ERROR CORRECTION IN FLOATING POINT UNITS USING INFORMATION REDUNDANCY

by

Shehab Yomn Abdellatif Elsayed

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

ERROR CORRECTION IN FLOATING POINT UNITS USING INFORMATION REDUNDANCY

by

Shehab Yomn Abdellatif Elsayed

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Under the Supervision of

Assoc. Prof. Hossam A. H. Fahmy
Principal Advisor

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

ERROR CORRECTION IN FLOATING POINT UNITS USING INFORMATION REDUNDANCY

by

Shehab Yomn Abdellatif Elsayed

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Approved by the
Examining Committee

Assoc. Prof. Hossam A. H. Fahmy, Thesis Main Advisor

Prof. Dr. Mohammed Zaki Abd-El-Mageed, Member

Prof. Dr. Magdy M. S. Elsoudani, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

Acknowledgments

First of all, I would like to thank my advisor, Dr. Hossam A. H. Fahmy, whose guidance and directions helped me improve in all aspects of my life and not just the academic one.

I would also like to thank all my friends and colleagues for their great help and support throughout the journey of my Master's degree.

Last but not least, I would like to thank my parents, brother and my whole family for their continuous help and support throughout every single moment of my life.

Abstract

With the continuous shrinking of electronic devices, they become more prone to faults. For a reliable system, the errors caused by these faults can not be ignored. Therefore, the presence of fault tolerance techniques in modern processors and arithmetic circuits is vital for the reliability of these systems. This work represents an attempt using information redundancy to add fault tolerance capabilities to a combined IEEE decimal-64/binary-64 floating point adder.

Several techniques have been devised to achieve fault tolerance in current decimal/binary arithmetic circuits using time redundancy, hardware redundancy or both. Information redundancy in the form of residue codes was also used to achieve error detection in floating point units. Meanwhile, a lot of research is being conducted in designing arithmetic circuits which adopt the Residue Number System RNS instead of the Weighted Number System WNS to make use of its carry free operations and fault tolerant properties.

In the proposed technique, Residue codes are used for error detection and correction. Meanwhile, the same checker is used on different parts of the result to decrease the area overhead of the correction circuit. The technique depends on calculating the residues of the operands to the arithmetic circuit, performing the arithmetic operation on the residues as well as the operands and finally calculating the syndrome. Through the proper choice of the moduli, it can be guaranteed that each error has a unique syndrome pattern. Therefore, knowing the syndrome pattern, the corresponding error can be determined and hence the result can be corrected.

To our knowledge, this is the first implementation of a residue error correction scheme in decimal and binary arithmetic circuits. The proposed method is able to correct any 4-digit error in the 4-digit numbers being checked assuming that errors occur only in the main adder. The 4-digit checking process is repeated until all digits of the result of the addition process are checked.

The design has been synthesized using TSMC 65 nm LP technology. The proposed error correction process can be divided into two main stages based on whether it depends on the result of the main adder or not. For the 4-digit checker, the first stage was found to occupy an area of 2396.16 μm^2 and has a delay of 1.98 nsec. The second stage was found to occupy an area of 4206.24 μm^2 and has a delay of 5.17 nsec.

The proposed technique has great error detection and correction capabilities but the large checker area (compared to the original system being checked) and the long delay introduced by the checker represent a great challenge to efficiently use information redundancy for error correction. Therefore, it has to be wisely integrated within the floating point adder in specific and within the processor as a whole. Moreover, the synthesis results do not support the assumption that errors occur in the main adder only. Consequently, further work should be done in order to provide the proposed design with the ability to correct an error whether it occurred in the main circuit or the checker circuit.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past few decades, the electronics industry has greatly evolved due to the continuous shrinking of available technologies. Feature size of semiconductor devices has decreased from 165 nm in 2000 to 22 nm in 2011 [1]. This lead to a large increase in transistor density from 27.8 to 1701 Mtransistors/cm^2 [1] which was also accompanied by a large increase in power density due to the inconsistent voltage scaling with the dimension scaling [2]. As a result, scaling has increased the effect of many phenomena which tend to cause an erroneous operation of electronic circuits and may eventually cause the circuit to fail [3–5].

In this thesis, fault tolerance in arithmetic circuits is discussed. A scheme using residue codes is proposed for error detection and correction. The proposed technique is applied to the Redundant Decimal/Binary Floating Point Adder designed in [6].

In this introductory chapter, some of the basic definitions in the topic of fault tolerance are explained in section 1.1. In section 1.2, some of the main failure mechanisms in integrated circuits are discussed with an emphasis on the effect of scaling on each of these failure mechanisms. The main techniques used to achieve fault tolerance are explained in section 1.3.

1.1 Basic Definitions

Reliability	A system is said to be reliable if it is able to perform its task meeting the required specifications. Several metrics have been developed to evaluate the reliability of a certain system [7, 8] such as: Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF), Mean Time To Repair (MTTR), *n%* Time To Failure (*n*TTF) [8] and Failures In Time (FIT) [3, 4].
Fault	A fault can be defined as a deviation of a signal from its desired value. This might be due to external factors as *electromagnetic radiation* or it might be caused by a defective component of the system.
Error	A fault, if not detected and dealt with properly, might affect other data in the system causing the system to produce erroneous results. An error can be defined as the difference between the computed value and the correct one.
Failure	Errors might cause a system to fail. A system fails if it can no longer carry out its task subject to the required specifications.

The above explanation is based on the *Multilevel Model of Reliability* explained in [9]. In this model, a system can be in any of the following states; *ideal, defective, faulty, erroneous, malfunctioning, degraded* and *failure*. The system can be initially in any of those states and it can go from one state to another depending on its design as shown in figure 1.1.

1.2 Why Is Fault Tolerance Needed?

As explained earlier, with the continuous shrinking of electronic devices, they become more prone to faults. Faults in electronic circuits can be categorized into [10]:

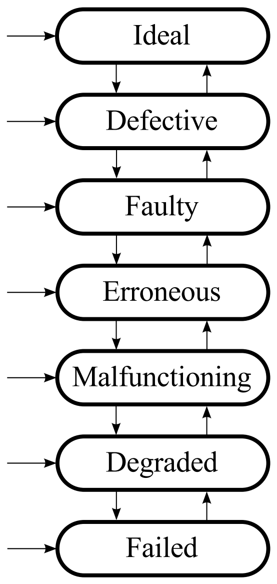


Figure 1.1: Mutilevel reliability model explained in [9]

Hard Errors	Also known as *permanent* errors. They result from component failure or malfunction. Once a circuit is affected with this type of error it can not be removed, but it can be treated in a way that allows the circuit to continue functioning properly.
Soft Errors	Also known as *transient* errors and *temporary* errors. They result from exposure of electronic circuits to external factors and sources of interference such as electromagnetic radiation. They only exist as long as the source of interference exists.

For a reliable system, the presence of these errors and their negative effects on performance is not acceptable. Therefore, the presence of fault tolerance techniques in modern processors in general and arithmetic circuits in specific is vital for the reliability of these systems. This thesis is mainly concerned with adding fault tolerance to arithmetic circuits and particularly to the floating point adder presented in [6].

1.2.1 Hard Errors

As mentioned earlier, hard errors refer to malfunction of the electronic device itself. For example, an interconnect which has turned into an open circuit due to electromigration* or a transistor which has suffered from breakdown. Once

*will be explained later

the device has suffered form that kind of errors, it can not go back to normal operation. However, these errors can be treated in a way that allows the whole system to function properly although some devices might have failed. In [3], a model called RAMP model was presented to study the reliability of microprocessors from an architectural perspective. It studies the reliability of the processor as a whole and not the reliability of single devices taking into consideration the effects of *Electromigration, Time Dependent Dielectric Breakdown (TDDB), Stress Migration* and *Thermal Cycling*. The RAMP model presented in [3] provides reliability measures for a certain technology node under different workloads. This model was extended in [4] to take into account the effect of scaling. Scaling affects the previous factors in three ways:

1. Change in temperature
2. Dimension scaling
3. Voltage Scaling

In [3], only the first effect was taken into consideration while in [4] the model was extended to account for the second and third effects. The base processor simulated in [4] is a 180 nm out-of-order 8-way superscalar processor and the results are reported for 16 traces of SPEC2K benchmarks (8 SpecFP and 8 SpecInt). This was used to study the effect of different workloads on the reliability of the processor. The effect of the different failure mechanisms on reliability is determined through studying its effect on MTTF and FIT (the number of failures seen per 10^9 device hours).

Electromigration (EM)

Electromigration [3–5, 11] refers to the displacement of the atoms of the metal of the interconnects due to the interaction between these atoms and the electron flow in the interconnect. As a result, metal atoms are depleted from some regions of the interconnect, called *voids*, resulting in high resistance or open circuit while they pile up in other regions, called *hillocks*, engendering the risk of short circuit. EM is mainly affected by the increase in temperature and the shrinking in interconnect

Table 1.1: Summary of the effects of scaling on EM, TDDB, SM and TC based on the RAMP model [4]. $E_{a_{EM}}$ is the activation energy for electromigration, κ is Boltzmann's constant, T is the absolute temperature in kelvin, T_0 is the stress free temperature of the metal, m and $E_{a_{SM}}$ are material dependent, a, b, X, Y and Z are fitting parameters, V is the voltage, q is the Coffin-Manson exponent (empirically determined constant that depends on the material), w and h are the width and height of the interconnect respectively and Δt_{ox} is the reduction in the gate oxide thickness

Failure Mechanism	**Major temperature dependence**	**Voltage dependence**	**Feature size dependence**
EM	$e^{\frac{E a_{EM}}{\kappa T}}$	NA	wh
TDDB	$e^{\frac{X+\frac{Y}{T}+ZT}{\kappa T}}$	$\left(\frac{1}{V}\right)^{(a-bT)}$	$10^{\frac{\Delta t_{ox}}{0.22}}$
SM	$\lvert T-T_0\rvert^{-m} e^{\frac{E a_{SM}}{\kappa T}}$	NA	NA
TC	$\frac{1}{T^q}$	NA	NA

dimensions when advancing from one technology node to the other [4]. These effects are quantitatively mentioned in table 1.1.

It was shown in [4] that the failure rate due to EM increases by an average of 303% and 447% for SpecFP and SpecInt respectively when migrating from 180 nm technology to 65 nm technology using a voltage of 1 volt. If the voltage used in the 65 nm technology is 0.9 volt, the increase in failure rates becomes 97% for SpecFP and 128% for SpecInt. This indirect dependence on voltage comes from the fact that voltage has a large effect on temperature which in turn affects electromigration according to table 1.1.

Time Dependent Dielectric Breakdown (TDDB)

Time Dependent Dielectric Breakdown (also known as Gate Oxide Breakdown) [3, 4, 12] is due to the generation of very strong electric field in the gate oxide. This is due to the continuous shrinking of the gate oxide thickness while the applied voltage is not scaled correspondingly. As a result a conductive path may be formed in the gate oxide allowing for a gate leakage current to flow and affecting the operation of the device. TDDB is mainly affected by the rise in temperature and the inconsistent voltage scaling with the shrinking of the gate oxide thickness when advancing from one technology node to the other. The dependence of TDDB on each of temperature, voltage and feature size scaling is stated in table 1.1.

It was shown in [4] that the failure rate due to TDDB increases by an average of 667% and 812% for SpecFP and SpecInt respectively when migrating from 180 nm technology to 65 nm using a voltage of 1 volt. If the voltage used in the 65 nm technology is 0.9 volt, the increase in the failure rate becomes 106% for SpecFP and 127% for SpecInt.

Stress Migration (SM)

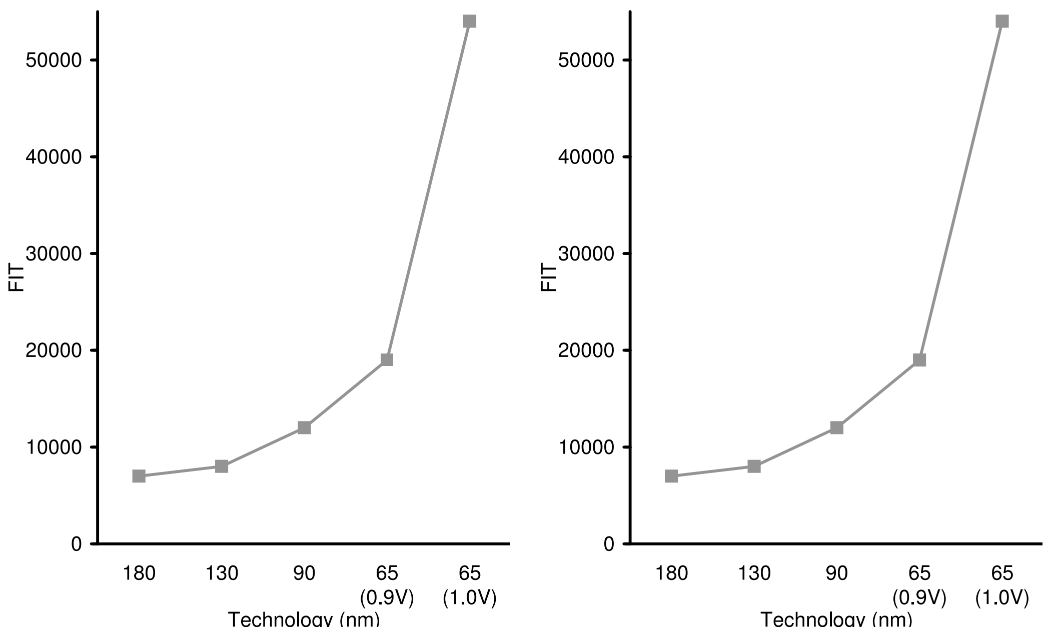
Stress Migration [3, 4] is the displacement of metal atoms because of thermo-mechanical stress which results from the difference in the thermal expansion rates of the different materials in the device. SM is mainly affected by the rise in temperature when advancing from one technology node to the other [4] as mentioned in table 1.1.

It was shown in [4] that the failure rate due to SM increases by an average of 76% and 106% for SpecFP and SpecInt respectively when migrating from 180 nm technology to 65 nm technology using a voltage of 1 volt. If the voltage used in the 65 nm technology is 0.9 volt, the increase in the failure rate becomes 43% for SpecFP and 52% for SpecInt.

Thermal Cycling (TC)

Thermal cycling [3, 4] refers to the permanent damage accumulating in the system every thermal cycle. This damage is most profound in the package and die interface. TC is mainly affected by the increase in temperature when advancing from one technology node to the other as shown in table 1.1.

It was shown in [4] that the failure rate due to TC increases by an average of 52% and 66% for SpecFP and SpecInt respectively when migrating from 180 nm technology to 65 nm technology using a voltage of 1 volt. If the voltage used in the 65 nm technology is 0.9 volt, the increase in the failure rate becomes 32% for SpecFP and 36% for SpecInt.



(a) SpecFP

(b) SpecInt

Figure 1.2: Combined failure rates for EM, TDDB, SM and TC under worst case scenario

Overall Effect

The effect of scaling on each of the above mentioned failure mechanisms is summarized in table 1.1. Figure 1.2 shows the total effect of these mechanisms on the reliability of the processor* being studied under worst case scenario combining the worst case conditions from all the workloads† considered in the experiments in [4]. Moreover, the average increase in failure rate due to the the above mentioned mechanisms as the technology advances from 180 nm to 65 nm with either a 1 volt supply voltage or a 0.9 volt supply voltage is summarized in table 1.2. Taking into consideration all of the previous failure mechanisms [4], the FIT values for the processor studied increased by an average of 274% and 357% for SpecFP and SpecInt respectively when migrating from 180 nm technology to 65 nm technology using a voltage of 1 volt. If the voltage used in the 65 nm technology is 0.9 volt, the increase in the FIT values becomes 70% for SpecFP and 86% for SpecInt.

*As mentioned earlier, the base processor simulated in [4] is a 180 nm out-of-order 8-way superscalar processor

†As mentioned earlier, results were reported for 8 SpecFP and 8 SpecInt workloads

Table 1.2: Summary of the average increase in failure rate due to EM, TDDB, SM and TC as the technology advances from 180 nm to 65 nm with either a 1 volt supply voltage or a 0.9 volt supply voltage for 8 SpecFP and 8 SpecInt workloads

	1 volt supply voltage		**0.9 volt supply voltage**	
	SpecFP	**SpecInt**	**SpecFP**	**SpecInt**
EM	303%	447%	97%	128%
TDDB	667%	812%	106%	127%
SM	76%	106%	43%	52%
TC	52%	66%	32%	36%
Total Effect	274%	357%	70%	86%

1.2.2 Soft Errors

As mentioned earlier, soft errors are temporary. They exist as long as their causes exist resulting in a temporary faulty operation of the electronic devices affected by them. One of the main causes of soft errors is the interaction between energetic particles and the semiconductor devices [13–16]. In [14] three types are mentioned as examples of these energetic particles. These types can be summarized as:

1. Alpha particles emitted by impurities in the packaging material of the integrated circuits
2. High energy neutrons from cosmic radiation
3. Low energy cosmic neutrons which interact with the isotope boron-10 used in the insulating layers in the integrated circuit industry.

When a semiconductor device is exposed to such particles, a charge, Q_{coll}, may be collected in the device. If this charge exceeds a certain limit, Q_{crit}, the device may suffer from faulty operation [15]. These errors can also be a result of electrical noise and other environmental effects [17]. Probabilistic models for the occurrence of soft errors have been presented in [16, 17] but they are out of the scope of this thesis.

Q_{coll} is proportional to the area and collection efficiency of the semiconductor device [13]. Therefore, as technology is down-scaled the ability of the device to collect charges decreases. This decrease is counteracted by a decrease in Q_{crit}

which is proportional to the supply voltage and node capacitance [13]. As a result, the *Soft Error Rate (SER)* of a single device is almost unchanged or slightly decreases as the technology is down-scaled [13]. This almost constant behavior of the SER along with the great increase in the transistor density form one technology node to another lead to an increase in the SER of the system as a whole.

The impact of exposure to the before mentioned particles on the system reliability depends on the nature of the circuits being affected whether they are RAM circuits, sequential or combinational devices [13, 15]. RAMs are prone to soft errors at any time of operation while latches in a sequential circuit will only affect the reliability of the system if it suffered from a soft error while it is storing data. Likewise, a combinational circuit would affect the system reliability if its soft error was captured by its successive latch. The probability of this happening depends on three factors [13, 15] often referred to as three types of *masking*:

1. **Electrical Masking**
   Concerned with the strength of the erroneous signal when it reaches the latch, whether it is still strong enough to be stored in it or not.

2. **Latch-Window Masking**
   Concerned with the time at which the erroneous signal reaches the latch, whether it is a suitable time for storage or not.

3. **Logical Masking**
   Concerned with the role played by the erroneous signal in determining the outcome of the logical operation

Due to these masking mechanisms, the combinational circuits are generally less affected by soft errors than memory elements [15, 18, 19]. However, these three masking mechanisms become less effective as the manufacturing technology is down-scaled [15] and the SER increases.

