FAST, AREA-EFFICIENT 32-BIT LNS FOR COMPUTER ARITHMETIC OPERATIONS

Rizalafande Che Ismail

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
AT NEWCASTLE UNIVERSITY, UNITED KINGDOM

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING
FACULTY OF SCIENCE, AGRICULTURE & ENGINEERING

September 2012

ABSTRACT

The logarithmic number system has been proposed as an alternative to floating-point. Multiplication, division and square-root operations are accomplished with fixed-point arithmetic, but addition and subtraction are considerably more challenging. Recent work has demonstrated that these operations too can be done with similar speed and accuracy to their floating-point equivalents, but the necessary circuitry is complex. In particular, it is dominated by the need for large lookup tables for the storage of a non-linear function.

This thesis describes the architectures required to implement a newly design approach for producing fast and area-efficient 32-bit LNS arithmetic unit. The designs are structured based on two different algorithms. At first, a new co-transformation procedure is introduced in the singularity region whilst performing subtractions in which the technique capable to generate less total storage than the co-transformation method in the previous LNS architecture. Secondly, improvement to an existing interpolation process is proposed, that also reduce the total tables to an extent that allows their easy synthesis in logic. Consequently, the total delays in the system can be significantly reduced.

According to the comparison analysis with previous best LNS design and floating-point units, it is shown that the new LNS architecture capable to offer significantly better in speed while sustaining its accuracy within floating-point limit. In addition, its implementation is more economical than previous best LNS system and almost equivalent with existing floating-point arithmetic unit.

ACKNOWLEDGEMENTS

First of all, I would like to express my special gratitude to my great supervisor, Dr. Nick Coleman, deeply for giving me an opportunity to pursue in this research work. His excellent advice, priceless support and remarkable guidance as well as spending a great deal of time and energy for this thesis is gratefully appreciated. What I have learned from him will benefit me well beyond my graduation in my future research career.

I am greatly indebted in Dr. Robin Emery and Mr. Raa'ed Aldujaily for the support during the work especially with related to the Cadence and Synopsys design tools. I also want to thank to all colleagues at the Microelectronics Systems Design group in room E4.21, who contributed to the perfect working environment.

Special thanks go to Mr. Mark Wilmott and Mrs. Lisa Wong at the Rutherford Appleton Laboratory in Harwell campus, Oxford. They have not only offered invaluable technical advice, but also offered CAD tool support in my thesis work.

In addition, I want to thank my beloved wife, Farah Fadhlina, and my gorgeous children, Irdina Rizqin, Iwani Rifqah and Irfan Rasyeeq, very much for their loving care, understanding and moral support. They gave me the boundless encouragement and motivation and led me to finish this thesis. I would like to extend thanks to my wonderful parents for all the love and support given.

Thanks are also due to University Malaysia Perlis and Ministry of Higher Education Malaysia for granting a study leave, and for financial assistance.

Finally, to all named and unnamed, for their support and understanding towards the completion of this research, thank you very much.

Life is good!

LIST OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1. Introduction

1.1. Motivation for the Research

The need for high-performance digital signal processing (DSP) in the area of image processing, computer graphics and robotics is highly demanding. High speed architecture allows DSPs to execute many operations with the lowest delay [1]. Since performance is a driving factor behind the use of the DSP, advances in executing arithmetic functions are the key to advances in the performance of DSP processors. Consequently, techniques to improve the computation of arithmetic functions have always been an interesting topic of exploration, as expressed in [2].

Most of DSP algorithms need to be computed in real-time and require a wide dynamic range of numbers. During the early stages of DSP, the fixed-point (FXP) number system was employed as the maths unit inside the DSP processor. This system performs well for high-speed applications whenever only limited precision is required by the application. Nevertheless, this implementation has a major limitation because of restricted accuracy, which is the result of finite word-length effects. Floating-point (FLP) DSP has therefore become an alternative used to overcome this restriction of precision of FXP architectures. Despite having a wide dynamic range, there are established international standards for FLP system [3]. One of the most efficient basic operations in existing high-speed FLP unit is the multiplication process. However, complex operations such as division and square root are often executed by software routines, and are possibly much slower. Moreover, arithmetic operations in FLP require a variable length of time due to the need for exponent alignment. As a result of this, DSP researchers have recently proposed a microprocessor based on the logarithmic number system (LNS) [4-7][94], which would guarantee superior performance in many arithmetic functions such as multiplication, division and square root.

LNS provides major advantages over FLP in terms of speed and accuracy in computing multiplication and division operations. This is because of the similarity of the architectures to perform these functions to FXP addition and subtraction. However, this inherent advantage was offset by the difficulty of implementing LNS addition and subtraction. Furthermore, it is also slow. Several authors have proposed techniques to improve this trade-off, and as a result the LNS is now able to operate with similar speed and accuracy to its FLP equivalent [6-11], despite its larger area. Due to these considerable achievements, research into LNS systems has been active ever since. Thus, it is of interest further to improve the LNS system relative to a FLP arithmetic unit.

1.2. An Overview of the LNS

Over the past four decades the LNS system has been a topic of continuing interest within the computer arithmetic area. As mentioned in previous section, multiplication and division operations become FXP addition and subtraction respectively. Unlike FLP counterparts, these operations are trivial and fast. Nevertheless, implementing addition and subtraction operations can be the main bottleneck, the evaluation of the non-linear functions (1.1) and (1.2). For $i = \log_2 x$, $j = \log_2 y$, $r = j - i$, and assuming $j \leq i$:

$$\log_2 (2^i + 2^j) = i + \log_2 (1 + 2^r) \tag{1.1}$$

$$\log_2 (2^i - 2^j) = i + \log_2 (1 - 2^r) \tag{1.2}$$

The functions $\log_2 (1 \pm 2^r)$, generically referred to as $F(r)$, are illustrated in Figure 1-1. In the earliest LNS design which is up to about 20-bit, the addition and subtraction function values can be stored directly in the lookup table. Beyond this, memory requirements become prohibitive, and instead the function is stored at intervals with intervening values obtained by interpolation. Typically, in constructing the LNS system, the objective has always been to keep within an FLP-equivalent error of 0.5 LSB, but this has not always been achieved. The problem is compounded by the singularity in the subtraction function, where the rapidly changing derivative as r approaches zero requires the use of successively smaller interpolation intervals that need a significant increase in storage, often to the point of impracticality. As well as that, applying the interpolation alone may also increase the delay of the LNS system.

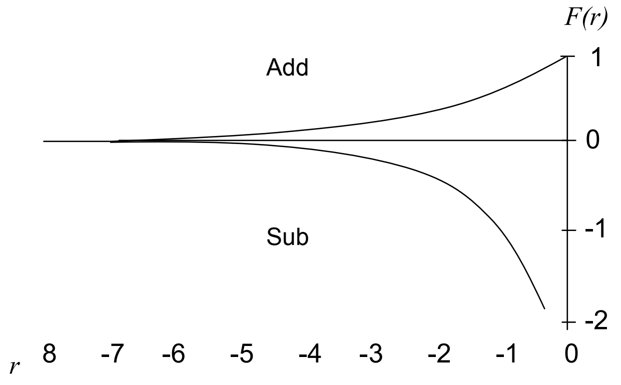


Figure 1-1 : LNS addition and subtraction functions.

However, as presented in 2000, an alternative approach was taken in a different interpolation technique. Dealing with 32-bit words and maintaining FLP-equivalent accuracy, it offered a much shorter delay path than using conventional interpolation architecture. In this approach, an interpolation was not used near the singularity. Instead, a co-transformation was used in the case of any subtraction with r close to zero (> -0.5), which it converted to an equivalent subtraction with r well away from zero. This 32-bit LNS system was based on the combination of the interpolation and the co-transformation procedure, and offered marginally better performance, in terms of both speed and accuracy, than a leading commercial FLP unit at that time. Nonetheless, two 2048 words of lookup tables were involved in the arrangement of the co-transformation architecture. Meanwhile, the interpolator itself then required 1024 words for one of its lookup tables. In practice, utilising these large lookup tables in the system could eventually introduce significant complications in floor planning. Hence, elimination of these components would not only yield a more compact architecture, but undoubtedly also a faster design.

Inspired by the above, therefore, the major objectives of this thesis can be summarised as follows:

- present a new development in the algorithm of the co-transformation procedure which can offer substantial improvement in area.
- enhance the interpolation architecture by exploring various existing techniques as to reduce the total storage and the delay of the system.
- demonstrate that the new LNS system will achieve much greater benefits in cost, speed and accuracy in comparison with FLP arithmetic units.

1.3. Contribution of the Thesis

The following points summarise the main contributions of the thesis.

- A novel approximation method, known as a second-order co-transformation procedure, is introduced in the crucial singularity region for performing the LNS subtraction function. Apart from the capability to sustain the same accuracy as FLP, implementing this new approach in conjunction with the existing interpolator reduces the total tables to 73% of the former LNS design. However, the proposed technique suffers from an increase in delay because it requires the interpolator to be used twice.

- An improvement in the interpolator design by reworking Chester's experiments [84] is proposed when computing the LNS addition and direct (i.e. non- co-transformed) subtraction. When merged with the second-order co-transformation, it yields a further reduction in total tables to 51% of previous LNS design. Through this new arrangement, the tables can now be readily synthesised in logic as a result of being smaller in size, for not more than 512 words. Consequently, this can contribute to a reduction in delay to 60% of the original LNS design when computing addition and direct subtraction. For subtractions with co-transformation, delay only increases by 12% compared to the previous work.

- An analysis is conducted between the new LNS design and equivalent FLP arithmetic units built using similar process technology. In terms of delay, the new LNS can be performed in 63% of the FLP time for executing addition and direct subtraction. Co-transformed subtractions require 131% of the FLP time but this is unlikely to be of great significance because it occurs in only a few percent of the total additive operations. Multiplication completes with 10% and division 3% of the FLP delays. In terms of total area, the new LNS can be built with fractionally less silicon, and worst-case accuracy is better than that of FLP arithmetic.

- At present, little work has been reported applying LNS design to word-lengths longer than 32-bit. The design and requirements of long format LNS arithmetic unit are therefore examined briefly in this thesis. The co-transform is developed further for this purpose.

1.4. Structure of the Thesis

The fundamental basis of computer arithmetic architecture and details of the FLP and LNS number systems are reviewed in Chapter 2. Previously published LNS designs are also discussed and analysed in terms of various aspects such as their design procedures, performance and suitability for DSP applications.

When evaluating and measuring the performance of the LNS system, several elements need to be considered, either the metrics required for measurement or the design methodology adopted to verify the design. Therefore, Chapter 3 explains the metrics involved for performance estimation during the simulation and synthesis processes. Besides that, the design flows of the simulation and synthesis procedures are also elaborated in detail.

In Chapter 4, the recent 32-bit arithmetic implementations are reviewed intensively. This includes exploring the leading published design of the LNS system before reconstructing the architecture using similar approaches as described in its original work. In addition, several FLP devices are also examined. These devices are independently designed and have been published. The performance of these arithmetic units is reported in this chapter, and later it will then be used for comparison with the new LNS system.

Chapter 5 presents a new development of the co-transformation architecture for executing LNS subtraction function, exploiting the previously published co-transformation concept and significantly elaborating on its architecture. The simulation and synthesis results of the proposed design are also reported in evaluating its efficiency in the light of previous work.

The different existing function approximation schemes are described in Chapter 6. An improved technique for the interpolator module is introduced. Accuracy and total area analyses are carried out and documented on the basis of worst-case error and total size of lookup tables respectively. It is shown that the improved version is able to provide a great reduction in total tables whilst sustaining accuracy within FLP limits.

The implementation of the suggested LNS arithmetic unit is explained in Chapter 7. The synthesis process is performed to determine the performance of the new LNS architecture in terms of speed and total silicon area, before a comparative study against FLP units and previous LNS design is discussed.

There is a lack of work on long word-length LNS, and a short survey of a possible long format system is therefore outlined in Chapter 8. This includes a proposal for another new co-transformation approach applicable to a long word-length system. Its implementation in logic gates and performance analysis against the standard 32-bit LNS number system are also described.

Finally, the main results of the thesis are summarised and conclusions are drawn in Chapter 9. Moreover, several possibilities for future work extending the present research are also offered.

CHAPTER 2

2. Background and Previous Work

2.1. Introduction

In this chapter, the current body of knowledge relevant to the present research is extensively reviewed. The fundamental basis of the computer arithmetic unit is briefly described. An overview is given of FLP and LNS numbers formats, and computing arithmetic units based on these number systems are elaborated in detail. Previously published techniques used to execute the LNS addition and subtraction are discussed and compared in various respects, since these operations are the main bottlenecks in LNS system.

Speed, accuracy and area are the three crucial variables in the efficiency of LNS arithmetic unit. Thereby, the performance of existing LNS systems is evaluated so that the results could be used as a benchmark for the novel architecture introduced in this thesis. Finally, the LNS systems adopted in numerous DSP applications are concisely described.

2.2. Computer Arithmetic Unit

Conventionally, most computer architectures include three basic hardware subsystems, namely the central processing unit (CPU), main-memory system and input/output (I/O) system [12-14]. A CPU carries out instructions sequentially by performing two distinct procedures known as the fetch and execute cycles, where at least one operation is conducted at a time. The main-memory system plays the vital role of holding the programs that control the computer's operations. The I/O system represents the various devices that can exchange information with the outside world.

As presented in Figure 2-1, the computer arithmetic unit is a component of a CPU system. It is commonly combined with logic functions, hence constituting an arithmetic logic unit. This arithmetic unit deals with the arithmetic functions needed to support various computer instructions, and thus it is a very important part of digital computer organisation. Agrawal and Rao [15] describe the computer arithmetic unit as always having been considered the heart of a digital computer system. Among the arithmetic operations that can be computed are addition, subtraction, multiplication, division, square root, exponentiation, logarithmic functions, complementation (negation), incrementation or decrementation, equality and magnitude comparison and shift operations. These numeric functions, and especially adders and multipliers, are also implemented in diverse ways in the data paths of digital signal processors which then form dedicated integer units and multiply-accumulate (MAC) structures. Moreover, adders, incrementers or decrementers, and comparators are often used for address and flag generation purposes in controllers.

Because the applications of arithmetic operations are manifold, much effort has been devoted to designing hardware algorithms and circuits to enhance the speed of these numeric operations [7, 16-18]. More recently, since the inception of portable electronic devices which require small and lightweight units, the demand for not only reduction in power consumption, but also the total area of the systems has increased dramatically. Therefore, the development of algorithms that can reduce delays and total area in arithmetic operations is a matter of great concern in today's arithmetic architecture [19-21].

The four basic numeric operations (addition, subtraction, multiplication and division) of the computer arithmetic unit are critically investigated in this thesis. New algorithms based on LNS which aims specifically at addition and subtraction functions are introduced which can significantly improve the overall performance of an arithmetic system.

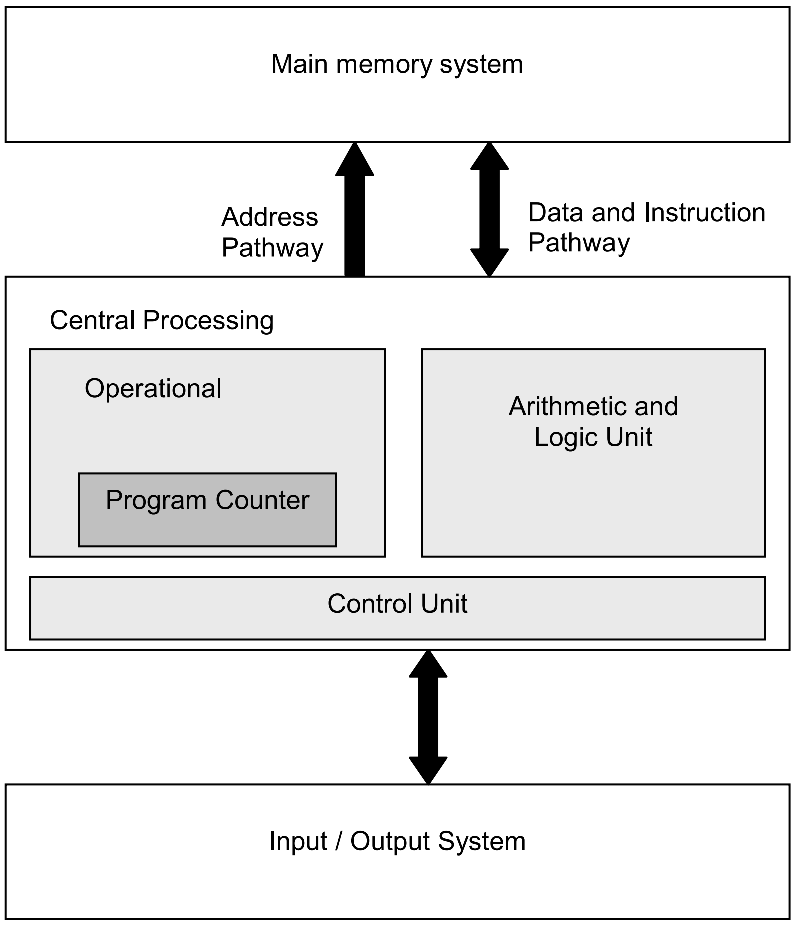


Figure 2-1: Main components of typical computer architectures.

2.3. Number Systems Representation

Integers and real numbers, also expressed as fractions, are the most common number system representations used in digital computers [14]. Traditionally, integers have been represented using FXP number systems that offer limited range and precision. When dealing with money and inventories in business and commercial activities, the use of integer numbers is adequate in estimating the results of calculations given the fact that usually only two places to the right of the decimal point will be occupied. Furthermore, in control problems which deal with measurements in degrees, minutes and seconds, the ranges involved can also fit into the FXP system [22]. Conversely, difficulties arise in scientific applications such as those needed by astronomers, engineers and physicists. The formulae used to represent length and mass, for instance, repeatedly consider differences between very large or very small numbers, and thus the FXP system fails [23]. In such situations real numbers have to be adopted to compute the functions.

Over the years, many computer manufacturers have implemented FLP system to represent real numbers [14, 24, 25]. An FLP system is capable of offering a wide dynamic range which can accommodate extremely large numbers and high precision for very small numbers. Nevertheless, over the last four decades, researchers have explored the use of LNS as an alternative to signify real numbers in computer systems [4, 6, 7, 26]. Despite the lack of standard formats, the accurate and inexpensive implementation of multiplication and division operations in LNS which only use FXP addition and subtraction, makes it more attractive compared to FLP [27]. In addition to higher speed, LNS has also been the subject of close attention for numerous applications as a result of its inherently better worst-case relative error compared to FLP [28].

In this thesis, LNS numbers are the main subject of the research, and the FLP format is also used for comparison purposes. Therefore, the basic fundamental features of both formats are described briefly below.

2.3.1. Floating Point

The IEEE 754 [3] is a standard used to represent FLP numbers and has been divided into single-precision format with 32-bit width, and double-precision format with 64-bit width. In this thesis, only single-precision format is considered. The three basic components of FLP numbers are the sign, exponent and mantissa as shown in Figure 2.2.

Sign (1-bit)	Exponent (8-bit)	Mantissa (23-bit)

Figure 2-2: Basic components of single-precision format.

The number denoted by the single-precision format is [29]:

value $= (-1)^{s}2^{e} \text{ x } 1.f$ (normalized) when E > 0 else (2.1)

$= (-1)^{s}2^{-126} \text{ x } 0.f$ (denormalized) (2.2)

where

f = fraction bits

s = sign bit (0 for positive, 1 for negative)

E = exponent fields (contains 127 plus the true exponent for single-precision)

e = unbiased exponent (e = E – 127 (bias))

The range of positive FLP numbers which can be split into normalized numbers (which preserve the full precision of the mantissa), and denormalized numbers (which occur when the exponent is all zeros, but the fraction is non-zero) are between $\pm 2^{-126}$ to $(2\text{-}2^{-23}) \cdot 2^{127}$ and $\pm 2^{-149}$ to $(1\text{-}2^{-23}) \cdot 2^{-126}$ respectively. Table 2-1 summarises the values than can be defined in the FLP system.

Table 2-1: Values represented in the 32-bit FLP format.

s	e	f	Value
0	0	0	+0
		Any non-zero	Positive Denormal, $0.f \cdot 2^{-126}$
	1 … 254	*Any*	Positive Normal, $1.f \cdot 2^{e}$
	255	0	$+\infty$
1	0	0	-0
		Any non-zero	Negative Denormal, $-0.f \cdot 2^{-126}$
	1 … 254	*Any*	Negative Normal, $-1.f \cdot 2^{e}$
	255	0	$-\infty$
Any	255	00’01 .. 01’11	SNaN
		10’00 .. 11’11	QNaN

2.3.2. Logarithmic Number System

In contrast with FLP numbers, LNS includes neither an integer exponent nor separate linear mantissa. It is much simpler because it uses a single scaled exponent and can be represented by [30]:

$$X = (-1)^{s} \text{ x } 2^{m.f} \tag{2.3}$$

where s, m and f indicate sign, integer and fractional bits respectively. Although there is no commonly accepted standard for the LNS format, the most widely used format is shown in Figure 2-3.

Sign (1-bit)	Fixed-Point Logarithmic Value	
	Integer (*m*-bit)	Fractional (*f*-bit)

Figure 2-3: LNS format [7].

Typically, base-2 logarithms are used in LNS computations though in principle any base can be used. When the real numbers represented are signed, LNS has a maximum and minimum range between 2^{-128} to $\approx 2^{+128}$, $\approx$ 2.9E – 39 to 3.4E + 38. A special arrangement of bits is used to indicate the real number zero.

2.4. Floating-Point Algorithms

The basic algorithms for arithmetic operations using FLP numbers are conceptually simple. Nevertheless, careful attention must be paid during hardware implementations in order to ensure correctness and to prevent excessive loss of precision [31].

Addition and subtraction are a lot more complex than the other FLP operations. In the following description, elementary binary FLP addition is explained, since subtraction can be converted to addition merely by flipping the sign of the subtrahend. In theory, addition is defined as:

$$(\pm m_1 \times 2^{e1}) + (\pm m_2 \times 2^{e2}) = \pm m \times 2^{e1} \tag{2.4}$$

where m, m_1 and m_2 are the mantissas and e, $e1$ and $e2$ are the exponents. Assuming $e1 \geq e2$, the exponents of the addends have to be made equal by right-shifting (divided by a power of two) the mantissa of the smaller number, m_2, by as many bits as its exponent, $e2$, is increased. Then the shifted mantissa, m_2, will be added to the other mantissa, m_1. After addition, the resulting mantissa is normalized back to the mantissa interval by multiplying it with the corresponding exponent, $e1$, as presented in (2.5) [32].

$$\begin{aligned}(\pm m_1 \times 2^{e1}) + (\pm m_2 \times 2^{e2}) &= (\pm m_1 \times 2^{e1}) + \left(\frac{\pm m_2}{2^{e1-e2}} \times 2^{e1}\right) \\ &= \left(\pm m_1 \pm \frac{m_2}{2^{e1-e2}}\right) \times 2^{e1} \\ &= \pm m \times 2^{e1}\end{aligned} \tag{2.5}$$

In contrast, binary FLP multiplication is a relatively straightforward procedure whereby the mantissas, m_1 and m_2, are first multiplied together [23]. Then, the exponents, $e1$and $e2$, are added. After multiplication has been computed, the product obviously has twice as many digits as the original operands. Hence, post-normalization procedure is needed to adjust the mantissa and the exponent of the result. Generally, the normalization process is executed by left-shifting the mantissa until it reaches the first bit 1. Simultaneously, for each bit left-shifted, the exponent must be reduced by 1. Therefore, the binary FLP multiplication is described as:

$$\begin{aligned}(\pm m_1 \times 2^{e1}) \times (\pm m_2 \times 2^{e2}) &= \pm (m_1 \times m_2) \times 2^{e1+e2} \\ &= \pm m \times 2^{e} \end{aligned} \tag{2.6}$$

The operation of FLP division is like that of multiplication, conducted by dividing the mantissas and subtracting the exponents and therefore presented as:

$$\begin{aligned}(\pm m_1 \times 2^{e1}) \div (\pm m_2 \times 2^{e2}) &= \pm (m1 \div m2) \times 2^{e1-e2} \\ &= \pm m \times 2^{e} \end{aligned} \tag{2.7}$$

In the case of division, the mantissas are first left-shifted according to their number of leading zeros. After being divided and subtracted for both mantissas and exponents, post-normalization is performed as in multiplication to produce the final result. Conceptually, division operations always consume a large proportion of area in any FLP system, therefore making it an inherently slow operation which should be used sparingly. Due to the fact that FLP division is an infrequent operation even in intensive FLP applications, many current architectures ignore its implementation [33, 34].

2.5. Logarithmic Number System Algorithms

Typically, computer arithmetic unit conducts four major operations, namely addition, subtraction, multiplication and division. In LNS arithmetic, multiplication and division are trivial operations due to the fact that they have equivalent architectures to either FXP addition or subtraction as illustrated in (2.8) and (2.9). Moreover, these operations are more accurate and there is no quantization error, thus returning an exact result [35], where as FLP often yields a half-bit rounding error [36].

Generally in the LNS system, two real numbers, x and y, are used and can be represented by the FXP values $i = \log_2 |x|$ and $j = \log_2 |y|$. In addition, an additional bit is used to show the signs of x and y, S_x and S_y. Thus, multiplication and division are computed as:

$$\text{Multiply:} L_1 = x \cdot y \quad \rightarrow \quad log_2 |L_1| = log_2 |x \cdot y| = log_2 |x| + log_2 |y| = i + j \tag{2.8}$$

where: $S_{L1} = S_x \oplus S_y$

$$\text{Divide:} \quad L_2 = x \div y \quad \rightarrow \quad log_2 |L_2| = log_2 |x \div y| = log_2 |x| - log_2 |y| = i - j \tag{2.9}$$

where: $S_{L2} = S_x \oplus S_y$

In contrast, LNS addition and subtraction become fairly complex procedures [26]. To perform these operations, Leonelli's algorithm [37] is used. The functions $s_b(r)$, for the addition algorithm (also known as Gaussian algorithm [38]), and $d_b(r)$, in the subtraction algorithm, are defined as:

$$s_b(r) = log_2 (1 + r) = log_2 (1 + 2^r), \; r < 0 \tag{2.10}$$

$$d_b(r) = log_2 (1 - r) = log_2 (1 - 2^r), \; r < 0 \tag{2.11}$$

Hence, these functions are plotted as in Figure 2-4.

Assuming that $|x| \geq |y| > 0$ and let $r = (log_2 |y| - log_2 |x|) = j - i$, therefore addition and subtraction can be computed using:

$$\begin{aligned} \text{Addition:} L_3 = x + y \quad \rightarrow \quad log_2 |L_3| &= log_2 | x + y | \\ &= log_2 | x (1 + (y / x) | \\ &= log_2 |x| + log_2 |1 + (y / x) | \\ &= log_2 |x| + log_2 |1 + (log_2 |y| - log_2 |x|)| \\ &= i + log_2 | 1 + 2^{j-i} | \\ &= i + log_2 | 1 + 2^r | \\ &= i + s_b(r) \end{aligned} \tag{2.12}$$

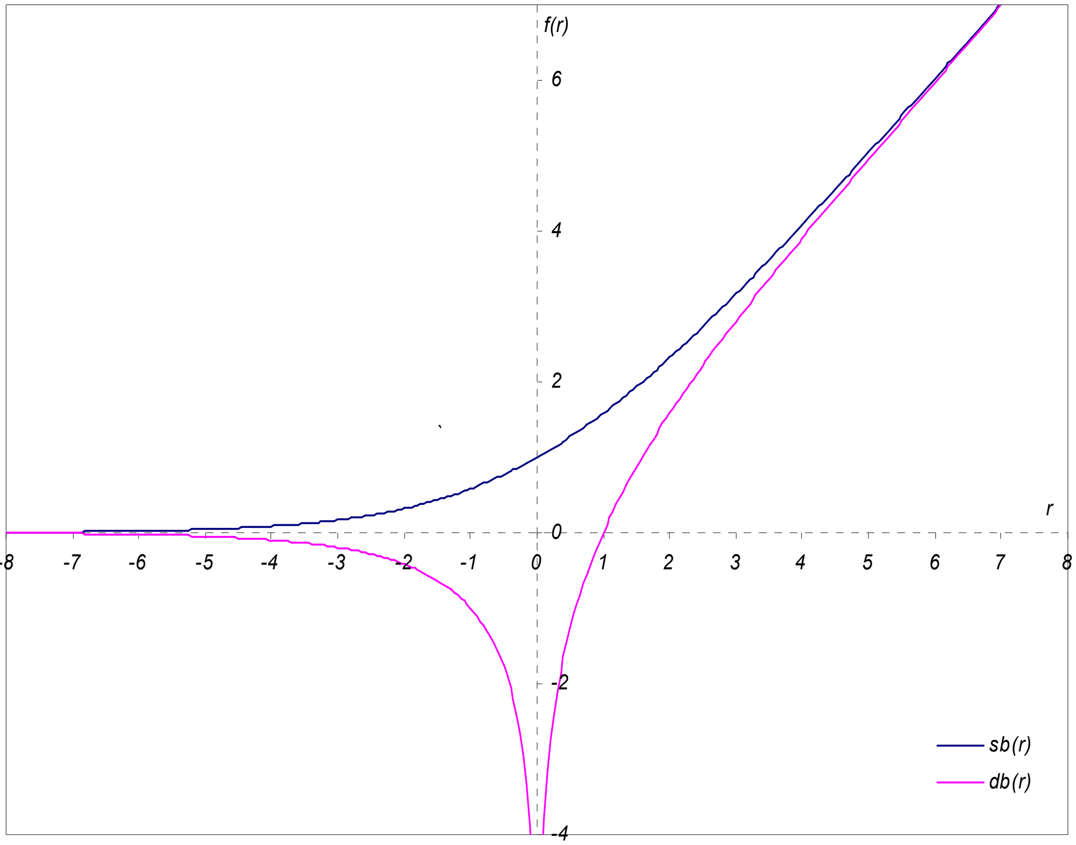


Figure 2-4: Transcendental functions $s_b(r)$ and $d_b(r)$.

$$\begin{aligned}
\text{Subtraction: } L_4 = x - y \quad \rightarrow \quad log_2|L_4| &= log_2|\,x - y\,| \\
&= log_2|\,x\,(\,1 - (\,y\,/\,x\,)\,| \\
&= log_2|x| + log_2\,|1 - (\,y\,/\,x\,)\,| \\
&= log_2|x| + log_2\,|1 - (log_2|y| - log_2|x|)| \\
&= i + log_2\,|\,1 - 2^{j-i}\,| \\
&= i + log_2\,|\,1 - 2^{r}\,| \\
&= i + d_b(r) \qquad (2.13)
\end{aligned}$$

It is clear that addition and subtraction operations are the main obstacle in an LNS system as a result of involving a lookup table in executing its non-linear function, $s_b(r)$ and $d_b(r)$. Potentially, with an increase in the word-length of LNS numbers, it can suffer from the requirement of a large lookup table in computing the function.

Therefore, over three decades, different ways of improving the addition and subtraction functions have been proposed, and these can be classified into seven distinct categories as follows.

2.5.1. Direct Lookup Table

The earliest and simplest LNS architecture for addition and subtraction was introduced in 1975 [39]. This was a direct implementation of equations (2.12) and (2.13) using lookup tables or so called Read Only Memory (ROM) based hardware covering all possible values of $s_b(r)$ and $d_b(r)$. The implemented structure based on this technique is as described in Figure 2-5.

In practice, the implementation of LNS add and subtract functions always has to limit the variable r to either positive or negative values. It is more usual to opt to restrict r to negative values because at a certain point (as shown in Figure 2-4), the functions of $s_b(r)$ and $d_b(r)$ have an output of zero or known as the *essential zero*. Consequently, $s_b(r)$ and $d_b(r)$ functions can yield a value that rounds to zero which is then easy to handle. As a result, the suggested procedure for addition and subtraction using the direct lookup table approach depends on two real numbers, x and y, as given below:

If $x \geq y \rightarrow r = j - i$:

Addition : $L = i + log_2 \,|\, 1 + 2^r \,|$ (2.14)

Subtraction : $L = i + log_2 \,|\, 1 - 2^r \,|$ (2.15)

If $y > x \rightarrow r = i - j$:

Addition : $L = j + log_2 \,|\, 1 + 2^r \,|$ (2.16)

Subtraction : $L = j + log_2 \,|\, 1 - 2^r \,|$ (2.17)

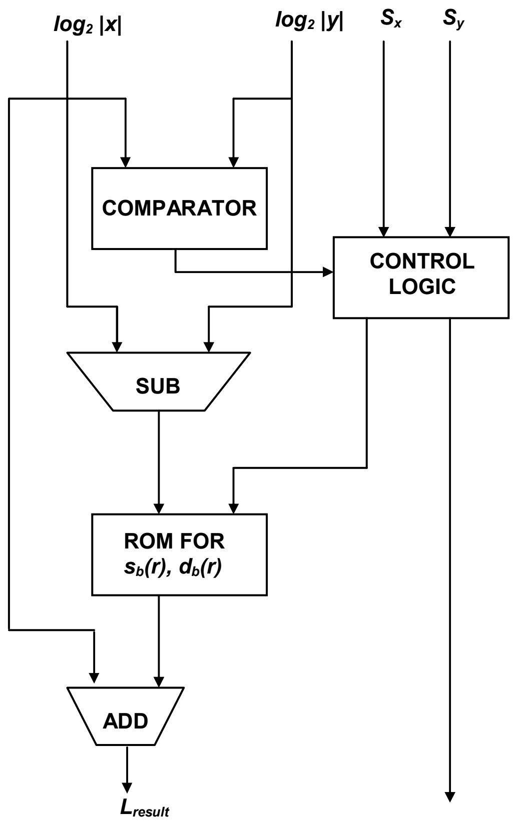


Figure 2-5: LNS adder/subtractor based on direct lookup table.

Using the technique considered here, the ROMs for $s_b(r)$ and $d_b(r)$ must each contain 2^f words of f bits each, and hence the total storage required can be computed as $f{\cdot}2^{f+1}$. With precision set to only 8-bit, a total of 4096 bits were achieved in [39] to compute LNS addition and subtraction. In evaluating the speed of the system, these operations were found to be approximately four times slower than conventional FLP methods. Although the direct lookup table approach has been successfully tested for a fast Fourier transform (FFT) application with the numbers rounded to 18-bit (plus sign bit) [40], it still yields an unreasonable size of ROM when it comes to long word-length numbers, especially at 32-bit, as a result of the required memory growing exponentially when the numbers increase linearly. In 1979, a state-of-the-art microcomputer, the FOCUS [41], was introduced that utilised the LNS system based on the direct lookup table method. It was reported that average execution cycles for 16-bit LNS add and subtract operations were 127 µsec and 125 µsec respectively when the FOCUS system was implemented in an Intel 8085 processor. In addition, 23,632 bits were needed for storage requirements in this architecture.

2.5.2. Interpolation

The memory space limitations of LNS addition and subtraction using a direct lookup table approach makes its use questionable. In order to overcome this problem, another technique, interpolation, is often used.

The direct interpolation technique [42] was first introduced to cater only for the addition algorithm, s_b, which requires a multiply unit in the hardware system. Using this technique, r is split into two parts, r_h and r_l, hence $r = r_h + r_l$. r_h encompasses the highest bits of the variable, whereas r_l represents the lowest bits. In the general case, the direct interpolation can be written as:

$$s_b(r) = s_b(r_h + r_l) \approx s_b(r_h) + C(r_h) \cdot r_l \tag{2.18}$$

where the slope $C(r_h)$ can be chosen from various methods such as Lagrange. Memory usage can be reduced by increasing the lower bits, r_l, but the accuracy of the approximation decreases too. Likewise, when the size of r_l increases, the same will happen with the size of the required multipliers. In effect, the use of an FXP multiplier can actually produce much higher costs, in terms of speed and area, which along with the greater expense due to its size can make the system even slower and larger. Therefore, direct interpolation in LNS is often limited to either first- or second-order coefficients.

Another notable interpolation technique was proposed by Taylor in 1983 [43], which is referred to here as linear interpolation. Taylor approximates $s_b(r)$ as:

$$s_b(r) = s_b(r_h) + s_b^{'}(r_h) \cdot r_l \tag{2.19}$$

As shown in Figure 2-6, with only addition operation shown for clarity, the linear interpolation method still needs a multiplier to compute the function. On top of that, two ROMs were introduced. Arnold *et al.* in 1988 [44] suggested a refined version of the interpolation procedure where they merge the direct interpolation method with the linear interpolation scheme. With the modified architecture, only one ROM is required and a shifter using powers of two is deployed as an alternative to the multiplier. However, once again, this technique is not feasible for the subtraction algorithm.

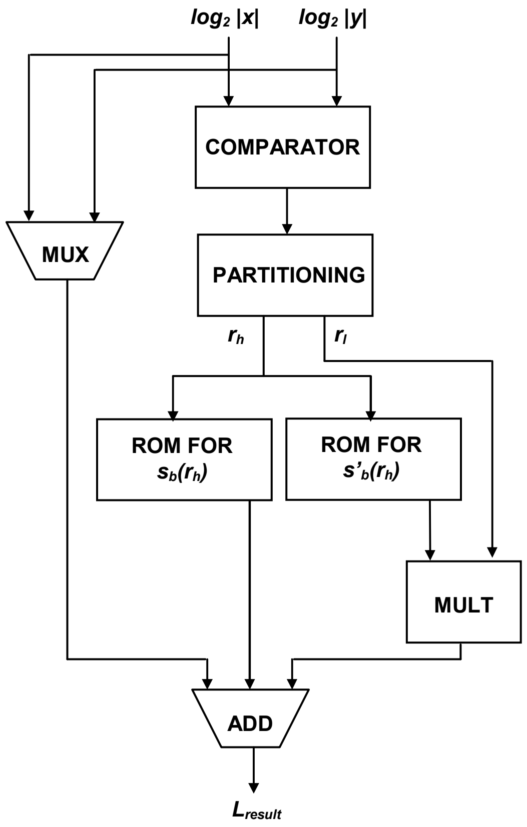


Figure 2-6: LNS adder implemented using linear interpolation.

A suggested interpolation procedure which can offer a wide dynamic range with an independently choosable signal-to-noise ratio was proposed by Henkel in 1989 [45]. The method was based on the Chebyshev approximation with unequally spaced partition points. This approach leads to significant memory reductions but still holds for the addition algorithm only. Note that there is a difference between the addition and subtraction algorithms in the $s_b(r)$ and $d_b(r)$ functions. While $s_b(r)$ is well-behaved, $d_b(r)$ has a singularity when r approaches zero (the function tends to $-\infty$, as shown in Figure 2-4). This can cause a large memory to be required to approximate the $d_b(r)$ function and it is therefore impractical to rely on the interpolation scheme to execute this operation. Furthermore, unacceptable error may also be introduced whenever interpolation is used in this particular region unless partitioning is applied.

A separate proposal in 1994 by Lewis [46] involved the use of a high-order coefficient in the interpolator function, also known as quadratic interpolation. In this technique, a novel scheme using an interleaved memory is introduced which can reduce the storage requirements when compared with linear interpolation. With design up to 32-bit and the accuracy of addition within FLP limits, the critical speed path of the architecture consists of a ROM, two multipliers, three barrel shifters and three stages of adders. Later in 2000, Coleman *et al.* [6] extended the idea of linear interpolation using an error correction algorithm for both addition and subtraction functions. This interpolation scheme for subtraction was incorporated with the newly proposed co-transformation method which will be further elaborated in Section 2.5.5 below. Using Coleman's technique, the speed path comprises of a ROM, a multiplier and three stages of addition process.

Aiming to minimise memory requirements and system complexity, therefore, Arnold [47] recommended in 2001 a multiple-of-four partitioning technique in quadratic interpolation. Nevertheless, even though the proposed address-generation circuit was simpler than that of Lewis and Coleman, this was unfortunately at the expense of a slight increase in approximation error. Still in 2001, Arnold [48] illustrated yet another improved version of Lewis's method [46], now with the advantage that only a single multiplication was required for addition and subtraction algorithms. The implementation of this technique is believed to have either similar or lower memory use than a previous interpolator [49], with corresponding accuracy better than linear interpolation. On the other hand, Fu et al. in [8, 28] described that the implementation of the minimax approximation for the interpolation process could significantly improve the total tables over Lewis and Coleman methods.

However, its worst-case delay was higher than Coleman due to the speed path consists of a ROM, two multipliers and three levels of adders.

2.5.3. Table Partitioning

Generally, partitioning is often combined with an interpolation scheme. Instead of using a single uniform partition (direct lookup table approach) [39], the technique can be realised by segregating the ROM into various sizes of interval mapping with the domain function of addition and subtraction algorithms. These intervals are distributed in smaller regions with similar widths of partition endpoints, hence providing substantial savings in ROM area.

In 1998, Taylor *et al.* [4] suggested a 20-bit LNS processor using a table partitioning method for both addition and subtraction functions. The range of r was divided into a number of smaller intervals with partition endpoints set at integer multiple-of-one for all regions less than -1. For regions close to zero, the multiple-of-half format was employed (i.e $-1 < r < -0.5$, $-0.5 < r < 0$), resulting in two smaller sizes of ROM. In total, 10 ROMs were used to accommodate $s_b(r)$ and $d_b(r)$ functions with total size of about 83.55 kbits, which is 75% less than in the direct lookup table implementation. However, the large size of these tables makes the practical limit for logarithmic arithmetic about 12-16 bits of fractions. Using Taylor approach, it was estimated that LNS add and subtract operations could be completed in 92 ns, a similar value to equivalent FLP processors in those days.

Meanwhile, Stouraitis [50] produced an enhanced version of Taylor's architecture by compressing the table lookup address space and inserting pipelining registers in the addition and subtraction data path. Therefore, with suggestion at a 24-bit LNS processor, the time taken for addition and subtraction could be reduced to 40 ns. Nevertheless, this procedure required a hidden bit to locate the ROM address, which would have an impact on the total area of the system when extending its precision.

One of the most noteworthy partition techniques was presented by Lewis in 1990 [9] using a partitioning procedure concurrently with linear interpolation. An integer multiple-of-two format was adopted at each interval of r less than -1 for subtraction, and in all cases of region r for addition. For subtraction in the region $-1 < r < 0$, the powers of two format was proposed. As tabulated in [9], nearly 2660 kbits were required in total for a 32-bit LNS design, which was impractical for implementation in a single chip using the 3 μm CMOS technology that was available at that time. The delay in the proposed method was assumed to be within two ROM accesses plus two FXP additions, which was slightly slower than the method in [4]. Thus, the implementation of this design might be unattractive for applications demanding high speed configuration.

In 1994, Lewis again [46] applied the table partitioning concept with an interleaved memory scheme. In the initial design, about 287 kbits of memory space were generated when using powers of two partition endpoints at each interval of r for addition and subtraction functions. Subsequently, an attempt was made by Lewis to minimise the area by rounding each table segment up to a multiple-of-eight, and thus only a total of 91 kbits of ROM were needed. Although efficient ROM size can be achieved through Lewis's technique, the introduction of two multipliers in this architecture can potentially increase the cost of the system, either in area or speed.

The other notable approach was suggested by Coleman *et al.* [6], using a partitioning scheme for error correcting interpolation with partition endpoints at powers of two for both addition and subtraction as depicted in Figure 2-7. For subtraction at the case $-0.5 < r < 0$, the co-transformation procedure was introduced. Using this architecture, 321 kbits of storage were required for a 32-bit LNS system. With application only to the addition algorithm, Arnold [47] presented the table partitioning method using a multiple-of-four format which then substantially diminished the total storage to one-third the size of Lewis [46] and one-sixth the memory of Coleman [6]. Regrettably, the implemented architecture exhibits a minor reduction in accuracy compared to a FLP system.

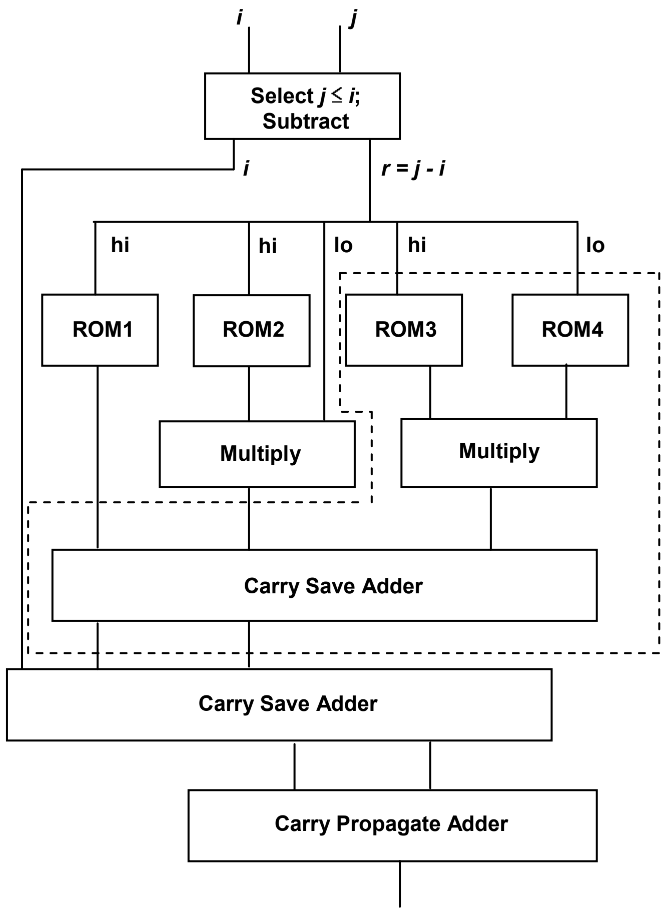


Figure 2-7: Coleman's LNS implementation.

