will be missed by approximate duplication, and large amounts of error may accumulate over time. Therefore, in order to correctly diagnose permanent failures, the multipliers that are protected by approximate duplication should be periodically tested using higher-level procedures such as full spatial duplication or built-in-self-tests.

5.5.5 *Dynamically Tunable Coverage*

The error semantics of approximate duplication (complete error coverage within a known tolerance) are well-suited to the low-cost, precision-proportional checking of floating-point arithmetic. The error injection results from Section 5.4, however, indicate that no single level of approximate fidelity is appropriate for all applications. This motivates the development of dynamically adaptable duplicate approximation schemes that can incorporate domain-specific sensitivity

knowledge or program analysis for superior error detection efficiency without risking silent data corruption or program failure.

Figure 5.6 illustrates the organization of dynamically tunable approximate duplication; the maximum relative checking error is flexible and can be determined at runtime. Dynamic adaptation of the checking tolerance at runtime may be especially useful for scientific applications, as there is some evidence that they can exhibit both a high sensitivity and a resilience to silent data corruption in the same program [177, 178, 179]. There are compelling iterative approximation and variable-precision multiplication approaches that might form the basis of this dynamic adaption [161, 180, 181, 182]. However, a complete description and evaluation of dynamically tunable approximate duplication is left for future work.

6 SUMMARY & BROADER APPLICABILITY

This dissertation presents and evaluates the concept of low-cost duplication for strong, efficient, fault-agnostic arithmetic error detection. Low-cost duplication employs a redundant arithmetic unit using a specialized number system to check the results of arithmetic. The use of an alternate number representation allows the duplicate checker to potentially be faster and more efficient than the protected arithmetic unit. Because the specialized arithmetic result is discarded after error detection, the downsides of non-standard number systems (such as redundant storage or imprecision accumulation) have no effect on the overall system. The high error coverage and superior speed of the low-cost duplicate organization may lend itself to other domains. Some future possibilities follow.

6.0.6 Security Applications

Low-cost duplication focuses on providing complete protection against single component errors—any feasible arithmetic fault can be detected so long as it is confined to either the main arithmetic unit or the low-cost duplicate checker. This error model represents complete protection against the rare, random, independent faults that typically impact system reliability,[32]. There is some interest in the security community in detecting arbitrary component errors to guard against laser fault injection, where a targeted fault is induced by a nefarious agent [183, 184, 185]. While an analysis of the security potential of low-cost duplication is outside the scope of this study, it seems that the scheme may be a amenable to such applications. This may be especially true for RNS duplication, as the residue number system has a long pedigree of highly fault-tolerant behavior [101, 120]—by adding an additional redundant residue (and further modifying the equality checking circuit) it should be possible to tolerate more than one faulty component.

32 As discussed in Section 2 design faults and timing violations may not be random and independent. Established techniques can be used to handle such faults using low-cost duplication.

6.0.7 *Stochastic or Timing-Speculative Computing*

It may be possible to exploit the benefits of low-cost duplication—strong, separable, low-latency error detection—to operate with unreliable hardware [65, 186] or with reduced timing and voltage margins [80, 81], increasing common-case efficiency while preserving correctness. The strength of low-cost duplication is crucial in such an approach, as it is necessary to guarantee correctness in the presence of errors.[33] Separability is important, as well, to allow for the most optimized and efficient main arithmetic unit to be used. Finally, low-latency error detection is important for this application of error detection, as the arithmetic error rate will be artificially inflated and low-latency detection cheapens fast microarchitectural replay [15].

33 Some prior work investigates the use of residue checking for such a purpose [187], but it cannot guarantee correctness.

REFERENCES

[1] M. B. Sullivan and E. E. Swartzlander, Jr., "Low-cost duplicate multiplication," in *Proceedings of the Symposium on Computer Arithmetic (ARITH)*, 2015.

[2] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, "BulletProof: a defect-tolerant CMP switch architecture," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2006, pp. 5–16.

[3] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent autonomous chip self-test using stored test patterns," in *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*, 2008, pp. 885–890.

[4] Y. Li, O. Mutlu, D. Gardner, and S. Mitra, "Concurrent autonomous self-test for uncore components in system-on-chips," in *Proceedings of the VLSI Test Symposium (VTS)*, 2010, pp. 232–237.