Figure 1.3 shows the results of an experiment conducted in [15]. In this experiment, the SER was calculated for a chip model based on the Alpha 21264 microprocessor which was designed for 350 nm process with 15.2 Mtransistors on the die. SER was calculated taking the effect of only high energy neutrons (Energy$>$1 MeV) and neglecting the effect of logical masking. Moreover,

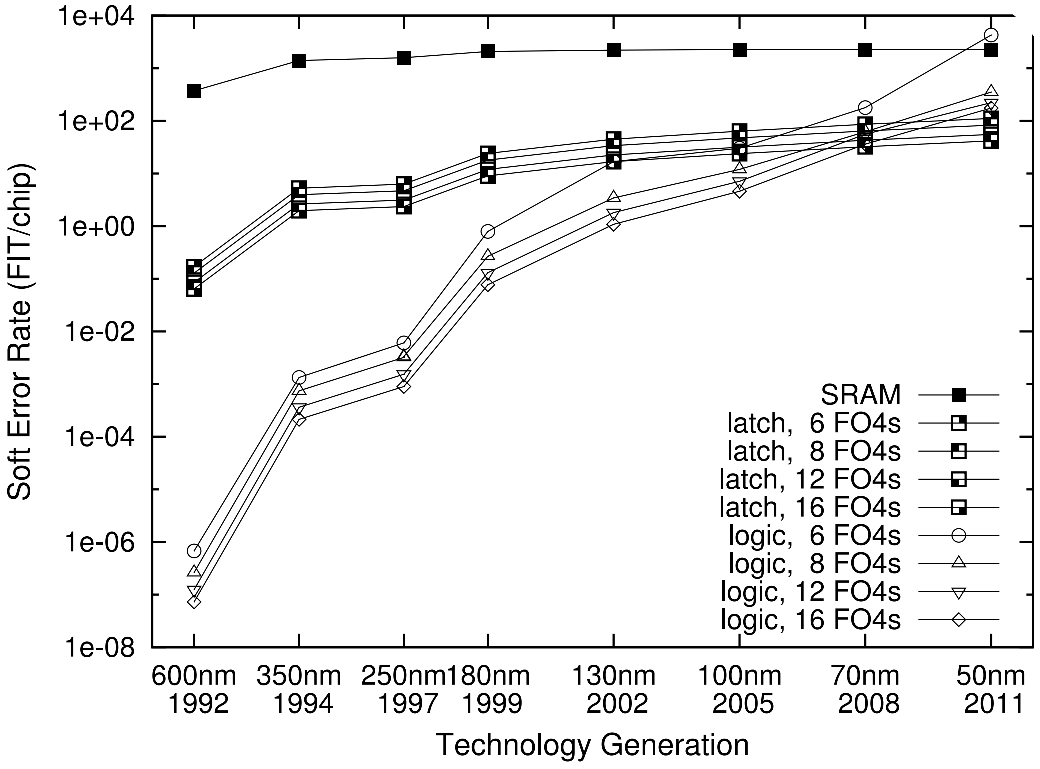


Figure 1.3: SER per chip for SRAMs, latches and combinational logic as mentioned in [15]

the experiment was carried out taking different depths of the processor pipeline into consideration. Generally, decreasing the pipeline stage delay requires more latches and less time for the signal to reach its designated latch from the combinational circuit that produced it. As a result, as the pipeline stage delay decreases, the SER increases as can be seen from figure 1.3. Given the trend shown in the figure, the SER in the highest performance logic circuits is already surpassing SER in memories and other logic circuits may surpass memories in the near future. Fault tolerance in such circuits in thus now a critical issue and this is the motivation of this thesis.

The soft error problem of semiconductor devices can be tackled at different levels [13]. Different technologies, like *Silicon on Insulator (SOI)*, can be used instead of traditional Bulk Substrate CMOS technologies. Devices fabricated by the SOI technology have less ability to collect charges than their conventional CMOS counterparts [13]. Another method is to tune the device parameters [20] to either increase Q_{crit} or decrease Q_{coll} leading to stronger immunity against neutrons or alpha particle strikes. Acting on a higher level, some facilities can be added to the system on the architecture level to be able to detect the errors and correct them. The method applied in this thesis applies this last concept in its attempt to enhance the reliability of the arithmetic circuits under test.

1.3 Methods of Achieving Fault Tolerance

There are many published works in the literature addressing the issue of fault tolerance and reliability in electronic circuits in general [17, 20–25] and in arithmetic circuits in specific[*] [10, 26–37]. Techniques devised to enhance the reliability of a system can be categorized into the following categories [13]:

1. **Process level techniques:**
   These techniques try to enhance the reliability through using other fabrication technologies that are less affected by the previously mentioned failure techniques than the conventionally used CMOS technology.
2. **Circuit level techniques:**
   These techniques try to tune the device parameters in the current technologies to make them more immune to wearing effects and external factors.
3. **Architecture level techniques:**
   These techniques try to mitigate the effect of the device failures by providing the system with error detection and correction capabilities. In other words, these techniques do not prevent the faults form happening, but they provide the system with the ability to recover so that these faults would not turn into errors that can endanger the overall system operation.

The main focus of this thesis will be on the techniques that fall under the third category. Generally speaking, these techniques depend on adding some sort of redundancy to the system. This redundancy can take any of the following forms:

1. Hardware Redundancy.
2. Time Redundancy.
3. Information Redundancy.
4. A combination of 1, 2 and 3

[*]will be discussed in more detail in chapter 2

1.3.1 Hardware Redundancy

Hardware Redundancy depends on the presence of multiple units that carry out the same task. The outputs from these units is then compared to detect whether an error has occurred or not. For error detection purposes, duplicating the units is sufficient to achieve the required goal whereas error correction requires at least three redundant units followed by a majority circuit to choose the correct output. This method has the fastest performance and can overcome both temporary and permanent faults, but it introduces large area, and consequently cost overhead.

1.3.2 Time Redundancy

Time redundancy depends on performing the required operation many times using the same hardware. As in hardware redundancy, a two-time redundancy is sufficient for error detection purposes but at least three-time redundancy is necessary for correction. Clearly, this method almost has no area overhead, but it introduces a large delay and can only overcome temporary faults.

1.3.3 Information Redundancy

Information redundancy depends on adding additional data in the form of a redundant part added to the code words used. Said differently, a code word will consist of two parts; a *data part* and a *redundant part*. These two parts can be separated or not as explained in section 1.4.1. Codes which have this property are called *Error-Correcting* codes in which any code word, m, is represented by a certain number of digits larger than the minimum number of digits needed to represent the data implied in that code. Let d_{max} be the largest number in the data set of *base-b*. Then, the minimum number of digits (k) needed to represent d_{max} is

$$k = \lceil \log_b(d_{max}) \rceil \tag{1.1}$$

Therefore, if m is represented in *n-digits* instead of *k-digits* where $n > k$, the code is said to be redundant. The redundant part is calculated from the data part, therefore, if any inconsistency appears between the calculated redundant part and the actual redundant part in the code word, that means that an error has occurred.

Error-correcting codes are used for the purposes of error detection and/or correction and they can be divided into two main categories [10]:

1. **Communication Error-Correcting Codes**
   These codes are used for error detection and correction in communication systems and memories. They check the integrity of stored data, in case of memories, and that of received data, in case of communication systems. This category includes parity codes, Hamming codes, Turbo codes, Reed-Solomon codes and many others.

2. **Arithmetic Error-Correcting Codes**
   These codes are used for error detection and correction in arithmetic circuits. They check the integrity of an arithmetic operation to make sure that the arithmetic circuit is working properly. Due to its importance for this thesis, this category will be explained in more detail in the following section.

Information codes can be used for various applications other than error-detection and correction. Not all of them depend on redundancy, in fact some of them try to remove as much redundant information as possible. Following are some of the most important applications of information coding.

- **Cryptography [38]:**
  Cryptography aims at providing a secure way of transmitting information form the transmitter to the receiver. The general idea depends on encrypting the information called *plaintext* using a certain *key* to produce what is known as *ciphertext* which is the actual transmitted data. The receiver then uses that key to decrypt the received data and get the original message. The key is only known to the transmitter and the receiver, therefore even if third parties overheard the transmission they would not be able to decipher it.

- **Data Compression:**
  The main goal of Data Compression, also known as *Source Coding*, is to represent the information in the least possible number of bits. This allows for a more efficient transmission of data using as few resources as possible. However, special decoders are required in order to re-expand the compressed data.

1.4 Arithmetic Error-Correcting Codes

There are a lot of similarities [10] between arithmetic codes and communication codes, but the redundant parts of arithmetic codes need to be closed under the arithmetic operations being checked. In this context closure means that if a certain arithmetic operation is applied to the redundant parts of two or more code words, the result should be equal to the outcome of calculating the redundant part of the result of applying that operation to the data parts of the same code words. This property is generally not present in communication codes and hence, applying them to arithmetic circuits requires complex circuits for code prediction and computation [22]. Assuming CW_1 and CW_2 are two codewords where,

$$CW_1 = (d_1, r_1), CW_2 = (d_2, r_2) \tag{1.2}$$

and $\diamond$ represents any kind of arithmetic operations and $\mathbf{Red}(d_i)$ is the operation that calculates the redundant part (r_i) from the data part (d_i). Then, CW_1 and CW_2 are valid arithmetic code words if the following relation holds true.

$$r_1 \diamond r_2 = \mathbf{Red}(d_1 \diamond d_2) \tag{1.3}$$

1.4.1 Types of Arithmetic Codes

Based on the structure of arithmetic codes, they can be classified into the following categories:

- **Systematic vs Nonsystematic**

Systematic Codes	An *n-digit* code is said to be systematic if each code word consists of a set of *k-digits* representing the data and a set of $(n-k)$-*digits* representing the redundant digits.
Nonsystematic Codes	In this type of codes the data is not apparent in the code word. In other words, the data can not be extracted directly from the code word.

Table 1.3: Examples of binary arithmetic codes

Binary Number $(\boldsymbol{N})$	**Decimal equivalent**	$5\boldsymbol{N}$ **code**	$(\boldsymbol{N}, \boldsymbol{N}\mathbf{mod}_4)$ **code**	$(9\boldsymbol{N})\mathbf{mod}_{24}$ **code**
000	0	000000	(000, 00)	00000
001	1	000101	(001, 01)	01001
010	2	001010	(010, 10)	10010
011	3	001111	(011, 11)	00011
100	4	010100	(100, 00)	01100
101	5	011001	(101, 01)	10101
110	6	011110	(110, 10)	00110
111	7	100011	(111, 11)	01111

Systematic codes can be further divided into:

- **Separate vs. Nonseparate**

Separate Codes	In separate codes, the data and redundant parts are treated independently while performing arithmetic operations.
Nonseparate Codes	In nonseparate codes, the data and redundant parts are treated as one operand while performing arithmetic operations.

Many examples for arithmetic codes are mentioned in [10] and following are three of the most important arithmetic codes.

1. **Product Codes**, also known as *AN* codes [35], are an example of nonsystematic arithmetic codes. In an *AN* code a number, *N*, is represented by the product *AN*, where *A* is an integer. As an example, $5N$ code in mentioned as an example of this class in the third column of table 1.3

2. **Residue Codes** are an example of systematic separate codes. The redundant part in this class of codes is the residue of dividing the data part with respect to a certain modulus, m. This class of codes will be discussed in more detail in chapters 2 and 3. The fourth column of table 1.3 contains an example for this class of codes with $m = 4$.

3. $(\boldsymbol{gAN})\mathbf{mod}_m$ codes which were introduced in [39]. These codes are based on the *AN* codes but have the properties of being systematic and nonseparate codes. $(9N)\text{mod}_{24}$ is mentioned as an example for this code class in the last column in table 1.3

Table 1.4: IEEE 754-2008 floating point formats

Format	**base (b)**	**Number of significand digits**	**Maximum exponent**	**Minimum exponent**
binary16	2	11	15	-14
binary32	2	24	127	-126
binary64	2	53	1023	-1022
binary128	2	113	16383	-16382
decimal32	10	7	96	-95
decimal64	10	16	384	-383
decimal128	10	34	6144	-6143

The main focus in this thesis is directed towards fault tolerance using information redundancy which is applied in the proposed design in the form of calculating the residues of the operands and comparing the result of applying the operation (addition/subtraction/multiplication) to these residues with the residue of the result. The circuit to which we apply these techniques uses floating point numbers so we present a brief introduction to those numbers next.

1.5 Floating Point Number Representation

In general, a floating number, N is represented as follows,

$$N = (-1)^s \times b^{exp_N} \times M_N \tag{1.4}$$

where, s is the sign of N, exp_N is its exponent, b is the base of the number system used and M_N is called the significand. Floating point number representation is very efficient in representing real numbers which suffer from great limitations when represented as a fixed point number. The most widely used standard in general purpose floating point arithmetic units is the IEEE 754-2008 standard [40] or its previous version, the IEEE 754-1985 standard [41]. The IEEE 754-2008 standard defines different formats of different lengths for both the binary and decimal floating point numbers together with the arithmetic operations performed on these numbers. The number of significand digits as well as the maximum

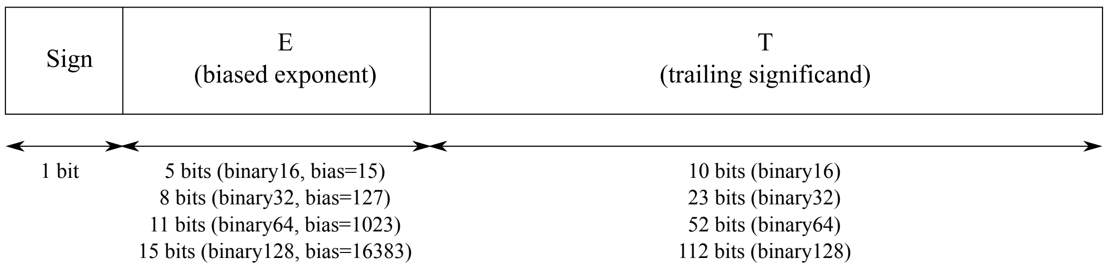


Figure 1.4: Binary floating point format

and minimum exponents of each format are shown in table 1.4. In general, the exponents are represented in the formats as a biased exponent which is the the true value of the exponent added to a certain bias that depends on the format.

1.5.1 Binary Floating Point Representation

In the binary format shown in figure 1.4, the significand is always normalized to have an integer part of '1'. This implies that each number has a unique representation. Since the integer part is always fixed, there is no need to reserve any place for it in the format. This implicit integer part is commonly known as *hidden one*. Hence, the T field in the binary format only carries the fraction part of the significand. Therefore, the value of a binary floating point can be calculated as

$$N = (-1)^{sign} \times 2^{(E-bias)} \times 1.T \tag{1.5}$$

The standard also defines several special values such as $\pm\infty$, NaNs, ± 0 and subnormal numbers. The values for the Sign, E and T fields for each of these values are shown in table 1.5. A subnormal number is a number smaller than the minimum number that can be represented in the normalized form for a certain floating point format. Its value is given as

$$N = (-1)^{sign} \times 2^{exp_{min}} \times 0.T \tag{1.6}$$

where exp_{min} is the minimum exponent for the used format.

Table 1.5: Special value representation in binary format

Value	**Sign field**	***E* field**	***T* field**
+0	+	all zeros	all zeros
-0	-	all zeros	all zeros
$+\infty$	+	all ones	all zeros
$-\infty$	-	all ones	all zeros
Nan	+/-	all ones	non-zero
Subnormal Numbers	+/- (according to number)	all zeros	non-zero (according to number)

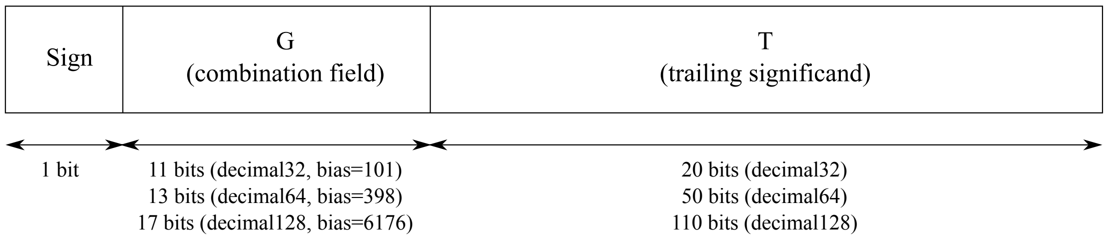


Figure 1.5: Decimal floating point format

1.5.2 Decimal Floating Point Representation

Unlike binary floating point numbers, decimal floating point numbers are not normalized. This means that the same real number might have more than one floating point representation. For example, a real number of 0.5 can be represented as 0.5×10^0, 5×10^{-1}, 0.005×10^2 and so on. These various equivalent representations of the same number are known as the floating point number's *cohort*.

As shown in figure 1.5, the decimal floating point format consists of a sign bit, a combination field (G) and a trailing significand field (T). The combination field encodes the biased exponent of the floating point number, special values and the most significant digit of the significand. The trailing significand field encodes the values for the least significant $p-1$ digits (assuming that the significand is a p-digit number). According to the IEEE standard, the significand can be encoded using two different techniques as follows:

Table 1.6: Combination field decoding in decimal floating point formats

	$G_0 \dots G_4$	Case
	11111	NaN
	11110	∞ $(sign = 0)$
	11110	$-\infty$ $(sign = 1)$
Decimal Encoding	110xx 1110x	$d_0 = 8 + G_4$ $leading\, biased\, exponent\, bits = 2G_2 + G_3$
Decimal Encoding	0xxxx 10xxx	$d_0 = 4G_2 + 2G_3 + G_4$ $leading\, biased\, exponent\, bits = 2G_0 + G_1$
Binary Encoding	0xxxx 10xxx	$biased\, exponent = G_0 \dots G_{w+1}$ $Most\, significant\, significand\, bits = G_{w+2}G_{w+3}G_{w+4}$
Binary Encoding	110xx 1110x	$biased\, exponent = G_2 \dots G_{w+3}$ $Most\, significant\, significand\, bits = 8 + G_{w+4}$

- **Decimal Encoding:**
  In this technique each three digits are encoded in ten bits according to the *Densely Packed Format* (DPD) [42].

- **Binary Encoding:**
  In this technique the trailing significand field is combined with the designated bits from the combination field and the result is considered as an unsigned binary integer.

If the width of the combination field is $w + 5$ $(G_0G_1 \dots G_{w+4})$ and the most significant significand digit is d_0, then the combination field can be decoded according to table 1.6.

1.6 Floating Point Addition

As mentioned in the beginning of this chapter, the error detection and correction technique proposed in this thesis was applied to the floating point adder designed in [6]. Therefore, for the sake of completeness, the binary and decimal floating point addition processes are discussed in the following section in order to provide an overall look of the main steps included in such processes.

Table 1.7: Determining the effective operation

Desired Operation	**Operands Signs**	**Effective Operation**
Addition	Same	Addition
Addition	Opposite	Subtraction
Subtraction	Same	Subtraction
Addition	Opposite	Addition

1.6.1 Binary Floating Point Addition

If two normalized binary floating point numbers were to be added, the resulting floating point number must also be normalized in order to be compliant with the IEEE 754 standard. Therefore, it is assumed, at first, that the exponent of the result will be equal to the larger of the two exponents of the two operands. This ensures that the number with the larger exponent remains unchanged while that with the smaller exponent is shifted to the right a number of positions equal to the exponent difference. Hence, the chances that the result of the addition is normalized increases. Moreover, a rounding process has to be performed according to the desired rounding mode as specified in the standard. The steps of binary floating point addition can be summarized as follows [6]:

1. **Calculating the exponent difference:**
   The exponent difference Δexp is calculated and the exponent of the result is set to the larger exponent of the two operands.

2. **Significand alignment:**
   The number with the smaller exponent is shifted Δexp positions to the right.

3. **Addition/Subtraction:**
   Performing the required arithmetic operation based on the effective operation which depends on the desired operation and the signs of the operands according to table 1.7. If the result is negative, then the resulting significand must be complemented.

4. **Normalization:**
   During addition, a final carry might be generated. This means that the result-

ing significand is not normalized. Hence, the resulting significand must be shifted one position to the right and the result exponent must be incremented by one. A similar problem could happen when performing subtraction, but in this case the problem is in the generation of leading zeros in the resulting significand. Therefore, the number of leading zeros has to be detected and then the significand must be shifted to the left by that number. Eventually, the result exponent must also be decremented by the number of the leading zeros.

5. **Rounding:**
   The resulting floating point number is rounded according to the desired rounding mode. In order to be able to perform rounding correctly, information about the bits that were shifted out during the alignment step is used. This information is kept in the form of three bits named the *Round*, *Guard* and *Sticky* bits.

6. **Final Adjustment:**
   After the rounding step, the result might need further normalization or it might even be a special value. Therefore, suitable actions must be taken in order to handle these situations.

1.6.2 Decimal Floating Point Addition

In general, decimal floating point addition is very similar to its binary counterpart. The main differences between them can be summarized as follows:

- As discussed in section 1.5.2, the significand and exponent of a decimal floating point number are implicit in the combination and the trailing significand fields. Hence, the first step would be to decode these fields in order to get the biased exponent and the significand of the floating number. Moreover, after performing the desired operation the result has to be encoded into its floating point representation.

- Due to the fact that decimal floating points are not normalized, the arithmetic operation being performed has to choose a member of the result's cohort to be its final result. The exponent of the chosen member is called the *preferred* *exponent*. In case of addition/subtraction with exact results, the preferred exponent is the smaller of the two operands. Meanwhile, if the result of the addition/subtraction is inexact, the preferred exponent is chosen to be the least possible exponent.