2.5.4. Bipartite Tables

Another method developed as an alternative to conventional lookup tables and linear interpolation is based on bipartite tables [51-54]. Despite requiring a multiplier, this technique only uses two lookup tables which are accessed in parallel, together with an adder for approximating $s_b(r)$ and $d_b(r)$ functions. As claimed in [51], an LNS system that uses bipartite tables will require significantly less memory than one that uses conventional lookup tables. Moreover, apart from only involving an addition operation at the final stage, the technique often has shorter overall delays since the smaller tables have shorter access times too.

Theoretically, to approximate $s_b(r)$ and $d_b(r)$ functions using bipartite tables, the input operand r is divided into three parts, which are denoted as r_0, r_1 and r_2, and have lengths of n_0, n_1 and n_2 respectively. Based on those three partitions, with the example of LNS addition, the function of $s_b(r)$ is approximated as:

$$s_b(r) = s_b(r_0 + r_1 + r_2) \approx a_0(r_0, r_1) + a_1(r_0, r_2) \qquad (2.20)$$

The coefficient $a_0(r_0,r_1)$ for the first table will receive n_0 + n_1 word-lengths, whereas n_0 + n_2 will act as inputs to the second table that provides the coefficient $a_1(r_0,r_2)$. The outputs from the two tables will therefore be added to estimate the $s_b(r)$ algorithm, as depicted in Figure 2-8.

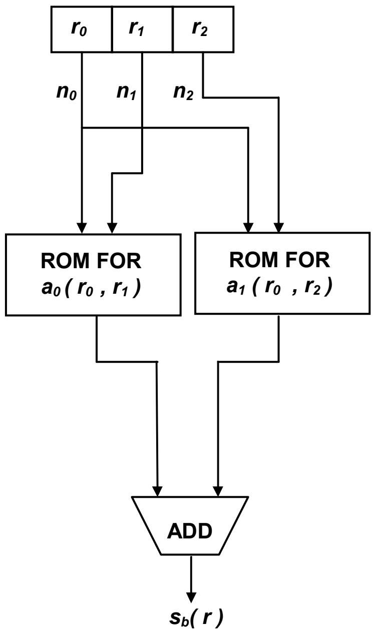


Figure 2-8: Bipartite table architecture.

Among the initial work implementing bipartite tables was a study by Das Sarma and Matula in 1995 [51]. A technique was proposed where the input operand was partitioned into high, middle and low fields of sizes $k+1$, k, k. For example, in the case of a 6-bit operand, the partition will be in the order of 3, 2, 2 of high, middle and low bits respectively. The partitioning concept presented was able to achieve substantial compression of lookup tables compared to the conventional direct lookup table approach, by factors over 4 with a 9-bit input operand. Further refinement of the bipartite table was achieved by Schulte and Stine in 1997 [55], utilizing the concept of symmetry in the table entries. Compared to a direct lookup table, this symmetric bipartite table was 5.6 times smaller with a 16-bit operand and 99.1 times smaller with a 24-bit operand, requiring an estimated total storage of nearly 35 kbits and 2031 kbits for 16-bit and 24-bit operands respectively. A separate proposal was illustrated by Dinechin and Tisserand in 2001 [56], where a multipartite table method was introduced. Instead of using dual tables, the technique employed multiple smaller tables to compute $s_b(r)$ and $d_b(r)$ functions. The synthesis results based on a parameterized library [57, 58] of LNS addition and subtraction using this technique proved that, even though the architecture is capable of achieving higher speed when compared with FLP, it was actually very bulky in size, and hence was limited in practice only to precisions up to 13-bit. Therefore, neither bipartite nor multipartite tables can realistically be considered for long word-length numbers. Furthermore, the multipartite method has the same issue with $d_b(r)$ singularity found in interpolation.

2.5.5. Co-transformation

As discussed earlier, most of the techniques presented so far have the problem of solving the $d_b(r)$ function when r is close to zero. They tend to be either higher in cost, in terms of memory size, or else lower in accuracy. One technique which can overcome this situation uses the co-transformation procedure. The idea behind this technique is to convert the argument of $d_b(r)$ into modified values that are guaranteed to avoid the singularity of the function.

The first noteworthy co-transformation technique was outlined by Coleman in 1995 [59], applying the concept in the region $-0.5 < r < 0$ for the $d_b(r)$ function. When employing this technique, the need for interpolation in the region $-0.5 < r < 0$

can be eliminated, thus substantially reducing the size and complexity of the lookup tables required. Note that for the $s_b(r)$ function, an interpolation scheme was applied through out all regions. In 2000, Coleman *et al.* [6] presented in details the implementation of this co-transformation together with interpolation in a 32-bit system. With significant improvements in accuracy over FLP, a total of 321 kbits were required in order to execute the LNS addition and subtraction. Recently, Coleman *et al.* [7] conducted an experiment to determine the feasibility of integrating the LNS system into a microprocessor based on the proposal in [6]. A chip of a 32-bit LNS microprocessor, named the European Logarithmic Microprocessor (ELM), was manufactured using 0.18 μm CMOS technology. This was compared with the existing FLP DSP device from Texas Instruments, which has one of the fastest speeds obtainable in 0.18 μm technology. Besides clearly verifying that the results were more accurate, the speed of the ELM was also substantially improved over the FLP device, at 24 ns whilst performing addition and direct subtraction, and 32 ns for subtraction using co-transformation.

A different but related co-transformation technique to Coleman's was given by Arnold *et al.* in 1999 [10]. Unlike Coleman's method, which transformed a value at the singularity to a negative argument of d_b that will fall in the region to the left of -0.5, Arnold's method avoids the singularity by transforming to a positive argument of s_b which does not have a singularity. Hence whenever $r > 0$, Arnold's technique is the most appropriate due to the positive value generated for the interpolation after being transformed. If $r < 0$, then Coleman's technique is the most natural to adopt because the transformed argument provided to the interpolation is negative. For that reason, Coleman's method is preferable given that many LNS researchers tended to apply a negative value of r, since this reduces the ROM size dramatically when approaching essential zero (as shown in Figure 2-4).

2.5.6. Hybrid Architecture

A combination of two different data formats, including elements from both LNS and FLP systems, has been exploited a new form of processors known as hybrid number system processors. These allow the multiply and divide operations to be rapidly computed using the LNS format, whilst addition and subtraction are processed efficiently in FLP representation. The first hybrid processor design was presented by Taylor [60], named the $(FU)^2$, which offered a 12-bit FLP datapath whose overall performance was found to demonstrate effectively when compared to that of the conventional FLP system.

With an extension to the 32-bit operands, Lai and Wu [61] proposed a hybrid system architecture that executed multiplication, division, square root and square in a fast manner using LNS. In contrast, the FLP number system was applied to resolve the input, output, addition and subtraction functions. Due to the consuming nature of the overhead operations whilst converting FLP-to-LNS and LNS-to-FLP, lookup tables and linear interpolation algorithms were inserted, whereupon the routine of this processor appeared to compare favourably with a 32-bit FLP DSP device. Since the main obstacle in this hybrid processor was the overhead of converting between number systems, Stouraitis [62] proposed a hybrid technique using a combination of signed-digit (SD) number representation and LNS, called a SD/LNS arithmetic unit. The addition/subtraction was now accomplished even faster than in the classical LNS processor, because the SD adder/subtractor was largely free from serial carry propagation. Figure 2-9 shows the principal concepts of the hybrid number system processor.

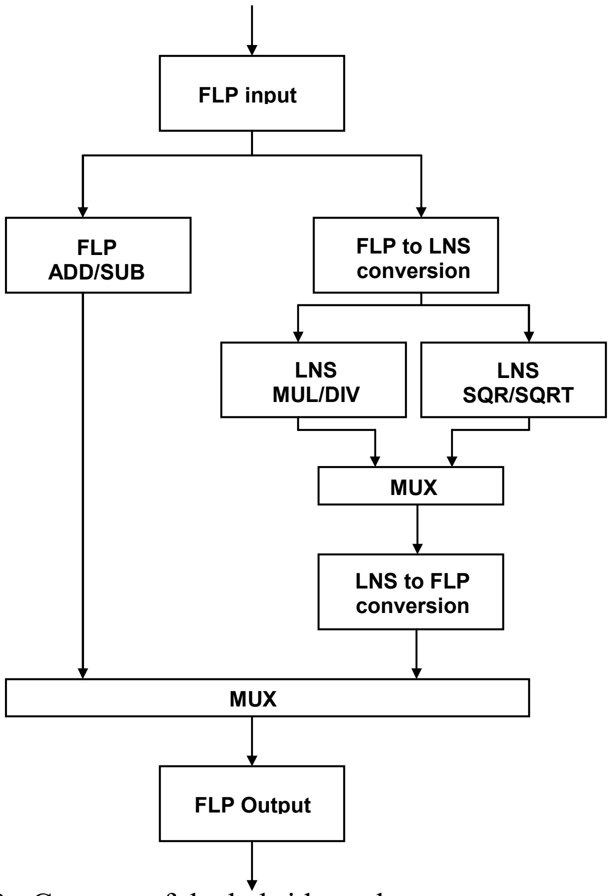


Figure 2-9 : Concept of the hybrid number system processor.

2.5.7. Related Variant Number Systems

Several other techniques have been suggested to minimise the architectural complexity in computing addition and subtraction operations. In 1990, Arnold *et al.* [63] proposed a new number system dubbed the dual redundant logarithmic number system (DRLNS) which was devised to mitigate the singularity issue in subtraction. As opposed to conventional LNS arithmetic, the DRLNS denotes a real number x in positive and negative components, X_p and X_n, similarly to a real number y which then gives Y_p and Y_n.

The exact values can then be represented as:

$$\begin{aligned} x &= b^{X_p} - b^{X_n} \\ y &= b^{Y_p} - b^{Y_n} \end{aligned} \tag{2.26}$$

where *b* indicates the base number. The advantage of adopting the DRLNS was that addition and subtraction shared the same execution process without involving a subtraction logarithm, $d_b(r)$. Hence, the function can be expressed as:

$$\begin{aligned} R_p = X_p + Y_p \quad \rightarrow \quad log_2|R_p| = log_2|X_p + Y_p| &= i_p + log_2|1 + 2^{r1}| \\ &= i_p + s_b(r1) \end{aligned} \tag{2.27}$$

$$\begin{aligned} R_n = X_n + Y_n \quad \rightarrow \quad log_2|R_n| = log_2|X_n + Y_n| &= i_n + log_2|1 + 2^{r2}| \\ &= i_n + s_b(r2) \end{aligned} \tag{2.28}$$

where

$$\begin{aligned} i_p &= log_2|X_p| \\ i_n &= log_2|X_n| \\ r1 &= log_2|Y_p| - log_2|X_p| \\ r2 &= log_2|Y_n| - log_2|X_n| \end{aligned}$$

The subtraction function is completed simply by interchanging the sign of X_p and Y_p with X_n and Y_n accordingly followed by the addition logarithm. However, in spite of being a trivial operation, the DRLNS often loses considerable accuracy as a result of requiring lookup tables when accomplishing the multiplication function. Moreover, the division operation is also difficult to carry out using this procedure [63]. Given these weaknesses, the DRLNS actually did not offer considerable advantages compared to a contemporary LNS system.

The semi-logarithmic number system (SLNS), introduced by Muller *et al.* in [64], is another variant of the new class of number systems. Assuming that a number *x* in the FLP and LNS can be represented by:

$$\begin{aligned} x_{FLP} &= (1-z)\cdot(-1)^{s_x}\cdot m_x \cdot 2^{e_x} \\ x_{LNS} &= (1-z)\cdot(-1)^{s_x}\cdot 2^{L_x} \end{aligned} \tag{2.29}$$

where z corresponds to zero, these two expressions can then be generalised in SLNS format by introducing new parameters:

$$x_{SLNS} = (1-z)\cdot(-1)^{s_x}\cdot \alpha m_x \cdot 2^{\beta e_x} \qquad (2.30)$$

Conceptually, the SLNS constitutes a compromise between FLP and LNS. In the case of $\alpha = \beta = 1$, the FLP format was applied to perform the operations, whereas for $\alpha = m_x = 1$ and $0 < \beta \leq 1$, LNS was adopted. The advantages of the SLNS are that multiplication and division can be easily completed as in the LNS, and a reduction in lookup tables can be obtained to perform addition and subtraction. According to the authors, slightly lower accuracy compared to LNS and FLP was deemed to be the only drawback, but the scheme was still pragmatically good enough for various DSP applications analogous to those using traditional LNS procedure.

Instead of using binary numbers to represent values in the classical LNS system, another approach proposed by Arnold in 2005 [65] was called the Residue Logarithmic Number System (RLNS). Here the values used to approximate the LNS operations were based on the residue number system. Although multiplication and division can be faster than any other operations, like that of conventional LNS, the RLNS still experienced the same issue in addition of huge lookup tables being required. As well as that, without an evaluation of the performance of the subtraction operation, its overall efficiency remains uncertain.

2.6. Performance Analysis

Three crucial elements dominate previous works when proposing new algorithms or architectures for an LNS system. Speed is always a key factor when producing any high performance LNS system. A high speed system can not only execute many operations with the lowest possible delay, but can also minimise the component and system related noise which occurs in DSP systems. Researchers have also strived to reduce the large areas involved in computing LNS addition and subtraction operations resulting from the lookup tables required to store the values for approximating the functions. However, an LNS system with high speed and reduced area but accuracy outside FLP limits would be worthless. Therefore, the accuracy of the results is of the utmost importance.

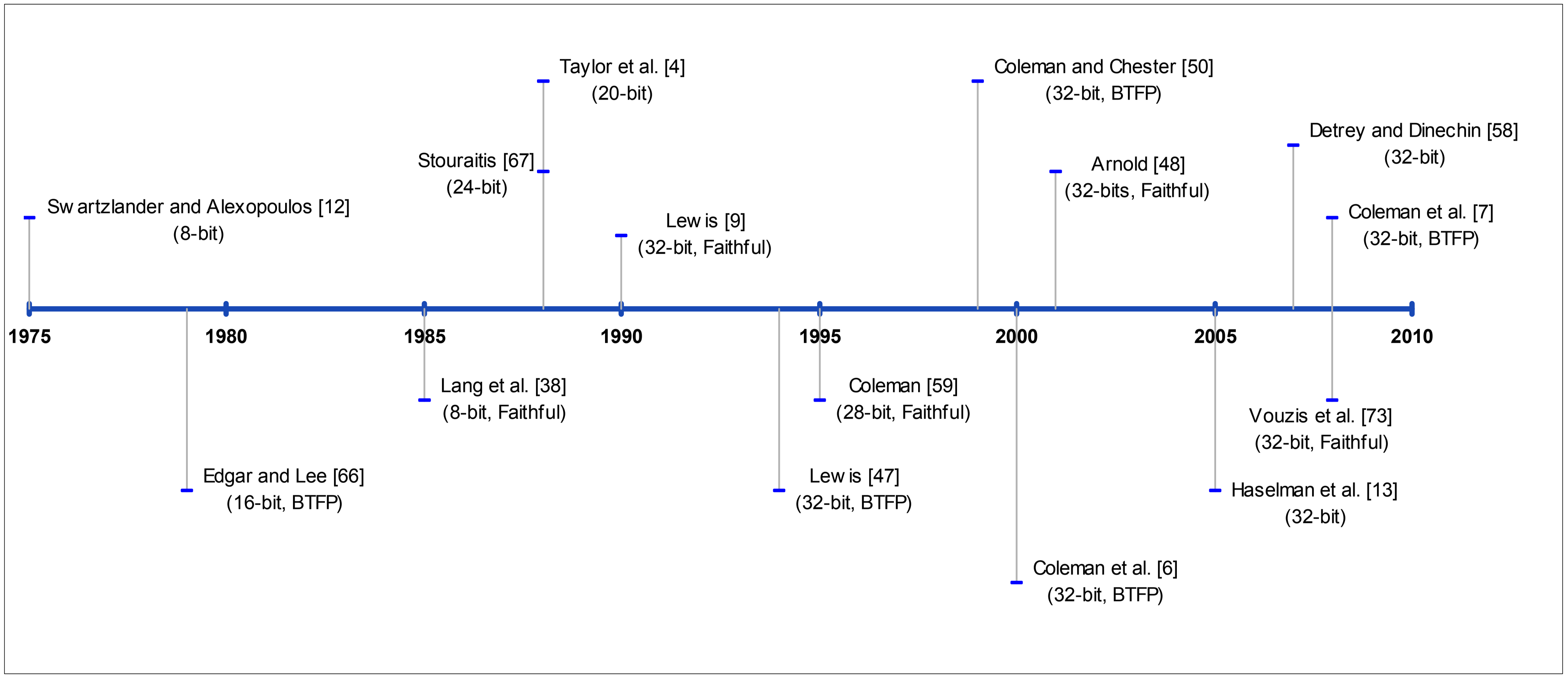
Based on the several different LNS techniques to compute addition and subtraction operations as discussed in Section 2.5, it can then be summarised as in Table 2-2. Obviously, it can be seen that by implementing the co-transformation approach with the interpolation process, less total storage can be achieved especially when subtractions near singularity region. Moreover, with significant improvements in accuracy, the worst-case delay in operating add and direct subtract functions was also found to be better than equivalent FLP units. Therefore, it can be concluded that this approach may now be the best technique to be used as a benchmark to improve further the LNS system. A summary of LNS designs over the years is also given in Figure 2-10.

2.7. LNS for Specific Applications

The ubiquity of the FLP unit in many DSP devices since the 1980s and rapid growth in the DSP market in every year has prevented much penetration of LNS arithmetic into various DSP applications. The lack of a standard format like, for example, the IEEE 754 for FLP [3], could be one of the main reasons that LNS systems have only appeared in limited classes of industrial applications. Furthermore, few LNS architectures have been shown to rival the speed and accuracy of existing FLP systems, which has also impeded their realisation as an alternative to FLP units. Nevertheless, numerous studies and several implementations of the LNS have proved that they work effectively for specific hardware designs.

Table 2-2 : Summary of the LNS techniques.

Technique	Advantage	Disadvantage
Direct Lookup Table	-	▪ Slower than FLP ▪ Not suitable for long word-length numbers
Interpolation	▪ Less total lookup tables than direct approach	▪ Less accurate than FLP and increase in the lookup tables when performing subtractions near singularity
Table Partitioning	▪ Less total lookup tables than interpolation alone	▪ Less accurate than FLP and increase in the lookup tables when performing subtractions near singularity
Bipartite Tables	▪ Faster than FLP	▪ Limited to short word-length numbers ▪ Bulky in size
Co-transformation with Interpolation	▪ Faster than FLP ▪ Accuracy better than FLP ▪ Reduce size and complexity of lookup tables when performing subtractions near singularity	-
Hybrid Architecture	▪ Execute add and subtract operations faster than conventional LNS	▪ Costly in converting between number systems
DRLNS	▪ Easy to compute subtraction function using addition algorithm	▪ Less accurate than FLP when performing multiplication and division


Swartzlander and Alexopoulos [12] (8-bit)
Edgar and Lee [66] (16-bit, BTFP)
Lang et al. [38] (8-bit, Faithful)
Stouraitis [67] (24-bit)
Taylor et al. [4] (20-bit)
Lewis [9] (32-bit, Faithful)
Lewis [47] (32-bit, BTFP)
Coleman [59] (28-bit, Faithful)
Coleman and Chester [50] (32-bit, BTFP)
Coleman et al. [6] (32-bit, BTFP)
Arnold [48] (32-bits, Faithful)
Haselman et al. [13] (32-bit)
Detrey and Dinechin [58] (32-bit)
Coleman et al. [7] (32-bit, BTFP)
Vouzis et al. [73] (32-bit, Faithful)
1975
1980
1985
1990
1995
2000
2005
2010

BTFP – accuracy better-than-floating-point (0.5 ulp error requirement)

Faithful – faithful rounding scheme (1 ulp error requirement)

Figure 2-10 : LNS trends vs time.

In 1983, Swartzlander *et al.* [40] suggested a Fast Fourier Transform (FFT) which would provide lower quantisation error than those of the FLP and FXP number systems. Another proposal examining LNS in filtering systems by Vainio and Neuvo in [67] took measurements from a constructed integrated circuit which showed that the sampling frequency of the LNS filter was comparable to other high performance DSP processors at that time.

Das *et al.* [68] identified ways of evaluating the trigonometric operations using LNS processors, which supports the arguments for the adaptability of the LNS system in a range of applications. In 2000, the development of the 32-bit LNS processor [6] demonstrated superior achievements over the equivalent 32-bit FLP system, where increases in speed and accuracy were gained. The simulations were then supported with an analytical study of a fabricated chip [7] which yielded similar outcomes when validated against a high performance FLP device using the same technology. Cost sensitive applications such as in multimedia always need a less costly architecture. In line with this, Arnold and Walter [69] produced a more compact LNS ALU with only a modest increase in error, whose unrestricted faithful rounding criteria is allowable in certain applications. The work in [70] therefore confirms the efficiency of this less accurate method when applied to Motion Picture Expert Group (MPEG) decoding architecture. Besides that, the implementation of the LNS approach for arithmetic operations in GRAPE-6 microprocessor design has contributed to a great success in terms of speed [94].

Moreover, LNS has also become convenient for calculating general matrix and complex arithmetic operations [71]. The robustness of the logarithmic multiply-accumulate operator can also be seen in digital hearing aid systems [72]. Furthermore, spam email now outnumbers legitimate messages by more than two-thirds, and so hardware architecture like the naïve Bayes inference engine has been proposed to monitor email content. Technically, such a system involves complex arithmetic operations which, in turn, produce computational noise. Therefore, the LNS number format has been proposed [73] as an attractive solution to simplify naïve Bayes computations.

Ultimately, the advantages of applying LNS arithmetic units for a wide variety of DSP applications as explicitly specified in a wealth of literature have been marked as a new trend in the evolution of the DSP world.

2.8. Summary

Lying at the heart of digital computer systems, a computer arithmetic unit can use either the FXP or FLP data format. Over the past three decades, LNS has also been used as a good alternative in computing basic arithmetic functions, especially for a large range of numbers. However, to date, its implementation is still restricted by the complexity of performing addition and subtraction resulting from the need for large lookup tables. Several schemes have been suggested to circumvent the singularity issue in the non-linear function of LNS subtraction. From this review of the literature, it can be concluded that the most notable method [6, 49] uses a mixture of co-transformation and error correcting interpolation, whereby reasonable storage requirements along with better speed and accuracy compared to FLP units are attained. As of now, it has been shown that LNS systems may be workable in a broad range of DSP applications and hence a new revolution in the DSP world is now underway.

CHAPTER 3

3. Metrics for Measurement and Design Methodology

3.1. Introduction

In this chapter, the metrics required for measurement whilst performing the simulation and synthesis processes are discussed. This includes the error analysis procedure and two types of performance estimation, relating to timing and area. Despite that, functional evaluation is also crucial, and hence for each circuit are compared between derived behaviour and desired behaviour as to confirm that the system works as expected.

In addition, the design flows of the simulation and synthesis processes are also explained. Typically, once functionally verified through the simulator program, each of the arithmetic designs is translated into VHDL code before being constrained synthesised in Faraday 0.18 µm CMOS technology based on a 32-bit system.

3.2. Metrics for Measurement

In making a selection of the most advanced LNS arithmetic unit for a particular application, several metrics must be considered. This will ensure that the performance of the chosen LNS system is justified and can be evaluated through a series of measurement processes. The criteria assessed in this thesis are explained below.

3.2.1. Error Analysis

Error characteristics are often used to justify the accuracy of the results produced in any arithmetic system. It is well known that in the LNS unit no errors occur in multiplication and division. However, addition and subtraction in LNS frequently suffer to sustain the error within the FLP boundary, in which has a worst-case relative error of 2^{-f-1} [28]. In order to measure the accuracy of the LNS system and compare it with the FLP system, the mathematical expressions defined in [49] are adopted.

First, let C and F be the exponent and f-bit mantissa of the FLP number system. An approximation result, $\hat{A}$, produced by a practical implementation is in error of the correct result, A, so that the absolute error can be represented as $e = \hat{A} - A$, with the assumption that the input operands are exact values. For a given operation, the maximum relative error of the system can be expressed as:

$$e_{max\ rel} = max\left(\frac{\hat{A} - A}{2^{C} \cdot 2^{-f}}\right) \tag{3.1}$$

and similar definitions apply for $e_{min\ rel}$ and $|e|_{max\ rel}$. Since these errors are directly related only to the absolute magnitude of the exact value, controlled by C, it is thus more realistic to define the error in terms of the exact value itself. Therefore, the maximum relative arithmetic error and an average relative arithmetic error can be written as:

$$e_{max\ rel\ arith} = \frac{max(\hat{A} - A)}{2^{-f} \cdot A} \tag{3.2}$$

$$e_{av\ rel\ arith} = \frac{1}{2^{-f} \cdot n} \sum_{i=1}^{n} \frac{\hat{A}_i - A_i}{A_i} \tag{3.3}$$

and again equally the same for $e_{min\ rel\ arith}$, $|e|_{max\ rel\ arith}$ and $|e|_{av\ rel\ arith}$.

When considering the error requirement in the LNS system, the expression can be quoted relatively identical with the equation (3.1), given that both logarithms forming the inputs to an operation are exact. Whilst the exact logarithm can be regarded as *I*, the result generated by the real implementation can therefore signified as $\hat{I}$ and hence in error by $e_{log} = \hat{I} - I$. Thus, the maximum relative error in the logarithm format can be quoted as:

$$e_{max\ rel\ log} = \frac{max(\hat{I} - I)}{2^{-f}} \tag{3.4}$$

and correspondingly so for the errors $e_{min\ rel\ log}$, $|e|_{max\ rel\ log}$, $e_{av\ rel\ log}$ and $|e|_{av\ rel\ log}$ as before. For a direct comparison between the error yielded in the FLP number and that in the equivalent LNS system, the error returned in the LNS format can be exponentiated and thus would provide a similar error to that of the FLP calculation.

$$e'_{max\ rel} = \frac{2^{max(\hat{I} - I)} - 1}{2^{-f}} \tag{3.5}$$

Since this is similar to $e_{max\ rel\ arith}$, thus $e_{min\ rel\ arith}$, $|e|_{max\ rel\ arith}$, $e_{av\ rel\ arith}$ and $|e|_{av\ rel\ arith}$ are also the same. With all classes of error clearly defined, the theoretical values of the errors [49] for each of the 32-bit FLP and LNS numbers are summarised in Table 3-1. Although the practical LNS results for addition and subtraction may differ in comparison to the theory, at least conceptually, the LNS has an inherent better worst-case relative error compared to FLP.

As can be observed in Figure 2-10, many studies have presented LNS addition and subtraction architecture that can achieve Better-Than-Floating-Point (BTFP) error behavior [6, 74, 75]. As its name implies, the LNS architecture in BTFP mode will guarantee the production of smaller worst-case relative error than FLP. Conversely, Arnold and Walter [69] suggested that, by relaxing the rounding criteria known as unrestricted faithful rounding, the resulting evaluation is the nearest or next nearest machine number representation. Eventually, this will reduce the total area of the LNS system and thus produce a more compact LNS ALU unit.

Table 3-1 : Best case theoretical errors.

Error Type	ADD/SUB		MUL/DIV	
	FLP	LNS	FLP	LNS
$e_{max\ rel}$	+0.5	▨	+0.5	▨
$e_{min\ rel}$	-0.5	▨	-0.5	▨
$e_{max\ rel\ arith}$	≈ +0.5	+0.3464	≈ +0.5	0
$e_{min\ rel\ arith}$	≈ -0.5	-0.3464	≈ -0.5	0
$e_{av\ rel\ arith}$	0	0	0	0
$\lvert e\rvert_{av\ rel\ arith}$	0.1733	0.1733	0.1733	0
$\lvert e\rvert_{max\ rel\ log}$	▨	0.5	▨	0
$\lvert e\rvert_{av\ rel\ log}$	▨	0.25	▨	0

However, this mode is more likely to be workable for certain DSP applications such as those in multimedia systems in which a reduced error constraint is acceptable. As the purpose here is to realise an LNS design that can serve a diverse range of DSP applications, in this work the BTFP mode is adopted for the evaluation of the addition and subtraction functions.

In order to do an error analysis for the addition and subtraction functions, it is not necessary to evaluate all possible combinations of operands *j* and *i*. The analysis has to be performed merely over all negative values of *j*, where *i* is restricted to zero in accordance to Theorem 1 as depicted in [49]:

*"**Theorem 1**. If the LNS addition and subtraction operations yield errors within a given $e_{max\ rel\ log}$ over all negative values of j for i = 0, then they yield the same $e_{max\ rel\ log}$ over all values of j for all values of i. An implementation can thus be regarded as fully verified if it can be verified over this subset."*

Coleman et al.

3.2.2. Functional Evaluation

In functional evaluation, a design can be certified as successfully verified when the simulation results are mathematically identical with the expected outcome. The process commonly starts by describing each circuit using either hardware description language (HDL) or schematic entry. In this thesis, VHDL (very-high-speed integrated circuit hardware description language) was used to construct the system as a result of its advantages over schematic based design such as the capability to implement the behavioural hardware description and the portability of the code due to a standardised language. Then, in simulating the design, a top-level simulation environment known as a testbench circuit was created, which consists of 100 random pattern numbers. The test vectors generated cover all the crucial cases that are expected to arise in the system. Using the ModelSim XE III/Starter 6.4b simulator, the system was simulated according to the specified test vectors.

The simulation results were then evaluated against the expected results retrieved from the simulation process based on a design in the C programming language. If a discrepancy was found, the description in VHDL code was modified accordingly before repeating the functional evaluation process. Whenever the expected and observed results matched, the system could be considered to be functionally correct.

3.2.3. Timing Evaluation

The main purpose of performing timing analysis is to investigate the delay characteristics, in terms of maximum or minimum delays, that occur in a design. In general, the maximum or so-called worst-case delay in a circuit results from the cell and the interconnection delays on the critical path. Conversely, the shortest signal propagation delay path in a combinational circuit represents the minimum delay in the system. The techniques adopted to evaluate propagation delay vary from manual verification, which is mainly used for a custom design, to applying automated timing analysis using specific CAD (computer-aided design) synthesis tools.

For more rapid and accurate results, an automated approach is selected in this study. The Synopsys Design Compiler tool based on a constrained synthesis automatically computes the maximum path delay required for the design whenever a relevant timing command is written. However, if a reported delay diverges from the desired goal, it can be improved by redesigning or optimising the circuit using a different topology. Additionally, several timing directive commands in the synthesis tool may also be used to reduce the critical path delay in the design. For ease of comparison, all timing estimations are given in nanosecond (ns) units.

3.2.4. Area Estimation

One of the design criteria currently receiving increased attention is the size of a circuit. A smaller total area can lead to the best implementation due to incurring lower costs. An exact estimation of the area is normally calculated after a circuit has been placed and routed, taking into consideration all the cells, wiring interconnections, and input and output pads. However, due to recent rapid increases in circuit complexity and the need to reduce the time-to-market, CAD tools that can help to produce an early estimation during the design process are now imperative. Therefore, the area information reported in this thesis was estimated from the total cell area data generated by the Synopsys Design Compiler tool during a constrained synthesis process.

Total cell area is typically approximately proportional to the number of the minimum standard cell size contained in a design, which in this case is the 2 input NAND gate. In order to convert the value of total cell area into square micron (μm^2) units, the height and the width of the 2 input NAND gate need first to be extracted from the *.lib* and *.lef* files. Then, these values are multiplied by the total cell area number before the final result can be derived as shown in equation (3.6):

$$area\ in\ \mu m^2 = total\ cell\ area \times width\ of\ NAND\ gate \times height\ of\ NAND\ gate \quad (3.6)$$

Below are the height and the width of the 2 input NAND gate based on Faraday 0.18 µm CMOS technology.

- Faraday 0.18 µm CMOS technology
  - minimum height of 2 input NAND gate = 5.04 µm
  - minimum width of 2 input NAND gate = 0.62 µm

Despite neglecting the circuit connectivity in the area estimation, the result still yields acceptable accuracy in representing the total area of a design. This argument is supported by the area evaluation technique which is most commonly used in the literature [16, 76], based on the unit-gate model. In addition, the area estimation adopted here has been found to be consistent with the result provided from actual routing, as it has been proven in [77].

3.3. Design Methodologies

The selection of an appropriate design flow and CAD tool is important in producing an efficient design. Typically, the choice of tools must complement the design flows. Therefore, the simulator design for LNS addition and subtraction was first explained which mainly written in C language. Then, the basic design flow in constructing the LNS arithmetic unit is briefly described along with the CAD tool implemented in this thesis. In addition, the procedures used for the synthesis process are also explained in detail.

3.3.1. Simulator Design Flow

In order to validate the workability of the LNS design illustrated in this thesis, two general simulator programs were modelled for addition and subtraction operations. The results produced by these simulators can then be verified against the published results [6], looking at the error characteristics of both functions. Besides this, the simulators were constructed to observe the best combinations of lookup tables for the interpolator by ensuring errors within FLP boundaries. Additionally, the flexibility of these tools has also made it viable to modify them repeatedly in order to verify the efficiency and practicability of implementing various types of interpolation procedures.

The designs of the simulator were written in C language and the compilation processes were executed in an Intel Core 2 processor using GNU Compiler Collection (GCC), the standard compiler software that supports C programming in the Linux operating system. For measuring the approximation error, an accurate result produced by the double-precision format of the FLP unit embedded in an Intel Core 2 processor was adopted as a benchmark.

Basically, the developments of the simulator for an addition and subtraction will most likely be the same, because the interpolator is used to approximate both functions. However, because of the difficulty in performing accurate interpolation in the region $-0.5 < r < 0$ for subtraction, a co-transformation procedure is employed as explained precisely in Chapter 5. Thus, the essential elements required in building the simulator are briefly indicated below and translated into the flow diagram in Figure 3-1:

- Create the support function algorithms, exponent and logarithm, which are two functions widely used throughout the simulator.
- Define the interpolator model which performs LNS addition and subtraction for the entire range of r, except for subtraction in the range of $r > -0.5$.
- Define a co-transformation scheme to compute the LNS subtraction in the region of $-0.5 < r < 0$ which is only applicable for the subtraction function.

- Create a table generation module for virtually developing the lookup tables to represent the memories.
- Compute approximation results of LNS addition and subtraction according to which region *r* falls into.
- Compute exact results of FLP addition and subtraction which will be used as a standard to compare with the approximated results of an LNS system.
- Calculate the error produced in the LNS design in comparison with that of the FLP unit and report the various error characteristics as detailed in Section 3.2.1.

Whilst the table sizes for the co-transformation architecture were constantly fixed for the entire process, a number of simulations were performed to determine the most appropriate lookup table sizes that need to be implemented for the interpolation procedure. The most suitable sizes will only be decided whenever the worst-case error falls below an equivalent of 0.5 FLP LSB. The most common powers of two partitioning concept was applied during the interpolation process, yielding six segments throughout the system before approaching the nearest point to an essential zero. Table 3-2 shows the general variables for the interpolator module which were modified in each simulation.

Table 3-2 : Simulation variables for the interpolator.

Parameter	Description
F	Stored function value at r_n
D	Stored function derivative at r_n
E	Stored maximum approximation error in (r_{n+1}, r_n)
P	Stored proportion of an error for the region that yields the largest absolute maximum error
δ	Current value of $r - r_n$
r	Current operand difference, in guarded format
r_n	Stored interpolation point

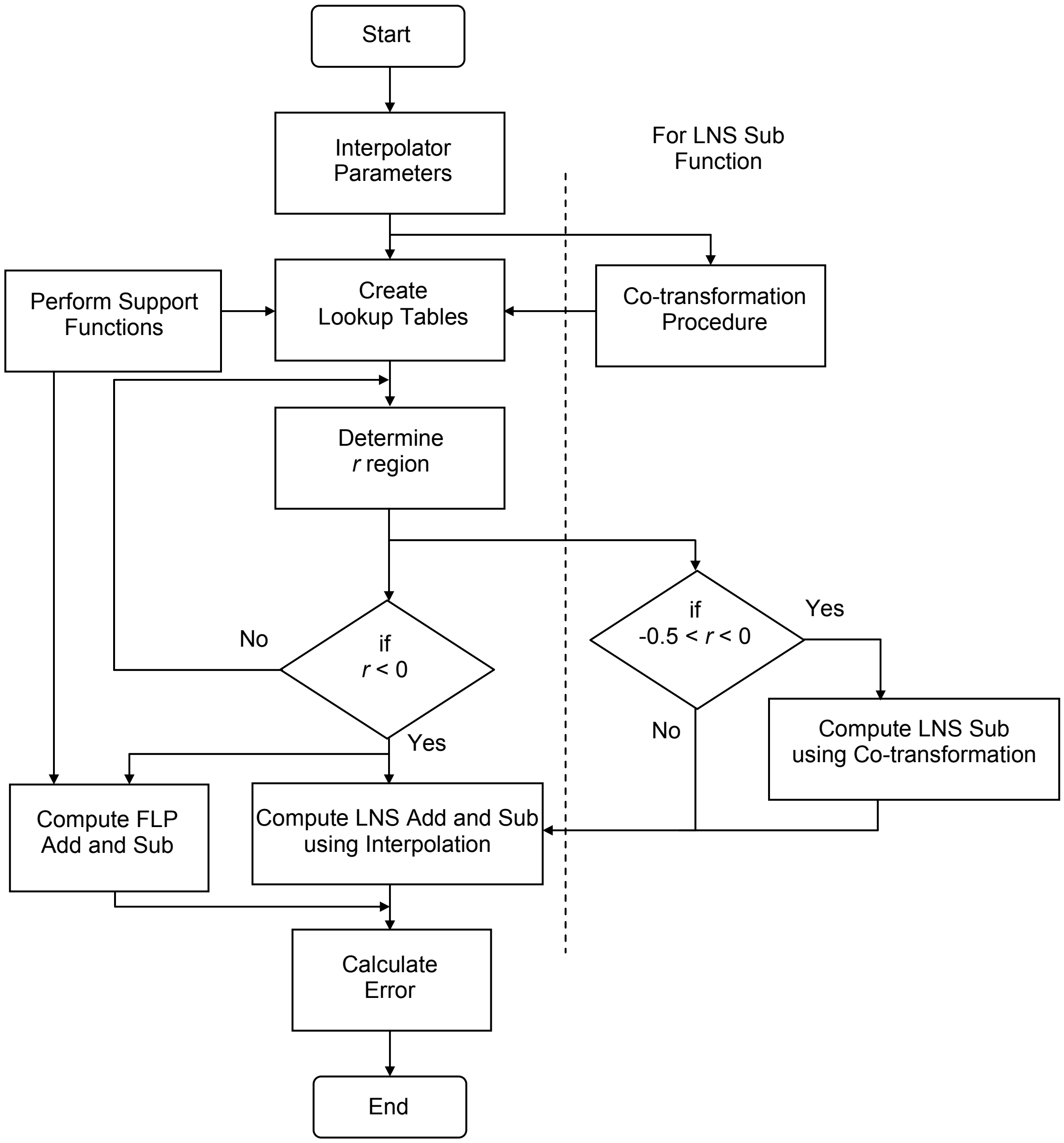


Figure 3-1 : Simulator design for the LNS addition and subtraction.

3.3.2. Circuit Design Flow

The flowchart in Figure 3-2 portrays the basic design flow of the LNS ALU system. The process is divided into two separate stages, namely functional verification and the synthesis process. Using ModelSim XE III/Starter 6.4b simulator as a CAD tool, the VHDL description of the LNS design was first written. The coding was then simulated in order to determine whether or not the design performs the desired functions. Whenever the design did not function as required, the VHDL code was modified accordingly before repeating the simulation process. Once the coding was functionally verified as correct, it was then transferred to the synthesis process.

While executing the synthesis process, the Synopsys Design Compiler tool was adopted. In this phase, the design was transformed into equivalent gates before timing evaluation and area estimation were performed. Whenever performance did not meet the desired goals, the design could be re-constructed or optimised before applying the process again. The design cycle was completed when the system met the defined objectives mentioned in Chapter 1. Further elaboration about the synthesis flow is given in Section 3.3.3.

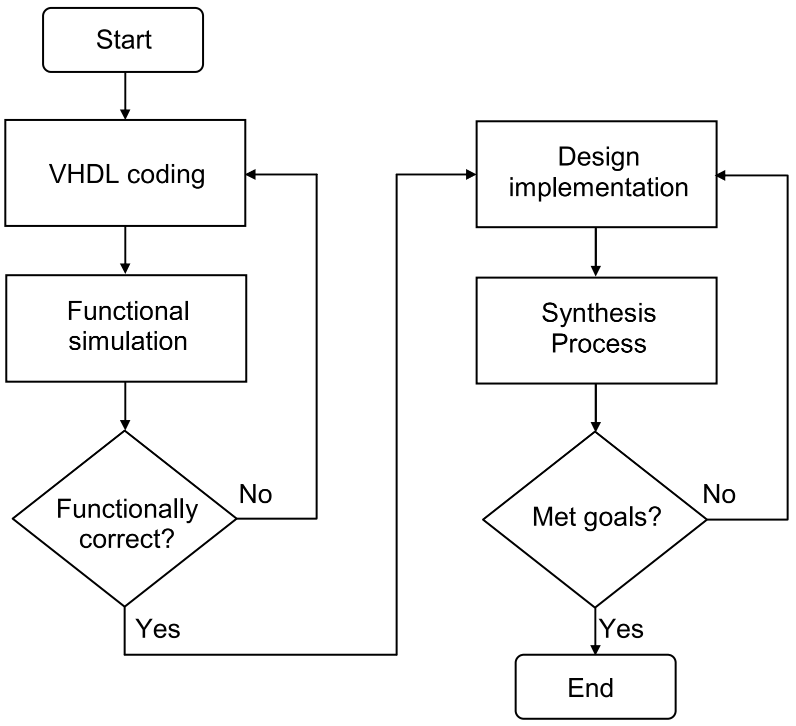


Figure 3-2 : Basic circuit design flow.

3.3.3. Synthesis Design Flow

The Synopsys Design Compiler tool was employed to perform the synthesis work in this thesis, and the steps applied in carrying out the process are depicted in Figure 3-3. The process begins by inserting the design files written in the VHDL language in the input files setting. Next, the links, targets and symbols for the libraries were specified accordingly. Conceptually, the relevant information about cells or gates based on the technology libraries applied was embedded in the link and target libraries settings. In this study, only one technology library was adopted in synthesising the circuit, Faraday 0.18 µm CMOS technology.

Then, reading the design written in the VHDL format can be accomplished by using two commands, namely *analyze* and *elaborate*. Using these commands, the pre-synthesis schematic design could now be viewed.

It is known that in most CMOS technologies, the performance of a system especially in terms of speed, may vary according to operating conditions such as temperature, voltage and process factors. Since variations in these factors were of no concern in this study, predefined sets of operating conditions in the technology library were used, as described in Table 3-3.

Table 3-3 : Operating conditions setting.

CMOS Technology	Temperature	Voltage	Process
Faraday 0.18 µm	25^oC	1.8 V	1.00

Another important procedure in controlling the synthesis of the design is the design constraint settings. Realistic design constraints will allow the compiler to achieve the design goals without violating design rules during the process. Here, constraints were added for timing (clock and delay) with the purpose of attempting to produce the best possible worst-case delay in the design.