[5] M. Majeed, D. Ahlström, U. Ingelsson, G. Carlsson, and E. Larsson, "Efficient embedding of deterministic test data," in *Proceedings of the Asian Test Symposium (ATS)*, 2010, pp. 159–162.

[6] N. Touba and E. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 783–789, 1997.

[7] M. Nicolaidis, R. Duarte, S. Manich, and J. Figueras, "Fault-secure parity prediction arithmetic operators," *IEEE Design and Test of Computers*, vol. 14, no. 2, pp. 60–71, April–June 1997.

[8] M. Nicolaidis and R. Duarte, "Fault-secure parity prediction Booth multipliers," *IEEE Design and Test of Computers*, vol. 16, no. 3, pp. 90–101, July–September 1999.

[9] M. Nicolaidis, "Carry checking/parity prediction adders and ALUs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, pp. 121–128, 2003.

[10] I. A. Noufal and M. Nicolaidis, "A CAD framework for generating self-checking multipliers based on residue codes," in *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*, 1999, pp. 122–129.

[11] J. Wakerly, "Partially self-checking circuits and their use in performing logical operations," *IEEE Transactions on Computers*, vol. C-23, pp. 658–666, 1974.

[12] A. Pan, J. Tschanz, and S. Kundu, "A low cost scheme for reducing silent data corruption in large arithmetic circuits," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2008, pp. 343–351.

[13] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment Domains: A scalable, efficient, and flexible resilience scheme for Exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[14] S. Tarnick, "Controllable self-checking checkers for conditional concurrent checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 547–553, 1995.

[15] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011, pp. 140–151.

[16] J. Bartlett, J. Gray, and B. Horst, "Fault tolerance in Tandem computer systems," in *The Evolution of Fault-Tolerant Computing*, Vienna, Austria: Springer, 1987, no. 1, pp. 55–76.

[17] M. Mueller, L. Alves, W. Fischer, M. Fair, and I. Modi, "RAS strategy for IBM S/390 G5 and G6," *IBM Journal of Research and Development*, vol. 43, pp. 875–888, 1999.

[18] L. Alves, M. Fair, P. Meaney, C. Chen, W. Clarke, G. Wellwood, N. Weber, I. Modi, B. Tolan, and F. Freier, "RAS design for the IBM eServer z900," *IBM Journal of Research and Development*, vol. 46, pp. 503–521, 2002.

[19] M. Fair, C. Conklin, S. Swaney, P. Meaney, W. Clarke, L. Alves, I. Modi, F. Freier, W. Fischer, and N. Weber, "Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990," *IBM Journal of Research and Development*, vol. 48, pp. 519–534, 2004.

[20] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 12–21.

[21] M. Hsiao, W. Carter, J. Thomas, and W. Stringfellow, "Reliability, Availability, and Serviceability of IBM computer systems: A quarter century of progress," *IBM Journal of Research and Development*, vol. 25, pp. 453–468, 1981.

[22] D. K. Pradhan, Ed., *Fault-Tolerant Computing: Theory and Technique*, Old Tappan, NJ: Prentice Hall Inc., 1986, vol. I.

[23] W. Clarke, L. Alves, T. Dell, H. Elfering, J. Kubala, C. Lin, M. Mueller, and K. Werner, "IBM System z10 design for RAS," *IBM Journal of Research and Development*, vol. 53, pp. 120–130, 2009.

[24] M. Maniatakos, P. Kudva, B. Fleischer, and Y. Makris, "Low-cost concurrent error detection for floating-point unit (FPU) controllers," *IEEE Transactions on Computers*, vol. 62, pp. 1376–1388, 2013.

[25] F. Sellers, M. Hsiao, and L. Bearnson, *Error Detecting Logic for Digital Computers*, New York, NY: McGraw-Hill, 1968.

[26] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1999, pp. 196–207.

[27] M. Hajkazem and A. Baniasadi, "A power-aware alternative for fault-tolerant multipliers," in *Proceedings of the Workshop on Resilient Architectures (WRA)*, 2012. [Online]. Available: http://wra.ece.utexas.edu/WRA2012/Program_files/LFM-Final-Submitted.pdf

[28] W. W. Peterson, "On checking an adder," *IBM Journal of Research and Development*, vol. 2, pp. 166–168, 1958.