1.7 Thesis Outline

In this chapter a brief introduction to the subject of fault tolerance (error detection and/or correction) was presented together with a brief overview of the floating point number representation and addition. In chapter 2, previous techniques for adding fault tolerance to arithmetic circuits are discussed together with a brief overview of the residue number system. The proposed design is discussed in chapter 3 and the results are discussed in chapter 4. Chapter 5 concludes this thesis together with suggestions for future research to improve the proposed design.

Chapter 2

Background and Related Work

2.1 Previously Developed Fault Tolerant Arithmetic Units

Several techniques can be found in the literature [10, 26–37] that add fault tolerance to arithmetic circuits. They differ in the capabilities they offer, whether error detection only or error detection and correction, and in their approach to the fault tolerance problem. In general, the ultimate goal of fault tolerance techniques is to make the system completely reliable at minimum cost overhead. This implies being able to detect as many errors as possible and correcting them while at the same time keeping time, area and power overhead to the minimum.

REMOD *(REprocessing with MicrO Delays)* has been proposed in [30] as a general scheme for fault detection, diagnosis and reconfiguration in arithmetic circuits composed of arrays of identical functional units. REMOD applies the concepts of both time and hardware redundancy. Each unit has a *cover* unit which carries out the same function of the original unit on the same set of inputs after some delay. Then, the outputs of both the original and the cover units are compared to check for errors. The check operation is done by circularly shifting the inputs such that the inputs are processed by the cover units. An additional shift operation is needed for diagnosis purposes such that each input is processed three times by three independent units. The idea of RESO *(REcomputation with Shifted Operands)* was first introduced in [31] where it was applied to adders. It was later applied to multiplier and divider arrays in [32].

Another scheme called REDWC *(REcomputation with Duplication and Comparison)* was introduced in [28]. It also uses a combination of time and hardware redundancy to achieve its goal of error detection. In [28], REDWC was applied to a 32-bit adder which was divided into two 16-bit adders. The addition and detection process is performed in two iterations. In the first iteration, the least significant halves of the operands are inputs to both of the 16-bit adders and then their outputs are checked together. In the second iteration, the most significant halves of the operands become the inputs and the process is repeated. The idea of REDWC was further extended to allow for error correction in [33] with the scheme called HPTR *(Hardware Partition in Time Redundancy)*, [34] with the scheme called RETWV *(REcomputation with Triplication With Voting)* and in [43] as the QTR *(Quadruple Time Redundancy)* scheme.

Designs for fault tolerant multipliers have been proposed in [27] and [29]. In [27], a *self-checking self-diagnosing 32-bit microprocessor multiplier* is proposed. It has error detection and correction capabilities with the ability to isolate the faulty unit. This technique considers the multiplication as a branch instruction allowing its result to be used while still being checked. If proved to be a wrong result, the processor pipeline is flushed and error correction mechanism is initiated. Its algorithm is based on using reconfigurable units recursively to detect the faulty unit and isolate it thus allowing only fault-free units to contribute to the result.

All of the above mentioned techniques are based on hardware redundancy, time redundancy or a combination of them. Another class of techniques depended on information redundancy in the form of arithmetic codes. In [26] and [37] residue codes were used to achieve error detection in floating point units while in [35] *AN* codes were used to do the job. Berger codes were also proposed as a coding scheme in [36] where the redundant data is the number of ‘0’ bits in the original data. Moreover, some techniques [44–48] adopted the *residue number system*, RNS[*], to be the number system used throughout the whole arithmetic circuit not just the error detection and correction process. RNS has many properties that are very useful to arithmetic circuits and leads to very fast arithmetic. Among these

[*]explained in more details in section 2.2

properties are [49] and [50]:

- The capability of performing carry-free addition, borrow-free subtraction and digit-by-digit multiplication.
- Fault tolerance capabilities can be easily added by introducing redundant residues resulting in what is known as *Redundant Residue Number System*, RRNS

On the other hand, RNS is not easily dealt with when it comes to sign detection, magnitude comparison, overflow detection, division and other complex arithmetic operations [49] and [50]. Moreover, since data in not normally stored in RNS representation, using it adds data conversion overhead which has a great impact on the overall performance of the system.

2.2 Residue Codes and Arithmetic

As mentioned earlier, a residue code is a systematic separate arithmetic code. It is formed [37] by appending the residue of the number, with respect to a set of moduli, to the number itself. Being a separate code, it has an advantage that the redundant part of the code can be treated independently form the data part. It also has the advantage that no further decoding is required to obtain the original data from its redundant representation. Apart from using residues as a coding scheme, they can be used to fully represent a set of numbers. The system in which numbers are expressed in the form of their residues with respect to a certain set of moduli is called the *Residue Number System* (RNS). Unlike the *Weighted Number System* (WNS), such as binary and decimal number systems, in RNS the digits have no weights and consequently no order which allows for carry-free arithmetic operations.

The first use of RNS in history can be probably traced back to the puzzle posed by the Chinese mathematician, *Sun Tzu* [49, 51]. In his third century book, *Suan-ching*, he wrote a verse where he asked a simple yet very important question in the RNS history. This question was; what is the number that when divided by three, five and seven has the remainders of two, three and two respectively? The method developed to reach the answer to this question, which is 23, is explained in

Sun Tzu's historic work. This method was later known as the *Chinese Remainder Theorem*, CRT.

If an RNS has the moduli set $(m_1, m_2, \ldots, m_n)$, then the maximum range of numbers, M, that can be represented by this RNS is achieved in the case of $m_1, m_2, \ldots, m_n$ being relatively prime. In this case, M can be calculated as,

$$M = \prod_{i=1}^{n} m_i \tag{2.1}$$

On the other hand, if the moduli were not relatively prime, then M would be calculated as,

$$M = \operatorname{lcm}(m_1, m_2, \ldots, m_n) \tag{2.2}$$

If the RNS is meant to represent unsigned integers, then these integers can take any value from zero to $M-1$. Meanwhile, if it is meant to represent signed integers, then any of these integers, N, may have any of the following values [52]:

$$-\frac{M-1}{2} \leq X \leq \frac{M-1}{2} \qquad \text{if } M \text{ is odd}$$

$$-\frac{M}{2} \leq X \leq \frac{M}{2} - 1 \qquad \text{if } M \text{ is even}$$

A system based on RNS is mainly composed of three levels; WNS to RNS conversion, RNS operations and finally RNS to WNS conversion. That last level being the most challenging, it is discussed in more detail in the following section.

2.2.1 RNS to WNS conversion

Several methods have been devised to convert a certain number from the *Residue Number System (RNS)* to the *Weighted Number System (WNS)* [49–51, 53–56]. These methods differ in their complexity, time needed to complete the conversion, etc. A comprehensive study of the different methods used in RNS to WNS conversion can be found in [49]. It is of crucial importance to find a conversion technique that is simple and fast enough so as not to degrade the overall performance of the system. In the following sections, some of most known conversion methods are discussed.

2.2.2 Chinese Remainder Theorem (CRT)

This theorem, along with the theory of residue numbers, was presented in the 19^{th} century by Carl Friedrich Gauss in his celebrated *Disquisitiones Arithmetical* [49, 51]. In this theorem, a number represented in the RNS as $(r_1, r_2, \ldots, r_n)$ with respect to the moduli $(m_1, m_2, \ldots, m_n)$ can be converted to its WNS equivalent, X, by applying equation 2.3 where $r_i = X\text{mod}_{m_i}$, $1 \leq i \leq n$.

$$X = \left(\sum_{i=1}^{n} M_i \times (M_i^{-1} r_i)\text{mod}_{m_i} \right) \text{mod}_M \tag{2.3}$$

where $M = \prod_{i=1}^{n} m_i$ and is called the dynamic range of the RNS, $M_i = {}^{M}\!/_{m_i}$ and M_i^{-1} is the modular multiplicative inverse[*] of M_i with respect to m_i.

Each term of the summation in equation 2.3 is independent of all other terms meaning that they can be calculated in parallel with each other. On the other hand, the large mod_M operation can be rather expensive in terms of area, time and energy.

There have been lots of efforts to simplify the CRT computation process but they have been dedicated to certain moduli sets. In [57] four three-moduli based RNSs are discussed which are $(2^n, 2_n + 1, 2_n - 1), (2n, 2n + 1, 2n - 1), (2^n, 2^n - 1, 2^{n-1} - 1)$ and $(2^{2n} + 1, 2^n + 1, 2^n - 1)$.

2.2.3 Mixed Radix Conversion (MRC)

This method depends on expressing numbers as a group of *Mixed Radix Digits* (MRD), $(a_1, a_2, \ldots, a_n)$ having the weights $(1, m_1, m_1m_2, \ldots, m_1m_2 \cdots m_n)$ respectively. In other words, a number, X, is represented in MRD representation as:

$$X = a_1 + a_2m_1 + a_3m_1m_2 + \cdots + a_nm_1m_2 \cdots m_n \tag{2.4}$$

If the RNS representation of X is $(r_1, r_2, \ldots, r_n)$, then the MRD, $a_i, 1 \leq 1 \leq n$ can be expressed in terms of the residues, $r_i, 1 \leq i \leq n$ as follows:

[*]If q is the modular multiplicative inverse of a number, n, with respect to a certain modulus, m, then $(nq)\text{mod}_m = 1$

$$a_1 = X\text{mod}_{m_1} = r_1 \tag{2.5a}$$

$$\begin{aligned} a_2 &= \left(\frac{X-a_1}{m_1}\right)\text{mod}_{m_2} \\ &= \left((r_2-a_1)m_1^{-1}\text{mod}_{m_2}\right)\text{mod}_{m_2} \end{aligned} \tag{2.5b}$$

$$\begin{aligned} a_3 &= \left(\frac{X-a_1-a_2m_1}{m_1m_2}\right)\text{mod}_{m_3} \\ &= \left((r_3-a_1)(m_1m_2)^{-1}\text{mod}_{m_3} - a_2m_2^{-1}\text{mod}_{m_3}\right)\text{mod}_{m_3} \\ &= \Big(\left((r_3-a_1)m_1^{-1}\text{mod}_{m_3} - a_2\right)m_2^{-1}\text{mod}_{m_3}\Big)\text{mod}_{m_3} \end{aligned} \tag{2.5c}$$

$$\vdots$$

$$\begin{aligned} a_n = \Bigg(\Big(&\cdots\left((r_k-a_1)m_1^{-1}\text{mod}_{m_n} - a_2\right)m_2^{-1}\text{mod}_{m_n} - \cdots \\ &- a_{n-1}\Big)m_{n-1}^{-1}\text{mod}_{m_n}\Bigg)\text{mod}_{m_n} \end{aligned} \tag{2.5d}$$

Unlike the CRT method, this method does not require the large mod_M for calculating the MRDs. On the other hand, the MRC method is sequential where each MRD depends on the preceding MRDs.

For the Basic MRC, the calculation of the MRDs requires $n(n-1)/2$ subtractions and multiplications which are then added in $(n-1)$ additions. This means that the complexity of the basic MRC is in the order of $O(n^2)$.

Several attempts [50, 53, 54, 56] have been made in order to decrease the complexity of the MRC method. The method proposed in [56] manages to decrease the complexity of the MRC method from $O(n^2)$ to $O(n)$ through simultaneous use of the different mod_{m_i} adders found in an RNS adder.

2.2.4 Generalized Matrix Method (MATR)

This is an MRC based method which was proposed in [49, 55] and depends on the periodicity property of RNS. It was mentioned by a specific example in [53, 54] and then it was generalized for any RNS number in [55]. In fact, the MATR method is very similar to the method mentioned in [56] but adopting a different paradigm in the explanation and derivation of the method. The equivalent number to an RNS number is obtained by jumping backwards in the residue table[*] to the nearest number having at least one residue equal to zero. The value of the jump is stored and the process is repeated until all residues are equal to zero. The value of the number is then calculated as the sum of the values of all the jumps.

By applying this technique, a number, X, can be calculated from its RNS representation, $(r_1, r_2, \ldots, r_n)$, with respect to the moduli set $(m_1, m_2, \ldots, m_n)$ as follows.

$$X = \sum_{i=1}^{n} p_i \tag{2.6}$$

where p_i is the value of the i^{th} jump and is given by:

$$p_i = \begin{cases} r_1 & \text{if } i = 1 \\ (m_1 m_2 \cdots m_{i-1}) & \\ \qquad \times \left((m_1 m_2 \cdots m_{i-1})^{-1} \text{mod}_{m_i} \times t_{(i-1)j} \right) \text{mod}_{m_i} & \text{if } i > 1 \end{cases} \tag{2.7}$$

where the value of $t_{(i-1)j}$ is calculated from the following matrix calculations.

After stepping backwards p_1 steps, the RNS number becomes:

$$X - p_1 = \begin{pmatrix} (r_1 - p_1)\text{mod}_{m_1} = 0 \\ (r_2 - p_1)\text{mod}_{m_2} = t_1 \\ (r_3 - p_1)\text{mod}_{m_3} = t_2 \\ \vdots \\ (r_n - p_1)\text{mod}_{m_n} = t_n \end{pmatrix} \tag{2.8}$$

The second jump, p_2 can then be calculated as

$$p_2 = m_1 \left(m_1^{-1} \text{mod}_{m_2} \times t_1 \right) \text{mod}_{m_2} \tag{2.9}$$

Being a multiple of m_1 guarantees that after stepping backwards p_2 steps, the first residue (with respect to m_1) remains zero. Moreover, equation 2.9 guarantees that $(t_1 - p_2)\,\text{mod}_{m_2} = 0$ meaning that after the second jump the second residue will be zero along with the first one.

After jumping backwards p_2 positions in the residue table, the RNS number

[*]a table listing all the possible numbers together with their corresponding residues with respect to the moduli of the RNS being studied

becomes:

$$X - p_1 - p_2 = \begin{pmatrix} (0 - p_2)\,\mathrm{mod}_{m_1} = 0 \\ (t_1 - p_2)\,\mathrm{mod}_{m_2} = 0 \\ (t_2 - p_2)\,\mathrm{mod}_{m_3} = t_{21} \\ \vdots \\ (t_n - p_2)\,\mathrm{mod}_{m_n} = t_{(n-1)1} \end{pmatrix} \tag{2.10}$$

This process continues until a position is reached in the residue table where all the residues are zeros. The equivalent number can then be calculated as the sum of all the jumps. This method is similar to a great extent to the process of calculating the MRDs in the previous method but the subtractions for calculating t_i are done in parallel.

For the worst case, p_i, for $2 \leq i \leq n$, need to be computed. Each p_i computation includes:

- $(k-1)$, mod_{m_i}, subtractions for calculating the values of t_i's
- Two multiplications; one for multiplying t_i by m_{i-1} and the other for multiplying the result of the previous multiplication by $m_{i-1}^{-1}\mathrm{mod}_{m_i}$.

The above mentioned subtractions can be carried out in parallel in the RNS adder. Moreover, the only variable in the multiplication processes is t_i. Therefore, the result of the multiplication processes can be precomputed and stored in a look up table for different values of t_i. Finally, the values of p_i, for $1 \leq i \leq n$, are added together in $(k-1)$ addition processes. In general, the complexity of the MATR method is in the order of $O(n)$.

It is worth mentioning that the above mentioned methods are only applicable on RNSs with relatively prime set of moduli.

2.3 Conclusion

In this chapter, some of the previously developed techniques for adding fault tolerance to arithmetic circuits were discussed. Moreover, RNS was briefly introduced with special emphasis on the conversion from RNS to WNS. Three methods were discussed for the conversion; CRT, MRD and MATR. CRT is a parallel process, but it includes a large mod_M operation which can be rather complicated. On the other hand, MRC does not include large mod_M operations, but it is a sequential process with $O(n^2)$ complexity in its basic form. MATR manages to improve this complexity to be $O(n)$ while maintaining the simplicity of the mod operations. Therefore, it was used in the proposed design for RNS to WNS conversion.

Chapter 3

Proposed Design

3.1 Objective

The main purpose of the work presented in this thesis is to add fault detection and correction capabilities to the mixed decimal/binary redundant floating point adder presented in [6].

3.2 Methodology

As mentioned in section 1.3, in order to achieve fault tolerance for arithmetic circuits, some sort of redundancy must be added to the circuit. This redundancy can take the form of hardware redundancy, information redundancy, time redundancy or any combination of these techniques.

The technique proposed in this thesis depends on information redundancy. It depends on calculating the residues of the operands to the arithmetic circuit, performing the arithmetic operation on the residues as well as the operands and finally calculating the syndrome of the result. The syndrome of a certain number is defined as

$$\begin{aligned}\mathbf{S}[X, r_1, r_2, \dots, r_n] &= \big((X - r_1)\text{mod}_{m_1}, (X - r_2)\text{mod}_{m_2}, \dots, (X - r_n)\text{mod}_{m_n}\big) \\ &= (s_1, s_2, \dots, s_n)\end{aligned} \tag{3.1}$$

where, X is the number, m_i are the moduli chosen for the calculation of the residues and $r_i = (X)\text{mod}_{m_i}$. Therefore, in the case being discussed there is one of four cases, assuming that X is the result of the arithmetic operation and r_i are the independently calculated residues of the result:

Case 1: **No Error Occurred**

In this case, X as well as r_i are calculated correctly. Therefore,

$$\begin{aligned} s_i &= (X - r_i)\text{mod}_{m_i} \\ &= (X)\text{mod}_{m_i} - (r_i)\text{mod}_{m_i} \\ &= 0 \end{aligned} \tag{3.2}$$

Case 2: **Error in Main Unit**

In this case, $X_{err} = X + e$, but $r_i = (X)\text{mod}_{m_i}$. Therefore,

$$\begin{aligned} s_i &= (X_{err} - r_i)\text{mod}_{m_i} \\ &= (X + e)\text{mod}_{m_i} - (r_i)\text{mod}_{m_i} \\ &= (e)\text{mod}_{m_i} \end{aligned} \tag{3.3}$$

Case 3: **Error in Residue Calculation**

In this case, one or more of the r_j's might not be correct. Let these erroneous residues be $r_{j_{err}} = r_j + e$, $j = 1, 2, \ldots, n$. Therefore,

$$\begin{aligned} s_j &= (X - r_{j_{err}})\text{mod}_{m_j} \\ &= (X)\text{mod}_{m_j} - (r_j + e)\text{mod}_{m_j} \\ &= (-e)\text{mod}_{m_j} \end{aligned} \tag{3.4}$$

while all other $s_i = 0$, where $i \neq j$

Case 4: **Errors in both Main Unit and Residue Calculation**

This case is a combination of cases 3 and 2. Hence, $X_{err} = X + e_{main}$ and the erroneous residues are $r_{j_{err}} = r_j + e_{res}$, $j = 1, 2, \ldots, n$. Therefore,

$$\begin{aligned} s_j &= (X_{err} - r_{j_{err}})\text{mod}_{m_j} \\ &= (X + e_{main} - r_j - e_{res})\text{mod}_{m_j} \\ &= (e_{eq})\text{mod}_{m_j} \end{aligned} \tag{3.5}$$

where, $e_{eq} = e_{main} - e_{res}$ while $s_i = (e_{main})\text{mod}_{m_i}$ for $i \neq j$.

Case 5: **Error is not Detectable**

If the error in any of the cases 2, 3 or 4 is a multiple of $m_1 \times m_2 \times \cdots m_n$, then s_i would be zero. Errors in this form are undetectable because there is no way to differentiate between this case and the first case where no error occurred.

Through the proper choice of the moduli for which the residues are computed, it can be guaranteed that each error in the error set of interest has a unique corresponding syndrome pattern. Therefore, knowing the syndrome pattern, the corresponding error can be determined and hence the result can be corrected. This will be further explained in section 3.2.1. The method described in this thesis covers cases 1, 2 and 5 while it has not been extended yet to address the other two cases.

3.2.1 Factors Affecting the Choice of the Moduli

The choice of the redundant moduli depends on the following factors [10] which will be discussed separately in the following sections:

1. Ease of calculation of residues based on these moduli
2. Closure under addition/subtraction and multiplication for these residues
3. Suitable range of detectable and correctable errors

Ease of Calculation of Residues

In order to decrease the overhead introduced by the residue calculation process, the moduli should be chosen such that this process is as simple as possible. In general residues with respect to a certain modulus, *m*, are calculated as the remainder of dividing a number by *m*. Carrying out this operation literally may introduce very large overhead form the point of view of area, power and time. Meanwhile, residues with respect to certain moduli can be calculated via simple arithmetic operations which can greatly decrease residue calculation overhead [58, 59]. Moduli of b^n, $b^n - 1$ and $b^n + 1$ are examples of such moduli where b is the base of the number system being used.