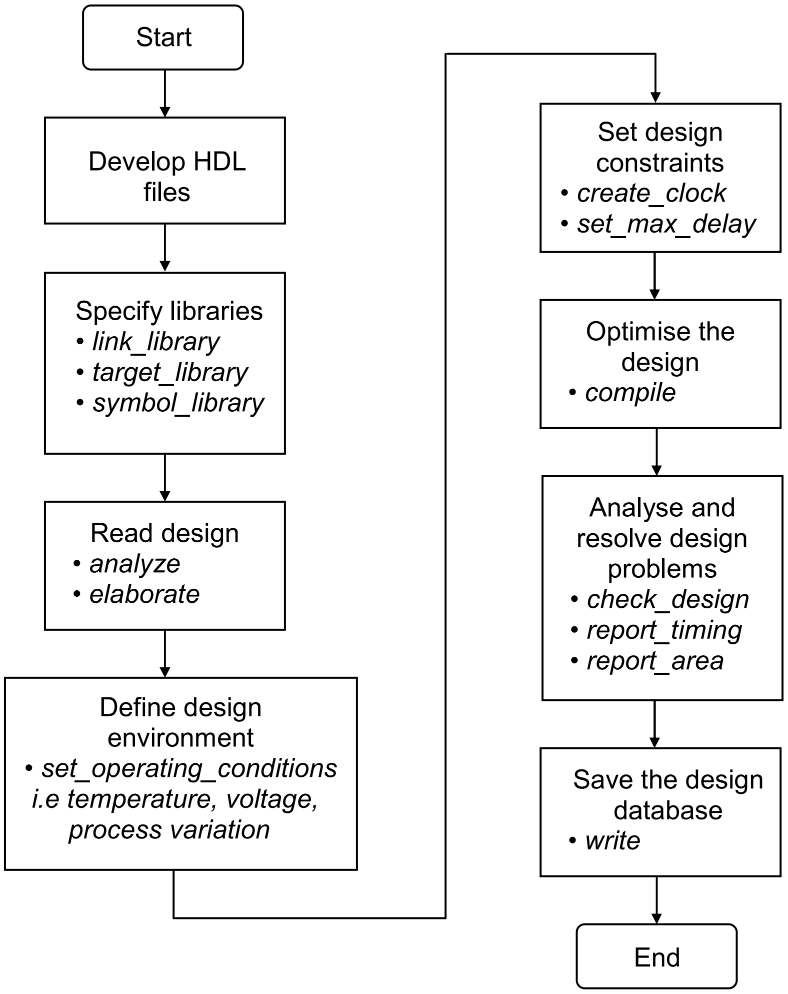


Figure 3-3 : Synopsys synthesis design flow [78].

Once all the requirements were loaded, the design was now ready for the synthesis and optimisation processes. So as to obtain the greatest optimisation, the *compile* option was invoked in the design compiler.

After the synthesis procedure, reports for timing and area were generated in order to analyse the characteristics of the optimised design. If the results needed to be improved, the design could be updated where possible before repeating the synthesis process. Finally, when the synthesis results had reached the specified goals, the final design was saved as a gate-level netlist in Verilog HDL format.

3.4. Summary

In order to evaluate the efficiency of the LNS arithmetic unit particularly for addition and subtraction, a metric such as worst-case error of the system is often examined and compared with the FLP equivalents. Besides that, the functional evaluation can be used to verify the simulation results against the expected results. Another important metric to be considered in this thesis was the timing evaluation. Through performing timing analysis, the worst-case delay of the system can be investigated. Area estimation was used to examine the total size of the architecture in silicon.

Apart from that, this chapter summarised three different design flows which will be applied in building the LNS system. The simulator design flow described the steps used to validate the workability of the design before being translated into circuit design. Commonly, C language was used to represent the design. On the other hand, the circuit design flow briefly explained the process involved in constructing the LNS arithmetic unit. There were two separate stages required, namely functional verification and the synthesis process. The details of the synthesis process were clearly elaborated in the synthesis design flow, where from this procedure, the performance of a system can be measured.

CHAPTER 4

4. Recent 32-bit Arithmetic Implementations

4.1. Introduction

In this thesis, the LNS system adopted in the ELM processor is chosen as a benchmark for comparison. This is due to the fact that the system is able to provide better accuracy and speed than FLP whilst performing addition and subtraction. Therefore, Chapter 4 reviews the design in detail before reconstructing the architecture using the same HDL model as used in the ELM itself. Particularly for the subtraction operation, the design consists of two separate architectures, a co-transformation and interpolation. However, only the interpolation process is described herein while the co-transformation procedure will be elaborated in Chapter 5. The summary of the design resulting from the simulation and synthesis processes are also discussed which are then used for analytical comparison. Apart from that, the performance of several FLP devices are also examined where the results can also be used for analytical study.

4.2. Leading Published Design: ELM processor

Many of the previously published LNS systems focus mainly on addition operations, and many fewer studies report solutions to compute the subtraction function, especially in the crucial region of $r > -1$. One system that promises better accuracy and speed than FLP in addition and subtraction is the LNS architecture presented in the ELM processor [6, 7]. Here, the co-transformation approach is combined with the error correcting interpolation scheme to execute the operations. Hence, it is worthwhile to acknowledge this technique as a leading published design, because its performance is much more appealing than the other methods. Thereby, the arithmetic unit adopted in the ELM system is reviewed to provide a benchmark design for comparison.

In order to calculate the addition and subtraction algorithms within the ELM, the *r* value is separated into various ranges of intervals at different widths of Δ. Due to the fact that the curves of the addition and subtraction tend to reach an essential zero point with decrease in *r*, the total storage requirements can be reduced by progressively increasing the width of Δ at each segment. For ease of implementation, the range of *r* is segmented at each powers of two, which then gives six segments. Table 4-1 illustrates the segmentation procedure and the corresponding Δ, whilst Figure 4-1 graphically depicts the partitioning concept. On the other hand, Figure 4-2 explains the definitions of intervals, regions and segmenting schemes, which are terms used repeatedly in this thesis.

An interval is a region that covers the width of Δ in which it is used to interpolate a function. When there is a set of one or more intervals, it can be formed into a single region. A segment can be understood as a formation of various regions and it is commonly partitioned in the range of powers of two.

Table 4-1 : Segments and Δ in the ELM system.

Segment	Addition		Subtraction	
	Region	Δ	Region	Δ
1	$-1 < r < 0$	1.0	$-1 < r < -0.5$	0.5
2	$-2 < r < -1$	1.0	$-2 < r < -1$	1.0
3	$-4 < r < -2$	2.0	$-4 < r < -2$	2.0
4	$-8 < r < -4$	4.0	$-8 < r < -4$	4.0
5	$-16 < r < -8$	8.0	$-16 < r < -8$	8.0
6	$-32 < r < -16$	16.0	$-32 < r < -16$	16.0

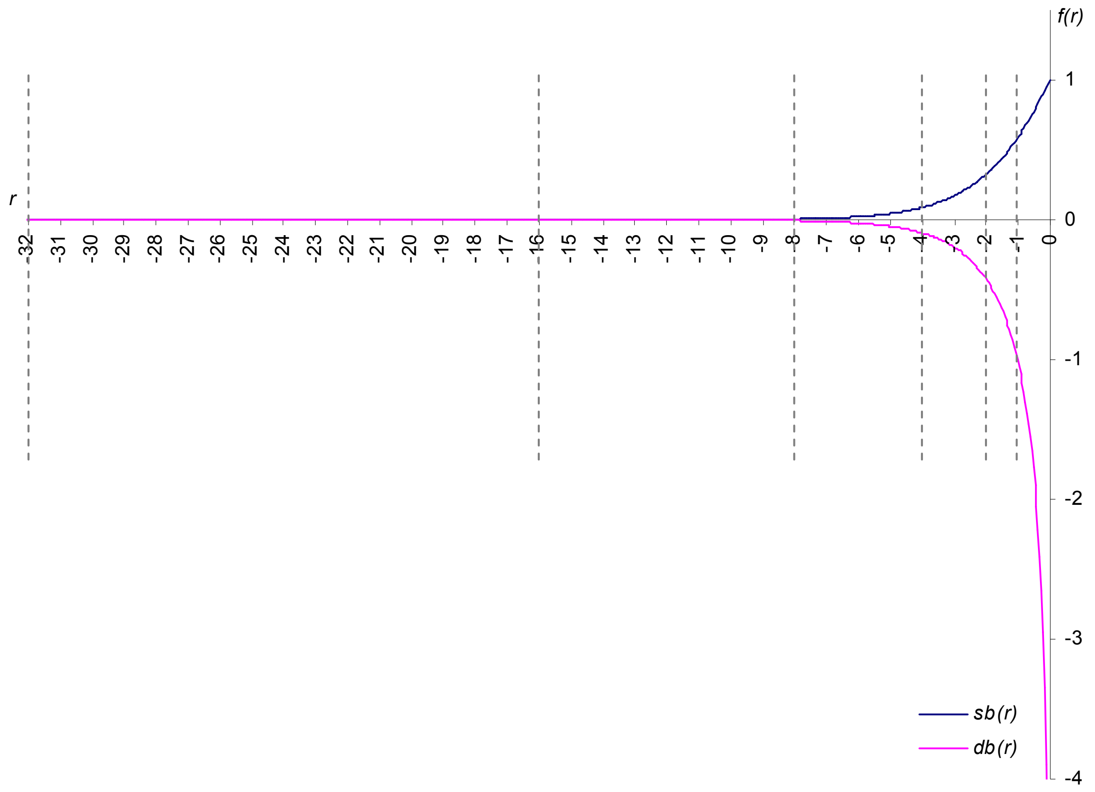


Figure 4-1 : Partitioning concept for addition and subtraction functions.

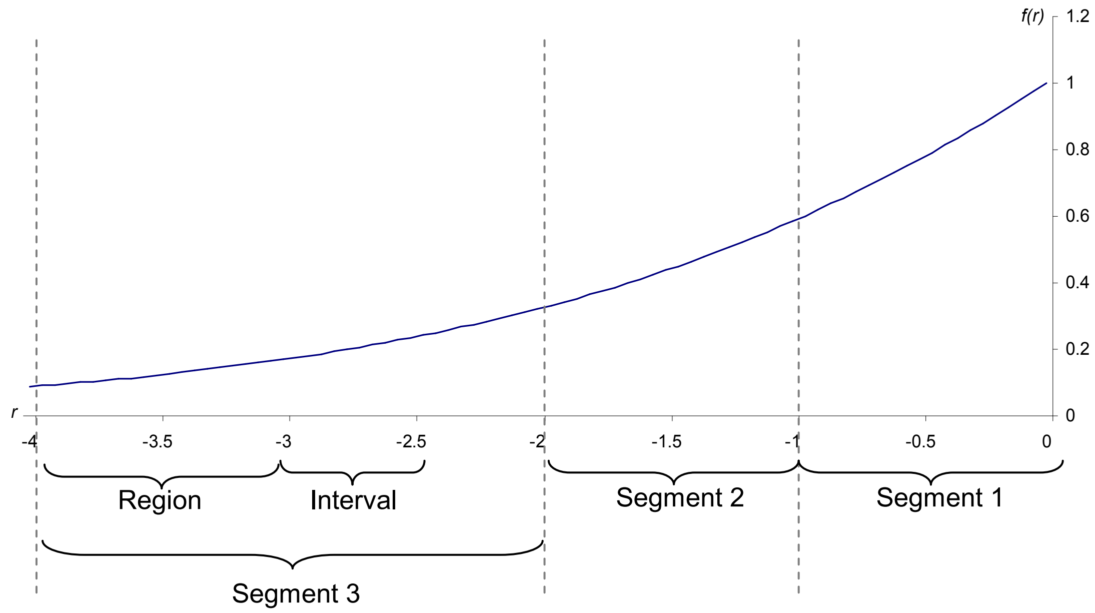


Figure 4-2 : Descriptions of interval, region and segment.

4.2.1. ELM Interpolation: Error Correction Algorithm

If LNS addition and direct subtraction have to be built so as to use as little memory as possible and must not be too complex, the most desirable function approximation technique to be applied is an interpolation scheme. Previous studies have suggested various types of interpolation techniques, ranging from direct interpolation, linear interpolation and non-linear interpolation approximation. However, the most notable scheme uses a high-order coefficient in the interpolator function as presented by Coleman *et al*. [6]. Apart from its capacity to dramatically reduce ROM size, using Coleman's approach can also yield better accuracy than FLP. Conventionally, a linear interpolation scheme can be expressed as:

$$f(r) = F(r_n) - \delta \cdot D(r_n) \tag{4.1}$$

where $F(r_n)$ represents either the addition or subtraction function in which their values are stored in an F table, and its derivative, $D(r_n)$, at that particular point is stored in a D table. Assuming that the intervening value of $r = r_n - \delta$, then δ is the difference between a value of r and the nearest more positive point in that specific region. However, the function approximation using linear interpolation usually yields error, as described in the inset of Figure 4-3, whereby:

$$\varepsilon(n,\delta) = F(r_n) - \delta \cdot D(r_n) - F(r_n - \delta) \tag{4.2}$$

and the maximum error at each interval can be written as:

$$E(n) \approx F(r_n) - \Delta \cdot D(r_n) - F(r_n - \Delta) \tag{4.3}$$

where Δ refers to the maximum width in that particular interval, which will usually be doubled at each increasing powers of two whenever r is gradually decreased. In order to compensate for error ε, a noteworthy solution is to implement the linear

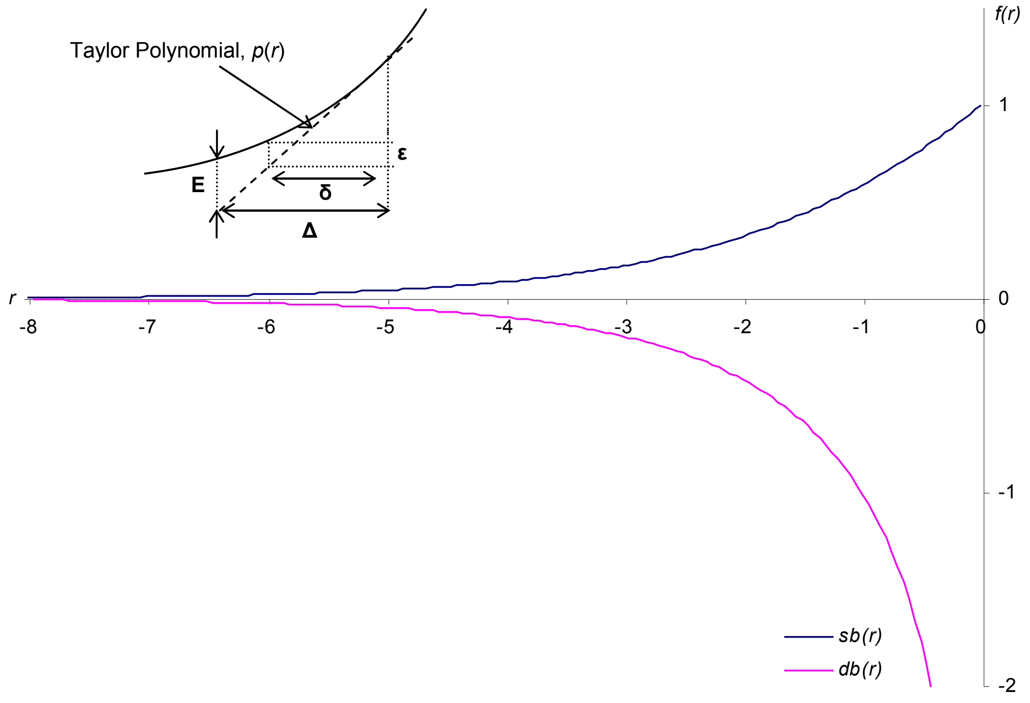


Figure 4-3 : Function approximation method for ELM.

interpolation in conjunction with an error correction algorithm as suggested in [6]. This exploits another table, known as E, to store a local maximum error value in each interpolation interval, as well as a P table which consists of the proportion of an error for the region that yields the largest absolute maximum error. Thereby, the error ε can be resolved as:

$$\varepsilon(n,\delta) \approx E(n) \cdot P(c,\delta) \tag{4.4}$$

where c is a constant, because only one P table is required in the system. By incorporating this function into equation (4.1), the error in the final result will then suppressed, hence a substantial saving in memory space is thereby possible.

Despite introducing two new tables, E and P, the adoption of this scheme has the advantage that these tables can be referred to concurrently with those from F and D. Moreover, the multiplication process of the value $E{\cdot}P$ can be computed at the same time as the multiplication in the linear interpolation, $\delta \cdot D$. In the final accumulation stage, because interpolation already involves an addition, the product of the error-correcting term can be accumulated by adding another level of a carry-save adder as portrayed in Figure 2-7. Overall, the correction procedure can therefore be completed with only a few extra gate delays, thus having the least impact on the critical speed path of the LNS system.

4.2.1.1. Taylor Approximation

The fundamental principle of function approximation in the ELM is based upon the linear Taylor approximation. In general, the Taylor approximation method can be illustrated as a tangent line that passes through a tabulated point, as portrayed in the inset of Figure 4-3. Conceptually, the basic formula of Taylor's theorem is written as:

$$p(r) = f(r_0) + \frac{(r - r_0)}{1!} f'(r_0) + \frac{(r - r_0)^2}{2!} f''(r_0) + \ldots\ldots + \frac{(r - r_0)^n}{n!} f^{(n)}(r_0) \quad (4.5)$$

In previous LNS designs, the term $f^{(n)}(r_0) \cdot (n!)^{-1}$ is stored as a computed value in ROM and n is often limited to 2. This ensures that the computation of the function approximation can be executed within less hardware complexity, as a result of each order of n in the Taylor polynomial requiring at least one multiplier and one adder to perform the function. Therefore, an increase in the order of n will not only involve additional hardware multipliers and adders, but at the same time will directly impact onto the cost of the hardware translation in silicon. Hence, the work presented in [6] restricted the Taylor polynomial to only the first degree, thereby the Taylor series can be formulated as:

$$p(r) = f(r_n) + (r - r_n) \cdot f'(r_n) \quad (4.6)$$

As a result of implementing the error correction algorithm in the approximation architecture, $\varepsilon(n,\delta)$ is therefore also added into equation (4.6). Through this arrangement, the published design in [6] is able to achieve a reasonable size of total storage, with its accuracy better than FLP.

4.3. Simulation Results

The simulation results for the ELM unit, focusing on the addition and subtraction functions, are summarised in Table 4-2. From the analysis, the worst-case error and lookup tables arrangement are analogous with the results published in [6]. This means that the illustrated simulator design as exhibited in Figure 3-1 has been able to yield results comparable with those in the original specification, which can hence be acknowledged to be fully verified and tested. The entry marked in bold italics in the table is the best composition of the total storage requirement, where the F, D and E tables are set to 256 words with the P table at 1024 words. Meanwhile, the greyed entries in the table signify the worst-case errors above the FLP limit of 0.5, which means that these lookup table configurations need not be considered. The error produced according to various sizes of lookup table formation is represented graphically in Figures 4-4, 4-5 and 4-6. Figure 4-7 shows the overall storage requirements for various combinations of the F, D, E and P tables able to sustain the worst-case error within FLP limit. For the purpose of this simulation, the F, D, E and P tables have been assumed to comprise of 32-bit words in 6 different segments based on the powers of two partitioning procedure.

4.4. Design Summary

In order to successfully achieve the BTFP mode, four guard bits are inserted into the ELM system to maintain precision whilst executing addition and subtraction operations. Once the computation is finished, the number is rounded back to the original 32-bits. As reported in [6], the system is partitioned into six segments at

Table 4-2 : The worst-case error of the ELM unit.

Parameters			ADD		SUB		Worst Case
F,D,E Sizes	P size	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
64	512	4	-2.0231	+0.9188	-1.2017	+3.8857	3.8857
128	512	4	-0.8134	+0.4501	-0.5401	+1.1579	1.1579
256	512	4	-0.4948	+0.4057	-0.4377	+0.5470	0.5470
512	512	4	-0.4589	+0.4042	-0.4366	+0.4286	0.4589
1024	512	4	-0.4287	+0.4081	-0.4355	+0.4265	0.4355
64	1024	4	-1.1933	+0.9188	-1.2017	+2.0972	2.0972
128	1024	4	-0.5937	+0.4501	-0.5401	+0.7206	0.7206
256	***1024***	***4***	***-0.4526***	***+0.4066***	***-0.4377***	***+0.4551***	***0.4551***
512	1024	4	-0.4457	+0.4032	-0.4375	+0.4286	0.4457
1024	1024	4	-0.4258	+0.4081	-0.4360	+0.4265	0.4360
64	2048	4	-0.7786	+0.9188	-1.2132	+1.1579	1.1579
128	2048	4	-0.5036	+0.4513	-0.5401	+0.5029	0.5401
256	2048	4	-0.4250	+0.4066	-0.4377	+0.4294	0.4377
512	2048	4	-0.4435	+0.4026	-0.4375	+0.4286	0.4435
1024	2048	4	-0.4258	+0.4091	-0.4360	+0.4265	0.4360

powers of two ranging from 0..-1, -1..-2, -2..-4, -4..-8, -8..-16 and -16..-32. However, for subtraction only, the co-transformation process is deployed over the range $-0.5 < r < 0$. At each segment, 256 words are used to store the F, D and E tables, whereas for the P table, its 1024 words accommodates the error profile for the subtraction logarithm in the range $-8 < r < -4$ since this is where the maximum absolute error occurs. At 32-bits, the total storage needed for the interpolation process is approximately 227 kbits, as shown in Table 4-3. Although the total bits cited in [6] is lower because there the number of bits were optimised with decreasing r in each table, that would actually lead to an impractical hardware implementation.

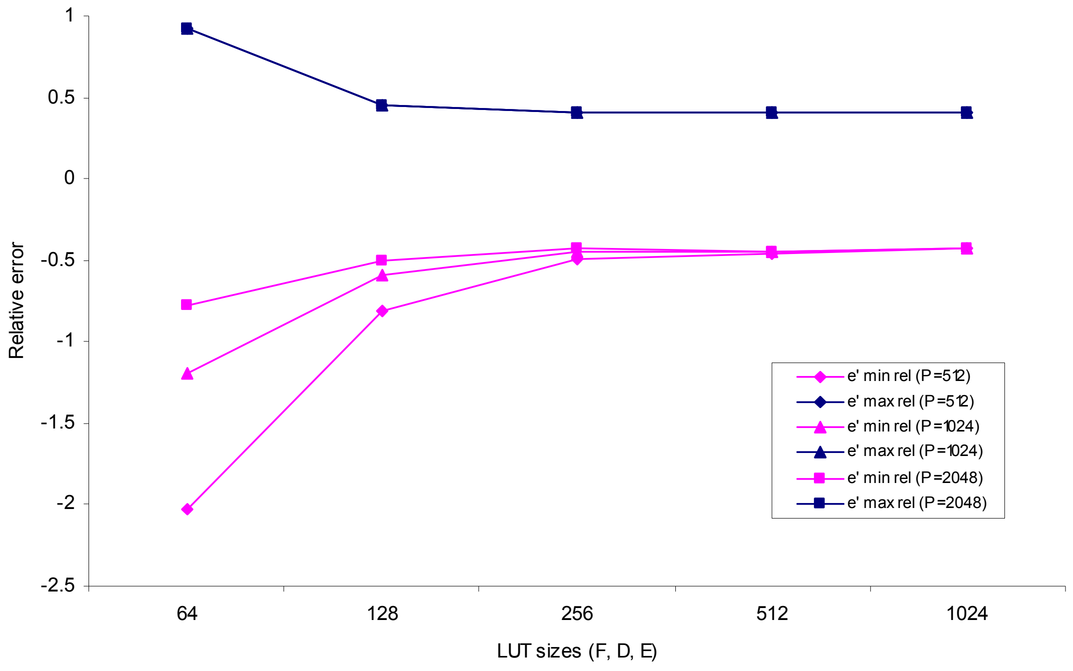


Figure 4-4 : Approximation error for the addition operation of the ELM unit.

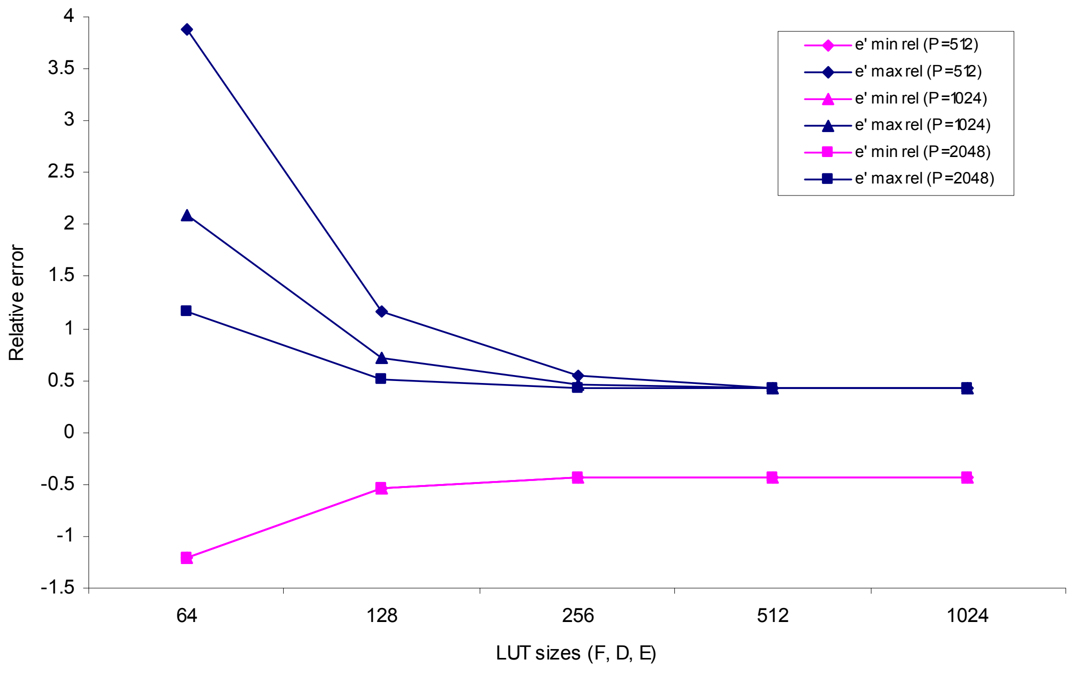


Figure 4-5 : Approximation error for the subtraction operation of the ELM unit.

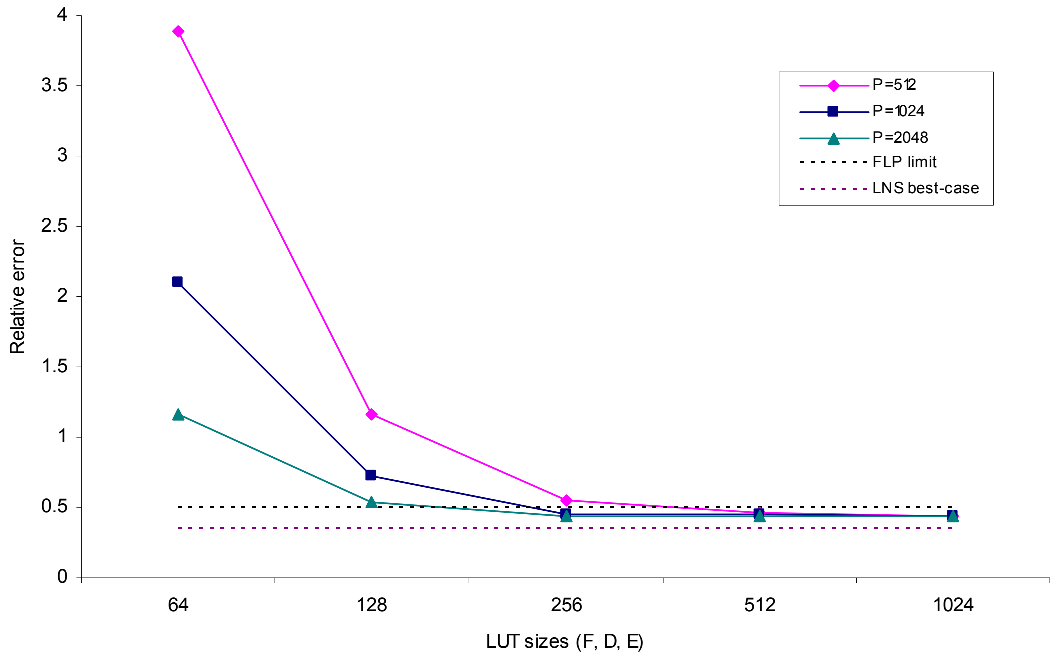


Figure 4-6 : Worst-case error of the ELM unit.

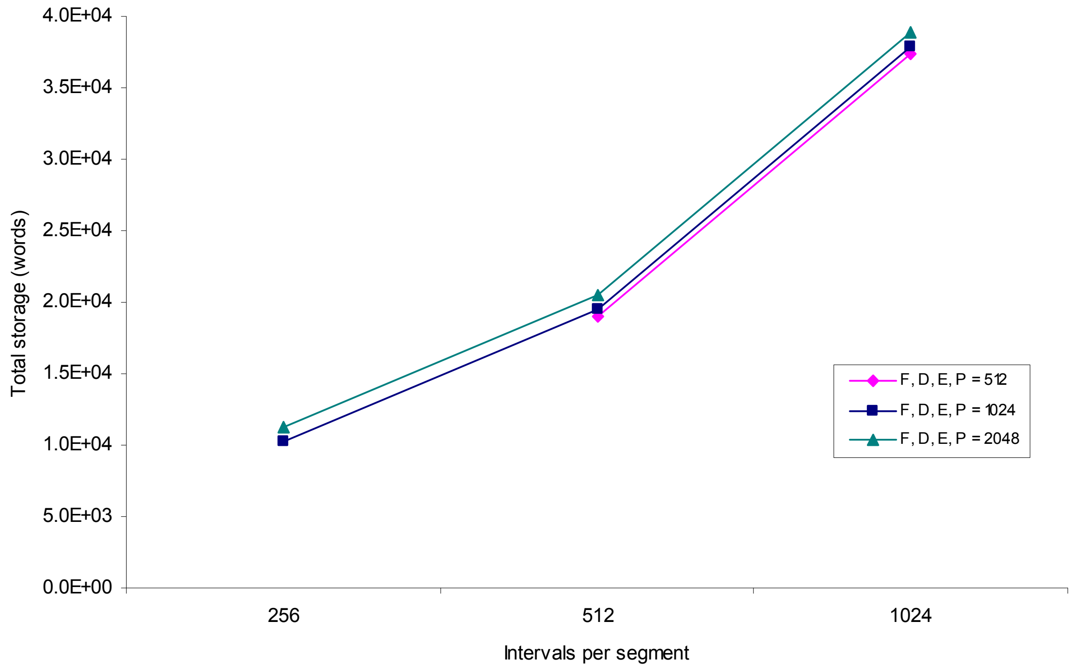


Figure 4-7 : Total storage requirement for the worst-case error within FLP limit.

Table 4-3 : ELM interpolation memory requirements.

Table	Words	Word Length	Segments	Total Bits
F Add	256	28-bit	6	43,008
F Sub	256	28-bit	6	43,008
D Add	256	27-bit	6	41,472
D Sub	256	28-bit	6	43,008
E Add	256	8-bit	6	12,288
E Sub	256	11-bit	6	16,896
P	1024	27-bit	1	27,648
Total				227,328

4.5. Synthesis Results

The previous ELM processor device based on 32-bit LNS arithmetic implementation was fabricated using 0.18 μm CMOS technology. The performance of this processor was compared with the commercial Texas Instruments (TI) FLP device, the TMS320C6711 DSP chip, itself fabricated in a similar technology. Examining the published results of the analysis of the ELM run at 125 MHz, multiplication and division were executed in a single cycle of 8 ns. Conversely, the 150 MHz TI device required 4 cycles, lasting 26.67 ns, to perform the multiplication operation and approximately 30 cycles for division. The ELM consumed 3 cycles, 24 ns, to compute addition and subtraction operations and 4 cycles whenever this involved the co-transformation procedure in subtractions. On the TI device, 4 cycles at 26.67 ns were needed to complete these functions.

Although the silicon area was not reported in the analysis, the author has confirmed that the overall dimensions of the ELM die area were 3,224 μm × 4,122 μm. Only the blocks labelled as MCALU, FDE, G and P, as illustrated in Figure 4-8, related to the organisation of the 32-bit LNS arithmetic unit, therefore measuring with a ruler gives an estimate of 862,550 μm^2 for that particular area.

For a realistic comparison, the LNS system incorporated in the ELM was resynthesised using the Faraday 0.18 μm process. The synthesiser was run twice, once without any target constraints, and again with a constraint to deliver the highest possible speed. These trial designs were not taken to a routed layout, but the synthesiser incorporates reliable modeling of routing and wiring, and is able to deliver an accurate prediction of the final area and delay, which is presented in Table 4-4. Note that the unconstrained results, 22.77 ns for addition and direct subtraction, 28.60 ns for subtractions making a co-transform, and an area of 842,433 μm^2, are very similar to those actually found on the fabricated device in [7].

The approximate doubling of speed when the target constraint is asserted probably reflects an improvement in synthesiser technology in the intervening years and appears to be comparable with current FLP performance which will be described in the next section. Therefore, this optimised synthesis has been taken as the starting point for further development.

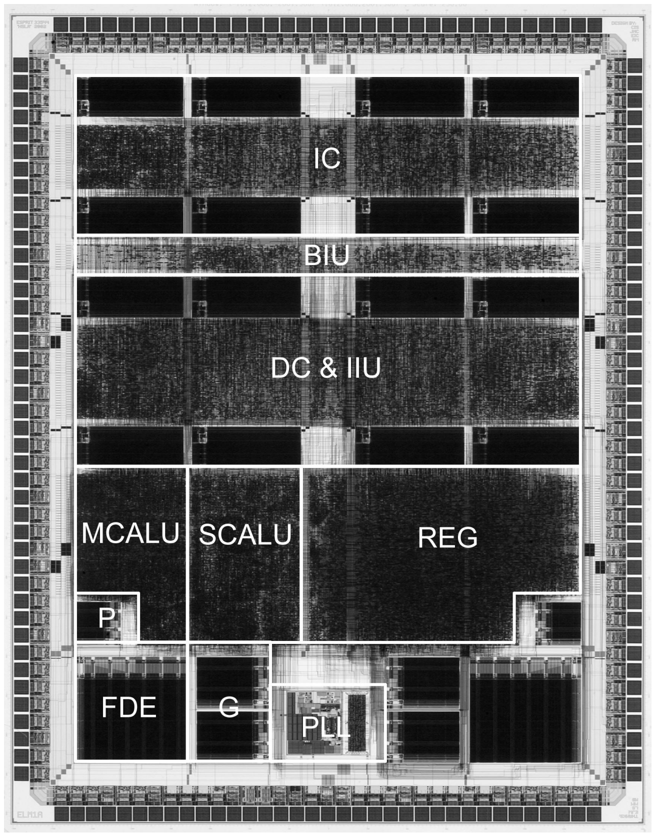


Figure 4-8 : Die plot of ELM.

Table 4-4 : Delay times and total device area of ELM.

Function	ELM			
	Unconstrained		Constrained	
	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)
Add / Sub	22.77	842,433	11.74	904,943
Sub (Co-transform)	28.60		13.15	
Mul / Div	2.27	8,337	1.16	10,514

4.6. FLP Devices

The development of the new 32-bit LNS arithmetic unit has to be validated against other 32-bit arithmetic implementations. For that reason, it is helpful to have an up-to-date FLP design fabricated with similar 0.18 μm technology for comparison.

Several downloadable FLP libraries [58, 79] are available online but the practicality of using them for comparison purposes is questionable without knowing to what extent optimisation efforts have been made in their designs. In addition, a majority of the presented FLP libraries were only targeted on FPGA's. To avoid a biased comparison, a FLP arithmetic device fabricated in a similar 0.18 μm technology is used for comparative analysis in this thesis.

Kwon et al. in [80] compared two FLP ALU architectures which had been optimised for different design goals. Both designs were synthesised and routed for 0.18 μm fabrication, as with the work reported in this thesis. The implementation of these two FLP arithmetic units, namely MONARCH and DIVA, followed the standard of the IEEE-754 format and supported single-precision numbers. Each system was able to execute add, subtract, multiply and divide operations.

The MONARCH FLP design was intended to achieve higher performance in the sense of the speed of executing arithmetic operations. In order to realise the design goal, therefore, every single arithmetic block operated independently with no data path shared between the addition/subtraction and multiplication/division modules. Consequently, each arithmetic unit could be optimised individually in order to obtain low instruction latency. It was reported that addition and multiplication had a delay of 3 clock cycles and division 9 clock cycles when clocked at 266 MHz. The estimated layout area was 600,000 µm^2.

The second design, DIVA, was optimised for minimal area. Several design considerations supported this, such as merging the exponent computation block for each arithmetic unit in one data path and also sharing rounding logic. As a result of these design strategies, 5 clock cycles were required to perform addition and multiplication, and 12 clock cycles for division when similarly clocked at 266 MHz. However, the total layout area was reduced to 481,635 µm^2. Table 4-5 summarises the delay and silicon area results for these FLP architectures.

Table 4-5 : Delay and area of FLP arithmetic unit at 266 MHz.

Function	FLP					
	MONARCH			DIVA		
	Delay (cycles)	Delay (ns)	Area (µm^2)	Delay (cycles)	Delay (ns)	Area (µm^2)
Add / Sub	3	11.28	600,000	5	18.80	481,635
Mul	3	11.28		5	18.80	
Div	9	33.83		12	45.11	

4.7. Comparison Analysis: ELM and FLP

Based on the constrained synthesis of the ELM design, its total silicon area is larger than FLP devices, DIVA and MONARCH, as illustrated in Figure 4-9. However, in terms of delay, the ELM computed addition and direct subtraction at almost the same speed with the faster of the two FLP units, MONARCH, as shown in Figure 4-10.

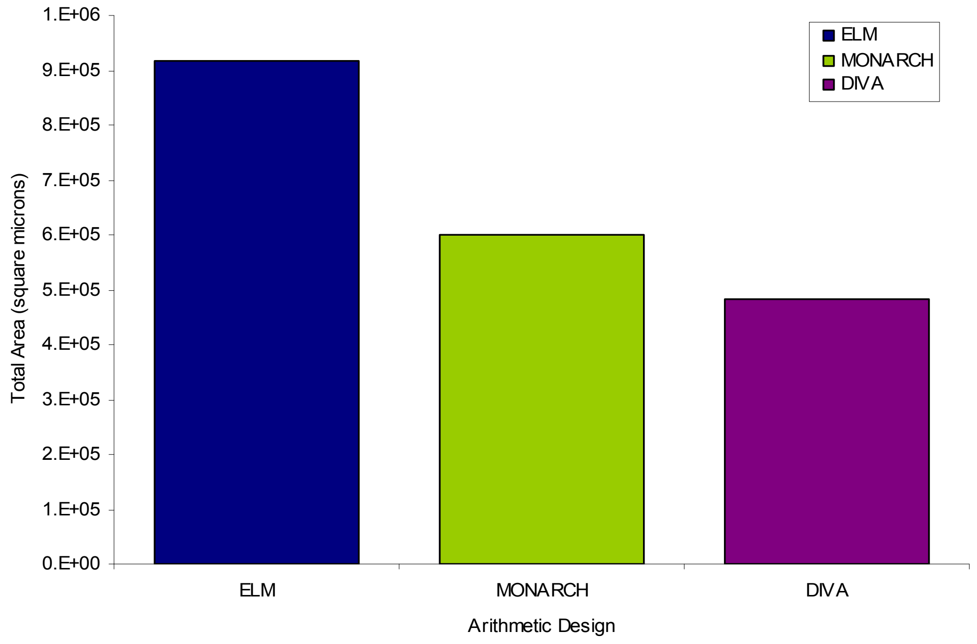


Figure 4-9 : Total silicon area between ELM, DIVA and MONARCH.

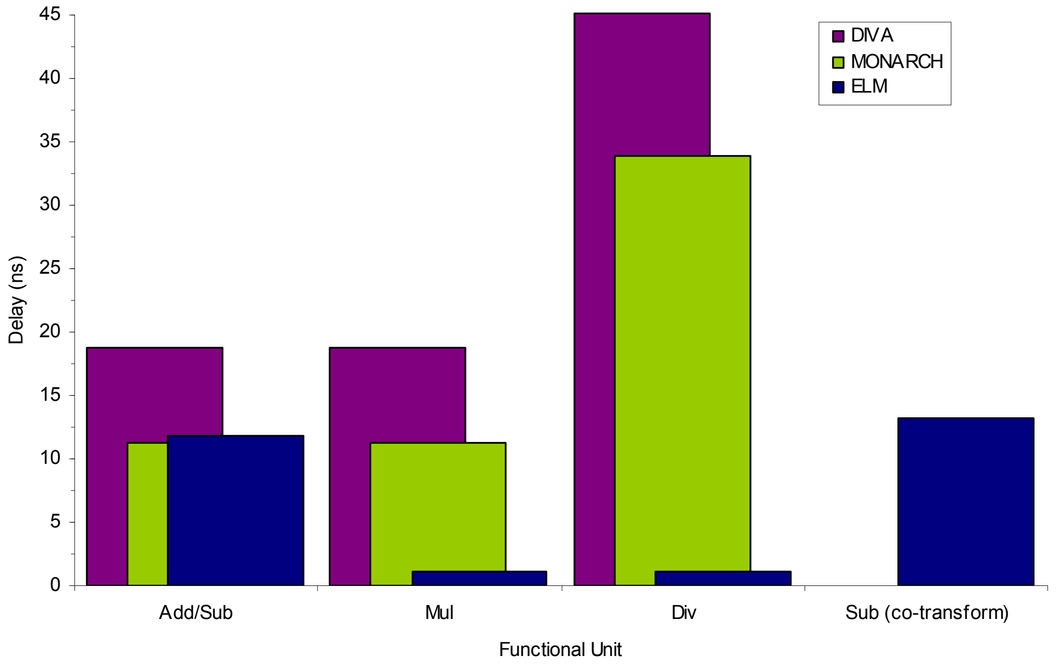


Figure 4-10 : Delay between ELM, DIVA and MONARCH.

For subtractions involving the co-transformation, the delay increases to 116% of the delay in MONARCH. On the other hand, the ELM executed multiplication and division in 1.16 ns, whereas MONARCH required 11.28 ns and 33.83 ns respectively. From the analysis described herein, it can be concluded that the result presented is in agreement with the analysis summarised in [7].

4.8. Summary

The hardware arrangement of the LNS design presented in the ELM processor was reviewed intensively in this chapter. Conceptually, a Taylor approximation method together with an error correction algorithm was employed to perform the interpolation process for executing addition and direct subtraction operations. By implementing this technique, significant reduction in total lookup tables can be obtained. According to the simulation and synthesis results based on the reconstructed ELM architecture, the results reported have been shown to be in agreement with those found in the original work. This means that the resynthesised design has been fully verified and the results generated can be used for comparison purposes.

In order to evaluate the performance of the new LNS system against other recent 32-bit arithmetic implementations, two FLP devices namely DIVA and MONARCH have been studied. These FLP units were synthesised in similar CMOS technology as that adopted in this thesis. Thus, fair and direct comparison can be obtained when performing the analytical work.

CHAPTER 5

5. Co-transformation Architecture for LNS Subtraction

5.1. Introduction

The co-transformation technique was first introduced in an ELM processor, and the resulting speed and accuracy of the complete system were better than in an FLP arithmetic unit. Therefore, for benchmarking purposes, Chapter 5 reviews this approach in detail before reconstructing the architecture. Using the concept implemented in the ELM as the initial idea, a new proposal for the co-transformation procedure is suggested [77]. The proposed design is then simulated accordingly and the results are reported. Finally, a comparative analysis is carried out to show the effectiveness of the proposed algorithm compared with the ELM, concentrating on total area used (in bits), worst-case delay and levels of error in the system.

5.2. First-order Co-transformation Procedure for LNS Subtraction

The difficulty of approximating the value of the subtraction function at values of r closer to zero, due to approaching $-\infty$ as depicted in Figure 4-1, will cause larger table sizes when using direct lookup tables, interpolation or even bipartite/multipartite tables. Thus, the co-transformation procedure introduced by Coleman [49] is applied in the region $-0.5 < r < 0$. The co-transformation scheme as outlined in Figure 5-1 can be called a first-order co-transformation procedure, which is constructed by introducing two new variables, $k1$ and $k2$, on top of the original subtraction function as explicitly derived in equation (5.1).