[29] A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers*, vol. C-20, pp. 1322–1331, 1971.

[30] U. Sparmann and S. Reddy, "On the effectiveness of residue code checking for parallel two's complement multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, pp. 227–239, 1996.

[31] A. Avizienis and J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, pp. 67–80, August 1984.

[32] E. Krimer, "Improving energy efficiency of reliable massively-parallel architectures," Ph.D. Dissertation, University of Texas at Austin, 2012.

[33] IEEE Technical Committee on Real-Time Systems, "Terminology and Notations," http://tcrts.org/education/terminology-and-notation/, 2014.

[34] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "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 (DSN)*, 2002, pp. 389–398.

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

[36] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2004, pp. 177–186.

[37] D. K. Schroder and J. A. Babcock, "Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing," *Journal of Applied Physics*, vol. 94, pp. 1–18, 2003.

[38] J. W. McPherson, "Reliability challenges for 45nm and beyond," in *Proceedings of the Design Automation Conference (DAC)*, 2006, pp. 176–181.

[39] D. F. Heidel, K. P. Rodbell, E. H. Cannon, C. Cabral, M. S. Gordon, P. Oldiges, and H. H. K. Tang, "Alpha-particle-induced upsets in advanced CMOS circuits and technology," *IBM Journal of Research and Development*, vol. 52, pp. 225–232, 2008.

[40] J. F. Ziegler, "Terrestrial cosmic ray intensities," *IBM Journal of Research and Development*, vol. 42, pp. 117–140, 1998.

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

[42] D. Munteanu and J.-L. Autran, "Modeling and simulation of single-event effects in digital devices and ICs," *IEEE Transactions on Nuclear Science*, vol. 55, pp. 1854–1878, 2008.

[43] T. Uemura, T. Kato, H. Matsuyama, K. Takahisa, M. Fukuda, and K. Hatanaka, "Investigation of multi cell upset in sequential logic and validity of redundancy technique," in *Proceedings of the Online-Testing Symposium (IOLTS)*, 2011, pp. 7–12.

[44] R. Harada, Y. Mitsuyama, M. Hashimoto, and T. Onoye, "Neutron induced single event multiple transients with voltage scaling and body biasing," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2011, pp. 3C.4.1–3C.4.5.

[45] L. Massengill, A. Baranski, D. Van Nort, J. Meng, and B. Bhuva, "Analysis of single-event effects in combinational logic-simulation of the AM2901 bitslice processor," *IEEE Transactions on Nuclear Science*, vol. 47, pp. 2609–2615, 2000.

[46] F. Wang and Y. Xie, "Soft error rate analysis for combinational logic using an accurate electrical masking model," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, pp. 137–146, 2011.

[47] E. Normand, "Single-event effects in avionics," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 461–474, 1996.

[48] T. Heijmen, "Radiation-induced soft errors in digital circuits–a literature survey," Philips Electronics B.V., Tech. Rep. #828, 2002.

[49] E. Cannon, D. Reinhardt, M. Gordon, and P. Makowenskyj, "SRAM SER in 90, 130 and 180 nm bulk and SOI technologies," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2004, pp. 300–304.

[50] F. Wang, Y. Xie, K. Bernstein, and Y. Luo, "Dependability analysis of nano-scale FinFET circuits," in *Proceedings of the Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*, 2006, pp. 399–404.

[51] L. Zeng and P. Beckett, "Soft Error Rate Estimation in Deep Sub-micron CMOS," in *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2007, pp. 210–216.

[52] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2008, pp. 114–122.

[53] M. Gadlage, J. Ahlbin, B. Narasimham, B. Bhuva, L. Massengill, and R. Schrimpf, "Single-event transient measurements in nMOS and pMOS transistors in a 65-nm bulk CMOS technology at elevated temperatures," *IEEE Transactions on Device and Materials Reliability*, vol. 11, pp. 179–186, 2011.

[54] W. Kuo, W. T. K. Chien, and T. Kim, *Reliability, Yield, and Stress Burn-in: A Unified Approach for Microelectronics Systems Manufacturing & Software Development*, Norwell, MA: Klewer Academic Publishers, 1998.