Let X be a *base-b* number of k digits and it is assumed for simplicity that k is a multiple of n, then it can be expressed as

$$X = \sum_{i=0}^{i=l} x_i (b^n)^i \tag{3.6}$$

where $0 \leq x_i \leq b^n - 1$ and $l = k/n$. Then the modulus of X with respect to $m = b^n$ can be calculated as:

$$\begin{aligned} (X)\text{mod}_m &= \left(\sum_{i=0}^{i=l} x_i (b^n)^i \right) \text{mod}_m \\ &= \left(x_l (b^n)^l + x_{l-1} (b^n)^{l-1} + \cdots + x_2 (b^n)^2 + x_0 (b^n)^0 \right) \text{mod}_m \\ &= x_0 \end{aligned} \tag{3.7}$$

In other words, residues with respect to b^n are simply the least significant n-digits of the number. Despite its simplicity, the most significant $k - n$ digits do not contribute to the residue calculation process. This leads to poor error coverage for the b^n moduli set.

On the other hand, the residue of X with respect to $m = b^n - 1$ can be calculated as:

$$\begin{aligned} (X)\text{mod}_m &= \left(\sum_{i=0}^{i=l} x_i (b^n)^i \right) \text{mod}_m \\ &= \left(x_l (b^n)^l \right) \text{mod}_m + \left(x_{l-1} (b^n)^{l-1} \right) \text{mod}_m + \cdots + \left(x_0 (b^n)^0 \right) \text{mod}_m \end{aligned} \tag{3.8}$$

and since in the case of $m = b^n - 1$,

$$\left((b^n)^l \right) \text{mod}_m = \left((b^n)^{l-1} \right) \text{mod}_m = \cdots = \left(b^n \right) \text{mod}_m = 1 \tag{3.9}$$

then

$$\begin{aligned}\big(X\big)\text{mod}_m &= (x_l)\text{mod}_m + (x_{l-1})\text{mod}_m + \cdots + (x_0)\text{mod}_m \\ &= \left(\sum_{i=0}^{i=l} x_i\right)\text{mod}_m\end{aligned} \tag{3.10}$$

Similar analysis can be done with $m = b^n + 1$, but in this case the remainder of dividing X by m would be calculated as follows

$$\big(X\big)\text{mod}_m = \left(\sum_{i=0}^{i=l} (-1)^i x_i\right)\text{mod}_m \tag{3.11}$$

In these cases $(m = b^n \pm 1)$, the residue is calculated by dividing the number into groups of n digits and adding/subtracting these groups together [26] as shown in equations 3.10 and 3.11. This process is repeated until the result is smaller than the modulus. This is due to the fact that the remainder from dividing $(b^n)^i$ by $b^n - 1$ is 1 and that resulting from dividing $(b^n)^i$ by $b^n + 1$ is either 1 or -1.

Example 1: The result of $(163258756)mod_{999}$ can be calculated as follows:

$$\begin{aligned}(163258756)mod_{999} &= (163 + 258 + 756)\text{mod}_{999} \\ &= (1177)\text{mod}_{999} \\ &= (1 + 177)\text{mod}_{999} \\ &= (178)\text{mod}_{999} \\ &= 178\end{aligned}$$

Example 2: The result of $(163258756)mod_{101}$ can be calculated as follows:

$$\begin{aligned}(163258756)mod_{101} &= (1 - 63 + 25 - 87 + 56)\text{mod}_{101} \\ &= (-68)\text{mod}_{101} \\ &= 33\end{aligned}$$

Closure under Addition/Subtraction and Multiplication[*]

As explained earlier in section 1.4, closure in this context means that the result of applying the arithmetic operation (addition, subtraction or multiplication) to the residues of the operands of the operation is equal to the residue of the result of applying the same operation to the operands themselves.

In order to be able to use the calculated residues directly to detect and correct possible errors, the residue system must be closed under addition and multiplication. Otherwise, some form of correction has to be applied in order to be able to use that residue code in error detection/correction. In other words, if r_x and r_y are the residues of two numbers, x and y, with respect to a certain modulus, m. Then $r_x +$ or $\times\, r_y$ must be equal to r_z in order to be able to use m directly as a modulus for error detection and correction where r_z is the modulus of $x +$ or $\times\, y$ with respect to m.

Due to the limited resources in arithmetic circuits, they actually perform modular addition and/or multiplication. For a modular addition to be closed under addition for a certain residue code, the range of the adder must be divisible by the modulus of that code [10]. Suppose the modulus of a certain residue code is m and the range of the adder is m_r. Then, if two numbers N_1 and N_2 were added, the result would be

$$N_{tot} = (N_1 + N_2)\,\mathrm{mod}_{m_r} \tag{3.12}$$

with a corresponding residue code of

$$N_{tot}\mathrm{mod}_m = ((N_1 + N_2)\,\mathrm{mod}_{m_r})\,\mathrm{mod}_m \tag{3.13}$$

On the other hand, the result of addition of the residues of N_1 and N_2 is

$$\begin{aligned} N'_{tot} &= (N_1)\,\mathrm{mod}_m + (N_2)\,\mathrm{mod}_m \\ &= (N_1 + N_2)\,\mathrm{mod}_m \end{aligned} \tag{3.14}$$

For $N_{tot}\mathrm{mod}_m$ to be equal to N'_{tot}, m_r must be divisible[†] by m. The proof used in

[*]Only closure under addition and subtraction are needed for the work presented in this thesis but closure under multiplication is mentioned for the sake of completeness

[†]Ref. [10, LEMMA 3.2]: For any N, $(N\mathrm{mod}_x)\,\mathrm{mod}_y = N\mathrm{mod}_y$ iff y divides x

[10] can be extended to include closure under multiplication as well as closure under addition. If the residue code is not closed under addition or multiplication some form of correction has to be applied. This is demonstrated in the following examples.

Example 1: In this example the closure under addition property for residue codes is investigated. Suppose the two decimal numbers, 7425 and 8231 are to be added with a decimal adder of range 4; meaning that the result of the adder belongs to the set $[-10^4+1, 10^4-1]$ (for example if the result of adding the numbers should be 10004, the output from the adder would be 4). The two residue codes under investigation have moduli of 101 and 50. The mod-50 system represents the case where the range of the adder is divisible by the modulus of the residue code system while the mod-101 system represents the case where the range of the adder is not divisible by the modulus of the residue code system and therefore a correction has to be applied. The operands in these residue systems are represented as

$$7425\text{mod}_{101} = 52)_{101} \qquad 7425\text{mod}_{50} = 25)_{50}$$

$$8231\text{mod}_{101} = 50)_{101} \qquad 8231\text{mod}_{50} = 31)_{50}$$

The $L)_b$ notation is used to indicate that L is in a mod-b residue number system. Then,

$$\begin{array}{rl} & 7425 \\ + & 8231 \\ \hline 1 & 5656 = 5656)_{10^4} \end{array} \qquad \begin{array}{rl} & 52 \\ + & 50 \\ \hline 1 & 02 = 1)_{101} \end{array} \qquad \begin{array}{rl} & 25 \\ + & 31 \\ \hline & 56 = 6)_{50} \end{array}$$

The closure under addition property for the mod-50 system can be shown by the fact that $5656\text{mod}_{50} = 6$. On the other hand, this is not the case for the mod-101 system where $5656\text{mod}_{101} \neq 1$. As stated before, a correction is needed in the latter case. A possible correction in that case is to take the output carry into consideration leading to $15656\text{mod}_{101} = 1$

Example 2: In this example the closure under multiplication property for residue codes is investigated. Suppose the two decimal numbers, 123 and 217 are multiplied with a decimal multiplier of range 4; meaning that the result of the multiplier belongs to the set $[-10^4+1, 10^4-1]$ (for example if the result of multiplying the numbers should be 10004, the output from the multiplier would be 4). Again, the two residue codes under investigation have moduli of 101 and 50. The operands in these residue systems are represented as

$123\text{mod}_{101} = 22)_{101}$ $\quad$ $123\text{mod}_{50} = 23)_{50}$

$217\text{mod}_{101} = 15)_{101}$ $\quad$ $217\text{mod}_{50} = 17)_{50}$

Then,

$$\begin{array}{rl} & 123 \\ \times & 217 \\ \hline 2 & 6691 = 6691)_{10^4} \end{array} \qquad \begin{array}{rl} & 22 \\ \times & 15 \\ \hline 3 & 30 = 27)_{101} \end{array} \qquad \begin{array}{rl} & 23 \\ \times & 17 \\ \hline 3 & 91 = 41)_{50} \end{array}$$

The closure under multiplication property for the mod-50 system can be shown by the fact that $6691\text{mod}_{50} = 41$. On the other hand, this is not the case for the mod-101 system where $6691\text{mod}_{101} \neq 27$. As stated before, a correction is needed in the latter case. A possible correction in that case is to take the output carry into consideration leading to $26691\text{mod}_{101} = 27$

Range of Detectable and Correctable Errors

As explained in section 3.2, the actual error is determined from its syndrome pattern. A non-zero syndrome pattern is enough to detect that an error has occurred but might not always give the correct value of the error (explained shortly). For error detection purposes, it is not required to be able to calculate the value of the error from the syndrome pattern. Suppose the set of possible errors is $\mathbf{E}$ and an error, $e \in \mathbf{E}$, has occurred. A residue code of modulus m would be able to detect any error, e, as long as $(e)\text{mod}_m \neq 0$. If more than one residue code was used with moduli $m_1, m_2, \ldots, m_i$, then it would be possible to detect any error,

e, as long as $(e)\text{mod}_{m_{eq}} \neq 0$, where i is the number of residue codes used and $m_{eq} = lcm(m_1, m_2, \ldots, m_i)$. In other words, an error equal to a multiple of m_{eq} will generate a zero syndrome pattern and hence will not be detected.

On the other hand, error correction requires the ability to calculate the error from its syndrome pattern. Therefore, each error value must have a unique syndrome pattern. For a group of residue codes with moduli $m_1, m_2, \ldots, m_i$, the number of unique residue (syndrome) patterns, $\mathbf{R}$ is given as:

$$\mathbf{R} = lcm(m_1, m_2, \ldots, m_i) \tag{3.15}$$

If $\mathbf{E}$ is the set of all possible errors, then the difference between any two elements in $\mathbf{E}$ must not be equal to $\mathbf{R}$ in order to be able to determine the error and correct it. In other words, if e_1 and e_2 are two errors that affect the system where, $e_2 = e_1 + i\mathbf{R}$, $i = 1, 2, 3, \ldots$, then the syndrome pattern corresponding to e_1 and that corresponding to e_2 will be identical. Hence, it will not be possible to calculate the value of the error. The elements in $\mathbf{E}$ depend on the values each digit is allowed to take and the number of erroneous digits.

Example: Consider a decimal number system where $\mathbf{E}$ is the set of single digit errors (i.e. $e = i \times 10^j$ where $1 \leq i \leq 9$ and $j = 0, 1, 2, \ldots$). Table 3.1, shows the residues of different errors with respect to $m_1 = 3, m_2 = 5$ and $m_3 = 7$. From table 3.1, several observations can be made:

(a) If m_1 and m_2 were the moduli of the residue code, then the detectable errors $\mathbf{E_{det}} \in \{1, 2, \ldots, 9, 10, 20\}$, while the range of correctable errors is $\mathbf{E_{cor}} \in \{1, 2, \ldots, 9, 10\}$. Outside of $\mathbf{E_{det}}$ some errors might occur and generate a zero syndrome pattern and therefore go undetected. Similarly, outside of $\mathbf{E_{cor}}$, some errors might occur and generate a syndrome pattern that can be decoded into more than one error value.

(b) If m_2 and m_3 were the moduli of the residue code, then the detectable errors $\mathbf{E_{det}} \in \{1, 2, \ldots, 9, 10, 20, \ldots, 60\}$, while the range of correctable errors is $\mathbf{E_{cor}} \in \{1, 2, \ldots, 9, 10, 20, 30\}$.

Table 3.1: Example for range of detectable and correctable errors

e	$m_1 = 3$	$m_2 = 5$	$m_3 = 7$	e	$m_1 = 3$	$m_2 = 5$	$m_3 = 7$
1	1	1	1	20	2	0	6
2	2	2	2	30	0	0	2
3	0	3	3	40	1	0	5
4	1	4	4	50	2	0	1
5	2	0	5	60	0	0	4
6	0	1	6	70	1	0	0
7	1	2	0	80	2	0	3
8	2	3	1	90	0	0	6
9	0	4	2	100	1	0	2
10	1	0	3	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(c) If m_1 and m_3 were the moduli of the residue code, then the detectable errors $\mathbf{E_{det}} \in \{1, 2, \ldots, 9, 10, 20, \ldots\}$, while the range of correctable errors is $\mathbf{E_{cor}} \in \{1, 2, \ldots, 9, 10\}$.

3.3 Fault Tolerance for Mixed Decimal/Binary Redundant Floating point Adder

In [6] a mixed decimal/binary redundant floating adder was proposed. The previously discussed fault tolerance technique will be applied to this adder to provide it with error detection and correction capabilities. As the name suggests, this adder is designed to deal with either the IEEE decimal-64 or binary-64 formats. It uses a redundant representation for the decimal/binary numbers in order to make use of the carry free addition property of the redundant numbers to make the addition operation faster. This redundancy is obvious in the choice of the digit set to be $\{-6, -5, \ldots, 6\}$ encoded in two's complement instead of the conventional digit set $\{0, 1, \ldots, 9\}$. Hence, a certain number can be represented in more than one form. Table 3.2 lists all the numbers in the chosen digit set with their equivalent binary encodings.

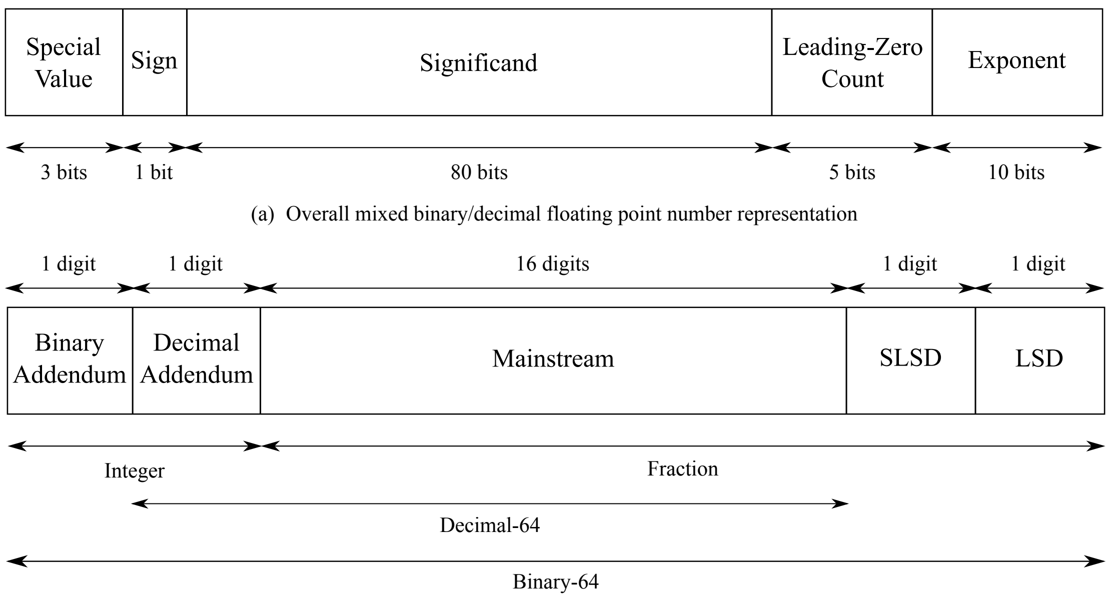
Figure 3.1 shows the internal representation of the floating point number used in [6]. As explained in section 1.5, The IEEE decimal-64 floating point format

Table 3.2: Redundant digit set representation

Digit	Binary Encoding	Digit	Binary Encoding
-6	1010	1	0001
-5	1011	2	0010
-4	1100	3	0011
-3	1101	4	0100
-2	1110	5	0101
-1	1111	6	0110
0	0000		

defines the decimal significand to be a 16-digit number with each digit belonging to the conventional digit set $[0, 9]$. This allows the significand to take a maximum value of 9999999999999999. In order to be able to represent this number using the redundant digit set proposed in [6], a 17th digit, *Addendum*, had to be appended to the sigificand as shown in figure 3.1(b). On the other hand, the IEEE binary-64 format defines the binary significand to be 52-bit fraction part in addition to a hidden integer part of one bit (always '1' fro normal numbers) called the *hidden one*. This binary number is represented in [6] using octal number representation. Therefore, two more bits are appended to the left of the hidden one to make a complete octal digit. Meanwhile, two bits are appended to the right of the fraction part to turn it into 18 complete octal digits. This octal representation can be further converted into the redundant octal representation by appending an addendum digit to the left of the integer part. The significand representation shown in figure 3.1(b) accommodates 17 redundant digit in case of a decimal operation and 20 redundant digits (2 for the integer part and 18 for the fraction part) in case of binary (octal operation).

The block diagram for the whole floating point adder is shown in figure 3.2. As explained in section 1.6, first, the exponent difference between the two operands is calculated. This difference determines the amount of shift that should be applied to the significands of the operands. In case of decimal addition, both operands might be swapped where the operand with the larger exponent might be shifted to the left and that with the smaller exponent might be shifted to the right. On



(a) Overall mixed binary/decimal floating point number representation

(b) Mixed binary/decimal significand representation

Figure 3.1: The floating point redundant representation proposed in [6]

the other hand, binary addition requires shifting (if needed) the operand with the smaller operand only since the significands are normalized. In order to decrease the area overhead of the shifter block, its design is based on the first operand (X) having a larger exponent than the second one (Y). Therefore, if the exponent difference unit indicates otherwise (the *swap* signal) the two operands have to be swapped. Special attention should be given to the significand that would be shifted to the right in order to determine the guard and round digits. For the binary format, these digits are included in the SLSD and LSD digits shown in figure 3.1(b). As for the sticky bit and sign generation, it runs simultaneously in order to decrease the delay of the adder. Afterwards, the aligned significands are added/subtracted using the *Signed-digit Decimal Redundant Adder*[*]. Then, several tests have to be carried out on the resulting significand that could affect the final result. These tests are:

- **Leading Zero Detection:** In the binary format, the resulting significand must be checked for leading zeros whose number must be counted. Based on their count, the significand should be shifted to the left to achieve normalization.

- **Negative Significand Detection:** If the resulting significand were negative,

[*]will be discussed in more detail shortly

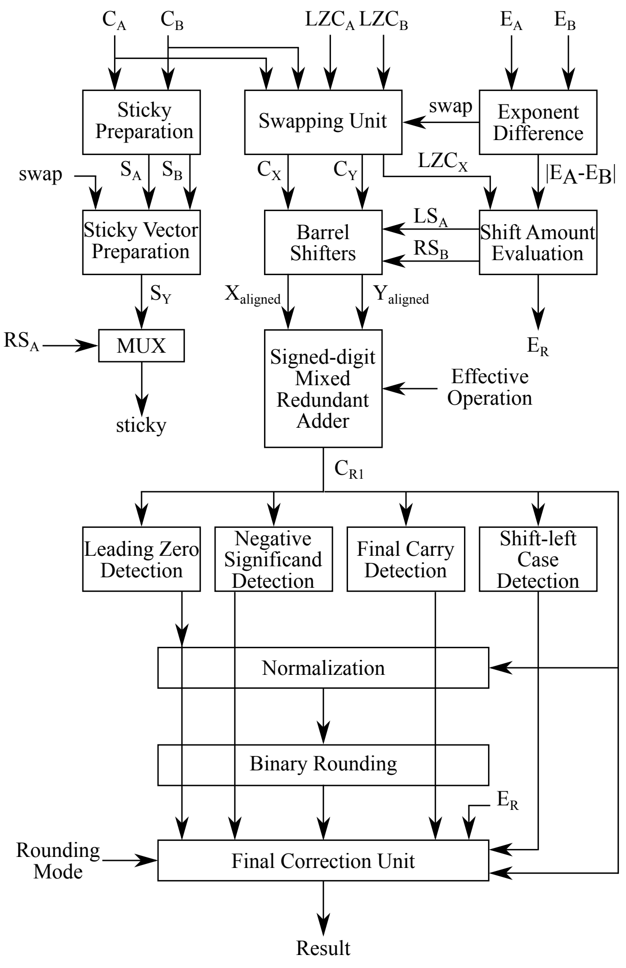


Figure 3.2: Block diagram of the Mixed floating point adder proposed in [6]

it would be represented in the complement form. Therefore, it should be converted back to its positive counterpart whereas the sign is maintained in the *sign* field of the floating point number representation.