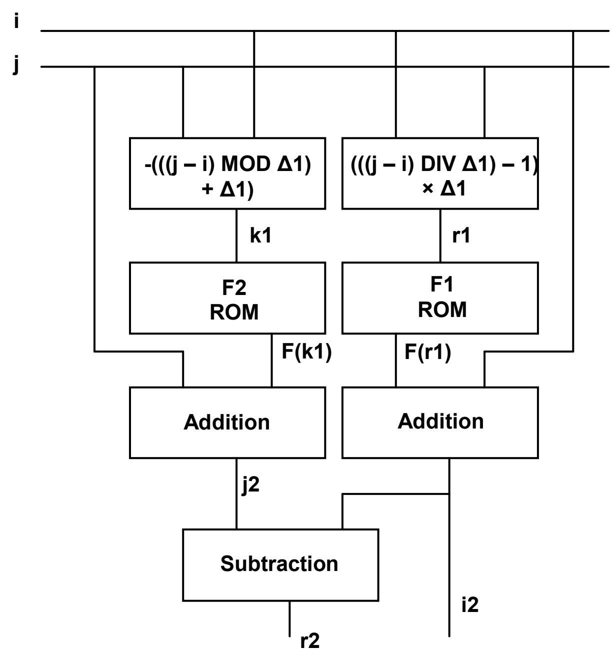


Figure 5-1 : ELM's co-transformation architecture.

$$2^i - 2^j = (2^i - 2^{j+k1}) - 2^{j+k2} \tag{5.1}$$

where

$$2^{k1} + 2^{k2} = 1, \text{ i.e } k2 = \log_2 (1 - 2^{k1}) \tag{5.2}$$

With $\Delta 1$ fixed at a large value, index $r1$ is calculated based on the individually chosen factor $k1$ which will then lie on the modulo-$\Delta 1$ boundary beneath r. Subsequently, $F(r1)$ can be retrieved from the lookup table F1, which stores $F(r)$ values in the region $-0.5 < r < -\Delta 1$. For all regions, the value of $k2$ is tabulated in the F2 table which includes all possible values of $k1$ that lie in the range $-\Delta 1 < k1 < 0$. Thus,

$$r1 = (((j - i)\ DIV\ \Delta 1) - 1) \times \Delta 1 = j + k1 - i \tag{5.3}$$

$$k1 = - (((j - i)\ MOD\ \Delta 1) + \Delta 1) = i - j + r1 \tag{5.4}$$

An index $r2$ results from the subtraction of the newly reformed $i2$ and $j2$ from their original values of i and j, and hence will give,

$$2^i - 2^j = 2^{i + F(r1)} - 2^{j + F(k1)} \tag{5.5}$$

$$\begin{aligned} r2 &= j - i + F(k1) - F(r1) \\ &= j - i + \log_2 [(1 - 2^{\,i - j + r1}) \div (1 - 2^{r1})] \end{aligned} \tag{5.6}$$

The value of *r2* can be considered in three regions, depending on the original operands *i* and *j*. For $j - i \leq -0.5$, *r2* is taken directly as $j - i$, and will lie in the linear region of *F* from which $F(r)$ can be obtained by interpolation. For $-0.5 < j - i < -\Delta 1$, *r2* is derived as shown in equation (5.6), and as it falls in the region less than -1 as illustrated in Figure 5-2, $F(r)$ is similarly obtained by interpolation. For the third region, $-\Delta 1 \leq j - i < 0$, the derived value of *r2* rises above −1. However, this range is covered by the F2 table, and $F(r)$ is therefore already available as *k2*. The modified values *r2* and *i2* are passed to the interpolator for completion of the outer subtraction. In adopting this approach, the bit partitioning scheme of the LNS format is illustrated in Figure 5-3.

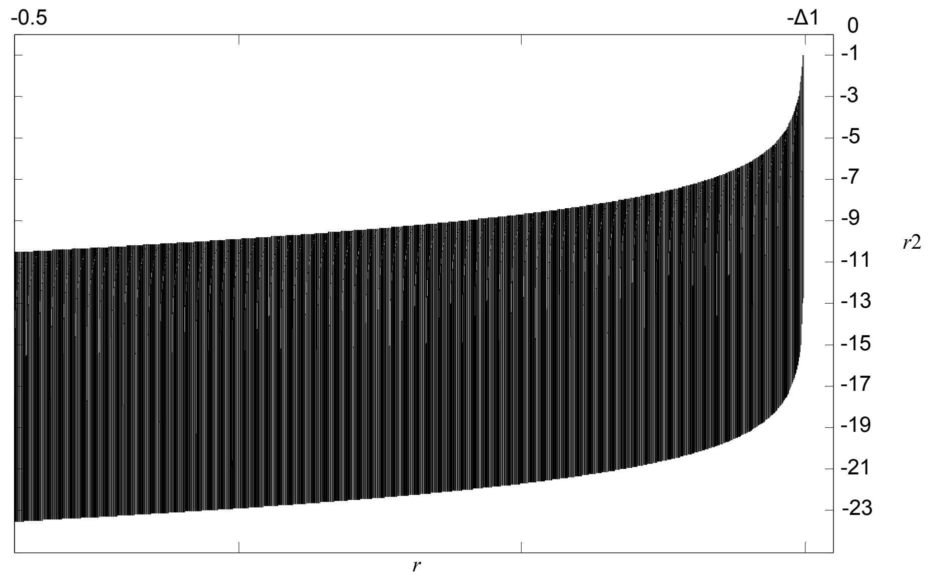


Figure 5-2 : Value of *r2* for -0.5 < *r* < -Δ*1*.

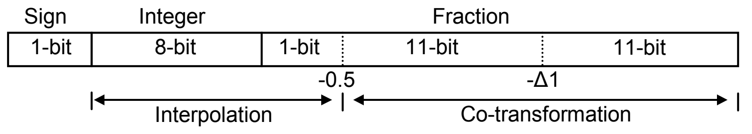


Figure 5-3 : Bit partitioning scheme for first-order co-transformation.

Now the total size of F1 and F2 tables are 2048 words each, details presented in Table 5-1, which is approximately one-seventh of the size of the tables that would be involved in the interpolation process for a similar region [59].

Table 5-1 : ELM co-transformation memory requirements.

Table	Words	Word Length	Segments	Total Bits
F1	2048	31-bit	1	63,488
F2	2048	32-bit	1	65,536
Total				129,024

5.3. Optimising Lookup Tables for LNS Subtraction

One of the key aspects in designing LNS addition and subtraction concerns the total storage requirements for the entire unit. Having such large lookup table requirements in previously published LNS systems made them unattractive for future DSP chip implementation, although they might be appropriate for some specific DSP applications. Therefore, if designers can reduce the total table requirements, this will then reduce the total area of the device commensurately.

The most challenging region is the subtraction function above -1, which approaches singularity and thereby requires a huge space for lookup tables when applying conventional methods to maintain precision within the FLP limit. The proposal for the ELM unit [6, 7] has introduced a promising architecture to compute subtraction over that particular range, but no such design so far has been able to further improve the technique in order to need less storage while achieving similar or better performance than this ELM. Given this situation, further exploration into the possibility of optimising the usage of lookup tables specifically in the region of $-0.5 < r < 0$ for subtraction is discussed in this section.

The algorithm proposed in this study, the so called second-order co-transformation procedure, derives from the basic principle of the first-order concept as used in an ELM. Despite having two similar sizes of tables, F1 and F2, this technique now incorporates another table in the system by applying the fractionating coefficient *k1* recursively. Thus, three much smaller tables are created in the range of $-0.5 < r < 0$ which will substantially reduce total storage as a result of the fractional bits being partitioned into three small regions. The segmentation procedure remains unchanged, as shown in Table 4-1. Additionally, five guard bits are added as to maintain accuracy within FLP limit and to keep the table sizes analogous to those of an ELM whilst performing the interpolation process.

5.3.1. The New Algorithm: Second-order Co-transformation Procedure for LNS Subtraction

Co-transformation as described for the ELM was introduced by replacing the subtraction $2^i - 2^j$ with two successive subtractions as shown in equation (5.1). However, the fractionating coefficient *k1* can be applied recursively. Substituting

$$2^{j+k2} = 2^j - 2^{j+k1}$$

into equation (5.1) gives:

$$\begin{aligned} 2^i - 2^j &= (2^i - 2^{j+k1}) - (2^j - 2^{j+k1}) \\ &= (2^i - 2^{j+k1}) - ((2^j - 2^{j+k1+k11}) - 2^{j+k1+k12}) \qquad (5.7) \end{aligned}$$

where,

$$2^{k11} + 2^{k12} = 1, \text{ i.e. } k12 = \log_2 (1 - 2^{k11}) \qquad (5.8)$$

The four subtractions in equation (5.7), and their respective indices *r*, will now be numbered as follows:

$$2^i - 2^j = \underbrace{\underbrace{(2^i - 2^{j+kl})}_{1} - \underbrace{(\underbrace{(2^j - 2^{j+kl+kll})}_{11} - 2^{j+kl+kl2})}_{12}}_{2}$$

At first, *k1* is selected such that the index *r1* falls on the nearest modulo-Δ*1* boundary beneath *j* − *i*, and *F*(*r1*) is obtained directly from the lookup table F1, containing *F*(*r*) for −1 < *r* < − Δ*1* at modulo-Δ*1* intervals. However, Δ*1* is now fixed at a larger value than was the case in the first-order arrangement, thereby shortening the index to the F1 table by the number of additional bits used. Previously, this would have caused a corresponding increase in size of the index to the F2 table. Now, however, the coefficient *k11* is similarly selected such that *r11* falls on the modulo-Δ*11* boundary beneath *j* + *k1* – *j* = *k1*, and *F*(*r11*) is obtained from table F11 which contains *F*(*r*) for -Δ*1* < *r* < -Δ*11* at modulo-Δ*11* intervals. This reduces the index to the F11 table by the number of bits representing Δ*11*. The final coefficient, *k12*, is obtained from the lookup table F12 indexed by *k11*, itself represented by the same number of bits as Δ*11*. This conceptual arrangement is shown in Figure 5-4, from which it may be seen that the index *r* has effectively been split into three partitions, each of which will optimally be about a third of the length of the original. For clarity, Figure 5-5 shows the bit partitioning scheme for the second-order co-transformation format.

Variables *r1*, *k1*, *r11* and *k11* can be represented as:

$$r1 = (((j - i)\ DIV\ \Delta 1) - 1) \times \Delta 1 = j + k1 - i \qquad (5.9)$$

$$k1 = -(((j - i)\ MOD\ \Delta 1) + \Delta 1) = i - j + r1 \qquad (5.10)$$

$$r11 = -(((j - i)\ \mathrm{MOD}\ \Delta 1) + \Delta 1) + ((j - i)\ \mathrm{MOD}\ \Delta 11) = k1 + k11 \qquad (5.11)$$

$$k11 = ((j - i)\ \mathrm{MOD}\ \Delta 11) = r11 - k1 \qquad (5.12)$$

Subtractions 11 and 1 are performed directly by lookup of their respective function tables. Subtraction 12 then generates an index:

$$\begin{aligned} r12 &= k1 + k12 - F(k1 + k11) \\ &= k1 + F(k11) - F(k1 + k11) \\ &= k1 + \log_2 ((1 - 2^{k11}) \div (1 - 2^{k1 + k11})) \qquad (5.13) \end{aligned}$$

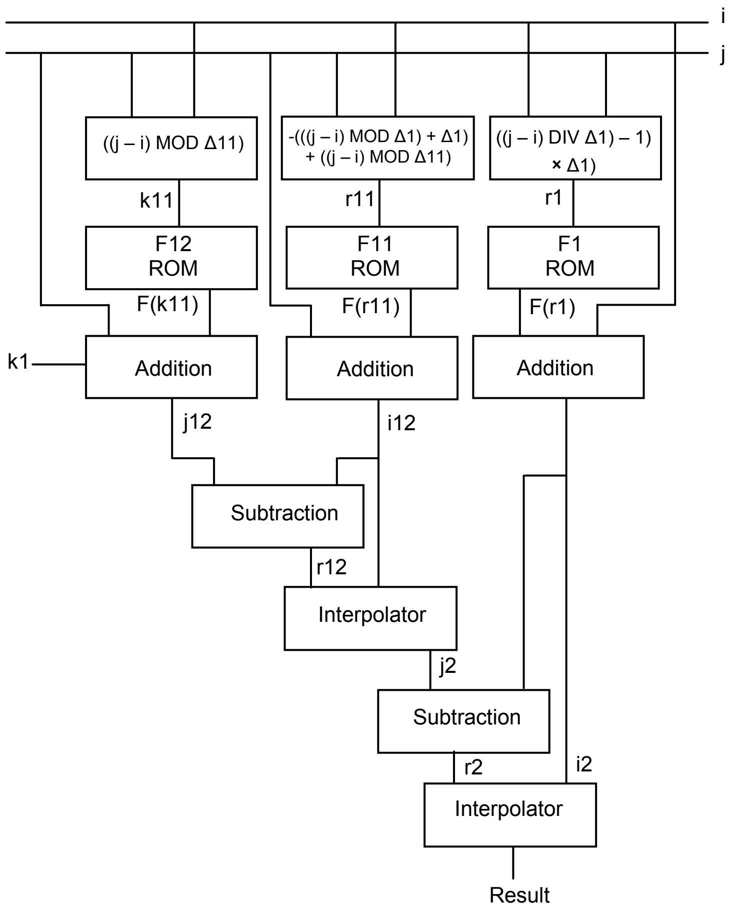


Figure 5-4 : Conceptual arrangement of second-order co-transformation.

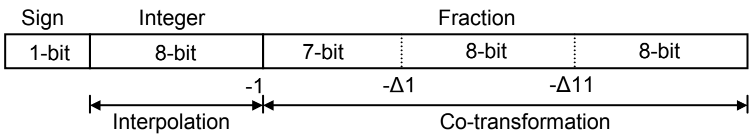


Figure 5-5: Bit partitioning scheme of second-order co-transformation.

The value of *r12* varies with the original *r* as shown in Figure 5-6, where $-2\Delta 1 < r < -\Delta 1$, i.e. *r* lies across the range of one $\Delta 1$. In the arrangement used for this illustration, $\Delta 11$ is 6 bits and $\Delta 1$ is 13, so that *r* is partitioned into low, middle and high-order segments of 6, 7 and 10 bits respectively. This is not the most optimal partitioning, but was chosen for this illustration in order to keep the graph to a manageable size. The modified value *r12* exhibits a repeating pattern of subintervals across each $\Delta 11$. With the exception, discussed below, of the extreme left subinterval, $r12 < -1$. Note, in fact, that for the point in each subinterval where $k11 = 0$, $r12 = -\infty$. These points have been omitted from the graph, and in practice they are ignored because the subsequent calculation of *F*(*r12*) is consequently zero. As regards the leftmost subinterval, it is necessary to consider the behaviour of *r12* as *r* progresses across the range of $\Delta 1$. In the first subinterval at the left of Figure 5-6, $k1 < \Delta 11$ and $k1 + k11 = \Delta 11$. To the far left of this subinterval, $k11 \approx \Delta 11$, and since *k1* is small, $r12 \approx 0$. Throughout this subinterval the middle partition is zero. It is therefore possible to treat this subinterval as a special case of the first-order arrangement, in which the second-order variable *k11*, table F12 and result *r12* are analogous to the first-order *k1*, F2 and *r2*. The new value *r2* bypasses the first interpolator and is passed directly to the second interpolation stage. Throughout the next subinterval, $k1 + k11 = 2\Delta 11$. To the far left of this subinterval, again, $k11 \approx \Delta 11$, but since *k1* and *k11* are both small, the exponential terms are approximately linear in behaviour, and *r12* is therefore ≈ -1. From here on, $r12 < -1$. Except in the special case just mentioned, subtraction 12 is then completed in the first interpolator, which is positioned as shown in Figure 5-4.

The result of subtraction 12 is then itself subtracted from the result of subtraction 1. Its index *r2*:

$$r2 = j - i + F(k1 + k11) + F(k1 + k12 - F(k1 + k11)) - F(r1) \qquad (5.14)$$

The value of *r2* is plotted over the range -1 < *r* < -Δ*1* in Figure 5-7. In all cases, *r2* < -1, and the subtraction can therefore be performed with a second iteration of the interpolator.

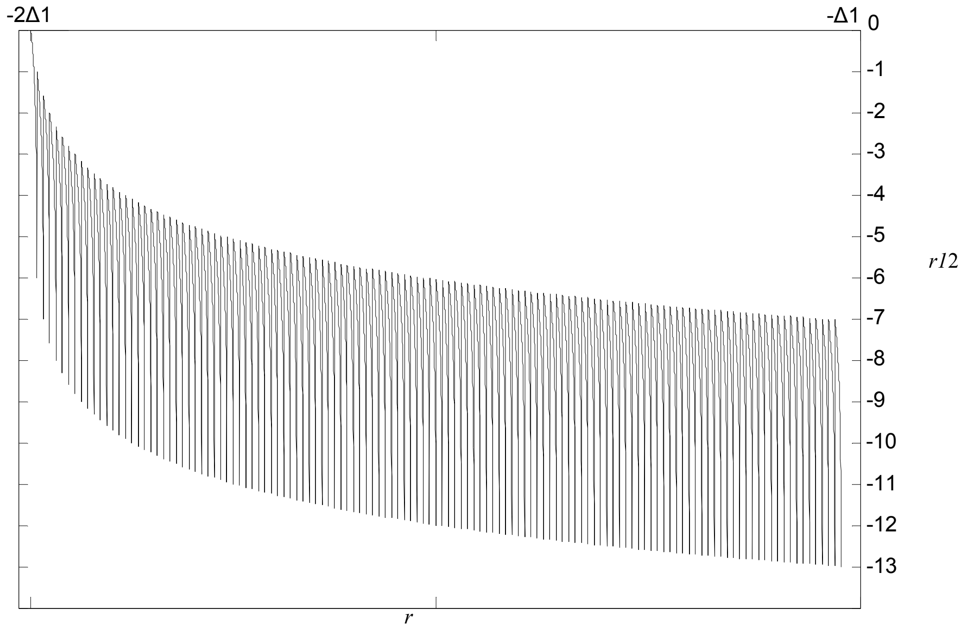


Figure 5-6 : Value of *r12* for -2Δ*1* < *r* < -Δ*1*.

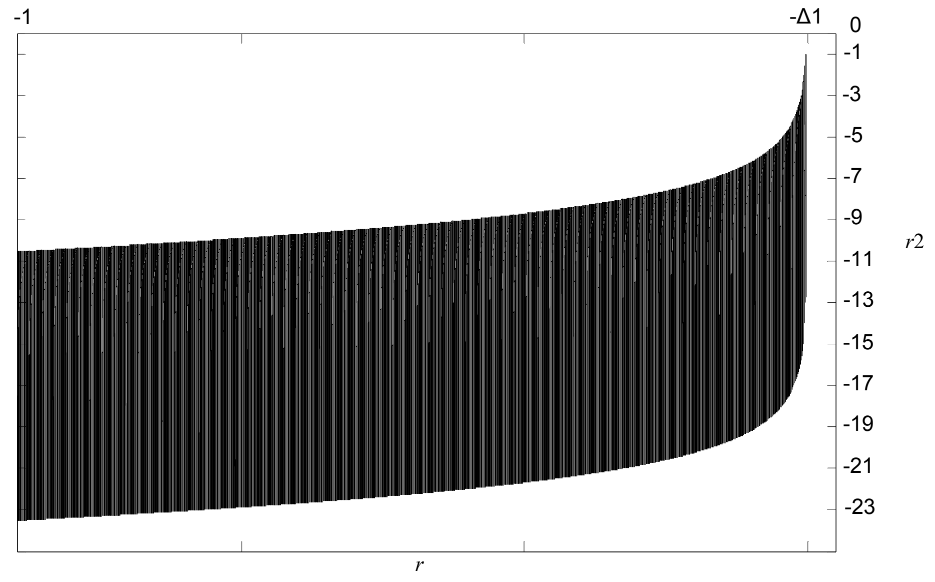


Figure 5-7 : Value of *r2* for -1 < *r* < -Δ*1*.

In the similar way that, in the first-order arrangement, the value of $r2$ falls into one of three regions, here it is separated into four. Again, this depends on the original operands i and j. For $j - i \leq -1$, $r2$ is taken directly as $j - i$, and will lie in the linear region of F from which $F(r)$ is obtained by interpolation. For $-1 < j - i < -\Delta 1$, $r2$ is derived as shown above, and now has a maximum of approximately -1. Thus it also lies in the linear region of F, and $F(r)$ is similarly obtained by interpolation. In the third region, $-\Delta 1 \leq j - i < -\Delta 11$, the high-order bits are zero and subtractions in this region can therefore be processed with a first-order technique using the F11 and F12 tables. Finally, $F(r)$ for $-\Delta 11 \leq j - i < 0$ is taken directly from the F12 table.

The overall co-transformation memory requirement is reduced from 4,096 words, as in the first-order shown in Table 5-1, to 640 as described in Table 5-2, or from 129,024 bits to 20,608, a reduction to about 16% of its original size.

Table 5-2 : Second-order co-transformation memory requirements.

Table	Words	Word Length	Segments	Total Bits
F1	128	31-bit	1	3,968
F11	256	32-bit	1	8,192
F12	256	33-bit	1	8,448
Total				20,608

5.3.2. Function Approximation Scheme

For fair and direct comparison, a first order Taylor approximation scheme incorporating an error correction algorithm is adopted to estimate the $F(r)$ in the region $-32 < r < -0.5$ for subtraction, and $-32 < r < 0$ for addition. This is equivalent to the approximation scheme in the ELM for those particular regions, as described in Section 4.2.1. Nevertheless, several alternative methods for the function approximation procedure will be investigated later in order to achieve further advantages over Taylor approximation, and these are elaborated in Chapter 6.

Using the same simulator model as constructed in Section 3.3.1, a number of simulations of error characteristics were performed through varying the sizes of the F, D, E and P tables. The work aimed to verify which table setting offers the best configuration to gain acceptable levels of error within the FLP boundary of 0.5.

5.3.3. Simulation Results

Table 5-3 outlines the simulation results for error characteristics of the combination of the newly proposed co-transformation scheme with interpolation using Taylor approximation for both addition and subtraction. It can be seen in Table 5-3 that the best worst-case error of the system still has the F, D and E tables at 256 words and the P table at 1024 words, as a result of applying an identical Taylor approximation method as in the ELM. However at this time, the number of guard bits is increased from 4 to 5. The worst-case relative error for the entire system is shown in Figure 5-8.

5.3.4. Design Summary

For ease of comparison, the LNS addition/subtraction architecture using the second-order co-transformation procedure applies the same configurations as those implemented in the ELM. Despite the existence of five guard bits in the device, the system presented is also divided into six segments at powers of two. Instead of partitioning into two as in the ELM, the subtraction operation in the range $-0.5 < r < 0$ is segmented into three regions ranging from $-\Delta 11 < r < 0$, $-\Delta 1 < r < -\Delta 11$ and $-0.5 < r < -\Delta 1$. Through this arrangement, a tremendous reduction in total storage can be gained whereby F1, F11 and F12 tables now only store 128, 128 and 256 words respectively. A maximum relative error nearly equivalent to the FLP limit is obtained when F, D and E tables are at 256 words,

Table 5-3 : The worst-case error of the optimised architecture.

Parameters			ADD		SUB		Worst Case
F,D,E Sizes	P size	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
64	512	5	-2.0248	+0.8757	-1.2671	+3.9264	3.9264
128	512	5	-0.7790	+0.4104	-0.4843	+1.2250	1.2250
256	512	5	-0.4802	+0.3543	-0.3705	+0.5787	0.5787
512	512	5	-0.4004	+0.3536	-0.3713	+0.4189	0.4189
1024	512	5	-0.3893	+0.3534	-0.3707	+0.4070	0.4070
64	1024	5	-1.1933	+0.8757	-1.2396	+2.1069	2.1069
128	1024	5	-0.5696	+0.4108	-0.4583	+0.7817	0.7817
256	***1024***	5	***-0.4277***	***+0.3543***	***-0.3705***	***+0.4719***	***0.4719***
256	1024	4	-0.4699	+0.3674	-0.3914	+0.5298	0.5298
512	1024	5	-0.3904	+0.3536	-0.3726	+0.4065	0.4065
1024	1024	5	-0.3893	+0.3569	-0.3707	+0.4070	0.4070
64	2048	5	-0.7786	+0.8857	-1.3452	+1.2056	1.2056
128	2048	5	-0.4632	+0.4118	-0.4670	+0.5458	0.5458
256	2048	5	-0.4113	+0.3549	-0.3723	+0.4321	0.4321
512	2048	5	-0.3884	+0.3542	-0.3730	+0.4065	0.4065
1024	2048	5	-0.3893	+0.3569	-0.3707	+0.4070	0.4070

with the P table at 1024 words. This is comparable with the ELM due to the application of a conceptually similar interpolation method for the addition and subtraction functions. As shown in Table 5-4, about 259 kbits would be required to compute LNS addition and subtraction using the new algorithm for the co-transformation process together with the previously published interpolation procedure.

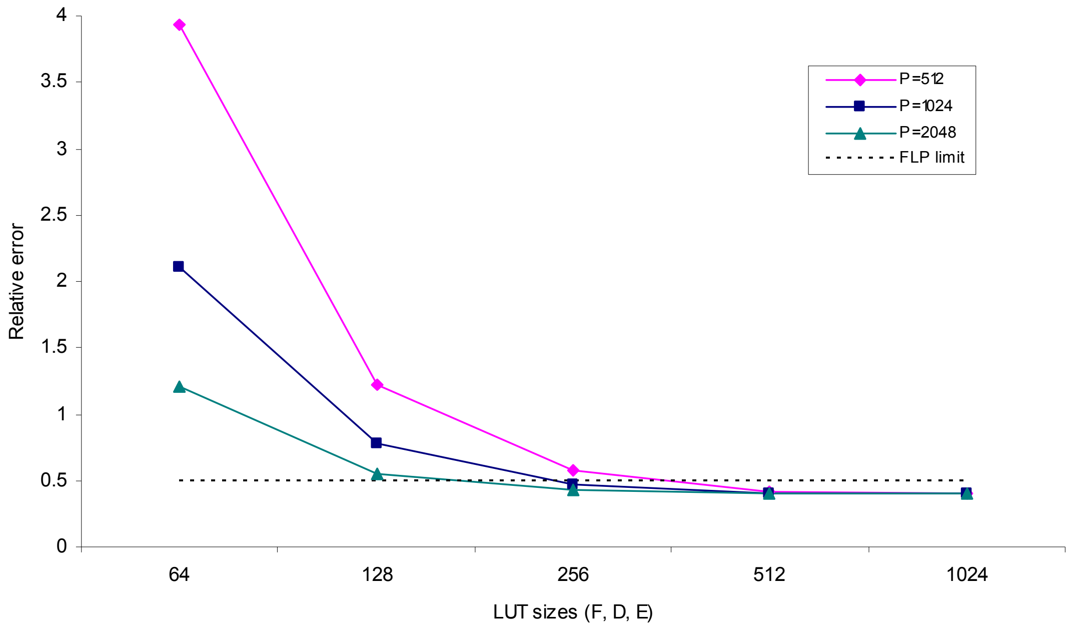


Figure 5-8 : The worst-case relative error of the proposed architecture.

Table 5-4: Total storage for the new algorithm.

Table	Words	Wordlength	Segments	Total Bits
F Add	256	29-bit	6	44,544
F Sub	256	29-bit	6	44,544
D Add	256	28-bit	6	43,008
D Sub	256	30-bit	6	46,080
E Add	256	9-bit	6	13,824
E Sub	256	14-bit	6	21,504
P	1024	28-bit	1	28,672
F1	128	31-bit	1	3,968
F11	128	32-bit	1	4,096
F12	256	33-bit	1	8,448
Total				258,688

5.4. Comparison Analysis: First-order and Second-order Co-transformation with the Taylor Interpolator

The analysis was conducted based on the constrained synthesis between the leading published design, the ELM, and the proposed LNS addition and subtraction architectures described in this chapter. The intention is to evaluate the efficiency of the new second-order co-transformation procedure when combined with the Taylor interpolator in three vital respects.

Firstly, the investigation focuses on the hardware costs in terms of the memory size required for each device in order to build a 32-bit system. This is due to the fact that ROM frequently dominates the silicon area of the unit and therefore very often becomes a major concern when developing an LNS system. Figure 5-9 graphically compares the total tables required for co-transformation and the interpolation module in the ELM as well as the new algorithm which comprised of second-order co-transformation together with the Taylor interpolator.

As can be observed in Figure 5-9, a dramatic reduction in table space to perform the co-transformation process can be achieved with the second-order co-transformation procedure. The effect of repeatedly applying the fractionating coefficient *k1* brings the total tables down to approximately one-eighth the size of those required in the ELM to execute the same function in the region $-0.5 < r < 0$. Previously, as reported in the first-order and second-order algorithms, the *r* value was transformed into a new value, *r2*, which lies in the region less than -1 after completing the co-transformation process. Thereby, it is now possible to extend co-transformation to cover the range $-1 < r < 0$, which then leads to 90% savings in total storage compared to the ELM. However, it is expected that the memory involved in the interpolation scheme in the new algorithm will be slightly higher than that in the ELM, as a result of using five guard bits.

Next, Figure 5-10 plots the worst-case error of the ELM and the new algorithm. It is found that the error characteristics of second-order co-transformation remain identical to those of the ELM because of the adoption of identical interpolation techniques. This means that, when utilising the new approach, the accuracy of the system is sustained. Nevertheless, ways have been sought to further improve the error behaviour of the system by simulating various types of interpolation method. These are elaborated upon in the next chapter.

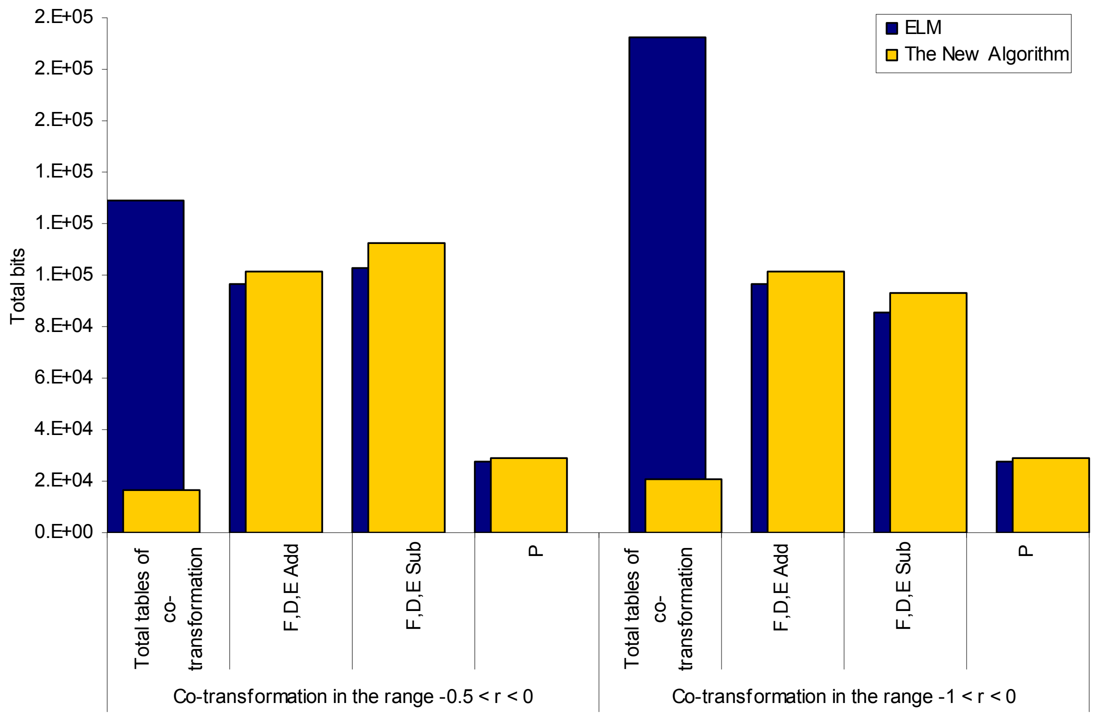


Figure 5-9 : Comparison of the total tables between ELM and new algorithm.

The third issue concerns the execution delay of the co-transformation process. Despite having worthwhile improvement in terms of table size, the implementation of the new algorithm comes at the expense of a vastly increased delay for subtractions using the co-transformation. This is mainly due to require two passes through the interpolator, which means that approximately twice the delay of a direct subtraction as graphically shown in Figure 5-11.

To conclude the analysis, it should be noted that implementing the new algorithm for the co-transformation procedure may lead to great savings in memory compared with the ELM. Nevertheless, it has a serious degradation in delay as a result of using the interpolator twice. Therefore, the adoption of the new co-transformation will only be feasible in conjunction with a faster interpolator. Hence, the next chapter examines several interpolation approaches to achieve this objective.

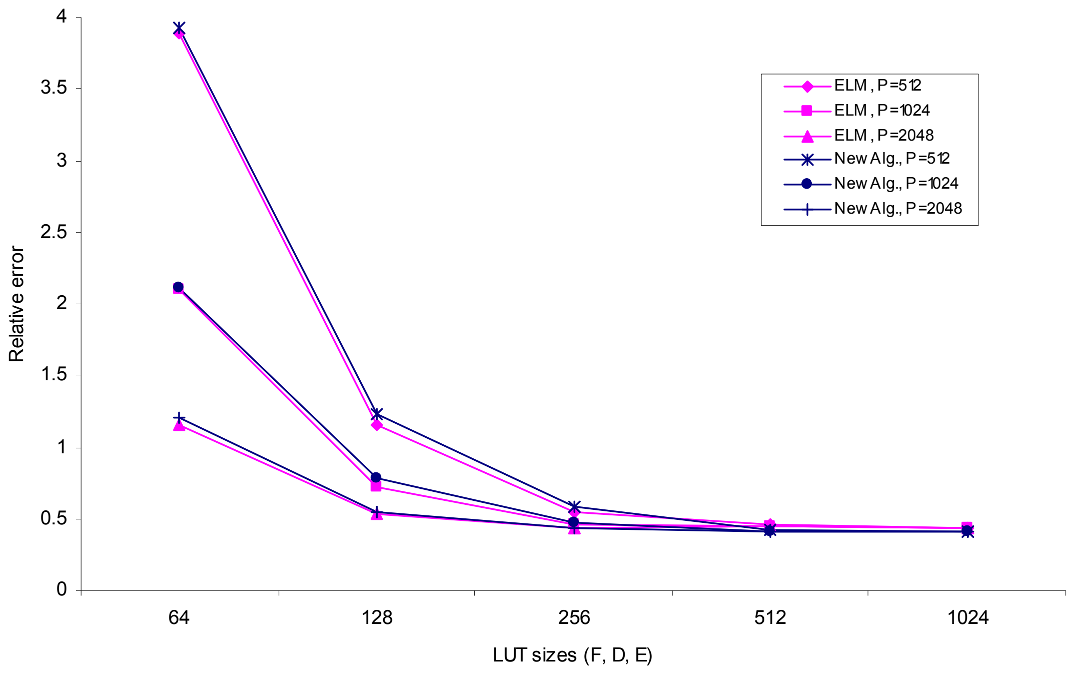


Figure 5-10: Worst-case error between ELM and the new algorithm.

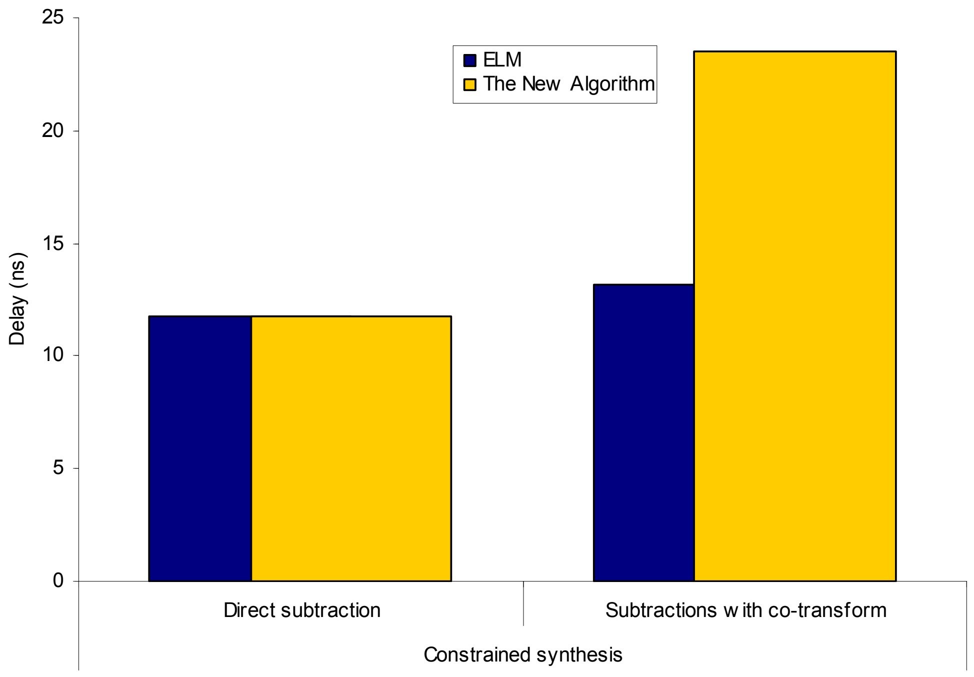


Figure 5-11 : Delay between ELM and the new algorithm.

5.5. Summary

In this chapter, a second-order co-transformation procedure has been introduced and the technique applied in the region $-0.5 < r < 0$ of the subtraction function, where the singularity issue normally arises. By exploiting a similar conceptual approach to that used in the ELM, the original value of r in the range $-0.5 < r < 0$ will be converted into a new value that is certain to lie in the linear region of the function of r, and thereby the singularity issue can be avoided. Not only can accuracy be sustained within FLP limit, but the use of the second-order concept is also capable of reducing the total storage needed to 73% of the total size in the ELM. However, it has a huge impact in terms of delay, much slower than ELM, for subtractions using the co-transformation as a result of using the interpolator twice. Hence, several interpolation schemes are investigated in Chapter 6 where further reductions in table size and delay may possibly be achieved.

CHAPTER 6

6. Function Approximation Scheme for LNS Addition and Subtraction

6.1. Introduction

Apart from improving the co-transformation procedure, as outlined in Chapter 5, which yielded less total storage than in the ELM design, the enhancement of the interpolator architecture is also of utmost importance for LNS addition and subtraction. Reducing the storage space needed in the interpolation process will have a significant impact on the total area of the LNS system. Therefore, several interpolation techniques are explored in this chapter in searching for the best design approach to implement.

Initially, three linear interpolation techniques are designed and compared in terms of the total storage generated and error characteristics. Subsequent analysis evaluates which design approach produces the smallest total area of tables whilst maintaining worst-case error level within FLP limit. A series of developments based on the selected interpolator architecture is then performed through adopting the non-linear interpolation process.

According to the work reported herein, the suggested interpolator module is shown to be able to provide a reduction in total storage in comparison with that needed in the ELM design. The improved version of the interpolator also incorporates tables small enough to be synthesised in logic rather than by using real ROM elements. As a result of this, the total execution delay of the interpolator can be reduced too.

6.2. Function Approximation using Interpolation

During the early days [39], LNS addition and subtraction were simply computed using the direct implementation of functions retrieved from the lookup tables which stored all possible values of $s_b(r)$ and $d_b(r)$. This approach is relatively easy and less complex, particularly for short word-lengths, based on the formula $f{\cdot}2^{f+1}$. However, the impact on the whole system when considering long word-lengths is hopelessly cumbersome. Assuming a 23-bit fractional part for 32-bit word-length, the total storage would be $23 \times 2^{23+1} \approx 3.86 \times 10^8$ words, which is clearly impractical. Therefore, an approximation procedure [6, 43, 47] is employed to overcome the issue of hardware complexity. It has been noted that approximations are widely performed in numerical analysis when difficulty is encountered in carrying out an analytical study involving an original function, due to the nature of the function itself [81].

Theoretically, in order to perform an approximation of LNS addition and subtraction, a new function which can be defined as *p(r)* is introduced which emulates the behaviour of the original function *f(r)*. Rather than directly keeping the complete curve, the *p(r)* is then segmented according to all the required points on each interval and these points are stored in the table. These stored values are then used to obtain an approximation of the calculated operation. There are several obvious ways in which an approximating function can be derived, but the easiest and most often being utilised in many applications is the use of an interpolation technique [82].

As illustrated in the literature, methods such as Taylor [6, 43], Lagrange [46] and Chebyshev [45] are among the interpolation schemes which have been adopted to approximate addition and subtraction functions. However, some of these techniques incorporate other optimisation procedures which can substantially reduce the total storage required. Hence, it is difficult to determine which design is more efficient and can produce less total storage. The solution is to temporarily disregard the optimised designs and instead return to the initial principle or conventional method used in the interpolation procedure. Thus, to select the interpolation technique most suitable for the co-transformation procedure, a preliminary study focuses on various types of interpolation method using conventional linear interpolation concepts. From there, an analysis is performed to determine which design demonstrates the greatest saving in memory space.

6.3. Linear Interpolation

A very brief introduction of linear interpolation has been included earlier in Section 4.2.1. Additional explanation is now necessary, starting by assuming an original function $f(r)$ which crosses at two points, for example r_n and r_{n+1}, as graphically shown in Figure 6-1. By applying an approximation concept, there exists a linear function $p(r)$, a so called unique straight line, which passes through these two locations, $(r_n,f(r_n))$ and $(r_{n+1},f(r_{n+1}))$. The function $p(r)$ can then be used to approximate any value of r that lies between r_n and r_{n+1}, and the result will be utilised to approximate the function $f(r)$. The flow of this process is therefore known as linear interpolation [83] and its mathematical expression can be formulated as in equation (4.1).

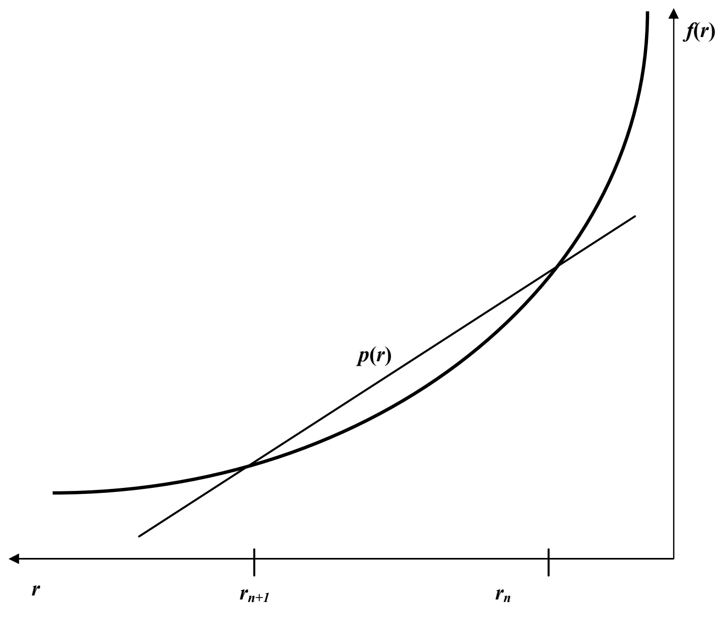


Figure 6-1 : Linear interpolation.

The Taylor approximation method was used to interpolate the function *f(r)* in Chapter 4, even though alternative techniques can be applied which are capable of producing even better error characteristics. This section of the study specifically aims to verify which linear interpolation techniques give better outcomes in terms of the total memory requirements compared to Taylor's approach while keeping the error behaviour within the acceptable limits of the half-bit ulp criterion as in the FLP. Hence, several interpolation procedures based on a linear method are investigated, before applying the non-linear technique to further reduce the total storage needed. Throughout the analysis, the co-transformation architecture proposed in Section 5.3.1 is incorporated into the simulation for subtraction in the range $-1 < r < 0$, and the same simulator model developed in Section 3.3.1 is used.

6.3.1. Linear Taylor Interpolation

Since the Taylor approach is used as a benchmark, the first-order Taylor approximation was modelled based on the theorem in equation (4.5) to yield the expression in equation (4.6). The simulation was conducted by varying the sizes of F and D tables accordingly until the errors reported were relatively similar to the FLP at 0.5. The errors returned by the simulator are equivalent to the FLP calculation due to the application of the formula given in equation (3.5). For illustration purposes, Figure 6-2 depicts the approximation error incurred repeatedly in the interval when interpolating using the Taylor procedure. The actual error results across the whole range of r are provided in Table 6-1 and also displayed graphically in Figure 6-3.

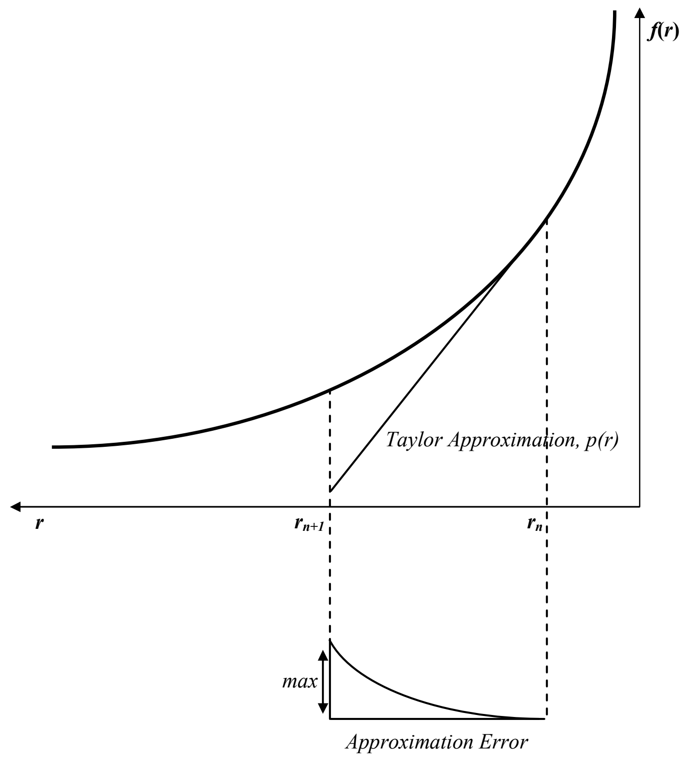


Figure 6-2 : Illustration of linear Taylor approximation error.