[55] T. Kim and W. Kuo, "Modeling manufacturing yield and reliability," *IEEE Transactions on Semiconductor Manufacturing*, vol. 12, pp. 485–492, 1999.

[56] C. Hu, D. Canaperi, S. Chen, L. Gignac, B. Herbst, S. Kaldor, M. Krishnan, E. Liniger, D. Rath, D. Restaino *et al.*, "Effects of overlayers on electromigration reliability improvement for Cu/low K interconnects," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2004, pp. 222–228.

[57] E. Takeda, R. Izawa, K. Umeda, and R. Nagai, "AC hot-carrier effects in scaled MOS devices," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 1991, pp. 118–122.

[58] J. H. Stathis, "Reliability limits for the gate insulator in CMOS technology," *IBM Journal of Research and Development*, vol. 46, pp. 265–286, 2002.

[59] H. Sharangpani and M. Barton, "Statistical analysis of floating point flaw in the Pentium processor," Intel Corporation, Tech. Rep., 1994. [Online]. Available: http://www.intel.com/support/processors/pentium/sb/cs-013005.htm

[60] "For Intel, it's a case of FPU all over again." *Electronic Engineering Times*, May 1997. [Online]. Available: http://www.fool.com/EETimes/1997/EETimes970516d.htm

[61] "Consumer Price Index (CPI) inflation calculator," Web. [Online]. Available: http://www.bls.gov/data/inflation_calculator.htm

[62] P. Larsson, "Power supply noise in future IC's: A crystal ball reading," in *Proceedings of the Conference on Custom Integrated Circuits*, 1999, pp. 467–474.

[63] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage noise in production processors," *IEEE MICRO*, vol. 31, no. 1, pp. 20–28, January 2011.

[64] V. Reddy, A. T. Krishnan, A. Marshall, J. Rodriguez, S. Natarajan, T. Rost, and S. Krishnan, "Impact of negative bias temperature instability on digital circuit reliability," *Microelectronics Reliability*, vol. 45, no. 1, pp. 31–38, 2005.

[65] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[66] "AMD reports potential heat problem with some Opteron chips," *Information Week*, April 2006. [Online]. Available: http://www.informationweek.com/amd-reports-potential-heat-problem-with-some-opteron-chips/d/d-id/1042705?

[67] R. Perry, "IDDQ testing in CMOS digital ASICs," *Journal of Electronic Testing*, vol. 3, pp. 317–325, 1992.

[68] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Preceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005, pp. 243–254.

[69] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Code Generation and Optimization*, vol. 2, pp. 366–396, 2005.

[70] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, pp. 135–148, 2009.

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

[72] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, New York, NY: Elsevier, 1978.

[73] J. Furuta, K. Kobayashi, and H. Onodera, "Impact of cell distance and well-contact density on neutron-induced multiple cell upsets," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2013, pp. 6C.3.1–6C.3.4.

[74] K. Zhang, J. Furuta, K. Kobayashi, and H. Onodera, "Dependence of cell distance and well-contact density on MCU rates by device simulations and neutron experiments in a 65-nm bulk process," *IEEE Transactions on Nuclear Science*, vol. 61, pp. 1583–1589, 2014.

[75] J. Ahlbin, T. Loveless, D. Ball, B. Bhuva, A. Witulski, L. Massengill, and M. Gadlage, "Double-pulse-single-event transients in combinational logic," in *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2011, pp. 3C.5.1–3C.5.6.

[76] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (Self Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Transactions on Computers*, vol. C-20, pp. 1312–1321, 1971.

[77] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007, pp. 210–222.

[78] P. Ndai, A. Agarwal, Q. Chen, and K. Roy, "A soft error monitor using switching current detection," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2005, pp. 185–190.

[79] A. Narsale and M. Huang, "Variation-tolerant hierarchical voltage monitoring circuit for soft error detection," in *Proceedings of the Symposium on Quality of Electronic Design (ISQED)*, 2009, pp. 799–805.

[80] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw, "RazorII: In situ error detection and correction for PVT and SER tolerance," *IEEE Journal of Solid State Circuits*, vol. 44, pp. 32–48, 2009.

[81] J. Tschanz, K. Bowman, S. Walstra, M. Agostinelli, T. Karnik, and V. De, "Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance," in *Proceedings of the Symposium on VLSI Circuits*, 2009, pp. 112–113.