- **Final Carry Detection:** If the resulting significand were larger than the maximum significand allowed by the the floating point format, a final carry would be generated. In this case the significand should be shifted to the right and the exponent should be increased be one. By doing that shift, the final carry becomes the most significant digit in the significand while the least significant digit is shifted out and affects the rounding process.

- **Shift-left Case Detection:** In the decimal format, it might be required to shift the resulting significand to the left in order to reach an exact representation of the result or to approach the preferred exponent (smallest possible exponent).

Table 3.3: Contribution of the different blocks to the total area of the floating point adder

Block	**Area Share**
Sticky Generation	12%
Redundant Adder	21%
Exponent Difference	3%
Swapping Unit	8%
Left, Right Shift Amount Evaluation	2%
Left Barrel Shifter	12%
Right Barrel Shifter	14%
Resulting Significand Factorization	16%
Final Block	6%
Rounding	6%

The following step is to normalize the result in case of binary addition and round it as well. Binary rounding requires special treatment due to the redundant octal representation of the binary significand. Finally, decimal rounding along with any remaining corrections are carried out to produce the final result.

The contribution of the different blocks of the floating point adder to the total area of the adder in shown in table 3.3. According to [6], the Redundant Adder block is the block with the largest area. Therefore, it is more prone to faults than other blocks and hence it was chosen to be the subject of the fault tolerance technique presented in this thesis. The building block for the redundant adder is a *Mixed Octal/Decimal Adder Cell* which deals with operands of one digit only. An Array of these cells can be used to add any two operands of arbitrary length.

3.3.1 Mixed Octal/Decimal Adder Cell

This block accepts as its inputs the two operands of the addition operation (each is one digit only), the desired operation (whether addition or subtraction), the input transfer digit from the lower digit addition and the base of the number system being used (whether decimal or binary). Thus, allowing the same hardware to be used for either decimal or octal addition operation. It then produces an output transfer digit that has a value of -1, 0 or 1 and sum that can take any value belonging to the set $\{-6, -4, \ldots, 6\}$.

The main operation of this block can be described simply as follows. It first adds the two input digits using a conventional 4-bit adder producing what is called the *interim sum*. This sum is then used to produce the *output transfer digit* (OTD) that represents the *input transfer digit* (ITD) to the next higher two digits being added. The OTD is calculated according to the following relation.

$$OTD = \begin{cases} 1 & interim\ sum \geq 6 \qquad (3.16a) \\ -1 & interim\ sum \leq -6 \qquad (3.16b) \\ 0 & otherwise \qquad (3.16c) \end{cases}$$

The transfer digits are actually represented as two signals called t_{pos} and t_{neg} where the numerical value of the transfer digits is calculated as,

$$transfer\ digit = t_{pos} - t_{neg} \qquad (3.17)$$

The final result is then calculated according to the following equation where $b = 10$ or 8 for decimal and octal systems respectively.

$$\begin{aligned} final\ result &= interim\ sum + (ITD_{from\ lower\ adder} - OTD \times b) \\ &= interim\ sum + correction\ digit \end{aligned} \qquad (3.18)$$

The *correction digit* only depends on the interim sum of the lower digits (in the form of ITD) and on the interim sum of the current digits (in the form of OTD). Therefore, noting that the interim sum itself is a function of only the input digits, it can be proved that any redundant numbers of any length can be added using a series of this mixed redundant adder without suffering from carry propagation. In fact, any addition process will be carried out in 2 addition steps; the *first* is adding corresponding digits to produce the interim sum and the *second* is adding the interim sum to the corresponding correction digit to produce the final sum.

Example: Suppose that the two decimal $(b = 10)$ numbers -4694 and -3155 are added by an array of Mixed Adder Cells with $ITD = 1$. These two numbers have redundant representations of $\bar{5}31\bar{4}$ and $\bar{3}\bar{2}55$ respectively where $\bar{x} = -x$. The result should be the redundant equivalent to

$-4694+(-3155)+1=-7848$. The addition process is performed as follows[*].

$$
\begin{array}{cccccccccccl}
 & & & & & & & & & & & ITD=1 \\
 & & & & & & & & & & \swarrow & \\
 & & & \bar{5} & & 3 & & 1 & & \bar{4} & & \\
 & & + & \bar{3} & & \bar{2} & & 5 & & \bar{5} & & \\
\hline
 & & & \bar{8} & & 1 & & 6 & & \bar{9} & & \text{interim sum} \\
 & \swarrow & & & \swarrow & & \swarrow & & \swarrow & & & \\
\bar{1} & & & 0 & & 1 & & \bar{1} & & & & \text{transfer digit} \\
 & \searrow & & & \searrow & & \searrow & & \searrow & & & \\
 & & & 10 & & 1 & & \bar{1}\bar{1} & & 11 & & \text{correction digit} \\
\hline
\bar{1} & & & 2 & & 2 & & \bar{5} & & 2 & & \text{final result}
\end{array}
$$

The final result is equivalent to $-10050+2202=-7848$ as expected.

3.3.2 Error Set and Chosen Moduli

As stated in section 3.3, the significand of the floating point number in its redundant form is formed of 17 or 20 redundant digits for the decimal and binary formats respectively. Generally, each digit can have any value in the set [-6,6]. Consequently, a digit can have an error ranging from to -12 to 12. In order to increase the range of detected and corrected errors without the need of using very large moduli, the significand is divided into groups of 4 digits. Each group is checked independently for errors. If it is assumed that an error could occur in any number of digits, then the possible errors in a 4-digit number can have any value ranging from -13332[†] to 13332[‡] for the decimal system and from -7020[§] to 7020[¶] for the binary system.

In order to cover the whole error range and at the same time have easy-to-calculate residues, moduli set of $\{999,101\}$ and $\{511,65\}$ were chosen for the decimal and octal systems respectively. For the decimal system, the product of the moduli is 100899 which, according to equation 3.15[||], is the same as the number of unique patterns formed by the residues with respect to these two moduli. This number of unique residue patterns covers the whole range of 4-digit errors in a

[*]Please recall that the maximum value for the interim sum is 5 and any larger value produces a transfer digit of 1 as indicated in equation 3.16

[†]$(-12)\times 10^3+(-12)\times 10^2+(-12)\times 10^1+(-12)=-13332$

[‡]$12\times 10^3+12\times 10^2+12\times 10^1+12=13332$

[§]$(-12)\times 8^3+(-12)\times 8^2+(-12)\times 8^1+(-12)=-7020$

[¶]$12\times 8^3+12\times 8^2+12\times 8^1+12=7020$

[||]$lcm(999,101)=999\times 101=100899$

4-digit decimal number. Similar analysis can be done with respect to the octal system to prove that the chosen moduli set produces enough residue patterns to cover the whole range of 4-digit errors in a 4-digit decimal number. Therefore, the chosen moduli sets satisfy the first and third conditions mentioned in section 3.2.1 but not the second one. Fortunately, the correction needed in order to overcome this is fairly simple as will be discussed in section 3.4.3.

The division of the significand into groups of four digits also facilitates the reuse of the same error detection and correction circuit to check for errors in different 4-digit groups. Moreover, this division and modularity in the design makes it easy to apply it on larger significands (for example in the IEEE decimal-128 and binary-128 formats). On the other hand, the independent treatment of each 4-digit group makes it harder to check for errors in the transfer digits from one group to another. This can be considered as a communication problem where various codes such as parity codes, turbo codes and many others have been developed to check the integrity of data transmission from a transmitter (a certain 4-digit group) to a receiver (the following group). However, applying these codes to the proposed design would greatly increase its area and delay overheads. Therefore, it was assumed that the transmission of the transfer digits from one 4-digit group to the other is error-free.

3.4 Block Diagram

The general block diagram for the proposed circuit is shown in figure 3.3.

3.4.1 Residue Generators

The first blocks in the checker circuit are the residue generator blocks. Two residue generators are needed for the proposed design. One is for calculating the residue with respect to 101 (in case of decimal system) and 65 (in case of binary system) while the other calculates the residue with respect to 999 (in case of decimal system) and 511 (in case of binary system). These two residue generators are explained in detail in the following subsections.

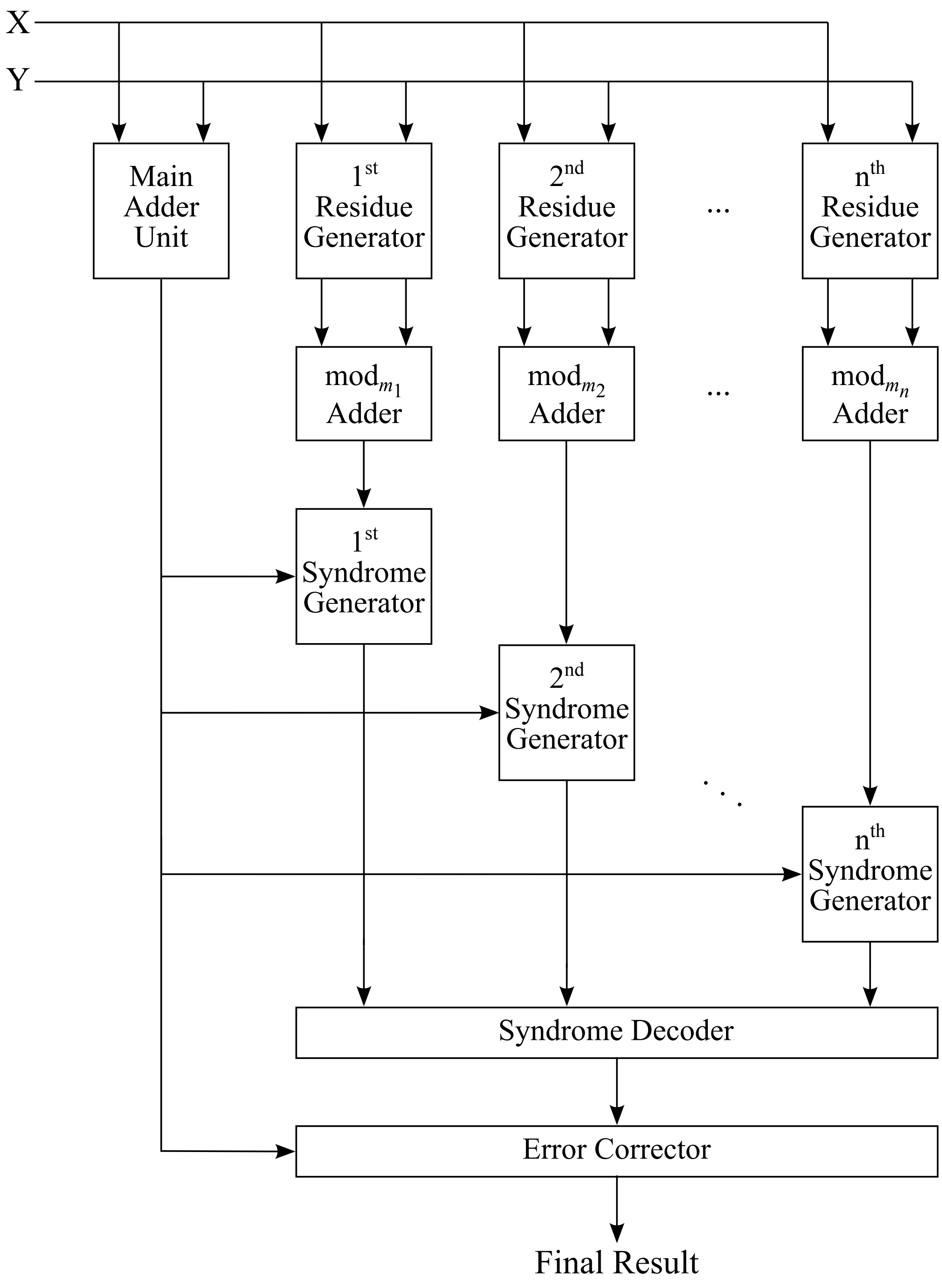


Figure 3.3: Block diagram of the proposed design

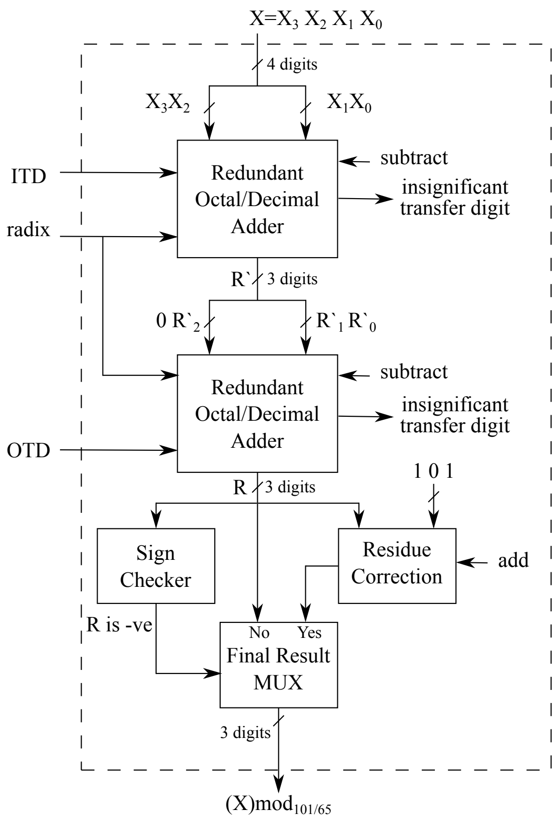


Figure 3.4: mod$_{101/65}$ residue generator block diagram

mod$_{101/65}$ Residue Generator

This block, shown in figure 3.4, calculates the residue of a 4-digit number with respect to either 101, in case of working in a decimal number system (base-10 system), or 65 in case of working in a binary/octal number system (base-8 system). The number system for the ongoing operation in determined by the *radix* signal. If *radix* $=$ ‘0’, then the current operation is a binary/octal operation and this block performs mod$_{65}$ operation. On the other hand, If *radix* $=$ ‘1’, then the current operation is a decimal operation and this block performs mod$_{101}$ operation.

According to equation 3.11, residues with respect to moduli $10^2 + 1$ or $8^2 + 1$ can be calculated by dividing the 4-digit number into two 2-digit numbers and subtracting the upper two digits from the lower two digits. In the redundant number system used in [6], a 2-digit decimal number can take any value from -66 to 66. Therefore, subtracting two 2-digit numbers from each other gives a result that ranges from -122 to 122. Similarly, a 2-digit octal number can take any value from* -54 to 54 giving a subtraction result that ranges from -108 to 108. As a result, it might be needed to perform the previous process (calculating the residue) one more time on the result of the first subtraction.

In this work all residues are assumed to be positive, therefore the output from this block would be a 3-digit number in the range from 0 to 100 (in case of decimal number system) or from 0 to 64 (in case of binary/octal number system). If the calculated residue is less than zero then a correction digit of 101 or 65 should be added to that calculated residue. It is worth mentioning that 65 is represented as 101 in an octal number system.

Example 1: $\bar{4}\bar{5}65\text{mod}_{101} = (65-(\bar{4}\bar{5}))\text{mod}_{101}$ (where, $\bar{x} = -x$)

$= 110\text{mod}_{101}$

$= (10-1)\text{mod}_{101} = 1\bar{1})_{101}$

Example 2: $65\bar{4}\bar{5}\text{mod}_{101} = (\bar{4}\bar{5}-65)\text{mod}_{101}$

$= (\bar{1}\bar{1}0)\text{mod}_{101}$

$= (\bar{1}0-\bar{1})\text{mod}_{101}$

$= \bar{1}1$ $(< 0, \therefore \text{add } 101)$

$= 1\bar{1}2)_{101}$

Two extra inputs are needed by this block. They are the *output transfer digit* OTD_{in} and the *input transfer digit* ITD_{in}. The need for these two inputs will be explained in section 3.4.3.

The previously proposed architecture for the residue generator can be further simplified to decrease both its area and delay overhead. The simplified block diagram for the residue generator is shown in figure 3.5. The main simplifications are:

- **Combining the first and second subtraction steps:**
  Suppose the 4-digit number is $X = x_3x_2x_1x_0$, then as mentioned before, the first subtraction step subtracts the upper two digits (x_3x_2) from the lower two digits (x_1x_0). The subtraction is carried out using two mixed adder cells. the first one has the operands x_0 and x_2, input transfer digit ITD_0 and generates

*-54 and 54 are the decimal equivalents to the octal numbers $\bar{6}\bar{6}$ and 66 respectively

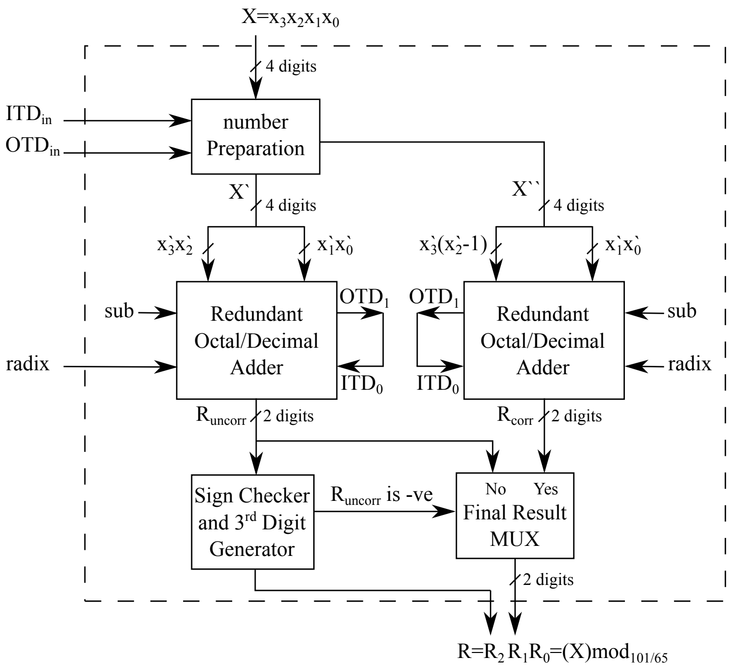


Figure 3.5: Simplified $\text{mod}_{101/65}$ residue generator block diagram

one result digit r_0 and an output transfer digit OTD_0. The second one has the operands x_1 and x_3, input transfer digit ITD_1 and generates one result digit r_1 and an output transfer digit OTD_1 If the result is allowed to occupy four digits, then the lower two digits would be the subtraction result (r_1r_0), the third digit would be OTD_1 and the fourth one would be zero. Consequently, the second subtraction step would effectively be subtracting the OTD_1 from r_3r_2. Therefore, the two subtraction steps can be combined by setting the ITD_0 to $-OTD_1$ and thus performing both subtraction steps in only one step and decreases the area overhead of the residue generator by the area of one subtracter.

In order to show that this modification would also decrease the delay overhead of the residue generator, it should be recalled from section 3.3.1 that the $OTDs$ depend only on its corresponding operands in the form of their interim sum. Moreover, the $ITDs$ affect the final result through the correction digit only whereas it does not affect the interim sum. Hence, the combined subtraction operation would have the following sequence:

1. Each digit is subtracted from its corresponding digit generating two interim sums.

2. OTD_1 can then be deduced from the upper interim sum.
3. ITD_0 is set to $-OTD_1$ and the two correction digits are calculated
4. The final result is calculated

Hence, the combined subtraction operation has an equivalent delay to one level of mixed adder cells compared to an equivalent delay of two levels of mixed adder cells before combination.