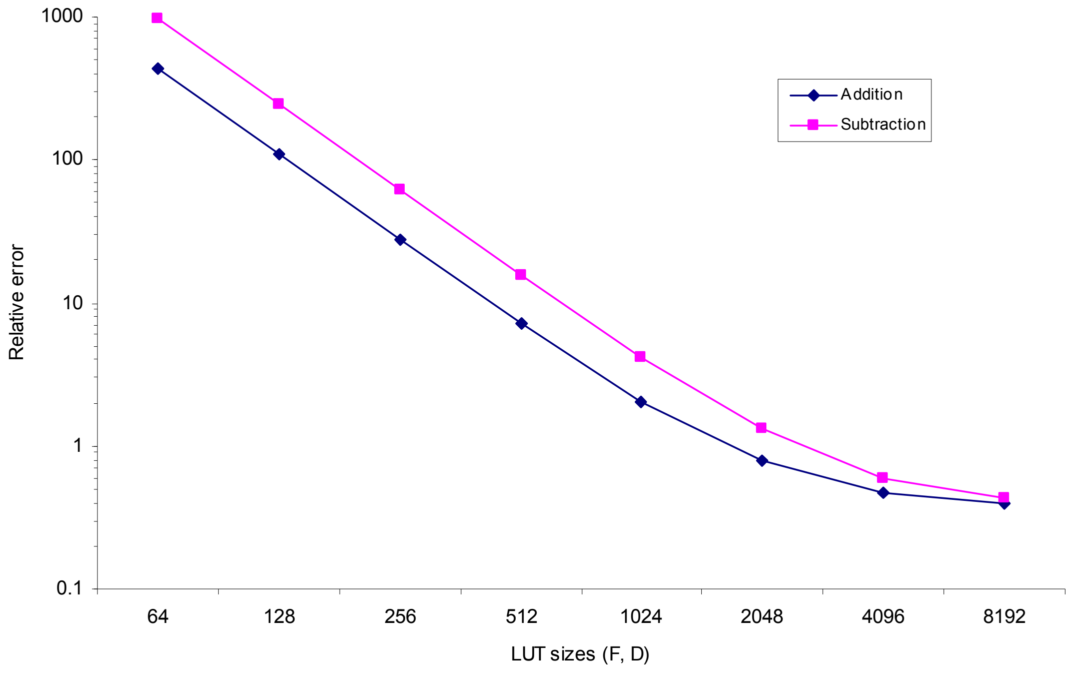


Figure 6-3 : Worst-case error of linear Taylor approximation.

Table 6-1 : Error of linear Taylor approximation.

Operation	F table	D table	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$
ADD	64	64	5	-430.6243	+0.3462
	128	128	5	-108.5963	+0.3464
	256	256	5	-27.4899	+0.3464
	512	512	5	-7.1379	+0.3464
	1024	1024	5	-2.0414	+0.3464
	2048	2048	5	-0.7806	+0.3464
	4096	4096	5	-0.4695	+0.3465
	8192	8192	5	-0.3922	+0.3465
SUB	64	64	5	-110.3230	+973.4232
	128	128	5	-26.3088	+244.6603
	256	256	5	-6.1552	+61.3282
	512	512	5	-1.6741	+15.5805
	1024	1024	5	-0.6551	+4.1404
	2048	2048	5	-0.3861	+1.3059
	4096	4096	5	-0.3540	+0.5989
	8192	8192	5	-0.3540	+0.4369

6.3.2. Linear Lagrange Interpolation

Apart from the linear Taylor approximation, there exists an even simpler type of interpolation approximation that can potentially reduce the total lookup table size whilst sustaining the accuracy within FLP limit. Typically, instead of *f* being focused at one point as in Taylor, it is actually more efficient to spread it over a number of points, which is similar to the technique described by Lewis [46] and Chester [84]. Therefore, the first alternative approximation procedure considered herein can be recognised as a secant line that intersects $f(r)$ at two calculated points as presented in Figure 6-4, an approach called a linear Lagrange interpolation. From an implementation of this approximation method, it is evident from the results in Figure 6-5 that its maximum error is significantly less than that in the Taylor procedure.

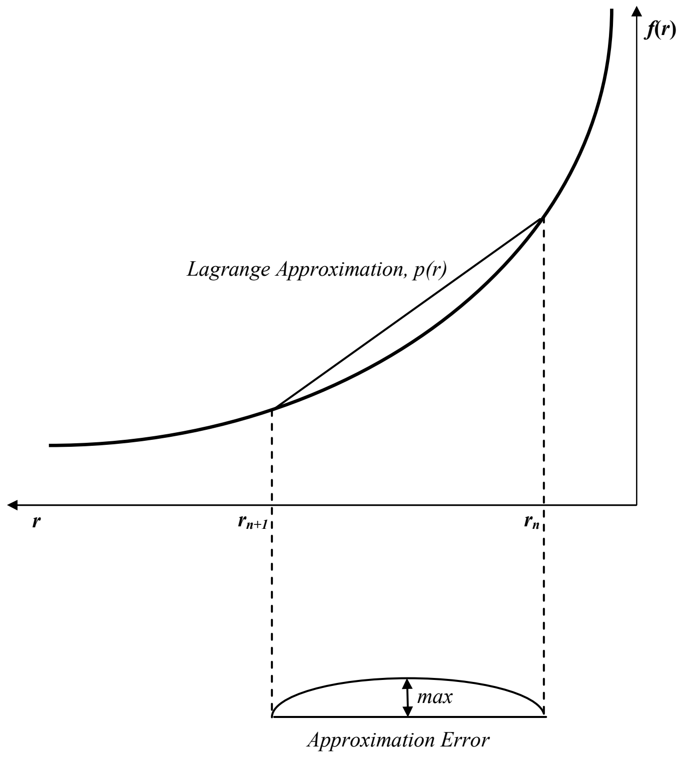
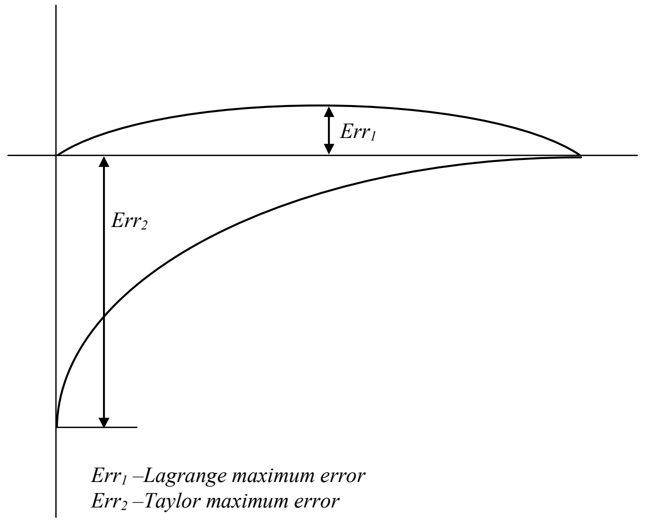


Figure 6-4 : Illustration of linear Lagrange interpolation.



Err_1
Err_2
Err_1 –Lagrange maximum error
Err_2 –Taylor maximum error

Figure 6-5 : Comparison of maximum error in Taylor and Lagrange.

Assuming that the unique straight line $p(r)$ passes through the function $f(r)$ at two distinct locations, say $(r_n,f(r_n))$ and $(r_{n+1},f(r_{n+1}))$, then the linear interpolating polynomial can be constructed as in [83]:

$$p(r)=\frac{(r_{n+1}\cdot f(r_n))-(r_n\cdot f(r_{n+1}))}{r_{n+1}-r_n}+r\left(\frac{(f(r_{n+1})-f(r_n))}{r_{n+1}-r_n}\right) \tag{6.1}$$

The expression can be rearranged in the Lagrange symmetric form giving:

$$p(r)=\left(\frac{r-r_{n+1}}{r_n-r_{n+1}}\right)f(r_n)+\left(\frac{r-r_n}{r_{n+1}-r_n}\right)f(r_{n+1}) \tag{6.2}$$

In order to apply the same arrangement as presented in the Taylor series as shown in equation (4.6), it is useful to note that $p(r)$ may also be written as:

$$p(r)=f(r_n)+(r-r_n)\cdot f'(r_n) \tag{6.3}$$

with the assumption that

$$f'(r_n)=\left(\frac{f(r_{n+1})-f(r_n)}{r_{n+1}-r_n}\right) \tag{6.4}$$

This results in an analogous hardware implementation of the interpolation architecture as described in the ELM design, although of course with different contents of the D table. The result for error characteristics simulated with different lookup table sizes using linear Lagrange interpolation is summarised in Table 6-2, and the worst-case errors for addition and subtraction are shown in Figure 6-6.

Table 6-2 : Error of linear Lagrange approximation.

Operation	F table	D table	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$
ADD	64	64	5	-0.3677	+107.2289
	128	128	5	-0.3677	+27.3178
	256	256	5	-0.3677	+7.1147
	512	512	5	-0.3677	+2.0361
	1024	1024	5	-0.3677	+0.7669
	2048	2048	5	-0.3682	+0.4493
	4096	4096	5	-0.3682	+0.3717
	4096	4096	4	-0.3898	+0.3717
SUB	64	64	5	-242.3960	+23.6166
	128	128	5	-61.3476	+5.5050
	256	256	5	-15.6575	+1.4741
	512	512	5	-4.1757	+0.6045
	1024	1024	5	-1.2991	+0.4128
	2048	2048	5	-0.5831	+0.3871
	4096	4096	5	-0.4056	+0.3871
	4096	4096	4	-0.4047	+0.4474

6.3.3. Linear Lagrange Interpolation – Modified Version

Exploiting the Lagrange format, another potentially useful approximating procedure can improve the maximum error even more. Whenever $p(r)$ is shifted down from the initial position of the Lagrange line, $p(r)$ eventually crosses $f(r)$ at two new values, in this case r_{k1} and r_{k2} as illustrated in Figure 6-7. With the width of the interval still intact between r_n and r_{n+1}, the curve of $f(r)$ is now divided into three different sections. By observation alone using Figure 6-8, small improvements in maximum error can clearly be obtained over the Lagrange scheme.

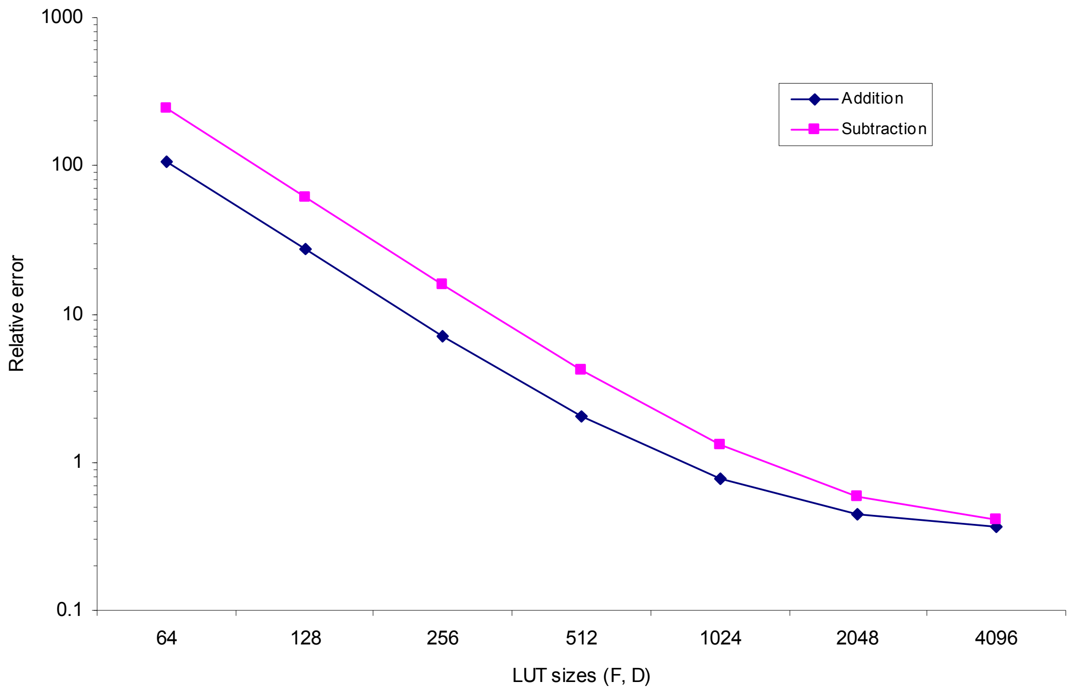


Figure 6-6 : Worst-case error of linear Lagrange interpolation.

Borrowing from equation (6.3) and considering the two different points that intersect *f*(*r*) at r_{k1} and r_{k2}, the formula for this approach is therefore expressed as:

$$p(r) = f(r_{k1}) + (r - r_{k1})\left(\frac{f(r_{k2}) - f(r_{k1})}{r_{k2} - r_{k1}}\right) \tag{6.5}$$

During the simulation process, the evaluation of the optimal values of r_{k1} and r_{k2} is not performed for every single interval in each segment. This is due to the fact that, whilst executing either addition or subtraction operations, the stored values of the F and D tables happen to be extremely similar for every interval within a segment [84]. For that reason, there is no need to optimise r_{k1} and r_{k2} values at each interval since this gives no significant benefit in the sense of hardware realisation. Rather, the optimisation of r_{k1} and r_{k2} are intensively computed for each segment using the powers of two partitioning procedure, *i.e.* 0..-1, -1..-2, -2 ..-4 ..... -16..-32. In order to execute this approximation procedure, a simulator was developed which fulfils the preceding argument concerning optimising r_{k1} and r_{k2} for every segment. Figure 6-9 presents the flow diagram of the simulator design.

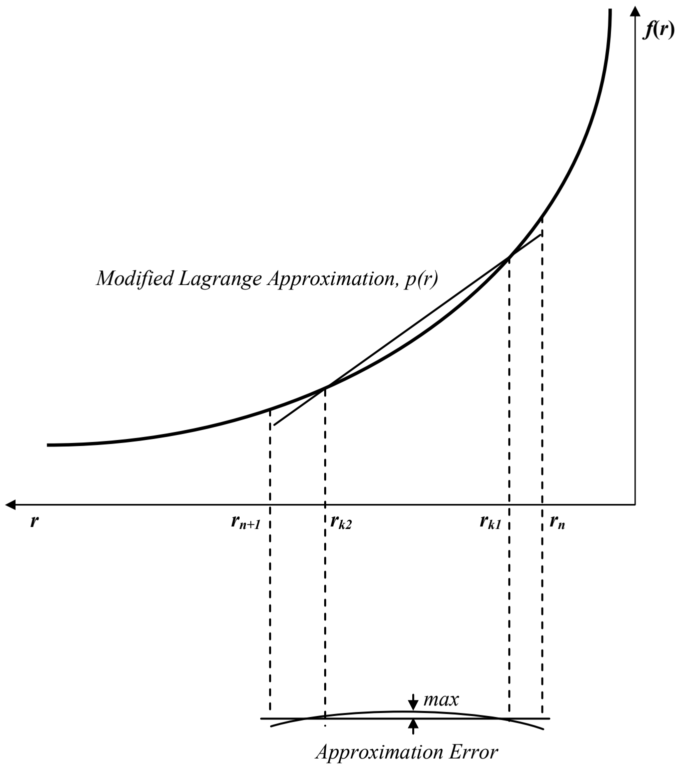


Figure 6-7 : Illustration of modified linear Lagrange interpolation.

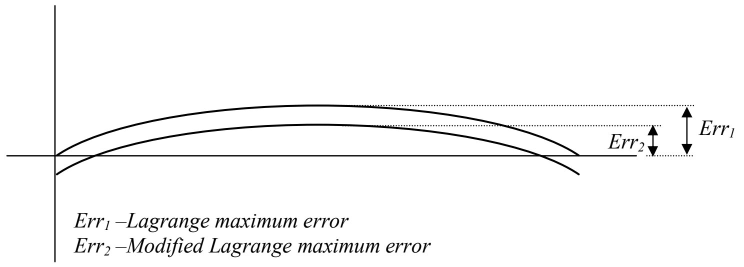
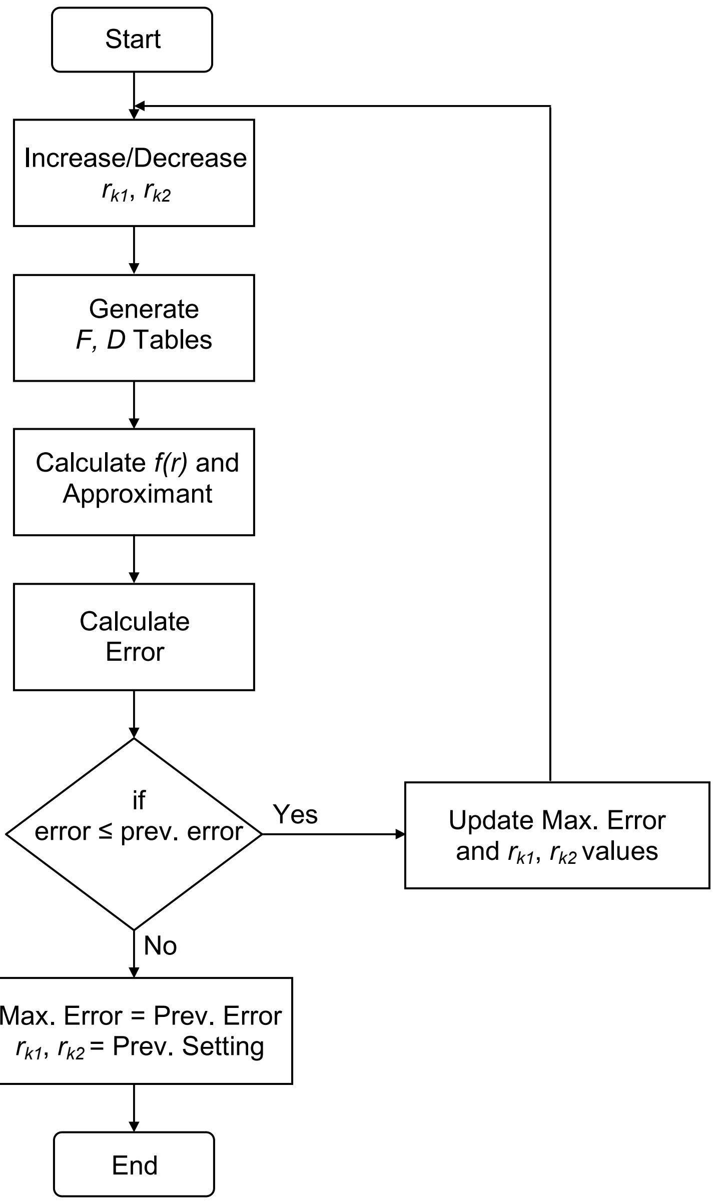

Err_1
Err_2
Err_1 –Lagrange maximum error
Err_2 –Modified Lagrange maximum error

Figure 6-8 : Illustration of maximum error between Lagrange and modified version.



Start
Increase/Decrease r_{k1}, r_{k2}
Generate F, D Tables
Calculate $f(r)$ and Approximant
Calculate Error
if error ≤ prev. error
Yes
Update Max. Error and r_{k1}, r_{k2} values
No
Max. Error = Prev. Error r_{k1}, r_{k2} = Prev. Setting
End

Figure 6-9 : Flow diagram for selection of r_{k1} and r_{k2}.

At first, the values of r_{k1} and r_{k2} were chosen so that the $p(r)$ line intersects $f(r)$ at two locations within the width of r_n and r_{n+1}. F and D tables were then generated according to the arrangement as in equation (6.5). The result of the addition and subtraction operations was then calculated simultaneously with the approximate value for FLP. The computed error was reported and compared with the previous maximum error. The simulator only ends the process whenever the error generated is greater than the previous value, otherwise r_{k1} and r_{k2} were either increased or decreased accordingly before applying the same procedure mentioned above. The analysed errors based on the most optimal values of r_{k1} and r_{k2} using various table sizes are tabulated in Table 6-3 and shown graphically in Figure 6-10.

Table 6-3 : Error of modified linear Lagrange approximation.

Operation	F table	D table	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$
ADD	64	64	5	-0.3723	+107.2102
	128	128	5	-0.3760	+27.2985
	256	256	5	-0.3811	+7.0971
	512	512	5	-0.3811	+2.0183
	1024	1024	5	-0.3816	+0.7481
	2048	2048	5	-0.3820	+0.4314
	4096	4096	5	-0.3836	+0.3566
	4096	4096	4	-0.4220	+0.3498
SUB	64	64	5	-242.3532	+23.6166
	128	128	5	-61.3032	+5.5050
	256	256	5	-15.6142	+1.4741
	512	512	5	-4.1308	+0.6078
	1024	1024	5	-1.2742	+0.4682
	2048	2048	5	-0.5577	+0.4005
	4096	4096	5	-0.3838	+0.4036
	4096	4096	4	-0.3707	+0.4849

6.3.4. Comparison of Linear Interpolators

With the aim of proposing an improved technique for the interpolation process that can dramatically reduce total memory space compared to Taylor's approach, two different types of linear interpolators have been described namely the Lagrange and

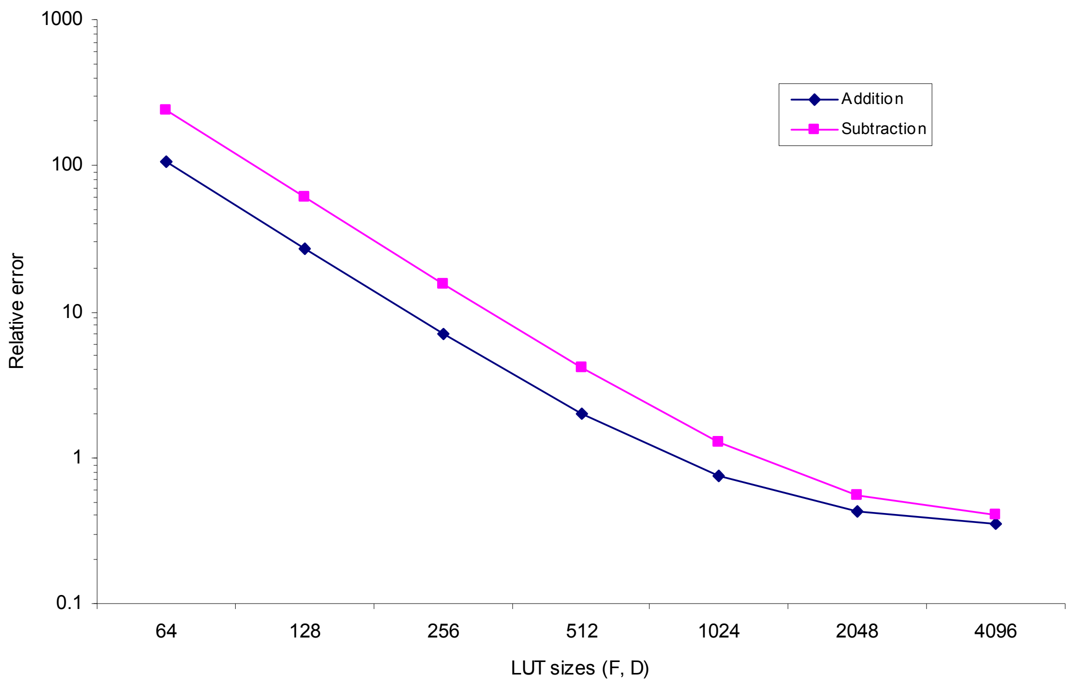


Figure 6-10 : Worst-case error of modified linear Lagrange interpolation.

modified Lagrange interpolation procedures. It should be noted that these linear interpolators do not actually represent any final solution for the implementation of 32-bit LNS add and subtract functions. Rather, this analysis is more likely to lay the basis for further exploration, especially when incorporated with a non-linear interpolation method.

The linear interpolators illustrated are being compared similarly in accordance with the measurement metrics specified in Chapter 3. The memory space has so far consumed a huge proportion of the silicon area of the LNS system. The initial analysis summarised in Table 6-4 focused on the hardware costs in terms of the total storage required by each interpolator technique. In practice, the lookup tables for addition and subtraction operations can be physically allocated to the same storage unit when they are in similar regions. However, for the purpose of this comparative study, the tables are split according to their functions so as to give more precise results. It is also assumed that 36-bits are stored in each address in the F and D tables due to the use of a 5-bit guarded format and since we are not considering the sign bit.

The segmentation scheme deployed is shown in Table 3-2, except for subtraction that covers only five regions because the co-transformation process is employed in the region from 0 to -1.

Table 6-4 : Linear interpolator storage requirements.

Interpolator Technique	Addition Words/Seg.		Subtraction Words/Seg.		Total Storage (kbits)
	F Table	D Table	F Table	D Table	
Taylor	4096	4096	8192	8192	4,718
Lagrange	2048	2048	4096	4096	2,359
Modified Lagrange	2048	2048	4096	4096	2,359

In the calculation of total storage, the total bits involved in co-transformation have been neglected since this does not vary with interpolator method. As far as can be seen from Table 6-4, implementing either the Lagrange or modified Lagrange method can potentially lead to a 50% saving in total space compared with the Taylor scheme. When considering real hardware implementation, the Lagrange and modified Lagrange interpolators can be implemented with similar arrangement as Taylor, but with less memory space. Thereby, it may potentially reduce the total execution delay whilst computing addition and subtraction operations. Evidently, whenever adopting the Lagrange scheme, the design is considerably more straightforward and less complex than modified Lagrange.

Although reduction in total storage is a priority, the error characteristics of the LNS system are also a crucial element. Yet, these two variables are interrelated. Figure 6-11 plots the worst-case errors for the Taylor, Lagrange and modified Lagrange interpolators. It can be seen that adopting the Lagrange interpolation procedure yields improved error behaviour compared to Taylor. The modified Lagrange approach produces error similar to that in Lagrange, but its more complicated design may make it less attractive.

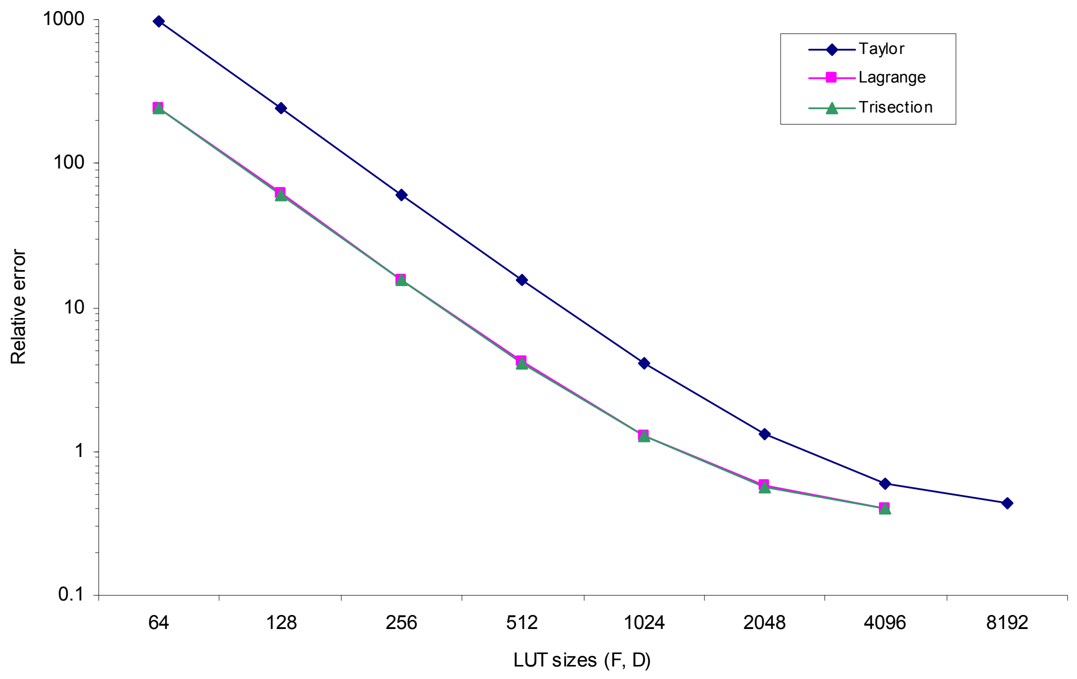


Figure 6-11 : Worst-case error of linear interpolator.

To conclude this analysis, the Lagrange approach is therefore selected as the best candidate for further development using non-linear interpolation, since it can be easily implemented and produces better error characteristic than the Taylor format.

6.4. Non-linear Interpolation

It is known that although linear interpolation entails relatively fast and simple computation, the results may be less accurate and to a certain extent the process requires larger memory space in order to maintain error within FLP limit. Thereby, to generate more precise results with minimal lookup table size while executing the LNS addition and subtraction, non-linear interpolation should be implemented. Only two non-linear interpolation schemes are considered here, a high-order degree method and the approach suggested in the ELM known as an error correction algorithm. Non-linear interpolation as proposed in the ELM has been reviewed in Section 4.2.1. Thus, the high-order method is explained in the next section before comparing it with the method implemented in the ELM. Subsequently, the best of these methods is selected for implementation in the 32-bit LNS system incorporated with the preferred approximation technique as described in Section 6.3.4.

6.4.1. High-Order Degree Method

Linear interpolation can also be categorised as a first-degree polynomial interpolation, because it merely involves two points in constructing a straight line in order to approximate a given function *f*. Therefore, whenever the constructed line passes through more than two locations, it can be defined as a polynomial interpolation of degree greater than one, or a so-called high-order degree interpolation procedure. For a more precise explanation, an example of the mathematical expression illustrated in [81, 85] is used and the generalisation of the equation is based on the Lagrange method, following the analysis presented in Section 6.3.4.

It is first assumed that $p(r)$ approximates $f(r)$ at $n + 1$ points which can be signified as at $r = r_0$, $r = r_1$, ... , $r = r_n$. Whenever $n > 1$, there will be more than two interpolating points and thus the conditions of a high-order degree interpolation process are met. From the mathematical function of the Lagrange format shown in equation (6.2), the polynomial *p* can be re-written as:

$$p(r) = S_0(r)f(r_0) + S_1(r)f(r_1) + \ldots.. + S_n(r)f(r_n) \tag{6.6}$$

where

$$S_0(r) = \frac{(r - r_1)(r - r_2)\ldots(r - r_n)}{(r_0 - r_1)(r_0 - r_2)\ldots(r_0 - r_n)} \tag{6.7}$$

$$S_1(r) = \frac{(r - r_0)(r - r_2)\ldots(r - r_n)}{(r_1 - r_0)(r_1 - r_2)\ldots(r_1 - r_n)} \tag{6.8}$$

This leads to the general form of function *S*:

$$S_i(r) = \prod_{j=0, j \neq i}^{n} \left(\frac{r - r_j}{r_i - r_j} \right) \tag{6.9}$$

Similarly for equation (4.6) it can be summarised as:

$$p(r) = \sum_{i=0}^{n} S_i(r) \cdot f(r_i) \tag{6.10}$$

From this it can be seen that, for each order of the polynomial interpolator, one multiplier and one adder are needed. As the number of the order is increased, there will potentially be a huge impact upon the hardware area and delay through additional multipliers which are connected in series, and possibly lookup tables too.

For these reasons and in order to maximise hardware performance, the non-linear interpolation technique suggested in the ELM is more appropriate, where the multiplication process for each polynomial is executed in parallel. Moreover, the result of the multiplication can be rearranged so that it can be accumulated in a chain of carry-save addition stages, hence potentially improving the execution delay in the system.

6.4.2. Error Correction Algorithm

The development of an error correction algorithm, as shown in [6], is built through the combined effect of linear interpolation together with an algorithm specially defined to correct approximation error. The details of this method are illustrated in Section 4.2.1. From initial observations in Table 4-3 and 6-4, it is obvious that far fewer total bits are involved when applying this technique compared to the linear interpolation with the same approximation format as in the Taylor procedure. If this technique is incorporated with Lagrange interpolation, it may be expected that the size of the lookup tables required can be significantly reduced even further.

6.4.2.1. Implementation of Error Correction Algorithm with Lagrange Interpolation

Despite being chosen for implementation with the Lagrange interpolation as a result of the analysis in Section 6.3.4, indeed the error correction algorithm can also be applied with any other linear interpolator provided that the maximum error within the interval remains proportionately equivalent throughout all regions. This is to ensure that the P table can be reused at each interval whilst computing the error correction process. Based on *p*(*r*) as in equation (6.3), incorporating the error correction algorithm with the linear Lagrange interpolation yields the following approximating function:

$$p(r) = f(r_n) + (r - r_n)f'(r_n) + \underbrace{P(r - r_n)(f(r_n + \Delta(n)) - f(r_n) - \Delta(n)f'(r_n))}_{\text{error correction algorithm}} \qquad (6.11)$$

where

$$\Delta(n) = \frac{r_{n+1} - r_n}{2} \qquad (6.12)$$

Through an implementation of the Lagrange format, the contents of the P and E tables as originally used in the ELM ALU unit consequently need to be replaced. This is merely due to the difference in the derivation of the maximum error in the Lagrange approach which occurs at the midpoint of each subinterval. Table 6-5 tabulates the error simulation results for LNS addition and subtraction based upon a combination of linear Lagrange interpolation together with an error correction algorithm as shown in equation (6.11). Meanwhile, the worst-case error of the system is plotted in Figure 6-12.

In practice, the shaded row in Table 6-5 is the most suitable arrangement to be selected because it uses less total storage compared to the other combinations. However, the required size for the P table is double that previously implemented in

Table 6-5 : Error of Lagrange interpolation using error correction algorithm.

Parameters			ADD		SUB		Worst Case
F,D,E Sizes	P size	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
64	512	5	-1.1474	+1.1674	-2.1395	+2.1157	2.1395
128	512	5	-0.5382	+0.5604	-0.8017	+0.7996	0.8017
256	**512**	**5**	**-0.3912**	**+0.4161**	**-0.4688**	**+0.4647**	**0.4688**
512	512	5	-0.3677	+0.3812	-0.3949	+0.4208	0.4208
1024	512	5	-0.3677	+0.3790	-0.3792	+0.4058	0.4058
64	1024	5	-0.7381	+0.7548	-1.2730	+1.2462	1.2730
128	1024	5	-0.4404	+0.4673	-0.5901	+0.5942	0.5942
256	1024	5	-0.3680	+0.3932	-0.4185	+0.4298	0.4298
512	1024	5	-0.3677	+0.3792	-0.3835	+0.4121	0.4121
1024	1024	5	-0.3677	+0.3790	-0.3792	+0.4058	0.4058
64	2048	5	-0.5393	+0.5537	-0.7833	+0.8566	0.8566
128	2048	5	-0.3920	+0.4136	-0.4764	+0.4682	0.4764
256	2048	5	-0.3677	+0.3824	-0.3940	+0.4208	0.4208
512	2048	5	-0.3677	+0.3792	-0.3791	+0.4121	0.4121
1024	2048	5	-0.3677	+0.3790	-0.3792	+0.4058	0.4058

the ELM. In order to prevent such a bulky size for a single memory in the system, therefore the other combination is taken into consideration. As illustrated in Figure 6-12, when F, D and E tables at 256 words, and P table at 512 words, its worst-case error still within FLP limit of 0.5 LSB. Consequently, this table arrangement, as bolded in Table 6-5, has been closely examined in the next section as to look for potential improvement in the total storage of the interpolator architecture. It is expected that the improved version should need small enough lookup tables to be conveniently synthesised rather than using the explicit ROM elements adopted in the ELM design.

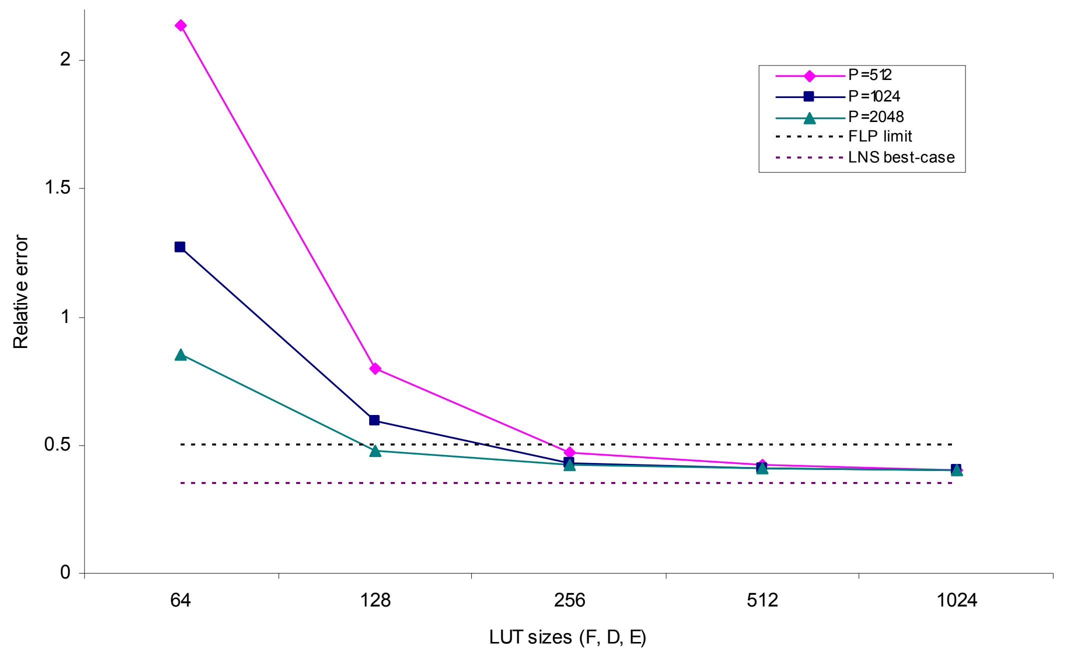


Figure 6-12 : Worst-case error of Lagrange interpolation using error correction algorithm.

6.5. Improvement of Non-linear Lagrange Interpolation

Several modifications are introduced in this section in order to further reduce the total table size needed when performing non-linear Lagrange interpolation. The first solution emphasises the possibility of reducing total storage through partitioning the intervals before the maximum error values stored in the E table can be shared between adjacent subintervals. Meanwhile, another technique is presented which minimises the size of lookup tables particularly in the region $-32 < r < -16$.

6.5.1. Partitioning the Intervals

Theoretically, in order to minimise the error characteristics of non-linear Lagrange interpolation, the $p(r)$ can actually be partitioned into a number of subintervals, for example two, as portrayed in Figure 6-13. Then each subinterval can be individually approximated by the interpolation polynomial. This type of approximation is normally known as a piecewise polynomial, sometimes called a spline-based format. In practice, to ensure that each subinterval has sufficiently small maximum error, the subintervals are divided into an equal space for each interval as described in Figure 6-13.

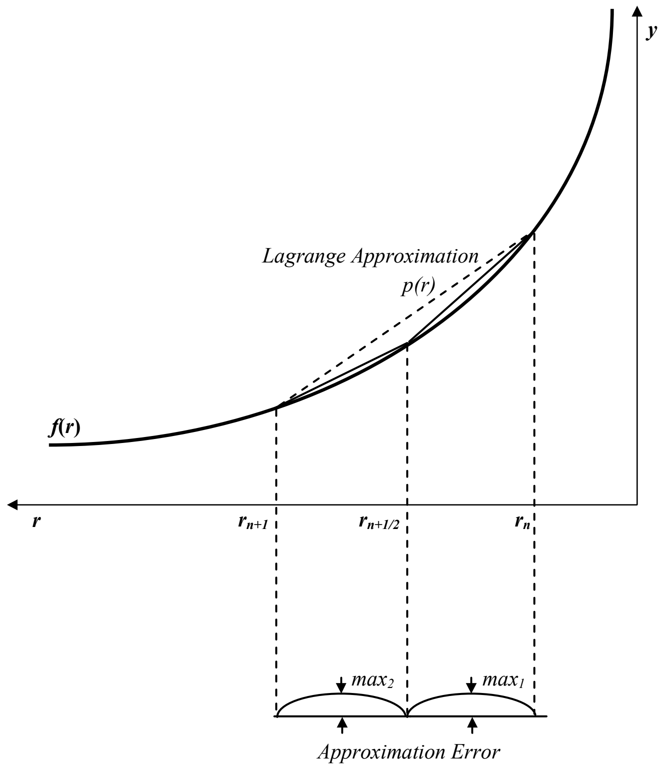


Figure 6-13 : Partitioning the interval based on Lagrange interpolation.

From the simulation of this piecewise polynomial approach based on non-linear Lagrange interpolation, one interesting fact is discovered. As can be observed in Figure 6-14, the maximum error stored in the E table for the first subinterval, *max_1*, appears to be almost equivalent to the maximum error in the second subinterval, *max_2*, which is kept in the other E table. Although the graph plotted in Figure 6-14 is only based on the subtraction function in the region $-2 < r < -1$ with E tables set at 128 words, yet similar conditions occur across every region for both addition and subtraction operations. Consequently the E table can be shared between adjacent subintervals, hence reducing the total storage required for the LNS addition and subtraction unit. In order to verify this suggested arrangement of the E table, Table 6-6 displays the error simulation results if the E table is shared at every interval while performing LNS addition and subtraction using non-linear Lagrange interpolation.

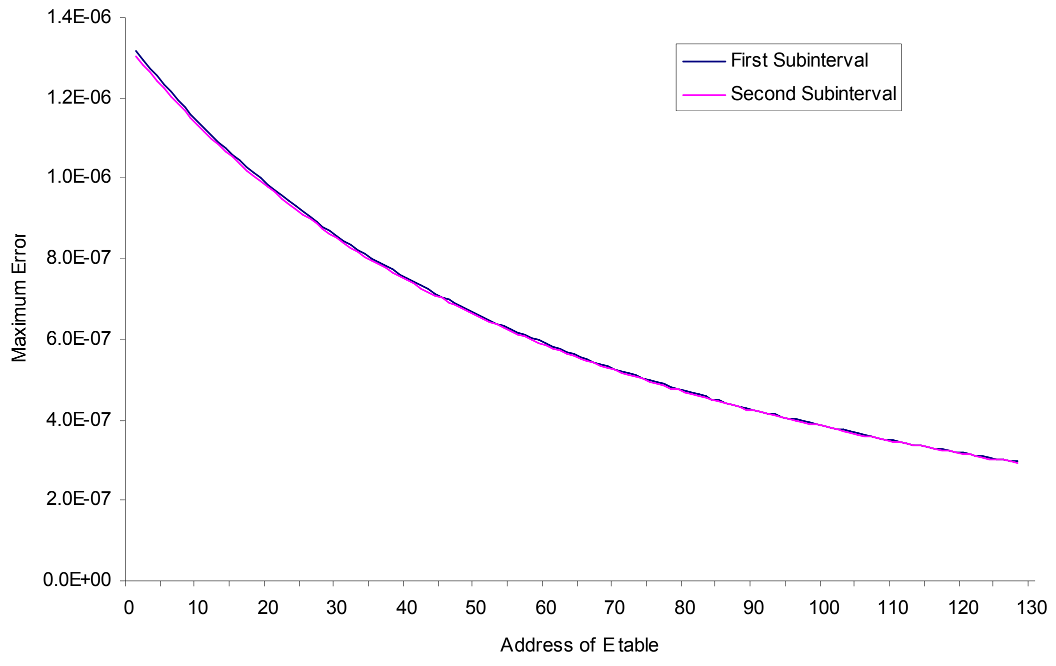


Figure 6-14 : Maximum errors of two adjacent subintervals when executing subtraction in the region $-2 < r < -1$.

From Table 6-6 it is apparent that the preferred combination for the lookup tables is a 512 words P table with 256 intervals for F and D tables. For the E table entries, subtractions now require 128 words per region with only 64 words for additions. A most significant benefit of this table arrangement is that all the tables are individually small enough to be conveniently synthesised in logic, and therefore the total execution delays in memory can be dramatically reduced. Nonetheless, the advantage of a reduction in table size comes at the expense of increasing the number of guard bits from five to six.