[82] E. Böhl, T. Lindenkreuz, and R. Stephan, "The fail-stop controller AE11," in *Proceedings of the International Test Conference (ITC)*, 1997, pp. 567–577.

[83] Y. Du, S. Chen, and J. Jianjun, "A layout-level approach to evaluate and mitigate the sensitive areas of multiple SETs in combinational circuits," *IEEE Transactions on Device and Materials Reliability*, vol. 14, pp. 213–219, 2014.

[84] B. Kiddie and W. Robinson, "Alternative standard cell placement strategies for single-event multiple-transient mitigation," in *Proceedings of the IEEE Symposium on VLSI (ISVLSI)*, 2014, pp. 589–594.

[85] D. Lardner, "Babbage's calculating engine," *Edinburgh Review*, vol. 59, pp. 263–327, 1834.

[86] S. Mitra and E. McCluskey, "Which concurrent error detection scheme to choose?" in *IEEE International Test Conference (ITC)*, 2000, pp. 985–994.

[87] M. B. Sullivan and E. E. Swartzlander, Jr., "On separable error detection for addition," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2013, pp. 2181–2186.

[88] C. Davies, Jr., "Data processing spheres of control," *IBM Systems Journal*, vol. 17, pp. 179–198, 1978.

[89] "Computing Community Consortium (CCC) Visioning study on cross-layer reliability: System-level, cross-layer cooperation to achieve predictable systems from unpredictable components." [Online]. Available: http://www.relxlayer.org/

[90] Synopsys Inc., "Design Compiler I-2013.12-SP5-2."

[91] Taiwan Semiconductor Manufacturing Company, "40nm CMOS Standard Cell Library v120b," 2009.

[92] S. Iacobovici, "End-to-end residue based protection of an execution pipeline," U.S. Patent #7 555 692 B1, 2009. [Online]. Available: http://www.google.com/patents/US7555692

[93] ——, "End-to-end residue-based protection of an execution pipeline that supports floating point operations," U.S. Patent #7 769 795 B1, 2010. [Online]. Available: http://www.google.com/patents/US7769795

[94] H. Naeimi, "An end-to-end ECC-based resiliency approach for microprocessors," in *Proceedings of the Workshop on Silicon Errors in Logic–System Effects (SELSE)*, 2011. [Online]. Available: http://softerrors.info/selse/images/selse_2011/Papers/selse7_submission_16.pdf

[95] T. R. N. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on Computers*, vol. C-19, pp. 398–402, 1970.

[96] J. L. Massey and O. N. García, "Error-correcting codes in computer arithmetic," in *Advances in Information Systems Science*, New York, NY: Plenum Press, 1972, pp. 273–326.

[97] J. M. Tahir, S. S. Dlay, R. N. G. Naguib, and O. R. Hinton, "Fault tolerant arithmetic unit using duplication and residue codes," *Integration, the VLSI Journal*, vol. 18, pp. 187–200, 1995.

[98] E. E. Swartzlander, Jr., "Fault-tolerant arithmetic via time-shared TMR," in *SPIE's International Symposium on Optical Science, Engineering, and Instrumentation*, vol. 3807, 1999, pp. 84–92.

[99] T. Ngai, C. He, and E. E.. Swartzlander, Jr., "Enhanced concurrent error correcting arithmetic unit design using alternating logic," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2001, pp. 78–83.

[100] M. B. Sullivan and E. E. Swartzlander, Jr., "Long residue checking for adders," in *Proceedings of the Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2012, pp. 177–180.

[101] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY: Oxford University Press, 2010.

[102] P. Kornerup and D. W. Matula, *Finite Precision Number Systems and Arithmetic*, Cambridge, UK: Cambridge University Press, 2010.

[103] J. Cortadella and J. Llaberia, "Evaluation of A+B=K conditions without carry propagation," *IEEE Transactions on Computers*, vol. 41, pp. 1484–1488, 1992.

[104] W. L. Lynch and G. R. Lauterbach, "Low-latency memory indexing method and structure," U.S. Patent #5 754 819 A, May, 1998. [Online]. Available: http://www.google.com/patents/US5754819

[105] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[106] M. Yilmaz, A. Meixner, S. Ozev, and D. Sorin, "Lazy error detection for microprocessor functional units," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2007, pp. 361–369.