It is worth mentioning that the result after performing the two subtractions is represented in two digits only. Thus, the OTD is not a part of the final result from the combined subtraction step. In order to prove this, the different scenarios of the two subtractions are considered. In general, an OTD of 1 means that the interim sum is between 6 and 12[*] resulting in a final sum of ranging from -4 to 2 and in case $ITD \neq 0$, this range becomes from $-4+ITD$ to $2+ITD$. Similarly, an OTD of -1 means that the interim sum is between -6 and -12 resulting in a final sum of ranging from 4 to -2 and in case $ITD \neq 0$, this range becomes from $4+ITD$ to $-2+ITD$. The different scenarios for the two subtractions can be summarized as follows where $ITD_0 = 0$ for both subtraction steps :

1. If $OTD_1 = 1$ and $OTD_0 = ITD_1 = 1$, this implies that $r_1 \in \{-3, \ldots, 3\}$ and $r_0 \in \{-4, \ldots, 2\}$. Therefore, the second subtraction subtracts 1 from r_0 giving a result $r_0^{\backslash} \in \{-5, \ldots, 1\}$ and an $OTD_0^{\backslash} = 0$ while zero is subtracted from r_1 with $ITD_1^{\backslash} = 0$ leaving r_1 unchanged (i.e. $r_1^{\backslash} = r_1$) and $OTD_1^{\backslash} = 0$. Thus, the final result is given by $r_1^{\backslash} r_0^{\backslash}$. Similar analysis can be done for the case where $OTD_1 = OTD_0 = -1$.
2. If $OTD_1 = 0$, then the result from the second subtraction is the same as the result form the first one and represented in two digits.
3. If $OTD_1 = 1$ and $OTD_0 = -1$, then $r_1 \in \{-5, \ldots, 1\}$ and $r_0 \in \{4, \ldots, -2\}$. After the second subtraction, $r_0^{\backslash} \in \{3, -3\}$ and $OTD_0^{\backslash} = 0$, hence $r_1^{\backslash} = r_1$. Thus, the final result is given by $r_1^{\backslash} r_0^{\backslash}$. Similar analysis can be done for the case where $OTD_1 = -1$ and $OTD_0 = 1$.

[*]For the redundant digit set, the maximum addition result is 6+6=12

4. If $OTD_1 = 1$ and $OTD_0 = 0$, then $r_1 \in \{-4,\ldots,2\}$ and $r_0 \in \{-5,\ldots,5\}$. After the second subtraction, $r_0^{\backslash} \in \{-4,\ldots,4\}$[*] and $OTD_0^{\backslash}$ might be 0 or -1. Hence, $r_1^{\backslash} \in \{-5,\ldots,2\}$ and $OTD_1^{\backslash} = 0$. Thus, the final result is given by $r_1^{\backslash} r_0^{\backslash}$. Similar analysis can be done for the case where $OTD_1 = -1$ and $OTD_0 = 0$.

- **Simultaneous calculation of the corrected result:**
  Instead of waiting for the outcome of the subtraction operations to be generated and then perform the correction step, the corrected and uncorrected residues can be calculated simultaneously. The lower two digits of the corrected result are calculated by adding one to the subtraction operation. In the proposed design, this is achieved by subtracting 1 from x_2 before performing the subtraction. Then based on the sign of the uncorrected result, one of them is chosen to be the lower two digits of the final residue. The uncorrected residue is chosen if the sign is positive while the corrected residue is chosen if the sign is negative. As mentioned in the previous point, the uncorrected residue is a 2-digit number. Therefore, the third digit of the final residue is either 0, if the sign of the uncorrected result were positive, or 1, if it were negative.

The *Number Preparation* block in figure 3.5 is responsible for taking the effect of ITD_{in} and OTD_{in} into consideration by calculating a new value for x_0 accordingly. Moreover, this block calculates the value of $x_2 - 1$ to be used in calculating the corrected residue.

Example 3: The residue calculation of $\bar{4}\bar{5}65$ with respect to 101 would be performed as follows:

$$\begin{tikzcd}
 & 6 & 5 & \\
 & -\ \bar{4} & \bar{5} & \\
 & 10 \arrow[dl] & 10 \arrow[dl] & \text{interim sum} \\
ITD_0 = OTD_1 = 1 \arrow[dr] & 1 \arrow[dr] & & \text{transfer digit} \\
 & \bar{9} & \overline{11} & \text{correction digit} \\
 & 1 & \bar{1} & \text{final result}
\end{tikzcd}$$

[*]If the interim sum is -6 then the final result becomes 4 with $OTD = -1$

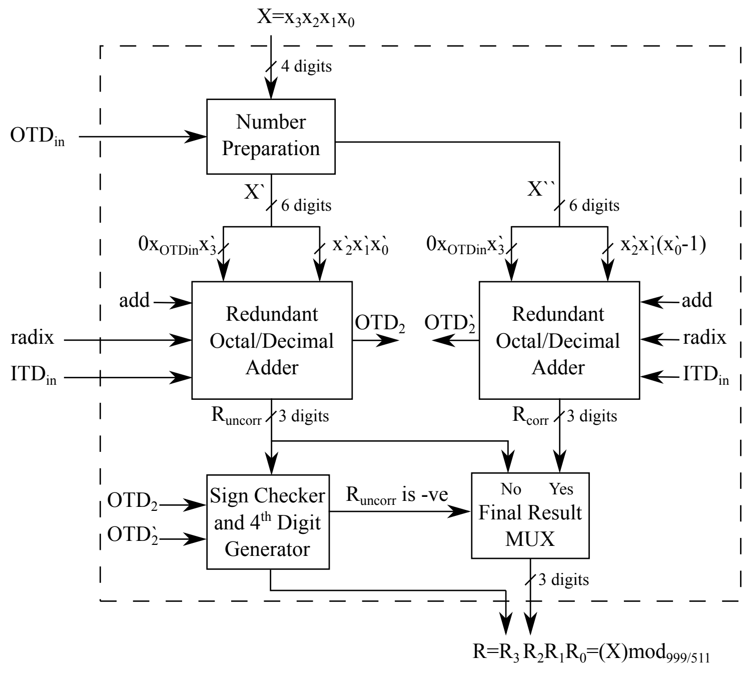


Figure 3.6: $\text{mod}_{999/511}$ residue generator block diagram

$\text{mod}_{999/511}$ Residue Generator

This block, shown in figure 3.6, is very similar to the $\text{mod}_{101/65}$ residue generator block. The difference is that this block calculates the residue of a 4-digit number with respect to either 999, in case of working in a decimal number system (base-10 system), or 511 in case of working in a binary/octal number system (base-8 system). According to equation 3.10, residues with respect to moduli $10^3 - 1$ or $8^3 - 1$ can be calculated by adding the most significant digit to the least significant three digits. In the redundant number system used in [6], a 3-digit decimal number can take any value from -666 to 666 while a single digit number ranges from -6 to 6. Therefore, adding a 3-digit number to a single digit number gives a result that ranges from -672 to 672. Similarly for the octal system, the addition result ranges from* -444 to 444. This means that no further residue calculation is needed. If the calculated residue is less than zero, then a correction number of 999 or 511 (both represented as $100\bar{1}$ in their corresponding system) should be added to that calculated residue. In order to decrease the delay caused by this block, the corrected residue is calculated in parallel with the uncorrected one then one of them is chosen based on the sign of the uncorrected residue.

*$-444 = -438 - 6$ and $444 = 438 + 6$, where -438 and 438 are the decimal equivalents to the octal numbers $\bar{6}\bar{6}\bar{6}$ and 666 respectively.

The final residue is a number ranging from 0 to 998 (in case of decimal number system) and 510 (in case of binary/octal number system). In order to represent this number using the redundant number system described in [6], four digits are needed. The lower three digits are the result from the redundant adders (either R_{uncorr} or R_{corr}) while the fourth is the output transfer digit (either OTD_2 or $OTD_2^{\backslash} + 1$).

The *Number Preparation* block is responsible for preparing the operands of the redundant adders for the uncorrected and corrected branches. The operands of the two branches are almost the same except for x_0 which becomes $x_0 - 1$ for the corrected branch. OTD_{in} is taken into consideration through calculating $x_{OTD_{in}}$ while ITD_{in} is considered as a direct input to the redundant adders.

Example 1:
$$\begin{aligned} 4565\text{mod}_{999} &= (565+4)\text{mod}_{999} \\ &= 6\bar{3}\bar{1})_{999} \end{aligned}$$

Example 2:
$$\begin{aligned} \bar{4}\bar{5}65\text{mod}_{999} &= (\bar{5}65+\bar{4})\text{mod}_{999} \\ &= \bar{5}61 \qquad\qquad (<0, \therefore \text{add } 100\bar{1}) \\ &= 560)_{999} \end{aligned}$$

3.4.2 Residue Adders

After calculating the residues, they are added/subtracted according to the operation being performed. Therefore two blocks are needed, one for the moduli of 101/65 while the other is for the moduli of 999/511

mod$_{101/65}$ Residue Adder Block

This block, shown in figure 3.7, is responsible for adding/subtracting the residues of two 4-digit numbers with respect to 101 or 65 and then calculating the residue of the result with respect to 101 or 65. In other words, if X and Y are two 4-digit numbers, then this block calculates $Z_{101/65} = (X_{101/65} \pm Y_{101/65})\text{mod}_{101/65}$, where $X_{101/65} = X\text{mod}_{101/65}$ and $Y_{101/65} = Y\text{mod}_{101/65}$. The operation to be performed is determined by the *operation* signal where ‘0’ and ‘1’ mean addition and subtraction respectively. Both operands of this block are 3-digit numbers ranging

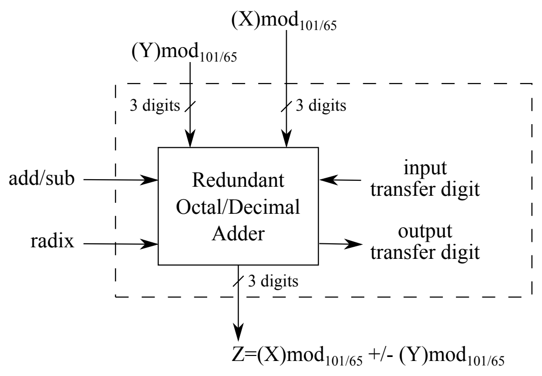


Figure 3.7: $\text{mod}_{101/65}$ residue adder block

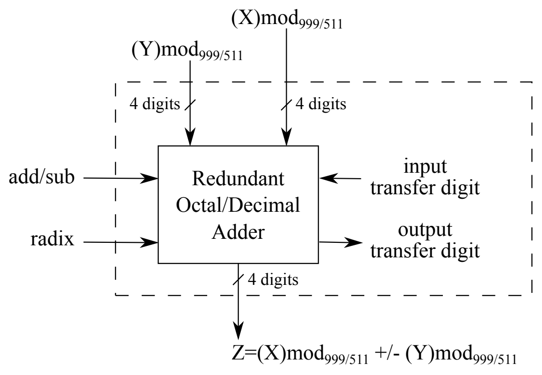


Figure 3.8: $\text{mod}_{999/511}$ residue adder block

form 0 to 100/64 for decimal and binary numbers respectively. Therefore, the result ranges from -100/-64 to 200/128 and hence can be accommodated in three digits as well. After adding/subtracting the two residues, the residue of the result is calculated by applying the same technique as in the $\text{mod}_{101/65}$ residue generator to a 3-digit number instead of a 4-digit one.

$\text{mod}_{999/511}$ Residue Adder Block

This block, shown in figure 3.7, is almost the same as its $\text{mod}_{101/65}$ counterpart except that its operands are 4-digit numbers ranging form 0 to 998/510. Therefore, the result ranges from -998/-510 to 1996/1020 and hence can be accommodated in four digits as well. After adding/subtracting the two residues, the residue of the result is calculated by applying the result to the $\text{mod}_{999/511}$ block.

It is worth mentioning that the residue calculation step after performing the addition/subtraction process in the previous two blocks is postponed until the syndrome pattern is generated. As explained in the next section, another residue calculation step is needed while generating the syndrome pattern. Therefore, both

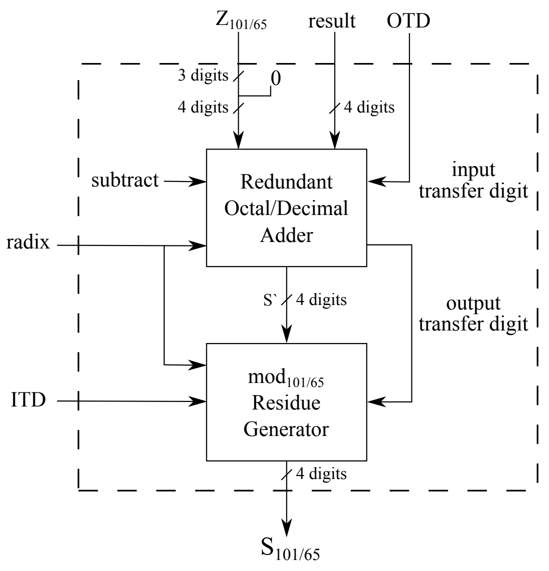


Figure 3.9: $\text{mod}_{101/65}$ syndrome generator block

residue calculation step can be combined into one calculation step only and hence removes some unnecessary steps. By applying this simplification, the only difference between the two residue adders is that one of them is a 3-digit adder while the other is 4-digit adder.

3.4.3 Syndrome Generators

In general, the syndrome is calculated according to equation 3.1. In the context of the proposed design, the syndrome calculation becomes,

$$\begin{aligned}
&\mathbf{S}[result, Z_{101/65}, Z_{999/511}] \\
&\quad = \big((result - Z_{101/65})\text{mod}_{101/65}, (result - Z_{999/511})\text{mod}_{999/511}\big) \qquad (3.19)\\
&\quad = (S_{101/65}, S_{999/511})
\end{aligned}$$

where, *result* is the result of the addition/subtraction generated by the main adder, $Z_{101/65}$ and $Z_{999/511}$ are the results generated by the residue adders. Two syndrome generators were designed, one to calculate $S_{101/65}$ and the other to calculate $S_{999/511}$.

Syndrome Generator with respect to 101/65

This block, shown in figure 3.9, is responsible for calculating the syndrome of the actual result with respect to 101 or 65. By applying the conditions discussed in section 3.2.1, it can be proved that the $\text{mod}_{101/65}$ system is not closed under addition using the 4-digit redundant adder. Therefore, a correction has to be applied when generating the syndrome. This correction can be achieved by taking the output transfer digit, generated by the main adder when adding the two 4-digit numbers under study, into consideration. Moreover, when adding two general 4-digit parts of the signifcands (general means not the least significant 4 four digits), the main adder adds these two parts together with the input transfer digit calculated from the addition of the preceding 4-digit pair. This means that the input transfer digit must also be taken into consideration when calculating the syndrome.

In order to determine how the output and input transfer digits affect the syndrome calculation, the syndrome is first calculated without taking them into consideration to find out how they affect the value of the calculated syndrome. Suitable actions are then taken to counteract these effects in order to get the expected syndrome which is $E\text{mod}_{101/65}$ (according to equation 3.3). Let OTD be the output transfer digit, ITD be the input transfer digit, $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ be the two 4-digit numbers to be added, $Z = z_3z_2z_1z_0$ be the result of the addition process and $Z^{\backprime} = z^{\backprime}_3z^{\backprime}_2z^{\backprime}_3z^{\backprime}_0$ be the result of addition ignoring ITD. Then, the main adder performs the following calculation

$$\begin{array}{rr} & x_3x_2x_1x_0 \\ + & y_3y_2y_1y_0 \\ \hline OTD^{\backprime} & z^{\backprime}_3z^{\backprime}_2z^{\backprime}_3z^{\backprime}_0 \\ + & ITD \\ \hline OTD & z_3z_2z_1z_0 \end{array}$$

On the other hand, the operation performed by the corresponding residue adder will be $X\text{mod}_{101/65} + Y\text{mod}_{101/65} = (OTD^{\backprime} \times b^4 + Z^{\backprime})\text{mod}_{101/65}$. If the syndrome was calculated without taking ITD and OTD into consideration, it would take the following form.

$$\begin{aligned}\mathbf{S} &= (Z + error - OTD^{\backslash} \times b^4 \text{mod}_{\beta^{\backslash}} - Z^{\backslash}\text{mod}_{\beta^{\backslash}})\text{mod}_{\beta^{\backslash}} \\ &= (error + Z^{\backslash} + ITD - (OTD^{\backslash} \times b^4)\text{mod}_{\beta^{\backslash}} - Z^{\backslash}\text{mod}_{\beta^{\backslash}})\text{mod}_{\beta^{\backslash}} \\ &= (error + ITD - (OTD^{\backslash} \times b^4\text{mod}_{\beta^{\backslash}}))\text{mod}_{\beta^{\backslash}} \\ &= (error)\text{mod}_{\beta^{\backslash}} + ITD - OTD^{\backslash}\end{aligned} \tag{3.20}$$

where $b = 10$ and $\beta^{\backslash} = 101$ for decimal number system and $b = 8$ and $\beta^{\backslash} = 65$ for octal number system. Using the property of no carry propagation in the redundant adder in [6], $OTD^{\backslash}$ can be replaced by OTD. Therefore the syndrome expression becomes,

$$\mathbf{S} = (error)\text{mod}_{\beta^{\backslash}} + ITD - OTD \tag{3.21}$$

This means that ITD must be subtracted from the syndrome and OTD must be added to it in order to be able to use the syndrome to calculate and correct the error. The OTD correction is simply a way of dealing with a 5-digit number using only 4-digit residue and syndrome generators. Therefore, two of such correction might be needed. One is to deal with the OTD from the main adder while the other is for the OTD generated from the subtraction operation during syndrome generation. In this design, the needed corrections are distributed among the subtraction and residue generation operation of the syndrome generation block. This is the reason of having the ITD_{in} and OTD_{in} as inputs to the residue generator block. These two inputs are only used in this case, otherwise they are zero. As explained in section 3.4.1, these two inputs are accounted for prior to residue calculation. For the sake of optimization, the residue generation step can be postponed and combined with another one in the syndrome decoder block.