6.5.2. Minimising the Lookup Tables

Another alternative that may help is to minimise the sizes of the tables, especially in the region that asymptotically approaches the essential zero condition. Referring to Figure 4-1, it is clear that whenever in the range $-32 < r < -16$, the addition and subtraction functions are very close to the zero line. Thus, the table sizes associated with this region can potentially be decreased due to the fact that certain parts of the contents of the tables will be packed with either zeroes or several repeated values. Considering only the region $-32 < r < -16$, Table 6-7 presents the error simulation results from various table formats, taking into account the concepts discussed earlier in Section 6.5.1.

Throughout the simulation of cases in the range $r > -16$, it is assumed that the sizes of the F, D and E tables remain the same as in the proposal in Table 6-6 where F and D are permanently set at 256 words with 128 and 64 intervals of the E tables for subtraction and addition respectively. The P table is also fixed at 512 words, and a six guard bits format is adopted. Evidently, the lookup tables involved in the region $-32 < r < -16$ can be minimised to only 32 words at each of the F, D and E tables, as depicted in Table 6-7. Although a minor development, this is important because it helps to further reduce the total size of the memory space needed.

6.5.3. Design Summary

As outlined in Table 6-8, the total bits required by the LNS system is 183,296 when the improvements illustrated in Sections 6.5.1 and 6.5.2 are applied. With the maximum size of the tables involved only containing 512 words, it seems that all the tables can individually be synthesised in logic instead of employing real ROM libraries. Elimination of these ROM elements in the LNS system would undoubtedly yield a faster and more compact result. Even with two extra guard bits in addition to the four in the ELM in order to sustain accuracy within the FLP limit will not actually have much impact on the total area of the design.

Table 6-6 : Error of non-linear Lagrange interpolator based on the E table sharing format.

Operation	F,D sizes	E size	P size	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$
Add	64	32	512	6	-4.4126	+1.1674
	128	64	512	6	-0.8761	+0.5538
	256	64	512	6	-0.4623	+0.4527
	256	128	512	6	-0.4237	+0.4019
	512	256	512	6	-0.3626	+0.3696
	64	32	1024	6	-4.3834	+0.7597
	128	64	1024	6	-0.8610	+0.4513
	256	128	1024	6	-0.4177	+0.3808
	512	256	1024	6	-0.3611	+0.3654
	64	32	2048	6	-4.3752	+0.5537
	128	64	2048	6	-0.8577	+0.4034
	256	128	2048	6	-0.4158	+0.3679
	512	256	2048	6	-0.3611	+0.3654
Sub	64	32	512	6	-3.5814	+8.0620
	128	64	512	6	-0.8251	+1.3694
	256	128	512	6	-0.4604	+0.4987
	256	128	512	5	-0.4688	+0.5559
	512	256	512	6	-0.3858	+0.3916
	64	32	1024	6	-3.0338	+7.9901
	128	64	1024	6	-0.6874	+1.3286
	256	128	1024	6	-0.4079	+0.4834
	512	256	1024	6	-0.3720	+0.3916
	64	32	2048	6	-2.6568	+7.9805
	128	64	2048	6	-0.6323	+1.3240
	256	128	2048	6	-0.3824	+0.4821
	512	256	2048	6	-0.3673	+0.3916

Table 6-7 : Error of non-linear Lagrange interpolator in the region -32 < r < -16.

Operation	Region -32 < r < -16			$e'_{min\ rel}$	$e'_{max\ rel}$
	F Size	D Size	E Size		
Add	128	128	128	-0.4623	+0.4527
	128	128	64	-0.4623	+0.4527
	64	64	64	-0.4623	+0.4527
	64	64	32	-0.4623	+0.4527
	32	32	32	-0.4623	+0.4527
	32	32	16	-0.8164	+0.4527
	16	16	16	-0.5742	+0.6297
	16	16	8	-3.0960	+0.6297
Sub	128	128	128	-0.4604	+0.4987
	128	128	64	-0.4604	+0.4987
	64	64	64	-0.4604	+0.4987
	64	64	32	-0.4604	+0.4987
	32	32	32	-0.4604	+0.4987
	32	32	16	-0.4604	+0.8225
	16	16	16	-0.6198	+0.5915
	16	16	8	-0.6198	+3.0880

6.6. Alternative Method: Minimax Interpolation

Since the minimax approximation is among the best techniques for minimising the maximum relative error in each region, Fu et al. in [8, 28] adopted it as a solution to interpolate $F(r)$, which they then implemented in an FPGA-based design. This is not directly comparable to the work in this thesis because equations (2.12) and (2.13) are rearranged to bring r onto the positive axis where the curves have different properties. Whereas in our work, the co-transform is applied to subtractions in the range $r > -1$, Fu has made special arrangements across a region four times this size, i.e. for $r < 4$. These subtractions are performed by decomposing $F(r)$ into two

Table 6-8 : Total storage using the improved interpolator.

	Region	Table	Organisation	Wordlength	Total Bits
Co-transform	$-1 < r < 0$	F1	128 words	32-bit	4,096
		F11	256 words	33-bit	8,448
		F12	256 words	34-bit	8,704
Interpolation	$-16 < r < -1$	F Sub	256 words × 4	30-bit	30,720
		D Sub	256 words × 4	29-bit	29,696
		E Sub	128 words × 4	11-bit	5,632
	$-16 < r < 0$	F Add	256 words × 5	30-bit	38,400
		D Add	256 words × 5	28-bit	35,840
		E Add	64 words × 5	8-bit	2,560
	$-32 < r < -16$	F Add	32 words	30-bit	960
		F Sub	32 words	30-bit	960
		D Add	32 words	28-bit	896
		D Sub	32 words	29-bit	928
		E Add	32 words	8-bit	256
		E Sub	32 words	11-bit	352
	$-32 < r < 0$	P	512 words	29-bit	14,848
Total					183,296

separate functions, both easier to interpolate than $F(r)$ itself. On the other hand, he was able to exploit the equivalence $r \approx F(r)$ at large r, where the need for interpolation was obviated. Over the remaining regions, an adaptive technique selected the most optimal intervals for the application of a minimax algorithm; for the addition function only 416 intervals were required to cover the interpolated range. However, the design incorporates an additional interpolator and tables for evaluation of the auxiliary functions involved in subtraction. It is particularly suitable for use on an FPGA where multiplication hardware is abundant, but it is difficult to extrapolate an estimate of its size or performance in a silicon implementation. Accuracy is within FP limits throughout.

With the aim to evaluate the effectiveness of the minimax approximation over the improved Lagrange interpolation as mentioned in Section 6.5, a second-order minimax-based interpolator has been developed. These interpolators are basically similar in complexity. The F, D and E tables of the improved Lagrange interpolator are replaced by tables (calculated by Maple software) of the 0th, 1st and 2nd order coefficients, and the P table by a multiplier that forms the square of its argument. As depicted in Table 6-9, the most suitable arrangement for each partition is at 128 intervals, for addition and subtraction operations, while the range shifter for subtraction is deployed over the range $-1 < r < 0$.

The total storage based on the minimax arrangement is shown in Table 6-10. When compared with the improved Lagrange scheme which requires 183,296 bits, this is now only 145,408 bits. Nevertheless, due to the fact that the multiplier is used in lieu of the P table, a lesser improvement in terms of speed will be expected when the design is synthesised. This will be evaluated next in the analysis section.

Table 6-9 : Error of the minimax interpolation.

Parameters		ADD		SUB		Worst Case
F,D,E Sizes	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
64	8	-0.5071	+0.4418	-0.5235	+0.5172	0.5235
128	8	-0.4350	+0.4245	-0.4405	+0.4381	0.4405
256	8	-0.4303	+0.4245	-0.4402	+0.4259	0.4402

6.7. ELM with the New Interpolator

For fair justification, it is now necessary to evaluate the new interpolators together with the original co-transformation as presented in the ELM design. Therefore, this section provides an analysis in terms of worst-case error and total tables

Table 6-10 : Total storage using the minimax arrangement.

	Region	Table	Organisation	Wordlength	Total Bits
Co-transform	$-1 < r < 0$	F1	128 words	34-bit	4,352
		F11	256 words	35-bit	8,960
		F12	256 words	36-bit	9,216
Interpolation	$-16 < r < -1$	F Sub	128 words × 4	33-bit	16,896
		D Sub	128 words × 4	33-bit	16,896
		E Sub	128 words × 4	31-bit	15,872
	$-16 < r < 0$	F Add	128 words × 5	31-bit	19,840
		D Add	128 words × 5	30-bit	19,200
		E Add	128 words × 5	28-bit	17,920
	$-32 < r < -16$	F Add	128 words	30-bit	3,968
		F Sub	128 words	30-bit	4,224
		D Add	128 words	28-bit	3,840
		D Sub	128 words	29-bit	4,224
Total					145,408

required when the improved Lagrange and minimax interpolators are implemented in conjunction with the first-order co-transformation.

6.7.1. Improved Lagrange Interpolation

Using a similar interpolation concept as described in Section 6.5 but this time in combination with the first-order co-transformation, again the F and D tables are also best implemented in 256 words for addition and subtraction functions. While the P table requires 512 words, the E table for subtractions need 128 intervals and 64 words for additions. Table 6-11 summarises the error of the combined architecture. It has to be noted that in the region $-32 < r < -16$, the F, D and E tables are permanently set to 32 words throughout the analysis. With this arrangement, the total storage is 315,776 bits as depicted in Table 6-12.

Table 6-11 : Error of ELM with improved Lagrange interpolator.

Operation	F,D sizes	E size	P size	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$
Add	64	32	512	6	-4.4126	+1.1674
	128	64	512	6	-0.8761	+0.5538
	256	64	512	6	-0.4623	+0.4527
	256	128	512	6	-0.4237	+0.4019
	512	256	512	6	-0.3626	+0.3696
	64	32	1024	6	-4.3834	+0.7597
	128	64	1024	6	-0.8610	+0.4513
	256	128	1024	6	-0.4177	+0.3808
	512	256	1024	6	-0.3611	+0.3654
	64	32	2048	6	-4.3752	+0.5537
	128	64	2048	6	-0.8577	+0.4034
	256	128	2048	6	-0.4158	+0.3679
	512	256	2048	6	-0.3611	+0.3654
Sub	64	32	512	6	-2.1395	+8.0491
	128	64	512	6	-0.7862	+1.3576
	256	128	512	6	-0.4749	+0.4904
	512	256	512	6	-0.4064	+0.3775
	64	32	1024	6	-1.2926	+7.9861
	128	64	1024	6	-0.5901	+1.3173
	256	128	1024	6	-0.4195	+0.4721
	512	256	1024	6	-0.4027	+0.3742
	64	32	2048	6	-0.8003	+7.9660
	128	64	2048	6	-0.4753	+1.3128
	256	128	2048	6	-0.4027	+0.4678
	512	256	2048	6	-0.4027	+0.3739

Table 6-12 : Total storage of ELM with improved Lagrange interpolator.

	Region	Table	Organisation	Wordlength	Total Bits
Co-transform	$-0.5 < r < 0$	F1	2048 words	34-bit	69,632
		F2	2048 words	33-bit	67,584
Interpolation	$-16 < r < -0.5$	F Sub	256 words × 5	30-bit	38,400
		D Sub	256 words × 5	29-bit	37,120
		E Sub	128 words × 5	11-bit	7,040
	$-16 < r < 0$	F Add	256 words × 5	30-bit	38,400
		D Add	256 words × 5	28-bit	35,840
		E Add	64 words × 5	8-bit	2,560
	$-32 < r < -16$	F Add	32 words	30-bit	960
		F Sub	32 words	30-bit	960
		D Add	32 words	28-bit	896
		D Sub	32 words	29-bit	928
		E Add	32 words	8-bit	256
		E Sub	32 words	11-bit	352
	$-32 < r < 0$	P	512 words	29-bit	14,848
Total					315,776

6.7.2. Minimax Interpolation

When applying the co-transformation scheme as outlined in the ELM design together with the minimax interpolation, the minimum size of F, D and E tables required to perform addition and subtraction is at 128 words as shown in Table 6-13. Consequently, the storage requirement of the LNS system based on this format is 280,704 bits as reported in Table 6-14.

Table 6-13 : Error of ELM with minimax interpolator.

Parameters		ADD		SUB		Worst Case
F,D,E Sizes	Guard Bits	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
64	8	-0.5071	+0.4418	-0.5235	+0.5052	0.5235
128	8	-0.4350	+0.4245	-0.4405	+0.4261	0.4405
256	8	-0.4303	+0.4245	-0.4402	+0.4139	0.4402

Table 6-14 : Total storage of ELM using the minimax arrangement.

	Region	Table	Organisation	Wordlength	Total Bits
Co-transform	$-0.5 < r < 0$	F1	2048 words	35-bit	71,680
		F2	2048 words	36-bit	73,728
Interpolation	$-16 < r < -0.5$	F Sub	128 words × 5	33-bit	21,120
		D Sub	128 words × 5	33-bit	21,120
		E Sub	128 words × 5	31-bit	19,840
	$-16 < r < 0$	F Add	128 words × 5	31-bit	19,840
		D Add	128 words × 5	30-bit	19,200
		E Add	128 words × 5	28-bit	17,920
	$-32 < r < -16$	F Add	128 words	30-bit	3,968
		F Sub	128 words	30-bit	4,224
		D Add	128 words	28-bit	3,840
		D Sub	128 words	29-bit	4,224
Total					280,704

6.8. Comparison Analysis: First-order and Second-order Co-transformation with the New Interpolator

As described in Figure 6-15, implementing the improved Lagrange and minimax interpolation schemes in conjunction with the first-order co-transformation would reduce total storage to 89% and 79% respectively, of the former size in the original ELM. However, applying the improved Lagrange together with the second-order co-transformation has significantly reduced the total bits to merely 183,296 bits, representing savings to 51% of the ELM design. Therefore, merging the first-order co-transformation with either improved Lagrange or minimax interpolator is still not really significant as those benefiting from the second-order co-transformation.

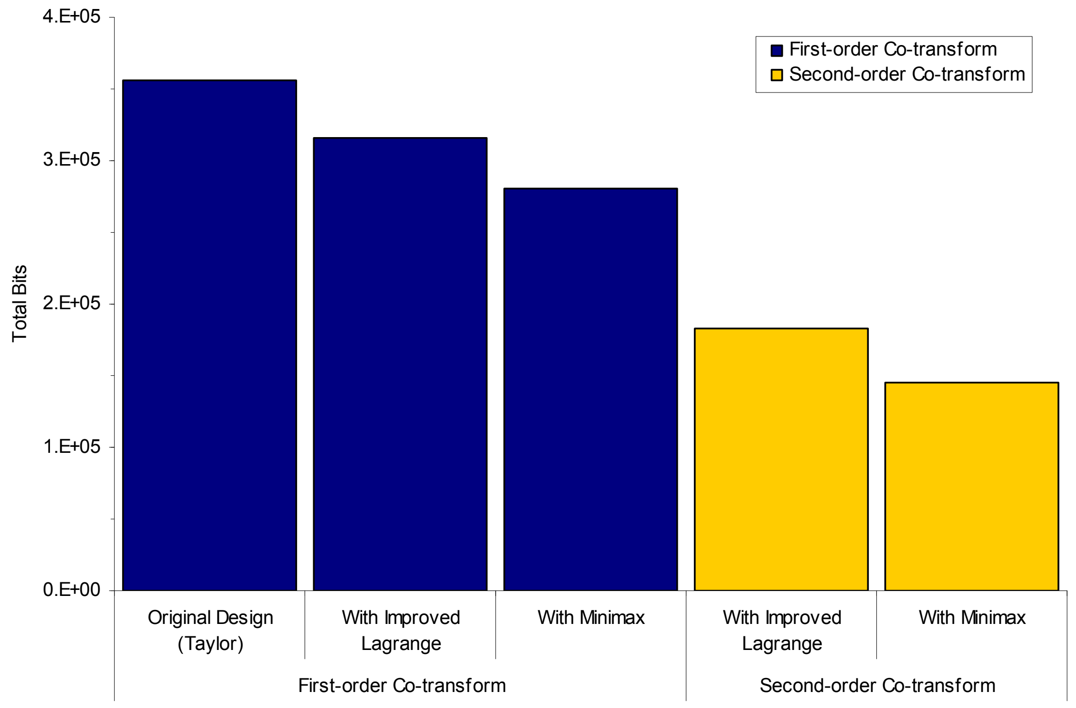


Figure 6-15 : Storage requirement for 32-bit LNS addition and subtraction.

On the other hand, although halving the size of tables when applying a minimax interpolation scheme together with the second-order co-transformation, the reduced storage is not so significant in comparison with using the improved Lagrange approach as illustrated in Figure 6-15. This is due to the fact that 8 guard bits are required as to achieve desired accuracy, and therefore the total storage of co-transform tables have now also increased as to accommodate the additional guard bits. Furthermore, the E tables, which previously held small values, now hold full-size coefficients. Thus, when compared with the improved Lagrange, the total storage based on the minimax arrangement has been reduced from 183,296 bits to only 145,408 bits.

In terms of speed, the implementation of the improved Lagrange scheme has shown to be able to provide the shortest delay in executing addition and direct subtraction as presented in Figure 6-16. Based on the constrained synthesis of this arrangement, the delay has been reduced to 60% of the delay in the constrained synthesis of the ELM. The reduction in delay does actually gain from the benefit that all the tables are now small enough to be conveniently synthesised in logic, which then may yield improvements in the critical speed path. However, the combination of the improved Lagrange with the second-order co-transformation has caused a slight increase in delay when subtractions using the co-transformation due to applying the interpolator twice. Delay increases in approximately by 12% of the delay in the ELM.

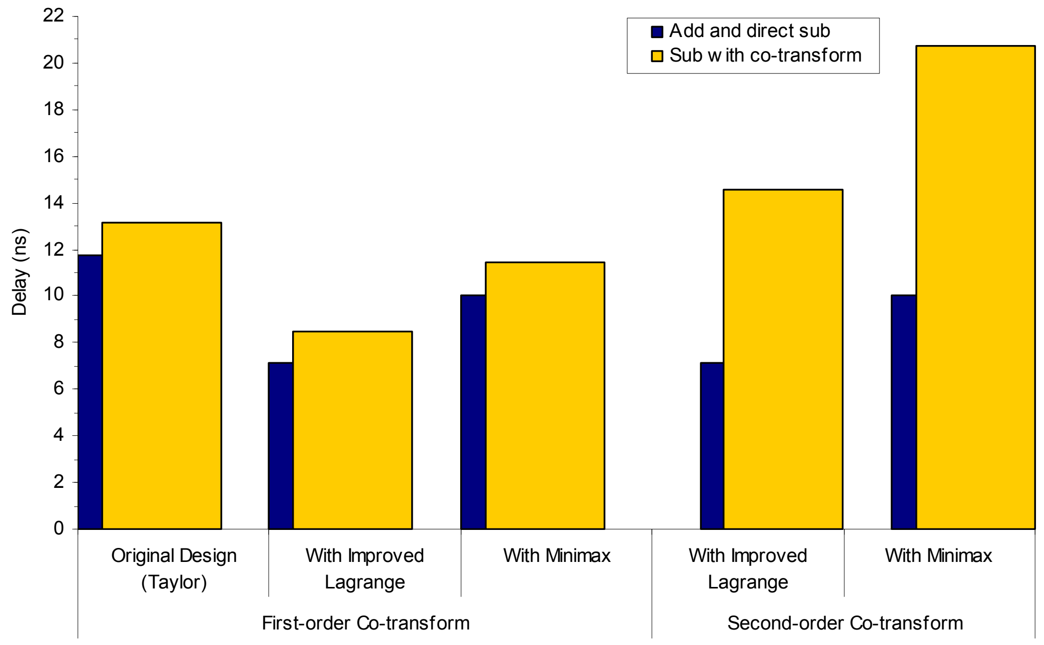


Figure 6-16 : Delay times for 32-bit LNS addition and subtraction.

In contrast, the minimax interpolator does not compare so favourably in terms of speed. It can be seen that its delay is approximately 3 ns more than that of the improved Lagrange interpolator when computing addition and direct subtraction. This is due to the inclusion in the critical path of the extra multiplier that forms the quadratic term. However, this multiplier can also be replaced with the dedicated squaring circuit as to reduce the delay but studies [86, 87] have shown that it can only reduce it by up to 25%. As this would amount to less than a nanosecond in this case it is unlikely to be worth the effort involved in designing it. Using the improved Lagrange interpolator which incorporates the error correction algorithm, on the other hand, is using the P table as to replace this extra multiplier. With the P table is independently designed based on the functions of nine terms and can be accessed in parallel with other lookup tables, therefore a great reduction in delay can be achieved. Consequently as can be observed in Figure 6-16, using the minimax interpolator in conjunction with the second-order co-transformation, it appears to be the slowest in performing subtractions using the co-transformation.

From the perspective of the area-delay product based on the subtractions using co-transformation, when the first-order co-transform design combined with the improved Lagrange or minimax interpolator, the new area-delay product is 57.3% or 68.7% respectively, of its value in the ELM. For combination between minimax interpolator and the second-order co-transformation, the new area-delay product is 64.4%. However, the lowest area-delay product is obtained when using the improved Lagrange in conjunction with the second-order co-transformation, whereby only 57.1% of the value in the ELM. Thereby, obviously it can be seen that the most suitable architecture of the LNS addition and subtraction can be constructed by merging the second-order co-transformation approach with the improved Lagrange interpolator.

6.9. Summary

Previously in the ELM design, the Taylor approximation method was applied during the interpolation process. However, the comparison analysis based on linear interpolation shows that a remarkable reduction in total storage, approximately 50%, can be gained when using Lagrange and modified Lagrange approximations. Furthermore, a very significant improvement in error characteristics has been achieved when implementing either Lagrange or modified Lagrange concept compared to the Taylor approach. In fact, applying Lagrange scheme is more attractive than modified Lagrange due to its simplicity in arrangement.

Improving the Lagrange approach using the non-linear interpolation has revealed further reduction in the size of the total tables. Through partitioning the interval into a number of subintervals, the E table can be shared between adjacent subintervals. Hence, only 128 words and 64 words are needed for the E tables in subtraction and addition respectively. Moreover the tables involved in the region $-32 < r < -16$ can be minimised to 32 words because the curves for addition and subtraction functions in this particular region almost approach the essential zero condition. An alternative method using minimax interpolation has also been examined. Although further reduction in total storage can be achieved, it suffers in terms of speed.

Ultimately, the results based on the comparative analysis conducted have indicated that the implementation of the improved interpolator together with the second-order co-transformation in the LNS addition and subtraction is the best arrangement.

CHAPTER 7

7. Logarithmic Number System Arithmetic Unit

7.1. Introduction

As concluded in Chapter 6, using the improved Lagrange interpolator in conjunction with the second-order co-transformation would produce a great improvement on the original ELM design. Although the improved Lagrange approach offers a modest reduction in area, it has a significant reduction in delay. The second-order co-transformation, on the other hand, offers a more substantial reduction in area, but is dependent upon a reduction in delay which the improved interpolator provides. This means that both techniques are needed to deliver a worthwhile advance for the new ALU system. Therefore, Chapter 7 outlines the details of the design and synthesis processes for the complete ALU system based on the new techniques as well as analysing the system based on comparisons with the ELM and FLP units.

An analysis of the system shows the effectiveness of the proposed LNS design compared with the ELM. It is expected that the new LNS arithmetic unit will be able to operate at the shortest time when performing addition and direct subtraction operations, as well as requiring a lower total silicon area in comparison with the ELM. Meanwhile, the design is also evaluated against two FLP arithmetic units built using a similar process technology, and the results will also demonstrate the suitability of the new LNS design in future DSP chips.

7.2. Arithmetic Unit Design

The simplest operations in an LNS arithmetic unit design are multiplication and division. The hardware implementation of these numeric functions is a direct translation from the algorithms into corresponding functional modules. However, the more intricate addition and subtraction operations require an additional understanding of physical requirements during the implementation stage. This section thus presents the practical hardware solutions for the four basic arithmetic operations of the LNS system which correspond with their fundamental algorithms. Figure 7-1 illustrates the conceptual arrangement of the LNS arithmetic unit.

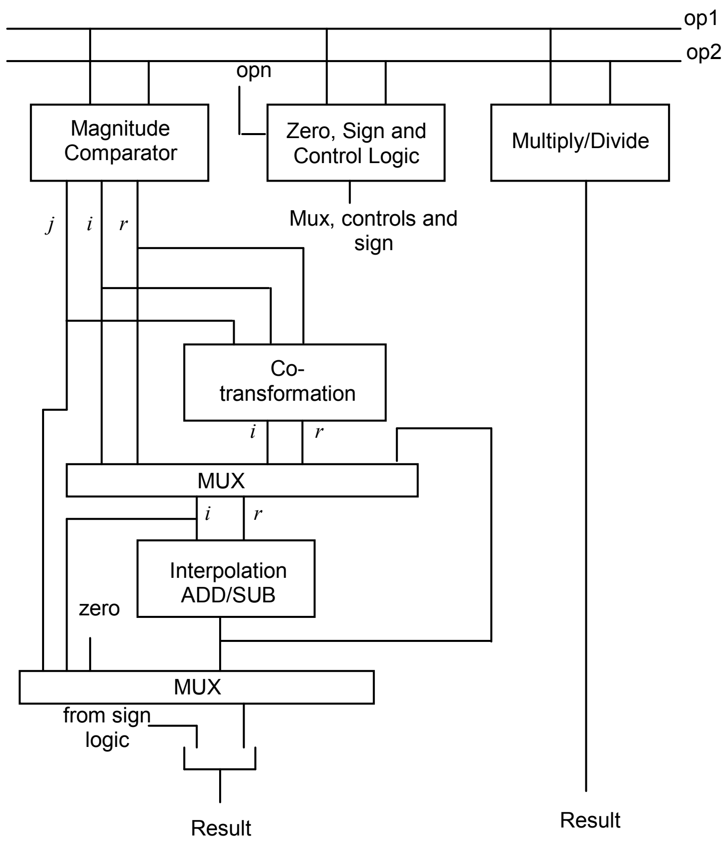


Figure 7-1 : Conceptual arrangement of the LNS arithmetic unit.

7.2.1. Multiply/Divide Unit

As shown in equations (2.8) and (2.9), the multiplication and division functions can be executed simply using FXP addition and subtraction units respectively. Thus, one possible hardware implementation for these operations is given in Figure 7-2. By adding the XOR (exclusive-OR) gate to the full adder circuit, addition and subtraction can be computed using a single hardware configuration. Consequently, this requires the use of fewer circuits than would be required for separate add and subtract functions. Conceptually, whenever in the LNS multiplication mode, the operand = ‘0’, and the outputs of the XOR gate will be the same as the B inputs. In this situation, the hardware performs addition process of the two numbers. In contrast, subtraction is accomplished by setting the operand to logic ‘1’ which therefore can be used to calculate the LNS division operation. In order to determine the sign bits, another XOR gate is inserted into the architecture and similarly a further XOR gate is used to set an overflow flag.

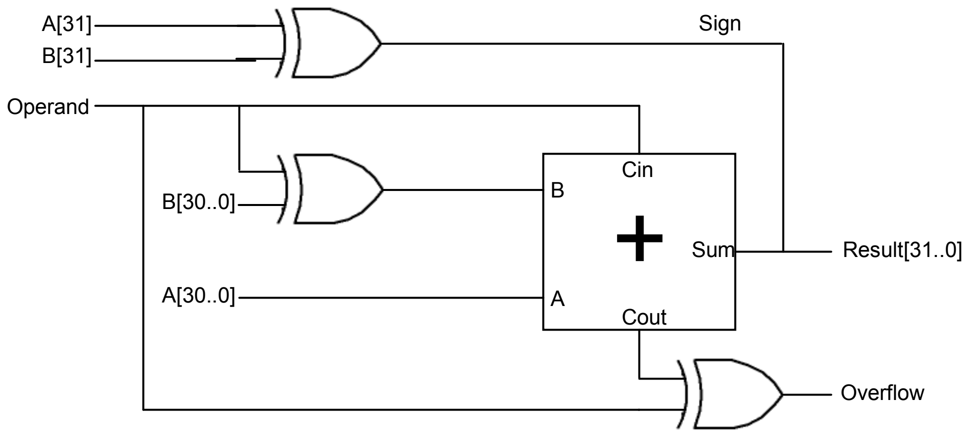


Figure 7-2 : Multiply/Divide hardware implementation.

7.2.2. Add/Subtract Unit

The add/subtract unit requires substantially more complicated functions to be implemented, potentially involving even more complex hardware than in the multiply/divide unit. Even worse, particularly in the range $r > -1$, the subtraction operation often demands a huge table size in order to maintain the accuracy of the system. However, the second-order co-transformation architecture shown in Section 5.3.1 leads to a significant reduction in the total lookup tables when the subtraction function is executed near singularity. Furthermore, the improved Lagrange interpolation method illustrated in Section 6.5 is also capable of reducing the table size needed for both addition and subtraction operations. The block diagram in Figure 7-3 depicts the hardware implementation of the LNS add/sub unit.

Assuming that LNS subtraction requires a complete co-transformation procedure, the entire unit in Figure 7-3 can be implemented with a worst-case delay of three ROM accesses, two FXP multiplications, four FXP additions with 2-inputs, and three FXP additions with 3-inputs. There are also other delays in supporting the logic and multiplexors. On the other hand, the critical speed path of LNS addition merely includes an ROM access, an FXP multiplication, and an FXP full adder with 2-inputs and 3-inputs. This is due to the fact that only the interpolation module is needed to perform the function.

It can be seen that the speed of the system predominantly relies on three main components, namely memory, the FXP full adder and the FXP multiplier. In order to maximise the speed of the LNS addition and subtraction operation, the lookup tables are now small enough to be individually synthesised, rather than using ROM libraries as in the ELM design. In addition, the implementation of the high speed FXP adder and multiplier in the system also contributes to minimising the overall delay. In this respect, a combination of the carry-lookahead and carry-select adder (CLA/CSLA) together with Booth-Wallace multiplier are selected.

7.3. Hardware Implementation of a 32-bit LNS System

With reference to Figure 7-3, three components tend to dominate the LNS design in the sense of timing and floor planning. Adopting dedicated ROM libraries into a design often introduces major complications in terms of the speed and area of the system. Thus, elimination of these elements should yield an increase in performance. Since only three small lookup tables are involved in this LNS design, 128, 256 and 512 words, it would be more realistic to use synthesised ROM during the practical implementation rather than real ROM libraries. Consequently, the system is capable of yielding faster and more compact results.

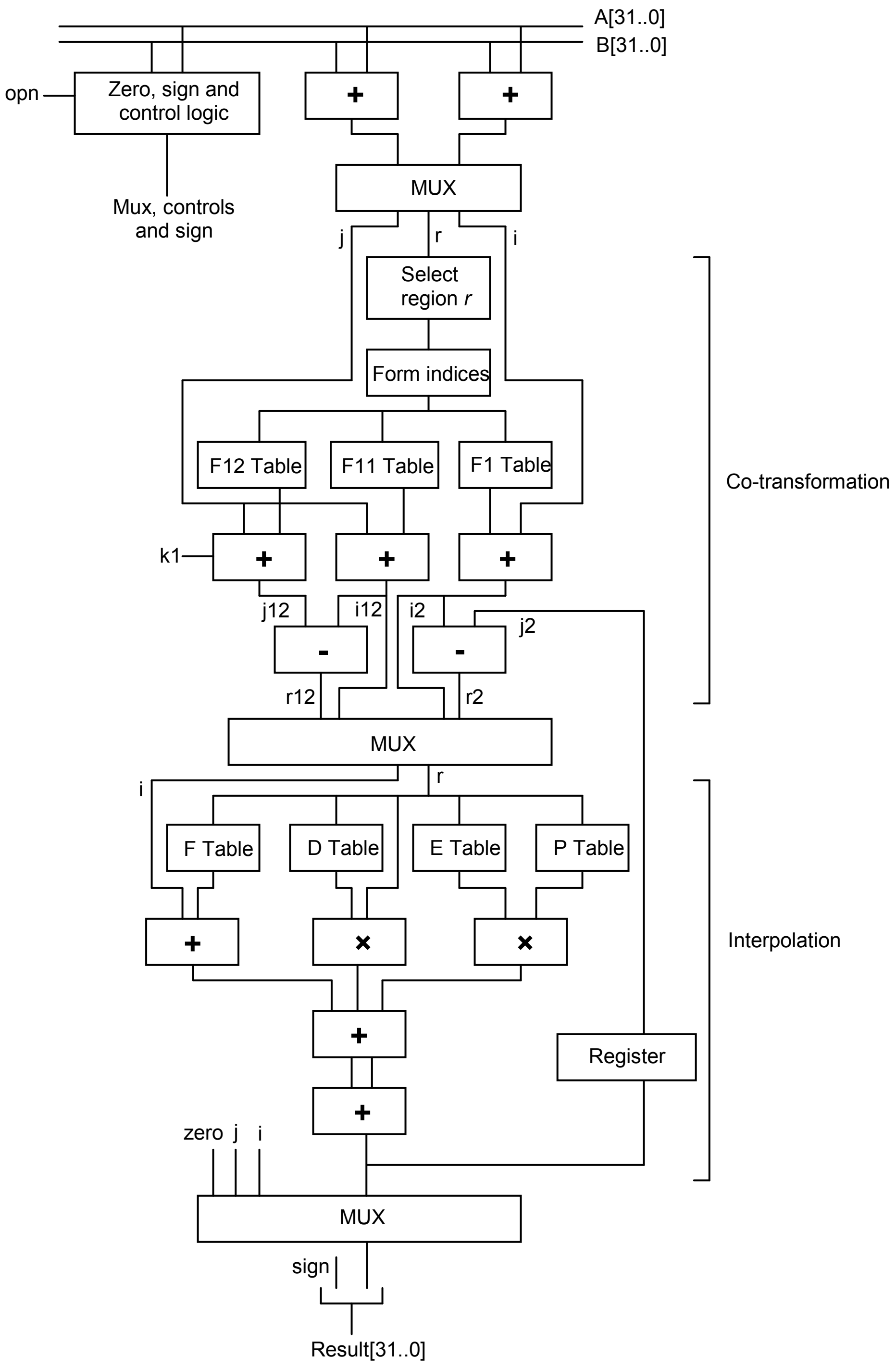


Figure 7-3 : The hardware implementation of the LNS add/sub unit.

The timing required to propagate the carry signal to output during the FXP addition operation could also contribute to the overall delay of the LNS system. Therefore, careful selection of the FXP adder architecture is needed in order to minimise the worst-case delay in the design. In this case, the CLA/CSLA adder [88, 89] has been adopted because it is able to operate effectively in reducing delay when performing the FXP addition process, and is therefore implemented in the hardware design of the LNS architecture.

For the FXP multiplication operation, two multiplier units are required in parallel to compute part of the interpolation process in the LNS system. As explicitly reported in [90], the combination of the Booth algorithm and the Wallace tree structure give the best speed and total device area in comparison with the other types of multipliers. Thereby in the case of the E × P multiplication process involving 12-bit by 29-bit inputs, a radix-4 modified Booth algorithm was employed to generate partial products before applying the Wallace tree structure to compute the final result. Whereas for the D × δ process which requires 29-bit by 26-bit multiplication operation, a higher radix multiplier is needed as to reduce the partial product rows commensurately. In this case a radix-8 Booth multiplier with Wallace tree was chosen.

7.4. Synthesis Results

The proposed LNS arithmetic unit based on the architecture described in Figure 7-3 was synthesised using Faraday 0.18 μm CMOS technology, and its area and delay metrics are tabulated in Table 7-1. When the design is constrained for maximum speed, the delay in addition and direct subtraction functions is at 7.10 ns. Nonetheless, for the small proportion of subtractions that require co-transformation procedure, particularly in the range of $r > -1$, the worst-case delay is approximately double the delay of addition and direct subtraction, at 14.79 ns. This is due to the re-use of the interpolator whenever r is in the region above -1. A delay of the 32-bit CLA/CSLA design is reported for multiplication and division, at 1.16 ns, since these operations can only be computed using a FXP adder unit. The total silicon area of this LNS would be 599,871 μm^2.

Table 7-1 : Delay times and total device area of 32-bit LNS arithmetic unit.

Function	32-bit LNS Arithmetic Unit	
	Delay (ns)	Area (μm^2)
Add / Sub	7.10	589,357
Sub (Co-transform)	14.79	
Mul / Div	1.16	10,514

7.5. Design Analysis

For an analysis comparable with the data presented for the MONARCH and DIVA FLP implementations, all the results described below for the ELM are based on the constrained synthesis. From the graph in Figure 7-4, the critical speed path of the new LNS shorter than that in the original ELM when executing addition or direct subtraction, a reduction from 11.74 ns to 7.10 ns. The delay has also been reduced by 4.18 ns and 11.7 ns of the delays in the MONARCH and DIVA respectively. Given that multiplication can be computed solely using FXP addition, the delay generated in the new LNS therefore only at 1.16 ns, in which 10% of MONARCH delay and 6% of DIVA delay. Similarly, division operation completes with better delay than in the MONARCH and DIVA, a reduction from 33.83 ns and 45.11 ns to 1.16 ns respectively. In the co-transformation involved during subtraction, there is a marginal increase in the delay in the new LNS from 13.15 ns as initially in the ELM to 14.79 ns.

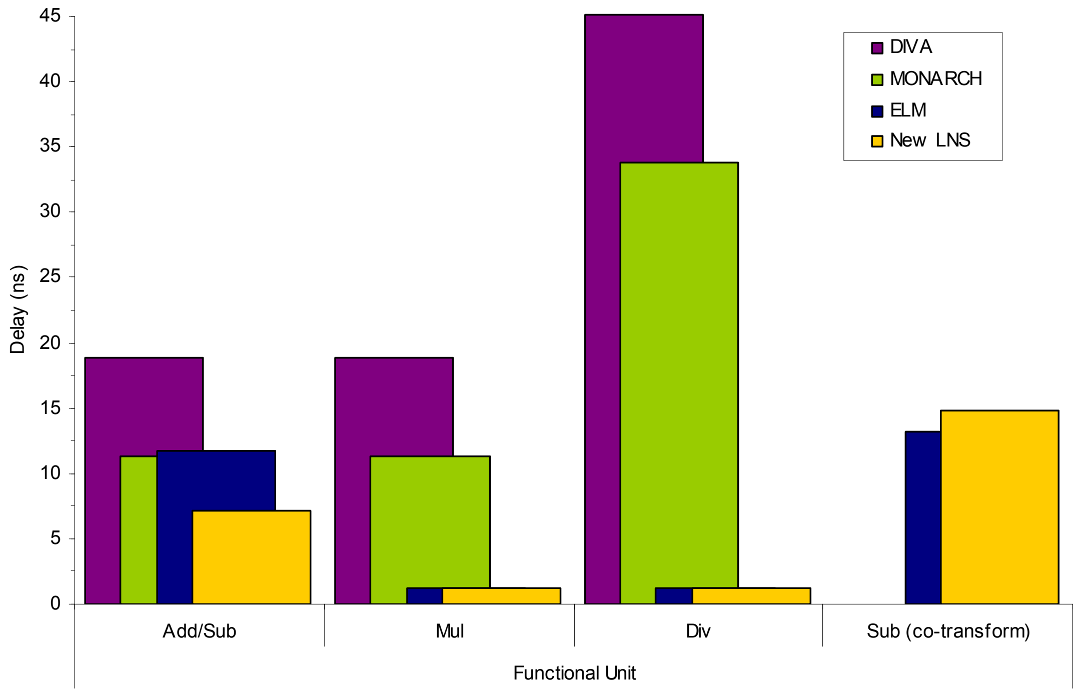


Figure 7-4 : Delays in nanoseconds and cycles of four different arithmetic implementations.

Nevertheless, the re-use of the interpolator in the new LNS is unlikely to be of practical significance in a microprocessor because operations would be fitted into a multiple of some clock cycles. At, say 266 MHz, addition and direct subtraction in the new LNS could be calculated in two cycles (7.52 ns), and multiplication and division in a single cycle (3.76 ns), whereas in subtractions using co-transformation four cycles (15.04 ns) are required. For the MONARCH and DIVA FLP formats, the numbers of cycles involved are much higher than the new LNS, at least three cycles for addition and subtraction, and more than three cycles for multiplication and division operations.

In terms of silicon area, it can be observed from Figure 7-5 that the area of the new LNS, including that of the multiplicative operators, has been reduced from 915,457 μm^2 to 599,871 μm^2, or 65% of the ELM design. In addition, the area of the new LNS is also slightly smaller than that of the MONARCH and only 24% larger than the area in the DIVA.

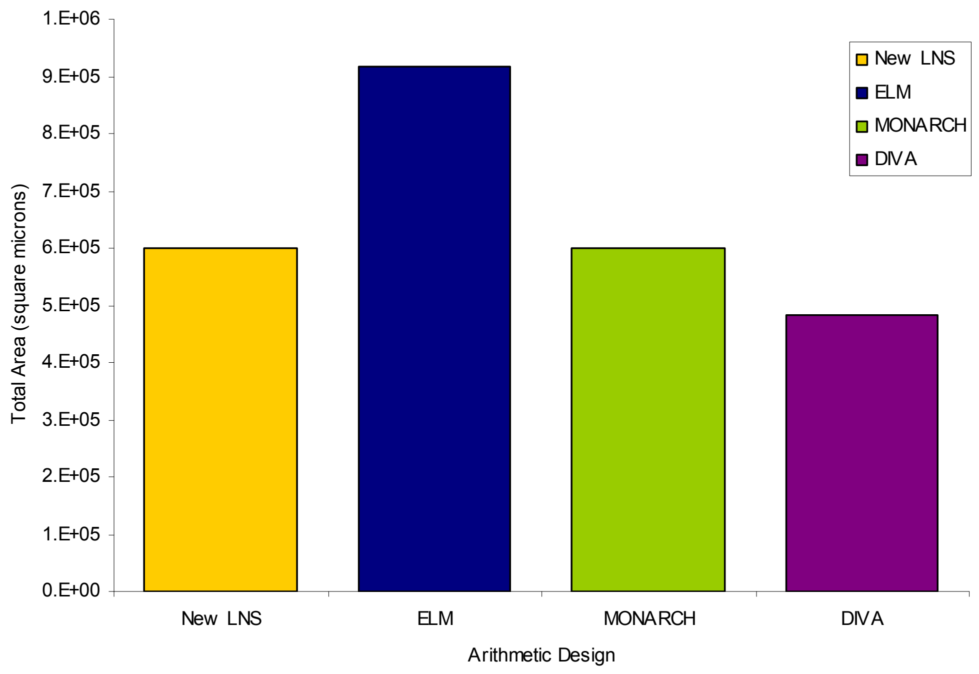


Figure 7-5 : Silicon areas (μm^2) in 32-bit arithmetic implementations.

Judging by this comparative analysis, the new LNS has been shown to be capable of executing addition and direct subtraction with less delay than the ELM and the other two FLP implementations. Much faster speeds have also been achieved when performing multiplication and division using the new LNS arithmetic in comparison with the MONARCH and DIVA. However, there is a slight increase in the delay of co-transformed subtractions when compared with the ELM. Nevertheless, less silicon area is consumed in the new LNS when compared with the ELM and MONARCH.

7.6. Summary

This chapter has described the hardware implementation of the new LNS based on a 32-bit system in detail. When synthesising the new LNS arithmetic in 0.18 µm technology, the critical delay path in computing addition and direct subtraction took 7.10 ns and only 1.16 ns for multiplication and division. In the event that co-transformation was required for subtraction, the worst-case delay was 14.79 ns. The total area for the complete LNS architecture was 599,871 μm^2.

In a controlled comparison with the previously published ELM design, the total delay in the new LNS system represented a reduction to 60% when executing addition and direct subtraction. A slight increase in delay occurred in co-transformed subtractions, by 12% of the delay in the ELM. In terms of silicon area, the implementation of the new LNS has been shown to be more cost effective, at 65% of the total area consumed in the ELM. The new area-delay product is 39% of its previous value in the ELM.

When compared with the faster of the two FLP units, the MONARCH design, the proposed LNS addition and subtraction can be performed in 63% of the time taken in FLP. For multiplication and division, the delays in the new LNS system were only 10% and 3% respectively of those in the MONARCH. The new LNS unit has also been built with fractionally less silicon than MONARCH, a reduction from 600,000 μm^2 to 599,871 μm^2.

CHAPTER 8

8. Implementation with Long Word-length Number

8.1. Introduction

Throughout this thesis so far, the discussion of the LNS system has only been concerned with a 32-bit architecture. This is a direct consequence of the objective of the research to investigate a direct alternative to the IEEE single-precision FLP arithmetic unit, a standard 32-bit number system. However, for applications where longer precision is required so as to increase the accuracy of the system, the 32-bit LNS may need to be extended. It is known that if longer word-lengths are applied, significant increases in the number and size of tables may be required. Nevertheless, to date, there has been a lack of analysis in long format numbers, except by Chen et al. [91].