[107] M. Alioto and G. Palumbo, "Analysis and comparison on full adder block in submicron technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, pp. 806–823, 2002.

[108] Nangate, "Open Cell Library v1.3," 2009.

[109] P. V. Mohan, *Residue Number Systems: Algorithms and Architectures*, Norwell, MA: Kluwer Academic Publishers, 2002.

[110] A. J. Martin, "Towards an energy complexity of computation," *Information Processing Letters*, vol. 77, pp. 181–187, 2001.

[111] R. Zimmermann, "Binary adder architectures for cell-based VLSI and their synthesis," Ph.D. Dissertation, Swiss Federal Institute of Technology (ETH), 1998.

[112] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, pp. 260–264, 1982.

[113] Synopsys Inc., "Designware IP library." [Online]. Available: http://www.synopsys.com/products/designware/

[114] R. Zimmermann, "Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication," in *Proceedings of the IEEE Symposium on Computer Arithmetic*, 1999, pp. 158–167.

[115] T. R. Rao and O. Garcia, "Cyclic and multiresidue codes for arithmetic operations," *IEEE Transactions on Information Theory*, vol. 17, pp. 85–91, 1971.

[116] G. Gaubatz, B. Sunar, and M. G. Karpovsky, "Non-linear residue codes for robust public-key arithmetic," in *Fault Diagnosis and Tolerance in Cryptography*, Berlin/Heidelberg, Germany: Springer, 2006, no. 4236, pp. 173–184.

[117] A. H. Syed, "Performance of different multipliers in the DesignWare building block IP," Synopsys Inc. [Online]. Available: http://www.synopsys.com/dw/dwtb.php?a=multiplier_bldg_block

[118] A. Karatsuba, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[119] H. L. Garner, "The residue number system," *IEEE Transactions on Electronic Computers*, vol. EC-8, pp. 140–147, 1959.

[120] N. S. Szabó and R. I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*, New York, NY: McGraw-Hill, 1967.

[121] A. Omondi and B. Premkumar, *Residue Number Systems*, Singapore: World Scientific, 2007.

[122] C. Efstathiou, N. Moschopoulos, K. Tsoumanis, and K. Pekmestzi, "On the design of configurable modulo $2n \pm 1$ residue generators," in *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, 2012, pp. 50–56.

[123] C. Efstathiou, H. T. Vergos, G. Dimitrakopoulos, and D. Nikolos, "Efficient diminished-1 modulo 2^n+1 multipliers," *IEEE Transactions on Computers*, vol. 54, pp. 491–496, 2005.

[124] B. Parhami, "On equivalences and fair comparisons among residue number systems with special moduli," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2010, pp. 1690–1694.

[125] W. Freking and K. Parhi, "Low-power FIR digital filters using residue arithmetic," in *Proceedings of the Asilomar Conference on Signals and Systems*, 1997, pp. 739–743.

[126] P. Ananda Mohan and A. Premkumar, "RNS-to-binary converters for two four-moduli sets $\{2^n-1, 2^n, 2^n+1, 2^{n+1}-1\}$ and $\{2^n-1, 2^n, 2^n+1, 2^{n+1}+1\}$," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, pp. 1245–1254, 2007.

[127] B. Cao, C.-H. Chang, and T. Srikanthan, "A residue-to-binary converter for a new five-moduli set," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, pp. 1041–1049, 2007.

[128] A. S. Molahosseini, C. Dadkhah, and K. Navi, "A new five-moduli set for efficient hardware implementation of the reverse converter," *IEICE Electronics Express*, vol. 6, pp. 1006–1012, 2009.

[129] B. Parhami, "RNS representations with redundant residues," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2001, pp. 1651–1655.

[130] M. Abdallah and A. Skavantzos, "On the binary quadratic residue system with noncoprime moduli," *IEEE Transactions on Signal Processing*, vol. 45, pp. 2085–2091, 1997.

[131] A. Skavantzos and M. Abdallah, "Implementation issues of the two-level residue number system with pairs of conjugate moduli," *IEEE Transactions on Signal Processing*, vol. 47, pp. 826–838, 1999.