Example: Let 4324 and 4521 be the two decimal numbers to be added with $ITD = 0$, then the correct result should be $\bar{1}\bar{2}45$ with $OTD = 1$. If there has been an error of $\bar{1}$ leading to an incorrect result of $\bar{1}\bar{2}44$ with $OTD = 1$, then syndrome calculation would be as follows:

The residues of the operands can be calculated to be:

$4324\text{mod}_{101} = 1\bar{2}2)_{101}$ $\qquad$ $4521\text{mod}_{101} = 1\bar{2}\bar{3})_{101}$

Then the result of the residue adder would be:

$1\bar{2}2)_{101} + 1\bar{2}\bar{3}_{101} = 1\bar{4}\bar{2})_{101}$

and the syndrome will be calculated as:

$$
\begin{aligned}
\mathbf{S} &= (\bar{1}\bar{2}44 - 1\bar{4}\bar{2} + 1)\mathrm{mod}_{101} \\
&= (\bar{1}\bar{2}\bar{2}6 + 1)\mathrm{mod}_{101} \\
&= (\bar{1}\bar{2}\bar{1}\bar{3})\mathrm{mod}_{101} \\
&= (\bar{1}\bar{3} - \bar{1}\bar{2})\mathrm{mod}_{101} \\
&= (\bar{1})\mathrm{mod}_{101} \\
&= \bar{1})_{101}
\end{aligned}
$$

$\therefore (error)\mathrm{mod}_{101} = \bar{1})_{101}$

Syndrome Generator with respect to 999/511

This block, shown in figure 3.10, is responsible for calculating the syndrome of the result with respect to 999 or 511. It can also be proved that the $\mathrm{mod}_{999/511}$ system is not closed under addition using the 4-digit redundant adder. Therefore, some corrections have to be applied when calculating the syndrome. These corrections can be determined by following the same method as in equation 3.20 but in this case $\beta' = 999$ and 511 for decimal and octal number systems respectively.

$$\mathbf{S} = (error)\mathrm{mod}_{\beta'} + ITD - OTD \times b \tag{3.22}$$

This means that ITD must be subtracted from the syndrome and $b \times OTD$ must be added to it in order to be able to use the syndrome to calculate and correct the error. Similar to the previous syndrome generator, two OTD corrections might be needed and the correction are distributed among the subtraction ans residue generation steps. As mentioned in section 3.4.1, the $\mathrm{mod}_{999/511}$ block needs only one stage of adders before the range correction stage. This adder adds the three *Least Significant* digits, $LS = ls_2 ls_1 ls_0$, to the *Most Significant* digit, $MS = ms_2 ms_1 ms_0$. The input transfer digit of this adder is set to $-ITD$. The two digits, ms_2 and ms_1 are normally zero, but in order to perform the required corrections in syndrome

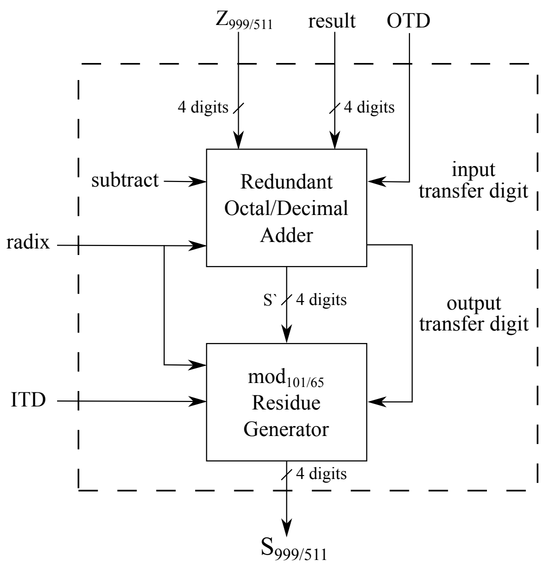


Figure 3.10: $\text{mod}_{999/511}$ syndrome generator block

calculation, ms_1 depends on OTD as explained in equation 3.23.

$$ms_1 = \begin{cases} 1 & \text{if } OTD = 1 \\ -1 & \text{if } OTD = -1 \\ 0 & otherwise \end{cases} \tag{3.23}$$

Example: Let 5324 and 4521 be the two decimal numbers to be added with $ITD = 1$, then the correct result should be $0\bar{2}46$ with $OTD = 1$. If there has been an error of $\bar{1}\bar{1}$ leading to an incorrect result of $0\bar{2}4\bar{5}$ with $OTD = 1$, then syndrome calculation would be as follows:

The residues of the operands can be calculated to be:

$5324\text{mod}_{999} = 33\bar{1})_{999}$ $\qquad$ $4521\text{mod}_{999} = 525)_{999}$

Then the result of the residue adder would be:

$33\bar{1})_{999} + 525)_{999} = 1\bar{2}54)_{999}$

and the syndrome before correction will be calculated as:

$$\mathbf{S}^{\backprime} = (0\bar{2}4\bar{5} - 1\bar{2}54)\text{mod}_{999}$$
$$= \bar{1}0\bar{2}1)_{999}$$

$\therefore LS = 0\bar{2}1, \quad \text{and} \quad MS = 01\bar{1} \quad (OTD = 1)$

$$
\begin{aligned}
\mathbf{S} &= (LS + MS - ITD)\text{mod}_{999} \\
&= (0\bar{2}1 + 01\bar{1} - 1)\text{mod}_{999} \\
&= \bar{1}\bar{1})_{999} \\
\therefore (error)\text{mod}_{999} &= \bar{1}\bar{1})_{101}
\end{aligned}
$$

3.4.4 Syndrome Decoder

This block, shown in figure 3.11, is responsible for decoding the syndrome pattern calculated by the *Syndrome Generator* blocks into the actual error. The calculated error is then used to obtain the correct result in the *Result Correction* block. Several methods for residue conversion are explained in section 2.2.1. For this design, the *MATR* method described in [55] is used. This method offers reduction in needed arithmetic operations to perform the conversion compared to other methods as the *Chinese Remainder Theorem (CRT)* and the *Mixed Radix Conversion (MRC)*. Therefore, using the *MATR* method improves this design from the point of view of complexity, area and speed.

For a residue number system based on two moduli, m_1 and m_2, the equivalent number, X, to the residue pair, (r_1, r_2), can be calculated as:

$$X = p_1 + m_1 \times ((m_1)^{-1}\text{mod}_{m_2} \times t_1)\text{mod}_{m_2} \tag{3.24}$$

where, $p_1 = r_1, t_1 = (r_2 - p_1)\text{mod}_{m_2}$ and $(m_1)^{-1}\text{mod}_{m_2}$ is the modular multiplicative inverse[*] of m_1 with respect to m_2.

For the decimal correction system being investigated, m_1 can be either 999 or 101 and the same for m_2 leading to the following two cases:

Case 1: If $m_1 = 999$ and $m_2 = 101$, then equation 3.24 becomes,

$$t_1 = (r_2 - p_1)\text{mod}_{101} \tag{3.25}$$

$$X = p_1 + 999 \times (55 \times t_1)\text{mod}_{101} \tag{3.26}$$

[*]If q is the modular multiplicative inverse of a certain number, n, with respect to a certain modulus, m, then $(nq)\text{mod}_m = 1$

Case 2: If $m_1 = 101$ and $m_2 = 999$, then equation 3.24 becomes,

$$t_1 = (r_2 - p_1)\text{mod}_{999} \tag{3.27}$$

$$X = p_1 + 101 \times (455 \times t_1)\text{mod}_{999} \tag{3.28}$$

Similarly, For the octal correction system the two cases become:

Case 1: If $m_1 = 511$ and $m_2 = 65$, then equation 3.24 becomes,

$$t_1 = (r_2 - p_1)\text{mod}_{65} \tag{3.29}$$

$$X = p_1 + 511 \times (36 \times t_1)\text{mod}_{65} \tag{3.30}$$

Case 2: If $m_1 = 65$ and $m_2 = 511$, then equation 3.24 becomes,

$$t_1 = (r_2 - p_1)\text{mod}_{511} \tag{3.31}$$

$$X = p_1 + 65 \times (228 \times t_1)\text{mod}_{511} \tag{3.32}$$

For the design presented in this thesis, the first case was the one implemented for each correction system.

The second term, p_2, of the addition operation in equations 3.24 is calculated using a *Look Up Table (LUT)*. This LUT is addressed with t_1 and returns the value of p_2. Two LUTs are needed; one for the decimal system and the other for the octal system. The size of the LUT depends on t_1 (which determines the number of entries in the LUT) and the number of digits in p_2. The latter is the same for all cases where the maximum value of p_2 is 99900 and 100899 for the first and second cases of the decimal system respectively and 32704 and 33150 for the first and second cases of the octal system respectively. All of these numbers (99900, 100899, 32704 and 33150) are represented by six digits in the redundant number system used in [6]. Therefore t_1 becomes the dominant factor in favoring one case over the other. The smaller the number of values that t_1 can take, the smaller the size of the LUTs. The range of t_1 is determined by the moduli of the residue systems. Hence, the first case for both correction systems needs a smaller LUT than the second case. The expected sizes for the LUTs for the decimal and octal

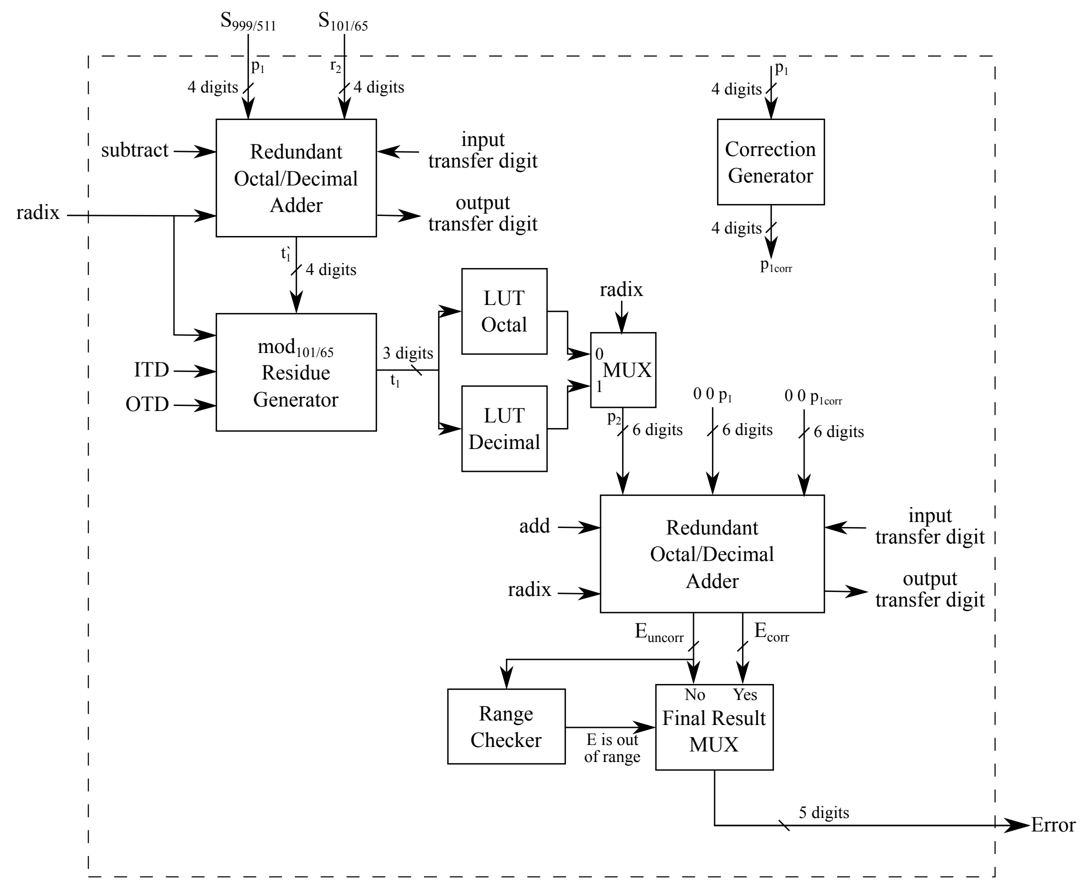


Figure 3.11: Syndrome decoder block

Table 3.4: Expected Sizes for the LUT for the decimal correction system

m_1	m_2	Number of t_1 values	Size of p_2 (in bits)	Size of LUT
999	101	101	24	2424 bits = 303 Bytes
101	999	999	24	23976 bits = 2997 Bytes

correction systems are shown in tables 3.4 and 3.5 respectively. The actual size will be larger than the expected ones due to the redundancy in the used number system.

The error calculated by the above mentioned method will be always positive. In order to account for the negative values of the error, a correction has to be made if the calculated error is larger than the maximum allowed positive value. This correction is made by subtracting either 101×999 or 65×511 from the calculated error for the decimal and octal systems respectively. This correction step is performed in parallel to the normal path in order to decrease the overall delay.

Table 3.5: Expected Sizes for the LUT for the octal correction system

m_1	m_2	Number of t_1 values	Size of p_2 (in bits)	Size of LUT
511	65	101	24	2424 bits = 303 Bytes
65	511	999	24	23976 bits = 2997 Bytes

Example: For the decimal correction system, if the syndrome pattern to be decoded is $(6\bar{2}2, 1\bar{1}\bar{2})$ where the first number is the syndrome with respect to 999 and the second one is with respect to 101. Then the process of syndrome decoding will be as follows:

$$p_1 = 6\bar{2}2$$

$$t_1 = (1\bar{1}\bar{2} - 6\bar{2}2)\text{mod}_{101} = 11$$

$$p_2 = 100\bar{1} \times (55 \times 11)\text{mod}_{101} = 100\bar{1}00 \qquad \text{(from LUT)}$$

$$error^{\backprime} = p_1 + p_2 = 6\bar{2}2 + 100\bar{1}00 = 100\bar{5}\bar{2}2 > 50449$$

$$\therefore error = error^{\backprime} + 101\bar{1}0\bar{1} = \bar{1}6\bar{2}3$$

$$\text{check:} \quad \bar{1}6\bar{2}3\text{mod}_{999} = 6\bar{2}2, \qquad \bar{1}6\bar{2}3\text{mod}_{999} = 1\bar{1}\bar{2}$$

3.4.5 Result Correction

In this final step, the error values form the different 4-digit groups are added together taking the weight of each error into consideration. The total error is then applied as one operand to the main adder with the second operand being the original erroneous result and the operation is set to subtraction.

It is worth mentioning that for this design specifically, the syndrome decoder can be used to obtain the result directly and not the error that occurred. This is achieved by setting the inputs of the syndrome decoder to be the be the outputs from the modular adders[*] instead of the syndrome pattern. The output of the modular adders are the residues of the result with respect to their corresponding moduli. Therefore, the syndrome decoder would be able to decode these residues into the actual result. This modification would work in this specific design because the range of possible results is already covered in the range of the allowable

[*]It is assumed here that the modular adders include both the addition/subtraction and the result residue calculation steps.

errors. Therefore, any value of the result has its unique residue pattern. On the contrary, if the error range did not cover the result range, not all values of the result would have their corresponding residue pattern. This could be the case if another error model was assumed,for example a single digit error model, instead of the all-digit error model assumed in this thesis. However, since the proposed design does not check whether the ITD to a certain 4-digit group is correct or not, the error might propagate from one group to another. Fortunately, this propagation does not go beyond the next group since the transfer digits depend only on their corresponding groups and not on all previous groups (no carry propagation). Hence, if a certain block in erroneous, both of its and its next group results should be corrected. The result of the next group would be erroneous if its ITD (OTD from the preceding block) is erroneous.

3.4.6 Conclusion

In this chapter, the main methodology for using residue codes in error detection and correction was discussed together with the main factors affecting the choice of the moduli for these codes. A mixed decimal/binary floating point adder [6] was then explained. This adder represents the main unit to which the proposed technique is applied. The different building blocks of the proposed design were then explained where two methods were proposed for the result correction.

Chapter 4

Results and Comparisons

4.1 Results

4.1.1 Functional Verification

This design was implemented using VHDL and simulated using the free ghdl simulator. The complete test vectors space for the proposed 4-digit unit consists of:

- 28561 possibilities[*] for each of the two operands and the result.
- 16 different combination for the input and output transfer digits, since each transfer digit is represented in two bits.
- Two possible operations, whether addition or subtraction.
- Two possible number systems, whether binary or decimal.

Consequently, the workspace consists of $28561^3 \times 16 \times 2 \times 2 = 1.491 \times 10^{15}$ test vectors. The average simulation time on a system using Intel® Core™ 2 Duo 2.13 GHz processor and 4 GB RAM is 3.3 msec/test vector. The time needed to test the design on all possible test vectors would then be 156021 years. Therefore only a subset of 17006112 out of the 1.491×10^{15} test vectors were tested. This subset was randomly chosen and the 4-digit checker passed all of its test vectors successfully. The test vectors were generated via a c++ program that mimics the desired behavior of the checker circuit. The chosen subset included both the

[*]Each of the two operands and the result is a 4-digit number and each digit can take any value from the 13 possible values from -6 to 6. Therefore, each number can have one of $13 \times 13 \times 13 \times 13 = 28561$ possible values

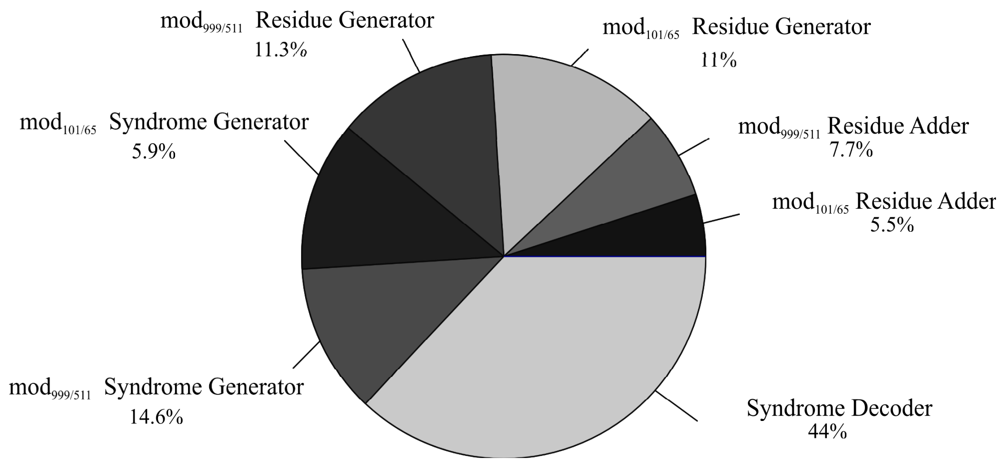


Figure 4.1: Area profiling of the proposed 4-digit error checker

binary and decimal systems for both addition and subtraction operations with different values for the input and output transfer digits.

4.1.2 Synthesis Results

The design has been synthesized using Synopsys Design Compiler and the TSMC 65 nm LP technology. The proposed error correction process can be divided into two main stages. The first stage runs in parallel with the main adder and includes the residue generation and modular addition steps. On the other hand, the second stage depends on the results from the main adder and hence can not start its operation until these results are generated. This stage includes the syndrome generation and syndrome decoding steps. As a result the constraints on the synthesis of both stages are different. The first stage synthesis should be optimized to occupy minimum area as long as its delay is less than or equal to that of the main adder. Conversely, the second stage should be optimized to have minimum delay in order to obtain the corrected result as soon as possible. By applying the before mentioned constraints to the 4-digit checker, the first stage was found to occupy an area of 2396.16 μm^2 and has a delay of 1.98 nsec. The second stage was found to occupy an area of 4206.24 μm^2 and has a delay of 5.17 nsec.

The original floating point adder was also synthesized and found to occupy an area of 16481 μm^2 and introduce a delay of 6.96 nsec. The large checker area (compared to the original system being checked) and the long delay introduced by the checker represent a great challenge to efficiently use information redundancy

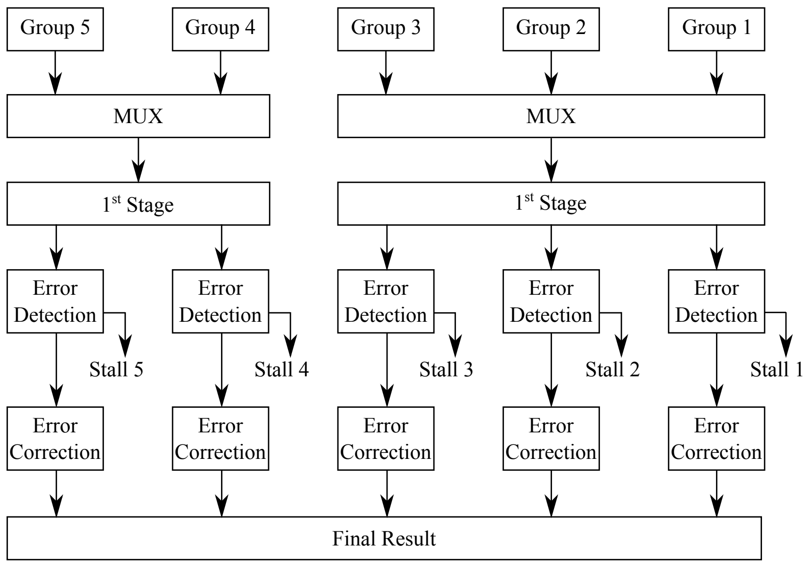


Figure 4.2: Overall error checker

for error correction. The following subsection attempts to tackle this challenge.

4.1.3 The Big Picture

In order to use the proposed error correction technique with minimum overhead to the overall performance of the processor, it has to be wisely integrated within the floating point adder in specific and within the processor as a whole. As mentioned before, the residue generation and modular addition steps are carried out in parallel with the main adder. After taking the *Guard*, *Round* and *Sticky* digits into account, the significand adder becomes a 19-digit adder. This means that the proposed 4-digit error checker has to be repeated five times (in order to keep the delay to its minimum possible value).

A deeper look in the synthesis results reveals that the first stage blocks can be reused for different digit groups since their delay is smaller than that of the main adder. Put differently, the 4-digit first stage blocks take almost 2 nsec to complete its job, therefore the same blocks can be reused on another group of four digits. Thus, the number of needed first stage blocks can be reduced from five to only two. On the other hand, five of the second stage blocks are used in parallel in order to minimize the overall delay. This delay can be taken off the normal operation path by allowing the processor to complete its operation based on the obtained results from the main adder until the checker finishes its operation. In fact, the checker determines whether an error has occurred or not after the syndrome generation step which takes 2 nsec out of the total 5.17 nsec of the second stage. Therefore, after the main adder finishes by 2 nsec the occurrence of an error can be detected and the processor operation is stalled. Otherwise, the processor continues its operation normally.The overhead for the overall error checker, shown in figure 4.2 is therefore expected to be 25823.25 μm^2.