Therefore, this chapter describes a longer word-length LNS design in a 40-bit format. In order to reduce the sizes of lookup tables, particularly for the co-transformation procedure, a third-order arrangement is introduced before the final results are computed using one of the interpolation techniques illustrated in Chapter 6. The LNS design is also synthesised and analysed in terms of area and critical path delay, and a comparative analysis is performed against the standard 32-bit LNS design suggested earlier.

In order to select either a long or short format representation, as indicated by Chester in [84], there is no criterion specified. This allows a designer freely to select and customise the number system according to the specific application. The format considered here has a 10-bit integer and 29-bit fraction.

8.2. The LNS System in a 40-bit Format

Inspired by the suggested 32-bit LNS architecture as illustrated in previous chapters, the building block for 40-bit LNS addition and subtraction again consists of a combination of the co-transformation procedure and interpolation process. For multiplication and division, the same adder module, using the CLA/CSLA architecture, is applied although this time the input bits need to be extended to suit operand size. Throughout the analysis, six guard bits are inserted in the system so as to maintain accuracy within the FLP limit.

Theoretically, by directly implementing a second-order co-transformation procedure in this 40-bit format to compute subtractions in the region $-1 < r < 0$, the fractional bits should be optimally partitioned into 9, 10 and 10 bits for the high, middle and low fields. Two lookup tables of 1024 words and one of 512 words of lookup tables are required during the co-transformation process. With only 7% of additive operators being subtractions with $r > -1$ [77], it seems impractical to implement such a large proportion of tables, approximately 2560 words, for only a small number of operations. Hence, third-order co-transformation is proposed to cater for the issue in the long format number, and the details of this are described in the next section.

8.2.1. Third-order Co-transformation Procedure for LNS Subtraction

The third-order co-transformation concept applies a similar approach as the second-order format detailed in Section 5.3.1, applying coefficient *k1* recursively.

$$\begin{aligned} 2^i - 2^j &= (2^i - 2^{j+k1}) - ((2^j - 2^{j+k1+k11}) - (2^{j+k1} - 2^{j+k1+k11})) \\ &= (2^i - 2^{j+k1}) - ((2^j - 2^{j+k1+k11}) - (2^{j+k1} - 2^{j+k1+k11+k111}) - 2^{j+k1+k11+k112}) \qquad (8.1) \end{aligned}$$

where,

$$2^{k111} + 2^{k112} = 1, \text{ i.e. } k112 = \log_2 (1 - 2^{k111}) \quad (8.2)$$

The block diagram of the suggested third-order approach is shown in Figure 8-1 together with the conceptual arrangement of bit partitioning in Figure 8-2. Conceptually analogous to the scheme presented in the second-order format, initially, index $r1$ is looked up from F1 table, containing $F(r)$ for $-1 < r < -\Delta 1$, where its value is guaranteed to fall on the nearest modulo-$\Delta 1$ based on the calculation of coefficient $k1$. Then, index $r11$ which falls on the modulo-$\Delta 11$ is approximated from the coefficient $k11$, and the resulting value of $F(r11)$ is stored in the F11 table which contains $F(r)$ for $-\Delta 1 < r < -\Delta 11$. With a similar number of bits as $\Delta 11$, the coefficient $k111$ is selected such that $r111$ falls on the modulo-$\Delta 111$, and the value of its function, $F(r111)$, is obtained from the F111 table which contains $F(r)$ for $-\Delta 11 < r < -\Delta 111$. The final coefficient, $k112$, however, is directly retrieved from the F112 table indexed by $k111$, and it also occupies exactly the same number of bits as represented in $\Delta 111$.

Variables $r1$, $k1$, $k11$, $r11$, $r111$ and $k111$ are:

$$r1 = ((j - i) \text{ DIV } \Delta 1) - 1) \times \Delta 1 = j + k1 - i \quad (8.3)$$

$$k1 = -(((j - i) \text{ MOD } \Delta 1) + \Delta 1) = i - j + r1 \quad (8.4)$$

$$k11 = ((j - i) \text{ MOD } \Delta 11) = r11 - k1 \quad (8.5)$$

$$r11 = -(((j - i) \text{ MOD } \Delta 1) + \Delta 1) + ((j - i) \text{ MOD } \Delta 11) = k1 + k11 \quad (8.6)$$

$$r111 = ((j - i) \text{ MOD } \Delta 11) + (-((j - i) \text{ MOD } \Delta 111)) = k11 + k111 \quad (8.7)$$

$$k111 = -((j - i) \text{ MOD } \Delta 111) = k11 - r111 \quad (8.8)$$

For ease of subsequent explanation, equation (8.1) is numbered as follows:

$$2^i - 2^j = \underbrace{\underbrace{(2^i - 2^{j+k1})}_{1} - (\underbrace{\underbrace{(2^j - 2^{j+k1+k11})}_{11} - \underbrace{(\underbrace{2^{j+k1} - 2^{j+k1+k11+k111}}_{111}) - 2^{j+k1+k11+k112})}_{112}}_{12}}_{2}$$

Theoretically, subtractions 1, 11 and 111 are completed instantaneously without the need for lookup tables. In contrast, subtraction 112 is written as:

$$\begin{aligned} r112 &= k11 + k112 - F(k11 + k111) \\ &= k11 + F(k111) - F(k11 + k111) \\ &= k11 + \log_2((1 - 2^{k111}) \div (1 - 2^{k11 + k111})) \qquad (8.9) \end{aligned}$$

This equation is identical to equation (5.12) for the second-order procedure, although this time with a different set of coefficients and therefore further analysis of this function is unnecessary. It is expected that the function will have similar characteristics as those discussed in Section 5.3.1. Subtraction 12 generates an index:

$$\begin{aligned} r12 &= k1 + F(k11 + k111) + F(r112) - F(k1 + k11) \\ &= k1 + F(k11 + k111) + F(k11 + k112 - F(k11 + k111)) - F(k1 + k11) \quad (8.10) \end{aligned}$$

Figure 8-3 depicts the value of *r12* when *r* is in the region $-2\Delta 1 < r < -\Delta 1$, and the same pattern occurs for each $\Delta 1$ in every subinterval. In this illustration, the fractional part has been partitioned into low, middle1, middle2 and high-order fields of 5, 5, 5 and 14 bits respectively. It should be noted that in the repeated cases of

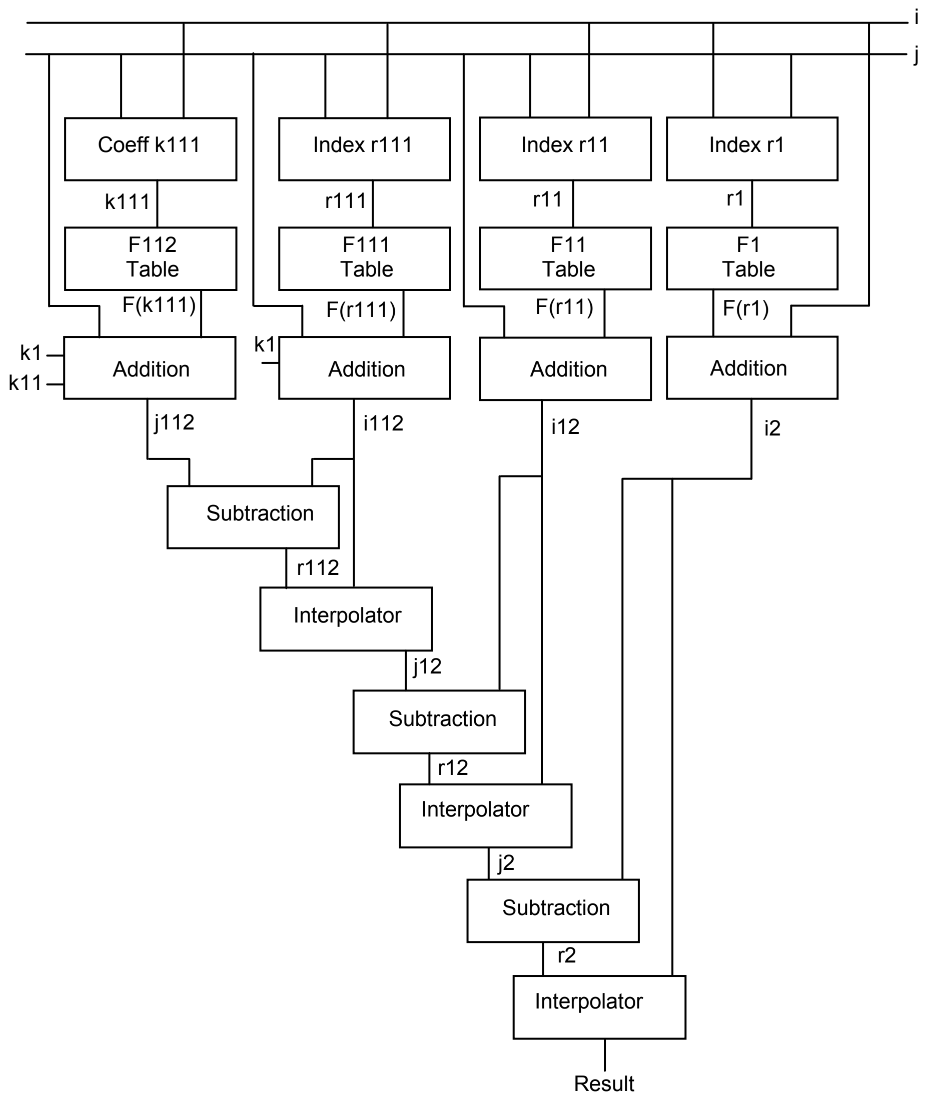


Figure 8-1 : Conceptual arrangement of the third-order co-transformation concept.

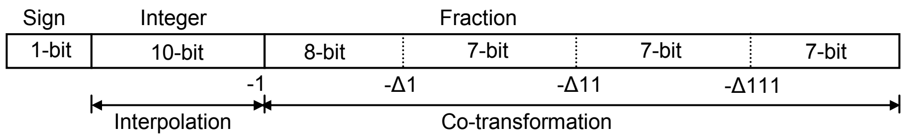


Figure 8-2 : Bit partitioning scheme of the third-order format.

k11 = 0 and *k111* = Δ*111*, *r12* has a positive value. At this stage, the computation of *F*(*r12*) is always zero and therefore the points are omitted from the graph. Further description is now needed of the behaviour of *r12* as *r* varies across the range of Δ*1*. First, consider the points at the left of the graph. At this leftmost subinterval, *k1* < Δ*11*, and *k1* + *k11* = Δ*11*. As it moves towards to the left of this subinterval, *k1* ≈ *k111*, and since *k1* is small, *r12* ≈ 0. However, in this particular subinterval, the value of the middle2 field is zero. Hence, the execution of the third-order format can be performed in the same way as presented for the second-order approach, where variable *k111*, *r11*, table F112 and F11 are analogous to the second-order *k11*, *r11*, table F12 and F11. Then, when in the subinterval of 2Δ*11*, *r12* is approximately -1, since *k1* and *k11* are now small enough and thus linear in behaviour. At this point, *r12* is completed in the second interpolator before the *j2* value is generated.

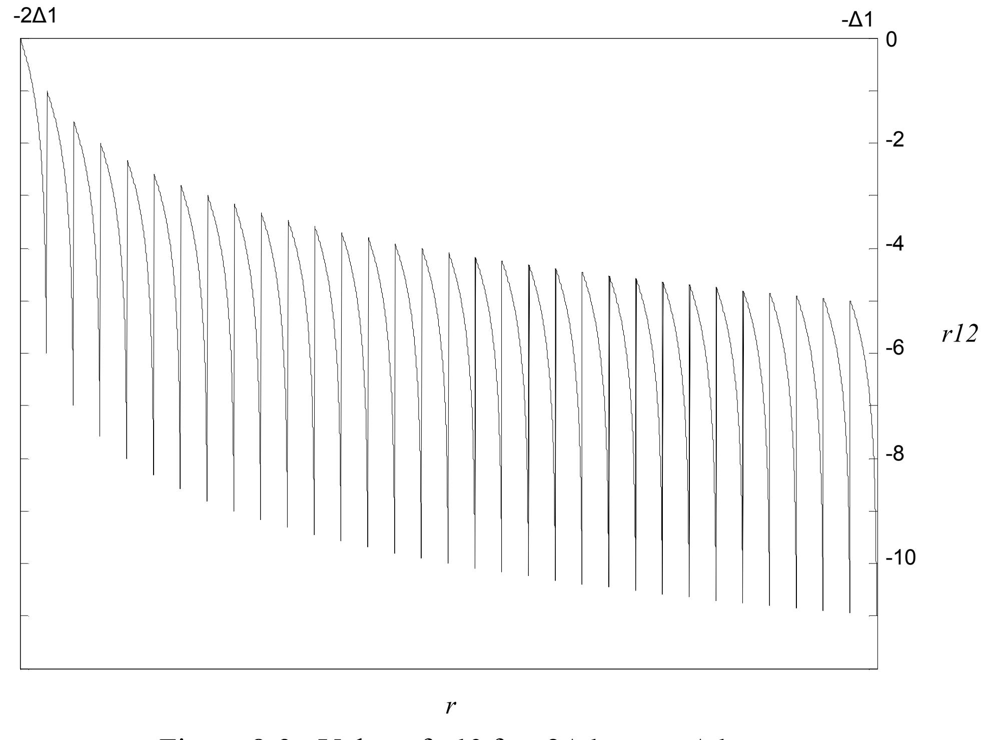


Figure 8-3 : Value of *r12* for -2Δ*1* < *r* < -Δ*1*.

In completing the co-transformation process, subtraction 12 is then subtracted with subtraction 1 before *r2* is produced as follows:

$$r2 = j - i + F(k1 + k11) + F(k1 + F(k11 + k111) + F(k11 + k112 - F(k11 + k111)) - F(k1 + k11)) - F(r1) \quad (8.11)$$

Once again, equation (8.11) is comparable with equation (5.13) in the second-order. Therefore, at all points, *r2* is definitely lower than -1, and can be accomplished through the third stage of the interpolator as positioned in Figure 8-1.

Depending on the operands of *i* and *j*, the value of *r2* typically falls in four different regions in the second-order arrangement. On the contrary, as a result of adding another table in the third-order concept, five regions have to be considered. In the first region, when $j - i \leq -1$, the value of *r2* is located in the linear region of *F*, so that *F*(*r*) can be executed directly using interpolation. Whilst in the region $-1 < j - i < -\Delta 1$, *r2* is computed based on the description mentioned above, which at the end always produces a maximum value of < -1. Consequently, *F*(*r*) can also be performed using interpolation. For $-\Delta 1 \leq j - i < -\Delta 11$, the high-order field is occupied with zero bits, and hence *F*(*r*) is completed in the same manner as in the second-order format. In the fourth region, $-\Delta 11 \leq j - i < -\Delta 111$, both high-order and middle1 bits are zero. *F*(*r*) is now accomplished using the first-order technique. Finally, *F*(*r*) is derived instantly from the F112 table when in the region $-\Delta 111 \leq j - i < 0$.

8.2.2. Interpolation

In terms of the interpolation procedure, at first two different formats, Lagrange and improved Lagrange as discussed in Chapter 6, have been adopted in the architecture. From there, the most suitable approach that can produce an optimal size of lookup tables is elected. These two interpolation techniques were chosen because both can produce less total storage than Taylor and can be implemented in hardware more easily than the modified Lagrange approach. Throughout the analysis, the implementation of the interpolation process is based on a non-linear scheme incorporating the error correction algorithm as in the ELM. The simulator design described in Section 3.3.1 has been applied in order to execute the error simulation of addition and subtraction functions. The results for worst-case errors in Lagrange and improved Lagrange formats are summarised in Tables 8-1 and 8-2. The row with the grey background in each table indicates the most suitable combination for implementation. By observation alone, the Lagrange approach is able to generate the most optimal size of lookup table and therefore is chosen to be implemented in the LNS addition and subtraction architectures.

Table 8-1 : Error of Lagrange interpolation.

Parameters			ADD		SUB		Worst Case
F,D Sizes	E size	P size	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
1,024	1,024	512	-0.4956	+0.5049	-0.7566	+0.7896	0.7896
2,048	2,048	512	-0.3830	+0.3928	-0.4696	+0.4764	0.4764
1,024	1,024	1,024	-0.4206	+0.4290	-0.5554	+05696	0.5696
2,048	2,048	1,024	-0.3643	+0.3751	-0.4007	+0.4322	0.4322
1,024	1,024	2,048	-0.3821	+0.3935	-0.4681	+0.4945	0.4945
2,048	2,048	2,048	-0.3573	+0.3656	-0.3769	+0.4329	0.4329

Table 8-2 : Error of improved Lagrange interpolation.

Parameters			ADD		SUB		Worst Case
F,D Sizes	E size	P size	$e'_{min\ rel}$	$e'_{max\ rel}$	$e'_{min\ rel}$	$e'_{max\ rel}$	e_{rel}
1,024	512	512	-0.4989	+0.5049	-0.7562	+0.7896	0.7896
2,048	1,024	512	-0.3844	+0.3928	-0.4696	+0.4764	0.4764
1,024	512	1,024	-0.4230	+0.4290	-0.5554	+0.5716	0.5716
2,048	1,024	1,024	-0.3656	+0.3751	-0.4007	+0.4322	0.4322
1,024	512	2,048	-0.3853	+0.3932	-0.4494	+0.5038	0.5038
2,048	1,024	2,048	-0.3582	+0.3656	-0.3747	+0.4319	0.4319

8.2.3. Design Summary

According to the analysis in Section 8.2.2, the combination of third-order co-transformation with the Lagrange interpolation procedure generates less total storage area. When implementing the third-order format, the tables involved in the co-transformation process are now segregated into four partitions. With this arrangement, the sizes of storage for F1, F11, F111 and F112 are therefore only 256, 128, 128 and 128 words respectively. Since the Lagrange approach is chosen as the best interpolation concept, the optimal sizes of the F, D and E tables that can produce the worst-case error approximately equivalent to FLP limit are 1024 words, with the P table at 2048 words. As summarised in Table 8-3, in total about 982 kbits would be required to compute LNS addition and subtraction in a 40-bit number system.

Table 8-3 : Total storage for the LNS 40-bit format.

Table	Words	Word length	Segments	Total Bits
F Add	1024	35-bit	6	215,040
F Sub	1024	35-bit	5	179,200
D Add	1024	33-bit	6	202,752
D Sub	1024	34-bit	5	174,080
E Add	1024	9-bit	6	55,296
E Sub	1024	12-bit	5	61,440
P	2048	34-bit	1	69,632
F1	256	36-bit	1	9,216
F11	128	38-bit	1	4,864
F111	128	39-bit	1	4,992
F112	128	39-bit	1	4,992
Total				981,504

8.3. Design Implementation

The hardware implementation of the LNS multiplication and division unit in a 40-bit format is identical with the design for a 32-bit system illustrated in Section 7.2.1, except for input size. However, due to applying the third-order co-transformation procedure, the LNS addition and subtraction module has a small modification in comparison with the 32-bit architecture. Another lookup table has been inserted in the co-transformation module to accommodate the four segmentations in fractional bits as suggested in Section 8.2.1. In addition, two more FXP adders are also needed before the third-order co-transformation process can be completed. Although the interpolator unit uses the Lagrange format, a similar arrangement in architecture as that proposed in the 32-bit design can still be implemented. In order to minimise the worst-case delay in the system, the CLA/CSLA and Booth-Wallace tree algorithm are adopted to perform FXP addition and multiplication operations respectively. The practical implementation of the LNS addition and subtraction unit in a 40-bit format is described in Figure 8-4.

8.3.1. Synthesis Results

The LNS arithmetic unit based on a 40-bit number system was synthesised using the constrained Faraday 0.18 μm CMOS technology, and the results are reported in Table 8-4. It shows that the worst-case delay for addition and direct subtraction is 7.71 ns. On the other hand, for subtraction using the co-transformation procedure, the delay sharply increases to roughly three times slower than that of direct subtraction due to the requirement to re-use the interpolator three times. In the case of multiplication and division, the functions can be completed in only 1.27 ns. Based on this 40-bit LNS design, the total area required is 1,542,976 μm^2.

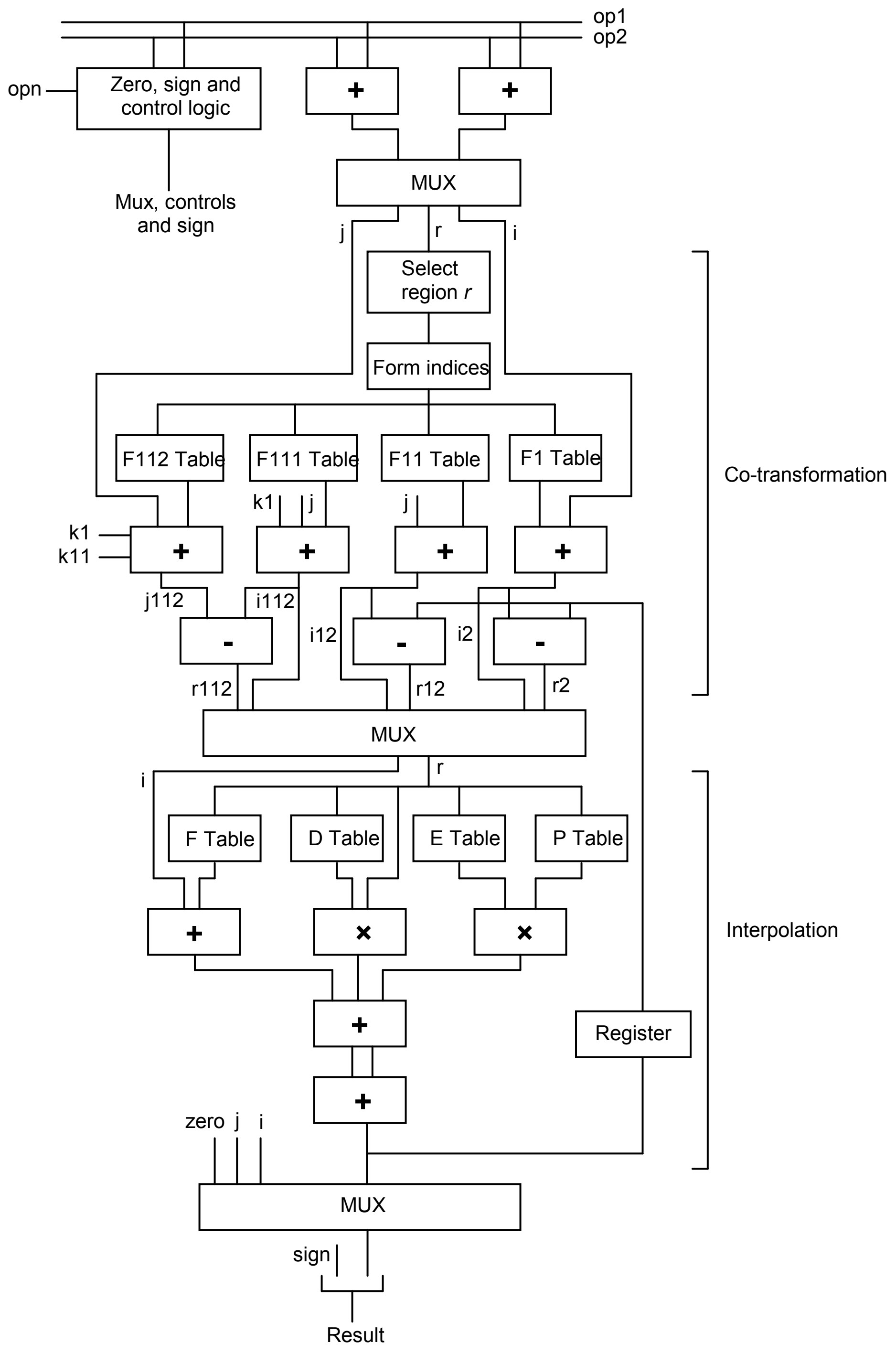


Figure 8-4 : The hardware implementation of the LNS addition and subtraction in a 40-bit format.

Table 8-4 : Delay times and total device area of a 40-bit LNS arithmetic unit.

Function	40-bit LNS Arithmetic Unit	
	Delay (ns)	Area (μm^2)
Add / Sub	7.71	1,528,956
Sub (Co-transform)	22.28	
Mul / Div	1.27	14,020

8.4. Performance Analysis

In order to evaluate the impact on overall performance of increasing the fractional bits in the LNS system, the results presented in Table 8-4 are compared with the synthesis results produced from a 32-bit LNS system as described in Table 7-1. As graphically displayed in Figure 8-5, the delay of an addition or direct subtraction has been increased by 9% of a 32-bit LNS design. Similarly, an increase by 9% is observed for multiplication and division operations. When subtraction requires co-transformation, the delay in the 40-bit design increases to 22.28 ns from 14.79 ns, i.e by 50% of the delay in a 32-bit LNS. This is mainly because three stages of interpolation are involved.

The implementation of the third-order co-transformation concept in a 40-bit LNS system that utilises three sets of 128 words and one of 256 words appears to have equivalent total sizes of co-transformation tables as needed for the 32-bit design. However, the requirements of 1024 words for each F, D and E table as well as a 2048-word P table during the interpolation process greatly influences the total area of the 40-bit LNS design. This can be clearly seen in Figure 8-6 where the total silicon area of a 40-bit number system has increased more than two fold over the area generated in a 32-bit architecture. It is estimated that the area-delay product of a 40-bit LNS is 0.0119 μm^2 sec, whereas it is 0.0042 μm^2 sec in a 32-bit LNS.

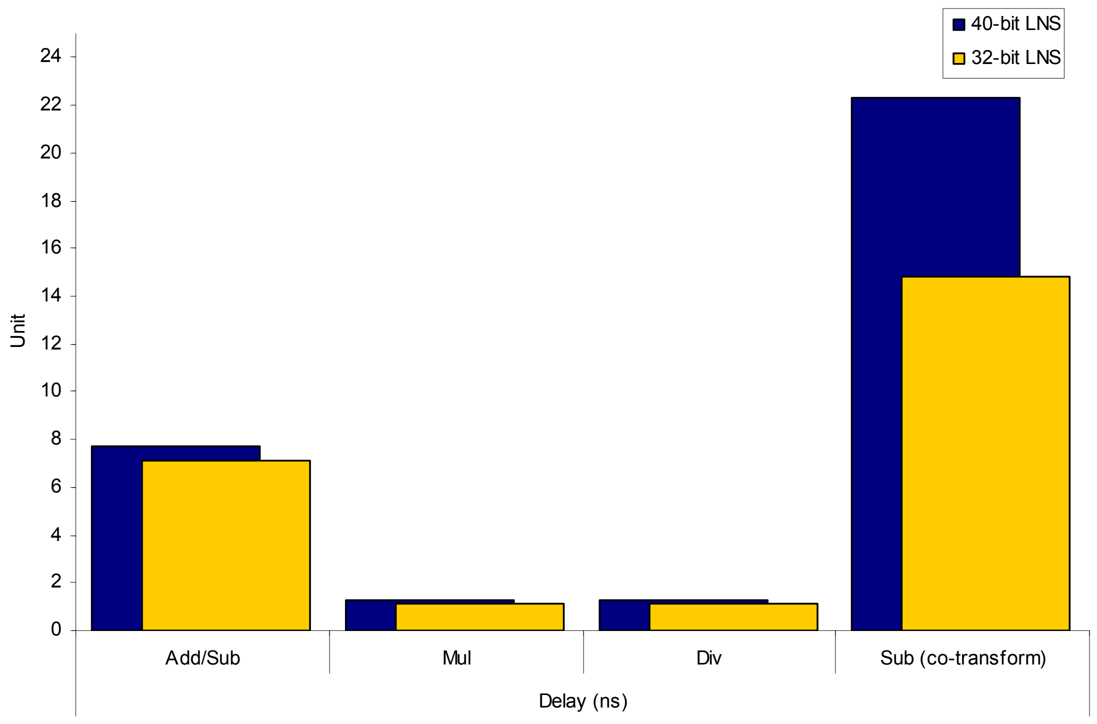


Figure 8-5 : Delays of a 32-bit and 40-bit LNS designs.

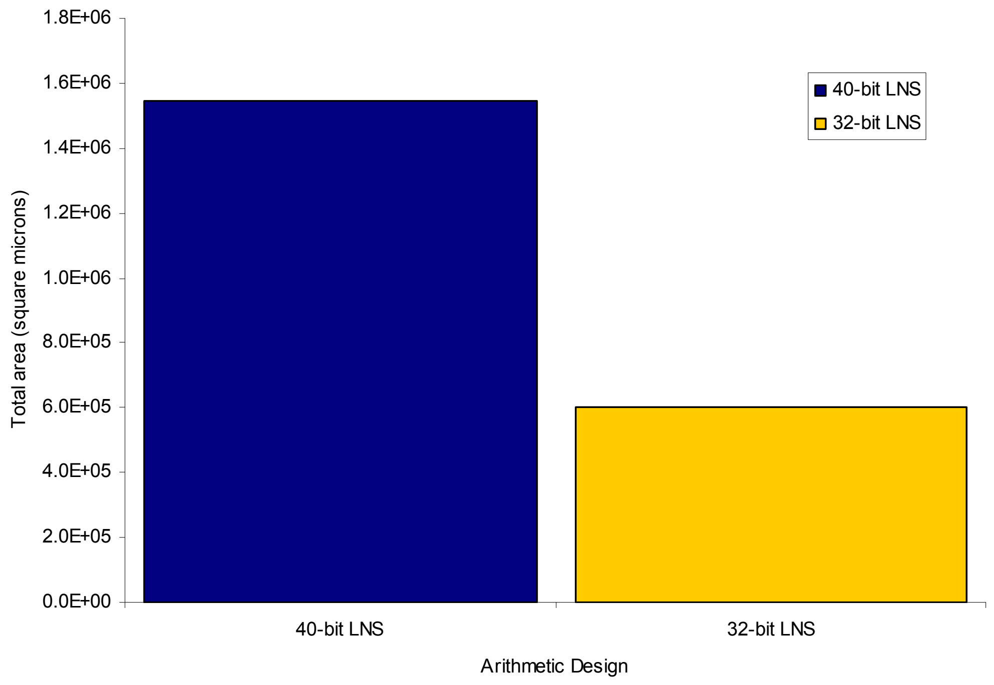


Figure 8-6 : Silicon areas in 32-bit and 40-bit LNS.

Even with only a minimal degradation in terms of speed, the total silicon area of the 40-bit LNS seems to be unwieldy in comparison with the 32-bit LNS. Thus, future work needs to concentrate on refining the interpolator module as to gradually reduce the total lookup tables for the long format number system.

8.5. Summary

A long word-length version of the LNS system has been designed and described in detail in this chapter. This 40-bit LNS format was segmented into 10-bit integer and 29-bit fraction and the third-order co-transformation concept was introduced to substantially reduce the total co-transformation tables to only 640 words, approximately equivalent to those presented for the suggested 32-bit system. From the analytical study, the best interpolation technique to be implemented was the Lagrange approach which employed 1024 words for the F, D and E tables and a 2048-word P table.

The delay of the 40-bit LNS design was increased to 109% of the 32-bit arrangement when executing addition and direct subtraction operations, and 150% during subtractions with co-transformation. For multiplication and division, an increase to 109% of a 32-bit system was reported. The estimated silicon area of the 40-bit LNS was roughly three times larger than that occupied in the 32-bit architecture.

CHAPTER 9

9. Conclusions and Recommendations

9.1. Conclusions of the Study

The primary objective of this thesis has been to present a new design approach for a high speed and reduced area 32-bit LNS arithmetic unit, and to show through design and simulation that the technique introduced is extremely competitive with commonly used FLP systems and better than the leading published LNS architecture.

According to the literature review, the main bottleneck in the LNS system arises from the complexity of executing addition and subtraction, particularly subtraction near the singularity region, which results in using large lookup tables. However, the LNS architecture proposed in the ELM design has been shown to be able to minimise storage requirements whilst computing addition and subtraction. Furthermore, when comparing delays of the ELM with those of an FLP device, addition and direct subtraction operations can be performed marginally better, at 90% of the corresponding FLP times. Although co-transformed subtractions were 120%, yet multiplication only required 30% of the FLP delays.

A new development of the co-transformation procedure presented in this thesis has vastly reduced the total storage requirements to 73% of the previously published ELM design. However, this in turn has a huge impact in terms of the critical path delay for subtractions using co-transformation due to the requirement to re-use the interpolator. It seems likely, therefore, that the new co-transformation will only be feasible in conjunction with an improved interpolator.

Hence, a smaller modification to the interpolator has been proposed. Combining the new co-transformation method with this improved interpolator has now reduced the total storage to 51% of that previous ELM implementation. With this new arrangement, it enabled a fully synthesised solution. A controlled comparison with the previous ELM design indicated a reduction to 60% of the delay and 65% of the silicon area. In addition, comparing it with the faster of the two independently designed FLP units has shown that LNS addition and direct subtraction can be performed in 63% of the FLP time. Multiplication completes with 10% and division 3% of the FLP delays. This new LNS design has also been built with fractionally less silicon, and worst-case accuracy is better than that of FLP arithmetic.

The present findings conclusively demonstrate that the new LNS system is now able to offer advantages in speed and accuracy over the FLP method. Moreover, it can be implemented at an equal cost in silicon. Furthermore, the performance of the new LNS is also found to be substantially better than the leading published LNS design.

9.2. Future Extensions

The new development of the LNS arithmetic unit has been fully designed and synthesised. Based on the results, various follow-on activities could be conducted in the future.

As presented in Chapter 6, a simple improvement in performing the interpolation process has been shown. However, although the total bits in the new LNS design can be reduced to 51% of the previous ELM, two FXP multipliers are still needed in the interpolator module.

In the current design described in Chapter 7, the FXP multiplication process has been completed using the traditional method of Booth and Wallace tree algorithms. Nevertheless, in order to further increase the speed of the LNS system, especially for addition and direct subtraction operations, many other multiplication techniques might be applied. For instance, a simple high speed multiplier design has been suggested in [92] in which the last additional partial product row can be avoided by utilising a fast method to find two's complement numbers. Besides that, combining the Booth recoded approach with the Dadda multiplier concept could possibly produce even better performance than using the Wallace tree method [93].

Therefore, it would be worth considering various combinations of algorithms and architectures to reduce delays in the multiplier design.

In this thesis, a brief proposal for long-format LNS as illustrated in Chapter 8 has been described. The new third-order co-transformation procedure has been proposed which can substantially reduce the total co-transformation tables when long precision numbers are involved. However, the design still suffers from a vast increase in delay when computing subtractions using co-transformation due to the requirement to pass through the interpolator three times. It has been shown that using the existing interpolation approach as described in Chapter 6 might not reduce the delay. Therefore, it is hoped that future researcher may suggest a technique which can improve further the interpolation process, or even better propose a new method to replace the interpolator module to deliver the faster speed still.

References

[1] Texas Instruments Inc., "High-Speed DSP Systems Design," *Reference Guide,* SPRU889-May 2005.

[2] H. L. Garner, "A Survey of Some Recent Contributions to Computer Arithmetic," *IEEE Transactions on Computers,* vol. C-25, pp. 1277-1282, 1976.

[3] IEEE Computer Society, "IEEE Standard 754 for Binary Floating-Point Arithmetic," 1985.

[4] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Transactions on Computers,* pp. 190-200, 1988.

[5] L. K. Yu and D. M. Lewis, "A 30-b Integrated Logarithmic Number System Processor," *IEEE Journal of Solid-State Circuits,* vol. 26, pp. 1433-1440, 1991.

[6] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Transactions on Computers,* pp. 702-715, 2000.

[7] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The European Logarithmic Microprocessor," *IEEE Transactions on Computers,* pp. 532-546, 2008.

[8] F. Haohuan, O. Mencer, and W. Luk, "FPGA Designs with Optimized Logarithmic Arithmetic," *IEEE Transactions on Computers,* vol. 59, pp. 1000-1006, 2010.

[9] D. M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System," *IEEE Transactions on Computers,* pp. 1325-1336, 1990.

[10] Mark G. Arnold, Thomas A. Bailey, John R. Cowles, and M. D. Winkel, "Arithmetic Co-Transformations in the Real and Complex Logarithmic Number Systems," *IEEE Transactions on Computers,* vol. 47, pp. 777-786, 1998.

[11] M. G. Arnold, "Improved Co-Transformation for LNS Subtraction," *IEEE International Symposium on Circuits and Systems,* vol. II, pp. 752-755, 2002.

[12] John L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, USA: Morgan Kaufmann, 2007.

[13] B. Wilkinson, *Computer Architecture Performance*. London, UK: Prentice Hall, 1996.

[14] Robert J. Baron and L. Higbie, *Computer Architecture*. New York, USA: Addison-Wesley Publishing Company, 1994.

[15] Dharma P. Agrawal and T. R. N. Rao, "Introduction: Computer Arithmetic," *IEEE Transactions on Computers,* vol. C-32, pp. 329-330, 1983.

[16] R. Zimmermann, "Binary Adder Architectures for Cell-Based VLSI and their Synthesis," in *PhD Thesis*: Swiss Federal Institute of Technology Zurich, 1997.

[17] G. W. Bewick, "Fast Multiplication: Algorithms and Implementation," in *PhD Thesis, Department of Electrical Engineering*: Stanford University, 1994.

[18] Nhon T. Quach and M. J. Flynn, "An Improved Algorithm for High-Speed Floating-Point Addition," Technical Report: CSL-TR-90-442, Stanford University 1990.

[19] N. Byeong-Gyu, K. Hyejung, and Y. Hoi-Jun, "A Low-Power Unified Arithmetic Unit for Programmable Handheld 3-D Graphics Systems," *IEEE Custom Integrated Circuits Conference,* pp. 535-538, 2006.

[20] I. Kouretas, C. Basetas, and V. Paliouras, "Low-power Logarithmic Number System Addition/Subtraction and their Impact on Digital Filters," *IEEE International Symposium on Circuits and Systems,* pp. 692-695, 2008.

[21] J. R. Sacha, "Arithmetic System for Low Power Signal Processing," in *Phd Thesis, Department of Computer Science and Engineering*: The Pennsylvania State University, 1998.

[22] I. Flores, *The Logic of Computer Arithmetic*. New York, USA: Prentice Hall Inc., 1963.

[23] J. B. Gosling, *Design of Arithmetic Units for Digital Computers*. London: The Macmillan Press Ltd, 1980.

[24] M. Darley, B. Kronlage, D. Bural, B. Churchill, D. Pulling, P. Wang, R. Iwamoto, and L. Yang, "The TMS390C602A Floating-Point Coprocessor for Sparc Systems," *IEEE Micro,* vol. 10, pp. 36-47, 1990.

[25] M. Atkins, "Performance and the i860 Microprocessor," *IEEE Micro,* vol. 11, pp. 24-27, 72-78, 1991.

[26] N. G. Kingsbury and P. J. W. Rayner, "Digital Filtering Using Logarithmic Arithmetic," *Electronics Letters,* pp. 56-58, 1971.

[27] Vouzis P.D., Collange S., and M.G Arnold, "Cotransformation Provides Area and Accuracy Improvement in an HDL Library for LNS Subtraction," *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools,* pp. 85-93, 2007.

[28] H. Fu, O. Mencer, and W. Luk, "Optimizing Logarithmic Arithmetic in FPGAs," *IEEE International Symposium on Field-Programmable Custom Computing Machines,* pp. 163-172, 2007.

[29] C. Inacio and D. Ombres, "The DSP Decision:Fixed Point or Floating?," *IEEE Spectrum,* vol. 33, pp. 72-74, 1996.

[30] H. Fu, O. Mencer, and W. Luk, "Optimizing Logarithmic Arithmetic on FPGAs " *IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 163-172, 2007.

[31] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York: Oxford University Press, 2000.

[32] J. Kontro, K. Kalliojarvi, and Y. Neuvo, "Floating-Point Arithmetic in Signal Processing," *IEEE International Symposium on Circuits and Systems,* vol. 4, pp. 1784-1791 vol.4, 1992.

[33] K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-Point Performance," *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays,* pp. 171-180, 2004.

[34] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units," *13th IEEE Symposium on Computer Arithmetic,* pp. 156-165, 1997.

[35] J. Lang, C. Zukowski, R. Lamaire, and C. H. An, "Integrated-Circuit Logarithmic Arithmetic Units," *IEEE Transactions on Computers,* vol. c-34, pp. 475-483, 1985.

[36] D. Golberg, "What Every Computer Scientist Should Know About Floating Point Arithmetic," *Computing Surveys,* 1991.

[37] Z. Leonelli, *Supplement Logarithmique,* Brossier, circa 1800.

[38] M. G. Arnold, T. A. Bailey, J. R. Cowles, and J. J. Cupal, "Redundant Logarithmic Number Systems," *Proceedings of the 9th Symposium on Computer Arithmetic,* pp. 144-151, 1989.

[39] Earl E. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Transactions on Computers,* pp. 1238-1242, 1975.

[40] E. E. Swartzlander, D. V. Satish Chandra, H. T. Nagle, Jr., and S. A. Starks, "Sign/Logarithm Arithmetic for FFT Implementation," *IEEE Transactions on Computers,* vol. C-32, pp. 526-534, 1983.

[41] A. D. Edgar and S. C. Lee, "FOCUS Microcomputer Number System," *Communications of the ACM,* vol. 22, pp. 166-177, 1979.

[42] M. G. Arnold, "Extending the Precision of the Sign Logarithm Number System," M.S. Thesis, University of Wyoming, Laramie, 1982.

[43] F. Taylor, "An Extended Precision Logarithmic Number System," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 31, pp. 232-234, 1983.

[44] M. G. Arnold, J. Cowles, and T. Bailey, "Improved Accuracy for Logarithmic Addition in DSP Applications," *International Conference on Acoustics, Speech, and Signal Processing, ,* pp. 1714-1717 vol.3, 1988.

[45] H. Henkel, "Improved Addition for the Logarithmic Number System," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 37, pp. 301-303, 1989.

[46] D. M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Transactions on Computers,* vol. 43, pp. 974-982, 1994.

[47] M. G. Arnold, "Design of a Faithful LNS Interpolator," *IEEE Euromicro Symposium on Digital Systems Design,* pp. 336-345, 2001.

[48] M. G. Arnold and M. D. Winkel, "A Single-Multiplier Quadratic Interpolator for LNS Arithmetic," *IEEE International Conference on Computer Design,* pp. 178-183, 2001.

[49] J. N. Coleman and E. I. Chester, "A 32-Bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point," *IEEE Symposium on Computer Arithmetic,* pp. 142-151, 1999.

[50] T. Stouraitis and F. J. Taylor, "Analysis of Logarithmic Number System Processor," *IEEE Transactions on Circuits and Systems,* vol. 35, pp. 519-527, 1988.

[51] D. Das Sarma and D. W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proceedings of the 12th Symposium on Computer Arithmetic,* pp. 17-28, 1995.

[52] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proceedings of the 12th Symposium on Computer Arithmetic,* pp. 10-16, 1995.

[53] M. J. Schulte and J. E. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation," *IEEE Symposium on Computer Arithmetic,* pp. 175-183, 1997.

[54] M. J. Schulte and J. E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Transactions on Computers, ,* vol. 48, pp. 842-847, 1999.

[55] M.J Schulte and J. E. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation," *Proceedings of the 13th Symposium on Computer Arithmetic,* pp. 175-183, 1997.

[56] F. de Dinechin and A. Tisserand, "Some Improvements on Multipartite Table Methods," *IEEE Symposium on Computer Arithmetic,* pp. 128-135, 2001.

[57] J. Detrey and F. de Dinechin, "A VHDL Library of LNS Operators," *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers,* vol. 2, pp. 2227-2231 Vol.2, 2003.

[58] J. Detrey and F. d. Dinechin, "A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic," *The Journal of VLSI Signal Processing,* pp. 161-175, 2007.

[59] J. N. Coleman, "Simplification of Table Structure in Logarithmic Arithmetic," *Electronic Letters,* vol. 31, pp. 1905-1906, 1995.

[60] F. Taylor, "A Hybrid Floating-point Logarithmic Number System Processor," *IEEE Transactions on Circuits and Systems,* vol. 32, pp. 92-95, 1985.

[61] F. S. Lai and C. F. E. Wu, "A Hybrid Number System Processor with Geometric and Complex Arithmetic Capabilities," *IEEE Transactions on Computers,* vol. 40, pp. 952-962, 1991.

[62] T. Stouraitis, "A Hybrid Floating-point/Logarithmic Number System Digital Signal Processor," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* pp. 1079-1082, 1989.

[63] M. G. Arnold, T. A. Bailey, J. R. Cowles, and J. J. Cupal, "Redundant Logarithmic Arithmetic," *IEEE Transactions on Computers,* vol. 39, pp. 1077-1086, 1990.