[132] E. E. Swartzlander, Jr. and A. G. Alexopoulos, "The sign/logarithm number system," *IEEE Transactions on Computers*, vol. 24, pp. 1238–1242, 1975.

[133] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, pp. 512–517, 1962.

[134] A. S. Fraenkel, "The use of index calculus and Mersenne primes for the design of a high-speed digital multiplier," *Journal of the ACM*, vol. 8, pp. 87–96, 1961.

[135] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[136] D. Lutz and N. Burgess, "Overcoming double-rounding errors under IEEE 754-2008 using software," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2010, pp. 1399–1401.

[137] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, pp. 5–48, 1991.

[138] J. C. Lo, "Reliable floating-point arithmetic algorithms for Berger encoded operands," in *Proceedings of the International Conference on Computer Design (ICCD)*, 1992, pp. 110–113.

[139] ——, "Reliable floating-point arithmetic algorithms for error-coded operands," *IEEE Transactions on Computers*, vol. 43, pp. 400–412, 1994.

[140] M. Maniatakos, Y. Makris, P. Kudva, and B. Fleischer, "Exponent monitoring for low-cost concurrent error detection in FPU control logic," in *Proceedings of the VLSI Test Symposium (VTS)*, 2011, pp. 235–240.

[141] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for Probabilistic Applications," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, 2010, pp. 546–558.

[142] P. Eibl, A. Cook, and D. Sorin, "Reduced Precision Checking for a Floating Point Adder," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2009, pp. 145–152.

[143] R. A. Davis, "A checking arithmetic unit," in *Proceedings of the Fall Joint Computer Conference*, 1965, pp. 705–713.

[144] M. Combet, H. Van Zonneveld, and L. Verbeek, "Computation of the base two logarithm of binary numbers," *IEEE Transactions on Computers*, vol. EC-14, pp. 863–867, 1965.

[145] A. Drozd, R. Al-Azzeh, J. Drozd, and M. Lobachev, "The logarithmic checking method for on-line testing of computing circuits for processing of the approximated data," in *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, 2004, pp. 416–423.

[146] A. Uhl and J. Becker, "Concurrent error detection in multipliers by using reduced wordlength multiplication and logarithms," in *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, 2013, pp. 129–135.

[147] J. M. Jou and S. R. Kuang, "Design of low-error fixed-width multiplier for DSP applications," *Electronics Letters*, vol. 33, no. 19, pp. 1597–1598, 1997.

[148] M. Schulte and E. E. Swartzlander, Jr., "Truncated multiplication with correction constant [for DSP]," in *Proceedings of the Workshop on VLSI Signal Processing*, 1993, pp. 388–396.

[149] S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *IEEE Transactions on Circuits and Systems*, vol. 43, pp. 90–95, 1996.

[150] E. King and E. E. Swartzlander, Jr., "Data-dependent truncation scheme for parallel multipliers," in *Proceedings of the Asilomar Conference on Signals and Systems*, 1997, pp. 1178–1182.

[151] M. de la Guia Solaz and R. Conway, "Comparative study on wordlength reduction and truncation for low power multipliers," in *Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics*, 2010, pp. 84–88.

[152] N. Petra, D. De Caro, V. Garofalo, E. Napoli, and A. Strollo, "Truncated binary multipliers with variable correction and minimum mean square error," *IEEE Transactions on Circuits and Systems*, vol. 57, pp. 1312–1325, 2010.

[153] K. Wires, M. Schulte, and J. Stine, "Variable-correction truncated floating point multipliers," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2000, pp. 1344–1348.

[154] N. Burgess, "Flagged prefix adder for dual additions," in *Proceedings of SPIE's International Symposium on Optical Science, Engineering, and Instrumentation*, 1998, pp. 567–575.

[155] S. Knowles, "Addition circuitry for calculating a sum and the sum plus one," United Kingdom Patent #2 342 193 A, 2000.

[156] ——, "Circuitry for performing operations on binary numbers," U.S. Patent #6 446 107 B1, 2002.

[157] ——, "Addition Circuitry," United Kingdom Patent #2 342 193 B, 2003.

[158] E. Hall, D. Lynch, and S. Dwyer, "Generation of products and quotients using approximate binary logarithms for digital filtering applications," *IEEE Transactions on Computers*, vol. C-19, pp. 97–105, 1970.