4.2 Comparisons

For the sake of comparison the proposed technique is compared to the error detection proposed in [26]. Table 4.1 shows the main differences between the two techniques. Although the proposed method in [26] is much smaller than our proposed design, it does not provide the floating point unit with the ability to correct errors. Moreover, it is not able to detect all possible errors.

Table 4.1: Proposed technique evaluation

Point of Comparison	Res 9-3 [26]	Proposed Technique
Area Overhead	5841 μm^2 ($\approx$ 29.7% of the FPU area in POWER7 processor)	25823.52 μm^2 ($\approx$ 62% of the FP adder in [6])
Error Detection and Correction Capabilities	Detection Only	Detection and Correction
Error Coverage	88% (for a totally random error pattern in a 64-bit number)	Almost 100% (for multiple digit error in the Mixed Adder case study)

4.3 Conclusion

In this chapter, the results of the functional verification and synthesis of the the proposed design were discussed. Synthesis results revealed the need to integrate the operation of the fault tolerant adder with the operation of the processor in order to minimize the overhead caused by the error checker. The proposed design was then compared to the error detection technique proposed in [26].

Chapter 5

Conclusion and Future Work

5.1 Conclusion

To our knowledge, this is the first implementation of a residue error correction scheme in decimal and binary arithmetic circuits. The proposed method is able to correct any 4-digit error in the 4-digit numbers being checked assuming that errors occur only in the main adder. The 4-digit checking process is repeated until all digits of the result of the addition process are checked.

This work represents an attempt using information redundancy to add fault tolerance capabilities to a combined IEEE decimal-64/binary-64 floating point adder. Several techniques have been devised to achieve fault tolerance in current decimal/binary arithmetic circuits using time redundancy, hardware redundancy or both. Information redundancy in the form of residue codes was also used to achieve error detection in floating point units. Meanwhile, a lot of research is being conducted in designing arithmetic circuits which adopt the Residue Number System RNS instead of the Weighted Number System WNS to make use of its carry free operations and fault tolerant properties.

In the proposed technique, Residue codes are used for error detection and correction. Meanwhile, the same checker is used on different parts of the result to decrease the area overhead of the correction circuit. The technique depends on calculating the residues of the operands to the arithmetic circuit, performing the arithmetic operation on the residues as well as the operands and finally calculating the syndrome. Through the proper choice of the moduli, it can be guaranteed that each error has a unique syndrome pattern. Therefore, knowing the syndrome pattern, the corresponding error can be determined and hence the result can be corrected.

According to the synthesis results in section 4.1.2, the area of the checker did not turn out to be small enough, compared to the area of the main adder, to justify the assumption that error occur only in the main adder. In other words, the area of the synthesized checker circuit obtained in chapter 4 does not imply that the probability of an error occurring in it will be neglected compared to the probability of an error occurring in the main adder circuit. Therefore, further work should be done in order to provide the proposed design with the ability to correct an error whether it occurred in the main circuit or the checker circuit.

5.2 Future Work

In this section, some ideas are suggested in order to improve the performance of the error correcting circuit.

- This work should be further improved by finding the optimum residue codes that produce maximum error coverage with minimum area and delay overheads.
- The proposed design should be improved to a allow for error detection and correction whether the error occurred in the main adder or in the checker circuits.
- More research should be done to provide error models for the arithmetic circuits and suggest which errors are more likely to happen, whether single digit, double digit, etc.
- Some work should also be addressed towards formulating mathematical relation that relates the range of detectable and correctable different kinds of errors (single-digit, double-digit, etc.) to the set of moduli used.
- Another important enhancement is to extend the proposed technique to include all blocks of the floating point adder and not just the significand addition.

Bibliography

[1] “The international technology road-map for semiconductors,” tech. rep., 2007, 2011. Available at http://www.itrs.net.

[2] W. Huang, M. Stan, S. Gurumurthi, R. Ribando, and K. Skadron, “Interaction of scaling trends in processor architecture and cooling,” in *26th Annual IEEE Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010*, pp. 198–204, Feb. 2010.

[3] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, “The case for lifetime reliability-aware microprocessors,” in *31st Annual International Symposium on Computer Architecture 2004 Proceedings*, pp. 276–287, Jun. 2004.

[4] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, “The impact of technology scaling on lifetime reliability,” in *International Conference on Dependable Systems and Networks, 2004*, pp. 177–186, Jun.-Jul. 2004.

[5] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martin-Martinez, B. Kaczer, G. Groeseneken, R. Rodriguez, and M. Nafria, “Emerging yield and reliability challenges in nanometer CMOS technologies,” in *Design, Automation and Test in Europe, 2008. DATE ’08*, pp. 1322–1327, Mar. 2008.

[6] K. Y. El-Ghamrawy, “A mixed decimal/binary redundant floating point adder,” Master’s thesis, Department of Electronics and Electrical Communications, Faculty of Engineering, Cairo University, 2011.

[7] B. B. Agarwal, M. Gupta, and S. P. Tayal, *SOFTWARE ENGINEERING TESTING An Introduction*, ch. 4. Jones Bartlett Learning, 2009.

[8] P. Ramachandran, S. Adve, P. Bose, and J. Rivers, “Metrics for architecture-level lifetime reliability analysis,” in *IEEE International Symposium on*

Performance Analysis of Systems and software, 2008. ISPASS 2008, pp. 202–212, Apr. 2008.

[9] B. Parhami, "Defect, fault, error,..., or failure?," *IEEE Transactions on Reliability*, vol. 46, pp. 450–451, Dec. 1997.

[10] T. R. N. Rao, *Error Coding for Arithmetic Processors*. Academic Press, Inc., 1974.

[11] J. Tao, J. Chen, N. Cheung, and C. Hu, "Modeling and characterization of electromigration failures under bidirectional current stress," *IEEE Transactions on Electron Devices*, vol. 43, pp. 800–808, May 1996.

[12] J. Stathis, "Physical and predictive models of ultrathin oxide reliability in CMOS devices and circuits," *IEEE Transactions on Device and Materials Reliability*, vol. 1, pp. 43–59, Mar 2001.

[13] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11*, pp. 243–247, Feb. 2005.

[14] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, pp. 258–266, May-Jun. 2005.

[15] 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 2002 International Conference on Dependable Systems and Networks, DSN 2002*, pp. 389–398, 2002.

[16] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced CMOS bulk devices," in *44th Annual IEEE International Reliability Physics Symposium Proceedings*, pp. 217–225, Mar. 2006.

[17] I. Polian, J. Hayes, S. Reddy, and B. Becker, "Modeling and mitigating transient errors in logic circuits," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, pp. 537–547, Jul.-Aug. 2011.

[18] J. Gaisler, “Evaluation of a 32-bit microprocessor with built-in concurrent error-detection,” in *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers*, pp. 42–46, Jun. 1997.

[19] P. Lidén, P. Dahlgren, R. Johansson, and J. Karlsson, “On latching probability of particle induced transients in combinational networks,” in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pp. 340–349, Jun. 1994.

[20] H. Deogun, D. Sylvester, and D. Blaauw, “Gate-level mitigation techniques for neutron-induced soft error rate,” in *Sixth International Symposium on Quality of Electronic Design, 2005. ISQED 2005*, pp. 175–180, Mar. 2005.

[21] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, “Impact of CMOS process scaling and SOI on the soft error rates of logic processes,” in *Digest of Technical Papers of 2001 Symposium on VLSI Technology*, pp. 73–74, 2001.

[22] D. Tran, A. Virazel, A. Bosio, L. Dilillo, P. Girard, S. Pravossoudovitch, and H. Wunderlich, “A hybrid fault tolerant architecture for robustness improvement of digital circuits,” in *2011 20th Asian Test Symposium (ATS)*, pp. 136–141, Nov. 2011.

[23] A. Nieuwland, S. Jasarevic, and G. Jerin, “Combinational logic soft error analysis and protection,” in *12th IEEE International On-Line Testing Symposium, 2006. IOLTS 2006*, p. 6, 2006.

[24] P. Reviriego, S. Liu, and J. Maestro, “Mitigation of permanent faults in adaptive equalizers,” *Microelectronics Reliability*, vol. 51, no. 3, pp. 703–710, 2011.

[25] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, “Architectural core salvaging in a multi-core processor for hard-error tolerance,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA ’09, pp. 93–104, ACM, 2009.

[26] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *20th IEEE Symposium on Computer Arithmetic*, pp. 73–76, Jul. 2011.

[27] M. Yilmaz, D. R. Hower, S. Ozev, and D. J. Sorin, "Self-checking and self-diagnosing 32-bit microprocessor multiplier," in *IEEE International Test Conference, 2006. ITC '06*, pp. 1–10, Oct. 2006.

[28] B. Johnson, J. Aylor, and H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder," *IEEE Journal of Solid-State Circuits*, vol. 23, pp. 208–215, Feb. 1988.

[29] L. Chen and T. Chen, "Fault-tolerant serial-parallel multiplier," *IEE Proceedings-E Computers and Digital Techniques*, vol. 138, pp. 276–280, Jul. 1991.

[30] S. Dutt and F. Hanchek, "REMOD: a new methodology for designing fault-tolerant arithmetic circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, pp. 34–56, Mar. 1997.

[31] J. Patel and L. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, vol. C-31, pp. 589–595, Jul. 1982.

[32] J. Patel and L. Fung, "Concurrent error detection in multiply and divide arrays," *IEEE Transactions on Computers*, vol. C-32, pp. 417–422, Apr. 1983.

[33] S. Al-Arian and M. Gumusel, "HPTR: Hardware partition in time redundancy technique for fault tolerance," in *IEEE Southeastcon '92, Proceedings*, vol. 2, pp. 630–633, Apr. 1992.

[34] Y. Hsu and E. Swartzlander Jr., "Time redundant error correcting adders and multipliers," in *1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings*, pp. 247–256, Nov. 1992.

[35] A. Avizienis, "Arithmetic algorithms for error-coded operands," *IEEE Transactions on Computers*, vol. C-22, pp. 567–572, Jun. 1973.

[36] J.-C. Lo, S. Thanawastien, and T. Rao, “Berger check prediction for array multipliers and array dividers,” *IEEE Transactions on Computers*, vol. 42, pp. 892–896, Jul. 1993.

[37] R. Forsati, K. Faez, F. Moradi, and A. Rahbar, “A fault tolerant method for residue arithmetic circuits,” in *International Conference on Information Management and Engineering, 2009. ICIME ’09*, pp. 59–63, Apr. 2009.

[38] S. Haykin, *Communication Systems*. Wiley Publishing, 5th ed., 2009.

[39] T. R. N. Rao, “Error correction in adders using systematic subcodes,” *IEEE Transactions on Computers*, vol. C-21, pp. 254–259, Mar. 1972.

[40] “IEEE standard for floating-point arithmetic,” *IEEE Std 754-2008*, 2008.

[41] “IEEE standard for binary floating-point arithmetic,” *ANSI/IEEE Std 754-1985*, 1985.

[42] M. Cowlishaw, “Densely packed decimal encoding,” *IEE Proceedings-Computers and Digital Techniques*, vol. 149, pp. 102–104, May 2002.

[43] W. Townsend, J. Abraham, and E. Swartzlander Jr., “Quadruple time redundancy adders [error correcting adder],” in *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings*, pp. 250–256, Nov. 2003.

[44] Y. Tang, E. Boutillon, C. Jégo, and M. Jézéquel, “A new single-error correction scheme based on self-diagnosis residue number arithmetic,” in *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 27–33, Oct. 2010.

[45] E. Di Claudio, G. Orlandi, and F. Piazza, “A systolic redundant residue arithmetic error correction circuit,” *IEEE Transactions on Computers*, vol. 42, pp. 427–432, Apr. 1993.

[46] F. Barsi and P. Maestrini, “Error correcting properties of redundant residue number systems,” *IEEE Transactions on Computers*, vol. C-22, pp. 307–315, Mar. 1973.

[47] S. Eivazi, A. Eivazi, and G. Javid, “Error detection via redundant residue number system,” in *2010 5th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pp. 641–643, Dec. 2010.

[48] V. T. Goh and M. Siddiqi, “Multiple error detection and correction based on redundant residue number systems,” *IEEE Transactions on Communications*, vol. 56, pp. 325–330, Mar. 2008.

[49] K. A. Gbolagade, *Effective Reverse Conversion in Residue Number System Processors*. PhD thesis, Delft University of Technology, 2010.

[50] H. Yassine and W. Moore, “Improved mixed-radix conversion for residue number system architectures,” *IEE Proceedings-G, Circuits, Devices and Systems*, vol. 138, pp. 120–124, Feb. 1991.

[51] F. Taylor, “Residue arithmetic a tutorial with examples,” *Computer*, vol. 17, pp. 50–62, May 1984.

[52] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*. Advances in Computer Science and Engineering: Texts, Imperial College Press, 2007.

[53] H. Yassine, “Matrix mixed-radix conversion for RNS arithmetic architectures,” in *Proceedings of the 34th Midwest Symposium on Circuits and Systems, 1991*, vol. 1, pp. 273–278, May 1991.

[54] H. Yassine, “Fast arithmetic based on residue number system architectures,” in *IEEE International Symposium on Circuits and Systems, 1991*, vol. 5, pp. 2947–2950, Jun. 1991.

[55] K. Gbolagade and S. Cotofana, “Generalized matrix method for efficient residue to decimal conversion,” in *IEEE Asia Pacific Conference on Circuits and Systems, 2008. APCCAS 2008*, pp. 1414–1417, Nov.-Dec. 2008.

[56] K. Gbolagade and S. Cotofana, “An *O(n)* residue number system to mixed radix conversion technique,” in *IEEE International Symposium on Circuits and Systems, 2009. ISCAS 2009*, pp. 521–524, May 2009.

[57] W. Wang, M. Swamy, M. Ahmad, and Y. Wang, “A study of the residue-to-binary converters for the three-moduli sets,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 50, pp. 235–243, Feb. 2003.

[58] O. Abdelfattah, *Data Conversion in Residue Number System*. PhD thesis, McGill University, 2011.

[59] A. Premkumar, E. Ang, and E.-K. Lai, “Improved memoryless rns forward converter based on the periodicity of residues,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, pp. 133–137, Feb. 2006.

ملخص البحث

نظراً للإنكماش المستمر فى أبعاد مكونات الدوائر الإلكترونية، تصبح هذه الدوائر أكثر عرضة للأخطاء أثناء التشغيل. حتى يمكن الوصول إلى أنظمة يمكن الإعتماد عليها، فإن هذه الأخطاء لا يمكن تجاهلها. لهذا السبب فإن وجود طرق للتعامل مع هذه الأخطاء فى المعالجات والدوائر الحسابية الحديثة ضرورة ملحة حتى يمكن الإعتماد على مثل هذه الأنظمة. هذا البحث هو محاولة لإستخدام المعلومات الزائدة وكذلك التكرار فى الوقت لإضافة القدرة على إكتشاف وتصحيح الأخطاء إلى دائرة جمع ثنائية/عشرية للأعداد الكسرية.

لقد تم تطوير عدة طرق للتغلب على أخطاء التشغيل فى الدوائر الحسابية الثنائية/العشرية الحالية بإستخدام التكرار فى الوقت أو التكرار فى الدوائر نفسها أو المزج بين الطريقتين. كما تم إستخدام المعلومات الزائدة فى صورة أكواد باقى القسمة لإكتشاف وتصحيح الأخطاء فى دوائر الأعداد الكسرية. هذا بالإضافة إلى أن هناك الكثير من الأبحاث التى تجرى لتصميم دوائر حسابية تعتمد على تمثيل الأرقام ببواقى القسمة بدلاً من تمثيلهم بالطريقة المألوفة بأرقام لها أوزان معروفة للإستفادة مما يتمتع به التمثيل الأول من سرعة عمليات الجمع والطرح وقدرته على تحمل الأخطاء.

فى التصميم المقترح، تم إستعمال أكواد باقى القسمة لإكتشاف الأخطاء وتصحيحها، بالإضافة إلى إستخدام جزء من دائرة تصحيح الخطأ على عدة أجزاء من النتيجة لتقليل المساحة الكلية للدائرة المقترحة. الطريقة المقترحة تعتمد على حساب بواقى قسمة الأرقام المجموعة/المطروحة ثم إجراء الجمع/الطرح على الأرقام وكذلك على بواقى القسمة وأخيراً حساب ما يعرف بالمتلازمة. من خلال الإختيار المناسب للمعاملات التى يتم القسمة عليها يمكن ضمان أن كل خطأ ممكن فى النتيجة تناظره متلازمة معينة. بالتالي، بمعرفة المتلازمة يمكن حساب الخطأ وبالتالي تصحيحه.

على حد علمنا، فإن هذا هو أول تنفيذ لطريقة التصحيح بإستخدام بواقى القسمة فى الدوائر الحسابية للأرقام الثنائية/العشرية. الطريقة المقترحة لها القدرة على تصحيح أى خطأ فى أى عدد مكون من أربع أرقام بفرض أن الأخطاء تحدث فقط فى الدائرة الرئيسية. يتم تكرار هذه العملية حتى يتم التأكد من صحة كل ألأرقام المكونة لناتج العملية الحسابية.

لقد تم بناء التصميم المقترح بإستخدام تكنولوجيا TSMC 65 nm LP. العملية المقترحة لتصحيح الأخطاء يمكن تقسيمها على مرحلتين بناءً على إحتياجها لمعرفة النتيجة من دائرة الجمع الرئيسية. بالنسبة للدائرة التى تفحص العدد المكون من أربعة أرقام فإن مساحة المرحلة الأولى $2396.16\ \mu m^2$ وتستهلك

1.98 nsec. أما المرحلة الثانية فمساحتها $4206.24\ \mu m^2$ وتستهلك 5.17 nsec.

إن الطريقة المقترحة لها قدرات كبيرة على إكتشاف وتصحيح الأخطاء ولكن المساحة الكبيرة لدائرة تصحيح الخطأ (مقارنة بالدائرة الرئيسية) والفترة الطويلة التى تستغرقها لإستكمال عملها يمثلان تحدى كبير أمام إستخدام المعلومات الزائدة بشكل فعال لتصحيح الأخطاء. لهذا السبب يجب أن يتم دمجها بحكمة مع الدائرة الرئيسية بشكل خاص ومع المعالج ككل بشكل عام. هذا بالإضافة إلى أن نتائج بناء الدائرة التى تم الحصول عليها لا تؤكد الفرض بأن الأخطاء تحدث فقط فى الدائرة الرئيسية. بالتالى يجب تطوير الدائرة حتى تتمكن من تصحيح الأخطاء سواء حدثت فى الدائرة الرئيسية أو فى دائرة التصحيح.

تصحيح الأخطاء فى وحدات الأعداد الكسرية بإستخدام معلومات زائدة

إعداد

شهاب يمن عبد اللطيف السيد

رسالة مقدمة إلى كلية الهندسة، جامعة القاهرة
كجزء من متطلبات الحصول على درجة الماجستير
فى هندسة الإلكترونيات والإتصالات الكهربية

يعتمد من لجنة الممتحنين

كلية الهندسة، جامعة القاهرة
الجيزة، جمهورية مصر العربية
٢٠١٢

تصحيح الأخطاء فى وحدات الأعداد الكسرية بإستخدام معلومات زائدة

إعداد

شهاب يمن عبد اللطيف السيد

رسالة مقدمة إلى كلية الهندسة، جامعة القاهرة

كجزء من متطلبات الحصول على درجة الماجستير

فى هندسة الإلكترونيات والإتصالات الكهربية

تحت إشراف

ا. م. د. حسام على حسن فهمى

أستاذ مساعد، كلية الهندسة، جامعة القاهرة

كلية الهندسة، جامعة القاهرة

الجيزة، جمهورية مصر العربية

٢٠١٢

تصحيح الأخطاء فى وحدات الأعداد الكسرية بإستخدام معلومات زائدة

إعداد

شهاب يمن عبد اللطيف السيد

رسالة مقدمة إلى كلية الهندسة، جامعة القاهرة

كجزء من متطلبات الحصول على درجة الماجستير

فى هندسة الإلكترونيات والإتصالات الكهربية

كلية الهندسة، جامعة القاهرة

الجيزة، جمهورية مصر العربية

٢٠١٢