[159] S. SanGregory, C. Brothers, D. Gallagher, and R. Siferd, "A fast, low-power logarithm approximation with CMOS VLSI implementation," in *Proceedings of the Midwest Symposium on Circuits and Systems*, vol. 1, 1999, pp. 388–391.

[160] K. Abed and R. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter," *IEEE Transactions on Computers*, vol. 52, pp. 1421–1433, 2003.

[161] M. B. Sullivan and E. E. Swartzlander, Jr., "Truncated error correction for flexible approximate multiplication," in *Proceedings of the Asilomar Conference on Signals and Systems*, 2012, pp. 355–359.

[162] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Programming Languages Design and Implementation (PLDI)*, 2005, pp. 190–200.

[163] E. Czeck and D. Siewiorek, "Effects of transient gate-level faults on program behavior," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1990, pp. 236–243.

[164] J. Boue, P. Petillon, and Y. Crouzet, "MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1998, pp. 168–173.

[165] A. Bosio and G. Di Natale, "LIFTING: A flexible open-source fault simulator," in *Proceedings of the Asian Test Symposium (ATS)*, 2008, pp. 35–40.

[166] Z. Kalbarczyk, R. Iyer, G. Ries, J. Patel, M. Lee, and Y. Xiao, "Hierarchical simulation approach to accurate fault modeling for system dependability evaluation," *IEEE Transactions on Software Engineering*, vol. 25, pp. 619–632, 1999.

[167] W. Chao, F. Zhongchuan, C. Hongsong, and C. Gang, "FSFI: A full system simulator-based fault injection tool," in *Proceedings of the International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC)*, 2011, pp. 326–329.

[168] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 265–276.

[169] M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh, "Fault injection tools based on virtual machines," in *Proceedings of the International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014, pp. 1–6.

[170] J. Guthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1995, pp. 196–206.

[171] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 27:1–27:10.

[172] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop FPGA-based fault injection technique," *Microelectronics Reliability*, vol. 54, pp. 1000–1008, 2014.

[173] Raghuraman Balasubramanian and Karthikeyan Sankaralingam, "Understanding the impact of gate-level physical reliability effects on whole program execution," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 60–71.

[174] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 57:1–57:11.

[175] "The NAS Parallel Benchmarks." [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[176] "Classical molecular dynamics (CoMD) proxy application," 2012. [Online]. Available: http://www.exmatex.org/comd.html

[177] M. A. Heroux and M. Hoemmen, "Fault-tolerant iterative methods via selective reliability," Sandia National Laboratories, Tech. Rep., June 2011.

[178] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the International Supercomputing Conference (ISC)*, June 2008, pp. 155–164.

[179] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the International Supercomputing Conference (ISC)*, 2011, pp. 152–161.

[180] Z. Babić, A. Avramović, and P. Bulić, "An iterative logarithmic multiplier," *Microprocessors and Microsystems*, vol. 35, pp. 23–33, 2011.

[181] S.-R. Kuang, K.-Y. Wu, and K.-K. Yu, "Energy-efficient multiple-precision floating-point multiplier for embedded applications," *Journal of Signal Processing Systems*, vol. 72, pp. 43–55, 2012.

[182] H. Zhang, W. Zhang, and J. Lach, "A low-power accuracy-configurable floating point multiplier," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2014, pp. 48–54.

[183] E. Trichina and R. Korkikyan, "Multi fault laser attacks on protected CRT-RSA," in *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2010, pp. 75–86.

[184] Z. Wang, M. Karpovsky, and A. Joshi, "Secure multipliers resilient to strong fault-injection attacks using multilinear arithmetic codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, pp. 1036–1048, 2012.

[185] D. Karaklajic, J. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 2295–2306, 2013.

[186] K. Palem, "Energy aware computing through probabilistic switching: a study of limits," *IEEE Transactions on Computers*, vol. 54, pp. 1123–1137, 2005.

[187] M. Neagu, G. Mois, and L. Miclea, "On-line error detection for tuning dynamic frequency scaling," in *Proceedings of the International Conference on Automation Quality and Testing Robotics (AQTR)*, 2012, pp. 290–295.