[64] J. M. Muller, A. Tisserand, and A. Scherbyna, "Semi-logarithmic Number Systems," *IEEE Symposium on Computer Arithmetic,* pp. 201-207, 1995.

[65] M. G. Arnold, "The Residue Logarithmic Number System: Theory and Implementation," *IEEE Symposium on Computer Arithmetic (ARITH),* pp. 196–205, 2005.

[66] W. Strauss, "Expectations Down for DSP Market Growth," Retrieved from http://www.fwdconcepts.com, Date Accessed 23rd February 2010.

[67] O. Vainio and Y. Neuvo, "Logarithmic arithmetic in FIR filters," *IEEE Transactions on Circuits and Systems,* vol. 33, pp. 826-828, 1986.

[68] D. Das, K. Mukhopadhyaya, and B. P. Sinha, "Implementation of Four Common Functions on an LNS Co-processor," *IEEE Transactions on Computers,* vol. 44, pp. 155-161, 1995.

[69] M. G. Arnold and C. Walter, "Unrestricted Faithful Rounding is Good Enough for Some LNS Applications," *IEEE Symposium on Computer Arithmetic,* pp. 237-246, 2001.

[70] M. G. Arnold, "Reduced power consumption for MPEG decoding with LNS," *IEEE International Conference on Application-Specific Systems, Architectures and Processors,* pp. 65-75, 2002.

[71] E. I. Chester and J. N. Coleman, "Matrix Engine for Signal Processing Applications using the Logarithmic Number System," *IEEE International Conference on Application-Specific Systems, Architectures and Processors,* pp. 315-324, 2002.

[72] R. E. Morley, Jr., G. L. Engel, T. J. Sullivan, and S. M. Natarajan, "VLSI Based Design of a Battery-operated Digital Hearing Aid," *International Conference on Acoustics, Speech, and Signal Processing,* pp. 2512-2515 vol.5, 1988.

[73] M. N. Marsono, M. W. El-Kharashi, and F. Gebali, "Binary LNS-based Naive Bayes Inference Engine for Spam Control: Noise Analysis and FPGA Implementation," *IET Computers & Digital Techniques* vol. 2, pp. 56-62, 2008.

[74] D. M. Lewis, "An Accurate LNS Arithmetic Unit using Interleaved Memory Function Interpolator," *IEEE Symposium on Computer Arithmetic,* pp. 2-9, 1993.

[75] B. R. Lee and N. Burgess, "A Parallel Look-up Logarithmic Number System Addition/Subtraction Scheme for FPGA," *IEEE International Conference on Field-Programmable Technology (FPT),* pp. 76-83, 2003.

[76] A. Tyagi, "A Reduced-Area Scheme for Carry-Select Adders," *IEEE Transactions on Computers,* vol. 42, pp. 1163-1170, 1993.

[77] R. Che Ismail and J. N. Coleman, "ROM-less LNS," *20th IEEE Symposium on Computer Arithmetic (ARITH),* pp. 43-51, 2011.

[78] Synopsys Inc., "Design Compiler User Guide," 2002.

[79] Jidan Al-Eryani, "32-bit Floating-Point Unit (FPU100)," Opencores.org (2006), Retrieved from http://www.opencores.org/projects.cgi/web/fpu100/overview, Date Accessed 20th May 2008.

[80] T.-J. Kwon, J. Sondeen, and J. Draper, "Design Trade-offs in Floating-point Unit Implementation for Embedded and Processing-in-memory Systems," *IEEE International Symposium on Circuits and Systems,* pp. 3331-3334, 2005.

[81] G.M Phillips and P. J. Taylor, *Theory and Applications of Numerical Analysis*. London, UK: Academic Press Limited, 1996.

[82] E. Meijering, "A Chronology of Interpolation: From Ancient Astronomy to Modern Signal and Image Processing," *Proceedings of the IEEE,* vol. 90, pp. 319-342, 2002.

[83] G. M. Phillips, *Interpolation and Approximation by Polynomials*. New York, USA: Springer, 2000.

[84] E. Chester, "The LNS and Its Application in a High-Performance Matrix Processor," in *PhD Thesis, School of Electrical, Electronic and Computer Engineering*: Newcastle University, 2002.

[85] Endre Suli and D. Mayers, *An Introduction to Numerical Analysis*. Cambridge, UK: Cambridge University Press, 2003.

[86] J. Pihl and E. J. Aas, "A Multiplier and Squarer Generator for High Performance DSP Applications," *IEEE 39th Midwest Symposium on Circuits and Systems,* vol. 1, pp. 109-112 vol.1, 18-21 Aug 1996 1996.

[87] K. J. Cho and J. G. Chung, "Parallel Squarer Design using Pre-calculated Sums of Partial Products," *Electronics Letters,* vol. 43, pp. 1414-1416, 2007.

[88] H. Morinaka, H. Makino, Y. Nakase, H. A. S. H. Suzuki, and K. A. M. K. Mashiko, "A 64 bit Carry Look-Ahead CMOS Adder using Modified Carry Select," *Proceedings of the IEEE Custom Integrated Circuits Conference,* pp. 585-588, 1995.

[89] Ohsang Kwon, Earl E. Swartzlander Jr., and K. Nowka, "A Fast Hybrid Carry-Lookahead/Carry-Select Adder Design," *Proceedings of the 11th Great Lakes symposium on VLSI,* pp. 149-152, 2001.

[90] T. K. Callaway and E. E. Swartzlander, Jr., "Power-delay Characteristics of CMOS Multipliers," *13th IEEE Symposium on Computer Arithmetic,* pp. 26-32, 1997.

[91] C. Chen, C. Rui-Lin, and Y. Chih-Huan, "Pipelined Computation of Very Large Word-length LNS Addition/Subtraction with Polynomial Hardware Cost," *IEEE Transactions on Computers,* vol. 49, pp. 716-726, 2000.

[92] J. Y. Kang and J. L. Gaudiot, "A Simple High-Speed Multiplier Design," *IEEE Transactions on Computers,* vol. 55, pp. 1253-1258, 2006.

[93] W. J. Townsend, E. E. Swartzlander, Jr., and J. A. Abraham, "A Comparison of Dadda and Wallace Multiplier Delays," *Proc. SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations XIII,* vol. 5205, pp. 552-560, 2003.

[94] J. Makino, T. Fukushige, M. Koga, K. Namura, "GRAPE-6 : The Massively-parallel Special-purpose Computer for Astrophysical Particle Simulations," *PASJ : Publ. Astron. Soc. Japan*, pp. 1-25, 2008.

A Appendices

A1. Authored and Co-authored Publications

During the course of this research, the following publications were written. The work of the co-authors and first authors where appropriate is acknowledged and appreciated. Internal departmental papers are not included in this list.

Journal Publication

- J.N Coleman and R.C Ismail, “Fast 32-bit Logarithmic Arithmetic”, submitted to *IEEE Transactions on Computers*.

Conference Publication

- R.C Ismail and J.N Coleman, “ROM-less LNS”, *20th IEEE Symposium on Computer Arithmetic (ARITH),* pp. 43-51, 2011.

A2. C Programming Language for 32-bit LNS Subtraction

The following C code is a simulator for 32-bit LNS subtraction unit using second-order co-transformation with the improved Lagrange interpolator.

```
//***************************************************//
//   32-BIT LOGARITHMIC SUBTRACTION WITH 6 GUARD BITS   //
//   (SECOND ORDER METHOD WITH IMPROVED INTERPOLATOR)   //
//***************************************************//

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <limits.h>
#include <stdint.h>
#include <inttypes.h>

#define maxcomp 9.2E18
#define g 64             //6 guard bits => 2^6 = 64;
#define f1 128           //7-bit of high field
#define f2 256           //8-bit of middle field
#define f3 16384         //8-bit of low field + 6 guard bits @ delta11
#define f2f3 4194304     //delta1
#define one 8388608.0
#define gone 536870912.0
#define step 1
#define p 512
#define f 128
#define fr8 128
#define fr16 16
#define m1 4194304      //(gone DIV f)
#define m2 8388608      //(gone DIV f)*2
#define m4 16777216     //(gone DIV f)*4
#define m8 33554432     //(gone DIV f)*8
#define m16 536870912   //(gone DIV f)*16

double ln2, log2e;

long long int arg1,arg2;
long long int arg;
long long int dividend, divisor;

long long int f1tab[128];
long long int f2tab[256];
long long int f3tab[256];
long long int fr1tab[256];
long long int dr1tab[256];
long long int er1tab[128];
long long int fr2tab[256];
long long int dr2tab[256];
long long int er2tab[128];
long long int fr4tab[256];
long long int dr4tab[256];
long long int er4tab[128];
long long int fr8tab[256];
long long int dr8tab[256];
long long int er8tab[128];
long long int fr16tab[64];
long long int dr16tab[64];
long long int er16tab[64];
long long int ptab[512];

double glog2 (double arg)
{
  return (log2e * log (arg) * (gone));
}

double gexp2 (double arg)
{
  return (exp (log (2) * (arg / (gone))));
}

void f1table (void)
{
  long long int i;
  long long int t;

  for (i = 1; i <= f1; i++)
    {
     t = -(i * f2f3);
     f1tab [i-1] = glog2 (fabs (1 - gexp2 (t)));
    }
  return;
}

void f2table (void)
{
  long long int i;
  long long int t;

    for (i = 1; i <= f2; i++)
    {
     t = -((i) * f3);
     f2tab [f2 - i] = glog2 (fabs (1 - gexp2 (t)));

    }

  return;
}

void f3table (void)
{
  long long int i;

  for (i = 0; i < f3/g; i++)
    {
      f3tab [i] = glog2 (fabs (1 - gexp2 (-i * g)));
    }
  return;
}

void fr1table (void)
{
  long long int i,j;
  long long int t,t2;

  for (i = 0; i < f; i++)
    {
      t = (-i * m1) - (gone);
      fr1tab[i] = glog2(fabs (1 - gexp2(t)));
    }

  for (j = 0; j < f; j++)//g1*16 = 65536
    {
      t2 = (-j * (m1)) - (gone) - (m1/2);
      fr1tab[j+f] = glog2(fabs (1 - gexp2(t2)));
    }

  return;
}

void dr1table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3;

  for (i = 0; i < f; i++)
    {
      t0 = -i * m1 - (gone);
      t1 = t0 - (m1/2);
      dr1tab[i] = (((glog2(fabs (1 - gexp2(t1)))) - fr1tab[i]) / (-m1/2))*-
                  gone;
    }

  for (j = 0; j < f; j++)
    {
      t2 = (-j * (m1)) - (gone) - (m1/2) - 1;
      t3 = t2 - (m1/2);
      dr1tab[j+f] = (((glog2(fabs (1 - gexp2(t3)))) - fr1tab[j+f]) / (-
                    m1/2))*-gone;
    }

  return;
}

void er1table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3,t4,t5;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m1)) - (gone);
      t1 = t0 - (m1/2);
      t2 = t0 + (-m1/4);
      er1tab[i] = round(-((fr1tab[i] + (((-m1/4) * -dr1tab[i])/gone)) -
                  (glog2(fabs (1 - gexp2(t2))))));
    }

  return;
}

void fr2table (void)
{
  long long int i,j;
  long long int t1,t2;

  for (i = 0; i < f; i++)
    {
      t1 = (-i * (m2)) - (gone * 2);
      fr2tab[i] = glog2(fabs (1 - gexp2(t1)));
    }
  for (j = 0; j < f; j++)
    {
      t2 = (-j * (m2)) - (gone * 2) - (m2/2);
      fr2tab[j+f] = glog2(fabs (1 - gexp2(t2)));
    }

  return;
}

void dr2table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m2)) - (gone * 2);
      t1 = t0 - (m2/2);
      dr2tab[i] = (((glog2(fabs (1 - gexp2(t1)))) - fr2tab[i]) / (-m2/2))*-
                  gone;
    }

  for (j = 0; j < f; j++)
    {
      t2 = (-j * (m2)) - (gone * 2) - (m2/2) - 1;
      t3 = t2 - (m2/2);
      dr2tab[j+f] = (((glog2(fabs (1 - gexp2(t3)))) - fr2tab[j+f]) / (-
                    m2/2))*-gone;
    }

  return;
}

void er2table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3,t4,t5;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m2)) - (gone * 2);
      t1 = t0 - (m2/2);
      t2 = t0 + (-m2/4);
      er2tab[i] = round(-((fr2tab[i] + (((-m2/4) * -dr2tab[i])/gone)) -
                  (glog2(fabs (1 - gexp2(t2))))));
    }

  return;
}

void fr4table (void)
{
  long long int i,j;
  long long int t1,t2;

  for (i = 0; i < f; i++)
    {
      t1 = (-i * (m4)) - (gone * 4);
      fr4tab[i] = glog2(fabs (1 - gexp2(t1)));
    }

  for (j = 0; j < f; j++)//g1*16 = 65536
    {
      t2 = (-j * (m4)) - (gone * 4) - (m4/2);
      fr4tab[j+f] = glog2(fabs (1 - gexp2(t2)));
      }

  return;
}

void dr4table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m4)) - (gone * 4);
      t1 = t0 - (m4/2);
      dr4tab[i] = (((glog2(fabs (1 - gexp2(t1)))) - fr4tab[i]) / (-m4/2))*-
                  gone;
    }

  for (j = 0; j < f; j++)//g1*16 = 65536
    {
      t2 = (-j * (m4)) - (gone * 4) - (m4/2) - 1;
      t3 = t2 - (m4/2);
      dr4tab[j+f] = (((glog2(fabs (1 - gexp2(t3)))) - fr4tab[j+f]) / (-
                    m4/2))*-gone;
    }

  return;
}

void er4table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3,t4,t5;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m4)) - (gone * 4);
      t1 = t0 - (m4/2);
      t2 = t0 + (-m4/4);
      er4tab[i] = round(-((fr4tab[i] + (((-m4/4) * -dr4tab[i])/gone)) -
                  (glog2(fabs (1 - gexp2(t2))))));
    }

  return;
}

void fr8table (void)
{
  long long int i,j;
  long long int t1,t2;

  for (i = 0; i < f; i++)
    {
      t1 = (-i * (m8)) - (gone * 8);
      fr8tab[i] = glog2(fabs (1 - gexp2(t1)));
    }

  for (j = 0; j < fr8; j++)
    {
      t2 = (-j * (m8)) - (gone * 8) - (m8/2);
      fr8tab[j+f] = glog2(fabs (1 - gexp2(t2)));
    }

  return;
}

void dr8table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m8)) - (gone * 8);
      t1 = t0 - (m8/2);
      dr8tab[i] = (((glog2(fabs (1 - gexp2(t1)))) - fr8tab[i]) / (-m8/2))*-
                  gone;
    }

  for (j = 0; j < fr8; j++)
    {
      t2 = (-j * (m8)) - (gone * 8) - (m8/2) - 1;
      t3 = t2 - (m8/2);
      dr8tab[j+f] = (((glog2(fabs (1 - gexp2(t3)))) - fr8tab[j+f]) / (-
                    m8/2))*-gone;
    }

  return;
}

void er8table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3,t4,t5;

  for (i = 0; i < f; i++)
    {
      t0 = (-i * (m8)) - (gone * 8);
      t1 = t0 - (m8/2);
      t2 = t0 + (-m8/4);
      er8tab[i] = round(-((fr8tab[i] + (((-m8/4) * -dr8tab[i])/gone)) –
                  (glog2(fabs (1 - gexp2(t2))))));
    }

  return;
}

void fr16table (void)
{
  long long int i,j;
  long long int t1,t2;

  for (i = 0; i < fr16; i++)
    {
      t1 = (-i * (m16)) - (gone * 16);
      fr16tab[i] = glog2(fabs (1 - gexp2(t1)));

    }

  for (j = 0; j < fr16; j++)
    {
      t2 = (-j * (m16)) - (gone * 16) - (m16/2);
      fr16tab[j+fr16] = glog2(fabs (1 - gexp2(t2)));
    }
  return;
}

void dr16table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3;

  for (i = 0; i < fr16; i++)
    {
      t0 = (-i * (m16)) - (gone * 16);
      t1 = t0 - (m16/2);//t1 = t0 - (m2/2);
      dr16tab[i] = (((glog2(fabs (1 - gexp2(t1)))) - fr16tab[i]) / (-
                   m16/2))*-gone;
    }

  for (j = 0; j < fr16; j++)
    {
      t2 = (-j * (m16)) - (gone * 16) - (m16/2) - 1;
      t3 = t2 - (m16/2);//t3 = t2 - (m2/2);
      dr16tab[j+fr16] = (((glog2(fabs (1 - gexp2(t3)))) - fr16tab[j+fr16])
                        / (-m16/2))*-gone;

    }

  return;
}

void er16table (void)
{
  long long int i,j;
  long long int t0,t1,t2,t3,t4,t5;

  for (i = 0; i < fr16; i++)
    {
      t0 = (-i * (m16)) - (gone * 16);
      t1 = t0 - (m16/2);
      t2 = t0 + (-m16/4);
      er16tab[i] = (-((fr16tab[i] + (((-m16/4) * -dr16tab[i])/gone)) -
                   (glog2(fabs (1 - gexp2(t2))))));

    }

  for (j = 0; j < fr16; j++)
    {
      t3 = (-j * (m16)) - (gone * 16) - (m16/2);
      t4 = t3 - (m16/2);
      t5 = t3 + (-m16/4);
      er16tab[j+fr16] = (-((fr16tab[j+fr16] + (((-m16/4) * -
                        dr16tab[j+fr16])/gone)) - (glog2(fabs (1 –
                        gexp2(t5))))));
    }

  return;
}

void ptable (void)
{
  long long int i;
  double t,error,error1,error2,temp;
  for (i = 0; i < p; i++)
    {
      t = (-gone * 2.0) - ((m2/2) / p) * i ;//- (m2/2)
      error1 = (fr2tab[0] + ((-m2/2) / p) * i * -dr2tab[0]/gone);
      error2 = glog2(fabs ( 1 - gexp2 (t)));
      error = error1 - error2;
      temp = (error / er2tab[0]) * gone;
      ptab[i] = temp;
    }

  return;
}

long long int lookupfr1 (long long int arg)
{
  long long int t,r;

  r = arg % m1;

  if (r < (m1/2))
    {
      t = (arg - gone) / (m1);
    }

   else
    {
      t = (arg - gone) / (m1) + f;
    }

  return fr1tab[t];
}

long long int lookupdr1 (long long int arg)
{
  long long int t,r;

  r = arg % m1;

  if (r < (m1/2))
  {
    t = (arg - gone) / (m1);
  }

  else
  {
    t = (arg - gone) / (m1) + f;
  }

  return dr1tab[t];
}

long long int lookuper1 (long long int arg)
{
  long long int t,r;

  r = arg % m1;

  if (r < (m1/2))
   {
      t = (arg - gone) / (m1);
    }

  else
```

```
    {
      t = (arg - gone) / (m1) ;
    }

  return er1tab[t];
}

long long int lookupfr2 (long long int arg)
{
  long long int t,r;

  r = arg % m2;

  if (r < (m2/2))
    {
      t = (arg - gone * 2) / (m2);
    }

  else
    {
      t = (arg - gone * 2) / (m2) + f;
    }

  return fr2tab[t];
}

long long int lookupdr2 (long long int arg)
{
  long long int t,r;

  r = arg % m2;

  if (r < (m2/2))
    {
      t = (arg - gone * 2) / (m2);
    }

  else
    {
      t = (arg - gone * 2) / (m2) + f;
    }

  return dr2tab[t];
}

long long int lookuper2 (long long int arg)
{
  long long int t,r;

  r = arg % m2;

  if (r < (m2/2))
    {
      t = (arg - gone * 2) / (m2);
    }

  else
    {
      t = (arg - gone * 2) / (m2) ;
    }

  return er2tab[t];
}
```

```
long long int lookupfr4 (long long int arg)
{
  long long int t,r;

  r = arg % m4;

  if (r < (m4/2))
    {
      t = (arg - gone * 4) / (m4);
    }

  else
    {
      t = (arg - gone * 4) / (m4) + f;
    }

  return fr4tab[t];
}

long long int lookupdr4 (long long int arg)
{
  long long int t,r;

  r = arg % m4;

  if (r < (m4/2))
    {
      t = (arg - gone * 4) / (m4);
    }

  else
    {
      t = (arg - gone * 4) / (m4) + f;
    }

  return dr4tab[t];
}

long long int lookuper4 (long long int arg)
{
  long long int t,r;

  r = arg % m4;

  if (r < (m4/2))
    {
      t = (arg - gone * 4) / (m4);
    }

  else
    {
      t = (arg - gone * 4) / (m4) ;
    }

  return er4tab[t];
}

long long int lookupfr8 (long long int arg)
{
  long long int t,r;

  r = arg % m8;
  if (r < (m8/2))
    {
      t = (arg - gone * 8) / (m8);
    }

  else
    {
      t = (arg - gone * 8) / (m8) + f;
    }

  return fr8tab[t];
}

long long int lookupdr8 (long long int arg)
{
  long long int t,r;

  r = arg % m8;

  if (r < (m8/2))
    {
      t = (arg - gone * 8) / (m8);
    }

  else
    {
      t = (arg - gone * 8) / (m8) + f;
    }

  return dr8tab[t];
}

long long int lookuper8 (long long int arg)
{
  long long int t,r;

  r = arg % m8;

  if (r < (m8/2))
    {
      t = (arg - gone * 8) / (m8);
    }

  else
    {
      t = (arg - gone * 8) / (m8) ;
    }

  return er8tab[t];
}

long long int lookupfr16 (long long int arg)
{
  long long int t,r;

  r = arg % m16;

  if (r < (m16/2))
    {
      t = (arg - gone * 16) / (m16);
    }
  else
    {
      t = (arg - gone * 16) / (m16) + fr16;
    }

  return fr16tab[t];
}

long long int lookupdr16 (long long int arg)
{
  long long int t,r;

  r = arg % m16;

  if (r < (m16/2))
    {
      t = (arg - gone * 16) / (m16);
    }

  else
    {
      t = (arg - gone * 16) / (m16) + fr16;
    }

  return dr16tab[t];
}

long long int lookuper16 (long long int arg)
{
  long long int t,r;

  r = arg % m16;

  if (r < (m16/2))
    {
      t = (arg - gone * 16) / (m16);
    }

  else
    {
      t = (arg - gone * 16) / (m16) + fr16;
    }

  return er16tab[t];
}

long long int lookupp (long long int arg)
{
  long long int t;

  t = arg;

  return ptab[t];
}

long long int sub1 (long long int arg1, long long int arg2)
{
  long long int s1,s2,s4,s8,s16;
  long long int t,t1,t2;
  long long int res;
  long long int r,fr,dr,er,pd;
  long long int k;
  double k1;
```

```
r = -arg2;

k1 = (r*2) / gone;
k = k1;

if (k <= 1)// -1 < r < 0
  {
    t = 0;
    fr = 0;
    dr = 0;
    er = 0;
    s1 = 0;
    pd = 0;
    goto mult;
   }

if (k <= 3)// -2 < r < -1
  {
    t = (r % (m1/2));
    fr = lookupfr1(r);
    dr = lookupdr1(r);
    er = lookuper1(r);
    s1 = ((r % (m1/2)) / ((m1/2) / p));
    pd = lookupp(s1);
    goto mult;
  }

if (k <= 7)// -4 < r < -2
  {
    t = (r % (m2/2));
    fr = lookupfr2(r);
    dr = lookupdr2(r);
    er = lookuper2(r);
    s2 = ((r % (m2/2)) / ((m2/2) / p));
    pd = lookupp(s2);
    goto mult;
  }

if (k <= 15)// -8 < r < -4
  {
    t = (r % (m4/2));
    fr = lookupfr4(r);
    dr = lookupdr4(r);
    er = lookuper4(r);
    s4 = ((r % (m4/2)) / ((m4/2) / p));
    pd = lookupp(s4);
    goto mult;
  }

if (k <= 31)// -16 < r < -8
  {
    t = (r % (m8/2));
    fr = lookupfr8(r);
    dr = lookupdr8(r);
    er = lookuper8(r);
    s8 = ((r % (m8/2)) / ((m8/2) / p));
    pd = lookupp(s8);
    goto mult;
  }


if (k <= 63) // -32 < r < -16
```

```
    {
      t = (r % (m16/2));
      fr = lookupfr16(r);
      dr = lookupdr16(r);
      er = lookuper16(r);
      s16 = ((r % (m16/2)) / ((m16/2) / p));
      pd = lookupp(s16);
      goto mult;
    }

  if (k <= 511)
    {
      fr = 0;
      dr = 0;
      er = 0;
      t = 0;
      pd = 0;
      goto mult;
    }

 mult:

 t1 = ((er) * pd);
 t1 = t1 / gone;
 t2 = t * dr;
 t2 = t2 / gone;
 res = arg1 + fr + t2 - t1;

  return res;
}


/*Generate floating point numbers */
double subx (long long int arg1, long long int arg2)
{

  long long int tmp;
  double fr,result;

  tmp = arg1 - arg2;

  if (tmp == 0)
    {
      result = -maxcomp;
      goto end;
    }
  else
    {
      fr = log2e * log (1 - exp (ln2 * (arg2 - arg1) / (gone))) * (gone);
      result = arg1 + fr;
    }
 end:

  return result;
}


/**************************************************************************/
/*          GENERATE LNS SUBTRACTOR USING RANGE SHIFTED METHODS           */
/**************************************************************************/
/*Select region of 'r'*/
```

```
long long int suby (long long int arg1,long long int arg2 )
{
  long long int a1,r;
  long long int j,j2;
  long long int k1,k11;
  long long int i1,i2;
  long long int t,t1,t2,t3,t4;
  long long int r1,r2,r11,r12;
  long long int result;

  a1 = arg1; //i
  r = arg2;  //j-i

  /*****************/
  /* REGION r = 0  */
  /*****************/
  if (r == 0)
    {
      result = -maxcomp;
      goto end;
    }

  /***************************/
  /* REGION -delta11 < r < 0 */
  /***************************/
  if (r > -f3)
  //The computation of result based upon accessing F3 table directly
    {
      t = (-(r % f3) / g);
      result = a1 + f3tab[t];
      goto end;
    }

  /*********************************/
  /* REGION -delta1 < r < -delta11 */
  /*********************************/
  t = -f2f3;
  //The computation of result based upon 1st order arch.
  if (r > t)
    {
      t1 = (-(r % f3) / g);

   if (t1 == 0)
     {
        t2 = (f3/g) - (-r / f3);
        t3 = 0;
     }
   else
     {
        t2 = (f3/g) - (-r / f3) - 1;
        t3 = (f3/g) - t1;
     }


     i2 = a1 + f2tab[t2];
     r2 = r + f3tab[t3] - f2tab[t2];
     result = sub1(i2,r2);

     goto end;
      }

   else
    {
      a1 = a1;
      r  = r;
    }

  /***************************/
  /* REGION -1 < r < -delta1 */
  /***************************/
  t = -gone;
  if (r > t)
  {
      if (((r % f2f3)) < -(f2f3 - f3))
      //when middle+low fields are lesser than FF000h, then executes
      //the operation based upon 1st order arch using only F1 and F3 tables
      {
         r1 = -r / f2f3;
         t1 = (((r % f2f3)/g) + (f2f3)/g);
         i2 = a1 + f1tab[r1];
         r2 = r + f3tab[t1] - f1tab[r1];
         result = sub1(i2,r2);
       }

      else
      //The computation of result based upon 2nd order arch.
      {
         r1 = -r / f2f3;
         k11 = (-(r % f3) / g);
         r11 = (-(r % f2f3) / f3);
         k1 = -((r % f2f3) + f2f3);
         r12 = k1 + f3tab[k11] - f2tab[r11];
         i1 = (r + a1) + f2tab[r11];
         j2 = sub1(i1,r12);
         i2 = a1 + f1tab[r1];
         r2 =  j2 - i2;
         result = sub1(i2,r2);
      }
      goto end;
    }

  /*****************/
  /* REGION r < -1 */
  /*****************/
 else
    {
      result = sub1(a1,r);
      goto end;
    }


  end:

  return result;
}

long long int yrnd (long long int arg)
{
  long long int rem,yrnd;

  rem = (arg % (g));
  if(rem < 0)
    {
      if (-rem <= (g / 2))
       {
        yrnd = ((arg - rem ) / g);
        }
       else
        {
          yrnd = ((arg - g - rem)/g);
        }
    }
  else
    {
      if (rem <= ((g) / 2))
       {
          yrnd = (arg - rem) / ( g);
       }
     else
       {
         yrnd = ((arg+(g)-rem))/(g);
       }
    }

  return yrnd;
}

void compare (void)
{
  long long int j,i,ry,y,n;

  double x, rx, errl, errgl, maxherrgl, maxlerrgl, maxlerrl, maxlerr2,
         maxherr2, maxlerr, maxherrl, maxherr, cumerrl, cummoderrl, err,
         cumerr, cummoderr;

  maxherrgl = 0;
  maxlerrgl = 0;
  maxherrl = 0;
  maxlerrl = 0;
  maxherr = 0;
  maxlerr = 0;
  cumerrl = 0;
  cumerr = 0;
  cummoderrl = 0;
  cummoderr = 0;
  n = 0;

//arg1 = 1; /* due to i sets to zero, =>(i*g==0*g) */

  for (j=0; j>=-201326592; j--) //executes from 0 to -24 (essential zero)
    {
      arg2 = (j*g);

      y = suby(arg1,arg2);
      x = subx(arg1,arg2);

     errgl = y - x;

    if (errgl>maxherrgl)
     {
       maxherrgl = errgl;
     }

    if (errgl<maxlerrgl)
     {
       maxlerrgl = errgl;
     }

    ry = yrnd(y);
    rx = x/(step*g);
    errl = ry - rx;

    if (errl<maxlerrl)
     {
        maxlerrl = errl;
        maxlerr = (exp(ln2*(errl/gone))-1)*gone;
        maxlerr2 = (errl/rx)*gone;
      }
    if (errl>maxherrl)
     {
        maxherrl = errl;
        maxherr = (exp(ln2*(errl/gone))-1)*gone;
        maxherr2 = (errl/rx)*gone;
     }

    cumerrl = (cumerrl+errl);
    cummoderrl = (cummoderrl + fabs(errl));
    err = (exp(ln2*(errl/gone))-1)*gone;
    cumerr = (cumerr + err);
    cummoderr = (cummoderr + fabs(err));
    n = n + 1;

       printf("~~~~~~~~~~######################~~~~~~~~~~~~~~\n");
       printf("SIMULATION OF ADVANCED LOGARITHMIC SUBTRACTION\n");
       printf("maxerr @ i %lli\t j %lli\n",arg1,j);
       printf("x %lf\t rx %lf\n",x,rx);
       printf("y %lli\t\t ry %lli\n",y,ry);
       printf("LSGB hi  e  %lf\t lo e %lf\n",maxherrgl,maxlerrgl);
       printf("LSB  hi  e  %lf\t lo e %lf\n",maxherrl,maxlerrl);
       printf("     av |e| %lf\t av e %lf\n",(cummoderrl/n),(cumerrl/n));
       printf("REL  hi  e  %lf\t lo e %lf\n",maxherr,maxlerr);
       printf("     av |e| %lf\t av e %lf\n\n",(cummoderr/n),(cumerr/n));
  }

}

main ()
{
  ln2 = log (2);
  log2e = 1 / ln2;
  f1table ();
  f2table ();
  f3table ();
  fr1table ();
  dr1table ();
  er1table ();
  fr2table ();
  dr2table ();
  er2table ();
  fr4table ();
  dr4table ();
  er4table ();
  fr8table ();
  dr8table ();
  er8table ();
  fr16table ();
  dr16table ();
  er16table ();
  ptable ();
  compare ();
  return (0);
}
```

A3. VHDL Model for 32-bit LNS Add/Subtract Unit

```
--------------------------------------------------------------------
-- Title    : LNS addsub (entity)
-- Filename : LNS AddSub with 2nd order with improved interpolation
--------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity LNSaddsub is
    port(
         -- Inputs
         s_addsub : in std_logic; --Operation to perform add(0)/sub(1)
         sa : in std_logic;       --Value-sign of operand A
         sb : in std_logic;       --Value-sign of operand B
         a  : in std_logic_vector(30 downto 0);
         b  : in std_logic_vector(30 downto 0);

         -- ROM interfaces:
        clk   : in std_logic;

        -- Outputs:
        sq    : out std_logic;
        q     : out std_logic_vector(30 downto 0);
        oflow : out std_logic;
        uflow : out std_logic
       );

end LNSaddsub;

architecture rtl of LNSaddsub is
-- Components

component checkops  is
   port(
         SA : in std_logic; -- value-sign bit of operand A
         SB : in std_logic; -- value-sign bit of operand B
         A  : in std_logic_vector(30 downto 0);
         B  : in std_logic_vector(30 downto 0);
         s_addsub : in std_logic; -- add#/sub (operation to perform)
         NEG : out std_logic;
         Azero : out std_logic;
         Bzero : out std_logic
        );

end component;

component setvalues  is
   port (
          A : in std_logic_vector(30 downto 0);
          B : in std_logic_vector(30 downto 0);
          i : out signed(31 downto 0);
          j : out signed(31 downto 0);
          r : out signed(31 downto 0);
          AltB : out std_logic;
          AeqB : out std_logic
        );
end component;
```

```
component rs_region is
    port (s_addsub : in STD_LOGIC;
           r : in  signed  (31 downto 0);
           i : in  signed  (31 downto 0);
           j : in  signed  (31 downto 0);
           F1_addr : out  STD_LOGIC_VECTOR (6 downto 0);
           F1 : in  STD_LOGIC_VECTOR (31 downto 0);
           F2_addr : out  STD_LOGIC_VECTOR (7 downto 0);
           F2 : in  STD_LOGIC_VECTOR (32 downto 0);
           F3_addr : out  STD_LOGIC_VECTOR (7 downto 0);
           F3 : in  STD_LOGIC_VECTOR (33 downto 0);
           r1a : out  signed  (37 downto 0);
           i1 : out  signed  (37 downto 0);
           i2 : out  signed  (37 downto 0);
           rs_cu : out std_logic;
           rs_infi : out std_logic;
           ResFromF3 : out signed (37 downto 0);
           val_near_zero : out std_logic;
           val_near_modtwo : out std_logic
         );
end component;

component cu_int is
   port(
        clk : in std_logic;
        s_addsub : in std_logic;
        val_near_modtwo : in std_logic;
        rs_cu : in std_logic;
        rs_infi : in std_logic;
        en_busA : out std_logic;
        sel_busA : out std_logic;
        en_busB : out std_logic
       );
end component;

component busA is
    port (
           clk : in std_logic;
           s_addsub : std_logic;
           en : in std_logic;
           sel : in std_logic;
           rs_infi : in std_logic;
           r1a : in  std_logic_vector (37 downto 0);
           i1 : in  signed  (37 downto 0);
           i2 : in  signed  (37 downto 0);
           r2 : in  std_logic_vector (37 downto 0);
           r : out std_logic_vector (37 downto 0);
           i : out signed (37 downto 0)
          );
end component;

component subR2 is
  port (
    a : in std_logic_vector (37 downto 0);
    b : in std_logic_vector (37 downto 0);
    result : out std_logic_vector (37 downto 0));
end component;


component partLookup2  is
   port(
        r_int : in std_logic_vector(37 downto 0);
        s_addsub : in std_logic; --sum#/diff
```

```
      -- ROM interface
      F1a_addr : out std_logic_vector(7 downto 0);
      D1a_addr : out std_logic_vector(7 downto 0);
      E1a_addr : out std_logic_vector(5 downto 0);
      F2a_addr : out std_logic_vector(7 downto 0);
      D2a_addr : out std_logic_vector(7 downto 0);
      E2a_addr : out std_logic_vector(5 downto 0);
      F4a_addr : out std_logic_vector(7 downto 0);
      D4a_addr : out std_logic_vector(7 downto 0);
      E4a_addr : out std_logic_vector(5 downto 0);
      F8a_addr : out std_logic_vector(7 downto 0);
      D8a_addr : out std_logic_vector(7 downto 0);
      E8a_addr : out std_logic_vector(5 downto 0);
      F16a_addr : out std_logic_vector(7 downto 0);
      D16a_addr : out std_logic_vector(7 downto 0);
      E16a_addr : out std_logic_vector(5 downto 0);
      F32a_addr : out std_logic_vector(4 downto 0);
      D32a_addr : out std_logic_vector(4 downto 0);
      E32a_addr : out std_logic_vector(4 downto 0);

      F2_addr : out std_logic_vector(7 downto 0);
      D2_addr : out std_logic_vector(7 downto 0);
      E2_addr : out std_logic_vector(6 downto 0);
      F4_addr : out std_logic_vector(7 downto 0);
      D4_addr : out std_logic_vector(7 downto 0);
      E4_addr : out std_logic_vector(6 downto 0);
      F8_addr : out std_logic_vector(7 downto 0);
      D8_addr : out std_logic_vector(7 downto 0);
      E8_addr : out std_logic_vector(6 downto 0);
      F16_addr : out std_logic_vector(7 downto 0);
      D16_addr : out std_logic_vector(7 downto 0);
      E16_addr : out std_logic_vector(6 downto 0);
      F32_addr : out std_logic_vector(4 downto 0);
      D32_addr : out std_logic_vector(4 downto 0);
      E32_addr : out std_logic_vector(4 downto 0);
      P_addr : out std_logic_vector(8 downto 0);

      delta : out std_logic_vector(27 downto 0)
   );

end component;

component addmul_WT2 is
   port(
        i : in std_logic_vector (37 downto 0);
        r2 : in std_logic_vector (37 downto 0);
        s_addsub : std_logic;
        F1a : in std_logic_vector (29 downto 0);
        D1a : in std_logic_vector (28 downto 0);
        E1a : in std_logic_vector (11 downto 0);
        F2a : in std_logic_vector (29 downto 0);
        D2a : in std_logic_vector (28 downto 0);
        E2a : in std_logic_vector (11 downto 0);
        F4a : in std_logic_vector (29 downto 0);
        D4a : in std_logic_vector (28 downto 0);
        E4a : in std_logic_vector (11 downto 0);
        F8a : in std_logic_vector (29 downto 0);
        D8a : in std_logic_vector (28 downto 0);
        E8a : in std_logic_vector (11 downto 0);
        F16a : in std_logic_vector (29 downto 0);
        D16a : in std_logic_vector (28 downto 0);
        E16a : in std_logic_vector (11 downto 0);
        F32a : in std_logic_vector (29 downto 0);
        D32a : in std_logic_vector (28 downto 0);
        E32a : in std_logic_vector (11 downto 0);

        F2 : in std_logic_vector (29 downto 0);
        D2 : in std_logic_vector (28 downto 0);
        E2 : in std_logic_vector (11 downto 0);
        F4 : in std_logic_vector (29 downto 0);
        D4 : in std_logic_vector (28 downto 0);
        E4 : in std_logic_vector (11 downto 0);
        F8 : in std_logic_vector (29 downto 0);
        D8 : in std_logic_vector (28 downto 0);
        E8 : in std_logic_vector (11 downto 0);
        F16 : in std_logic_vector (29 downto 0);
        D16 : in std_logic_vector (28 downto 0);
        E16 : in std_logic_vector (11 downto 0);
        F32 : in std_logic_vector (29 downto 0);
        D32 : in std_logic_vector (28 downto 0);
        E32 : in std_logic_vector (11 downto 0);
        Ptab : in std_logic_vector (29 downto 0);
        delta : in std_logic_vector(27 downto 0);
        result : out std_logic_vector (37 downto 0)
    );
end component;

component busB is
   port (
           en : in std_logic;
           s_result : in  std_logic_vector (37 downto 0);
           result_out : out  std_logic_vector (37 downto 0)
         );
end component;

component resultStatus is
   port (  s_addsub : in STD_LOGIC;
           s_result : in std_logic_vector (37 downto 0);
           j : in  signed  (31 downto 0);
           i : in  signed  (31 downto 0);
           ResFromF3 : in signed (37 downto 0);
           val_near_zero : in std_logic;
           val_near_modtwo : in std_logic;
           Azero : in std_logic;
           Bzero :in std_logic;
           NEG :in std_logic;
           AltB : in std_logic;
           AeqB : in std_logic;
           SQ : out std_logic;
           Q : out  STD_LOGIC_VECTOR (30 downto 0);
           Oflow : out std_logic;
           Uflow : out std_logic
         );
end component;

component LUT256a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(32 downto 0)
        );
end component;

component LUT256b is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(33 downto 0)
        );
end component;

component LUT128 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(6 downto 0);
          data_out     : out std_logic_vector(31 downto 0)
        );
end component;

component f1a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d1a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e1a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(5 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;


component f2a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d2a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e2a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(5 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f4a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d4a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e4a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(5 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f8a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d8a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e8a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(5 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f16a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d16a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e16a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(5 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f32a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d32a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e32a is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f2 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d2 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e2 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(6 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f4 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d4 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e4 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(6 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f8 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d8 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e8 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(6 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f16 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d16 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e16 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(6 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component f32 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

component d32 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(28 downto 0)
        );
end component;

component e32 is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(4 downto 0);
          data_out     : out std_logic_vector(11 downto 0)
        );
end component;

component ptable is
   port ( clk          : in  std_logic;
          address      : in  std_logic_vector(8 downto 0);
          data_out     : out std_logic_vector(29 downto 0)
        );
end component;

--wires checkops
signal NEG,Azero,Bzero : std_logic;
```

```
--wires setvalues
signal i,j,r : signed (31 downto 0);
signal AltB,AeqB : std_logic;

--wires rangeshifter -1 < r < 0
signal F1_addr_rs : std_logic_vector (6 downto 0);
signal F1_rs : std_logic_vector (31 downto 0);
signal F2_addr_rs : std_logic_vector (7 downto 0);
signal F2_rs : std_logic_vector (32 downto 0);
signal F3_addr_rs : std_logic_vector (7 downto 0);
signal F3_rs : std_logic_vector (33 downto 0);
signal r1a,i1,i2 : signed (37 downto 0);
signal ResFromF3 : signed (37 downto 0);
signal val_near_zero,val_near_modtwo : std_logic;
signal rs_cu : std_logic;

--wires control unit interpolator
signal en_busA : std_logic;
signal sel_busA : std_logic;
signal en_busB : std_logic;
signal rs_infi : std_logic;

--wires busA
signal i_int : signed (37 downto 0);
signal r_int : std_logic_vector (37 downto 0);
signal r1a_us : std_logic_vector(37 downto 0);

--sub r2
signal r2_new : std_logic_vector(37 downto 0);
signal i2_new_us : std_logic_vector(37 downto 0);

--wires partlookup
signal F1a_addr,F2a_addr,F4a_addr,F8a_addr,F16a_addr : std_logic_vector (7
downto 0);
signal D1a_addr,D2a_addr,D4a_addr,D8a_addr,D16a_addr : std_logic_vector (7
downto 0);
signal E1a_addr,E2a_addr,E4a_addr,E8a_addr,E16a_addr : std_logic_vector (5
downto 0);
signal F32a_addr,D32a_addr,E32a_addr : std_logic_vector (4 downto 0);
signal F2s_addr,F4s_addr,F8s_addr,F16s_addr : std_logic_vector (7 downto 0);
signal D2s_addr,D4s_addr,D8s_addr,D16s_addr : std_logic_vector (7 downto 0);
signal E2s_addr,E4s_addr,E8s_addr,E16s_addr : std_logic_vector (6 downto 0);
signal F32s_addr,D32s_addr,E32s_addr : std_logic_vector (4 downto 0);
signal P_addr : std_logic_vector (8 downto 0);
signal delta : std_logic_vector (27 downto 0);

--wires addmul
signal i_int_us : std_logic_vector (37 downto 0);
signal F1a_LUT,F2a_LUT,F4a_LUT,F8a_LUT,F16a_LUT,F32a_LUT : std_logic_vector
(29 downto 0);
signal D1a_LUT,D2a_LUT,D4a_LUT,D8a_LUT,D16a_LUT,D32a_LUT : std_logic_vector
(28 downto 0);
signal E1a_LUT,E2a_LUT,E4a_LUT,E8a_LUT,E16a_LUT,E32a_LUT : std_logic_vector
(11 downto 0);
signal F2s_LUT,F4s_LUT,F8s_LUT,F16s_LUT,F32s_LUT : std_logic_vector (29
downto 0);
signal D2s_LUT,D4s_LUT,D8s_LUT,D16s_LUT,D32s_LUT : std_logic_vector (28
downto 0);
signal E2s_LUT,E4s_LUT,E8s_LUT,E16s_LUT,E32s_LUT : std_logic_vector (11
downto 0);
signal P : std_logic_vector (29 downto 0);
```

```
--wires busB
signal result : std_logic_vector (37 downto 0);
signal result_tmp : std_logic_vector (37 downto 0);

begin

check_ops : checkops port map (sa,sb,a,b,s_addsub,NEG,Azero,Bzero);

set_values : setvalues port map (a,b,i,j,r,AltB,AeqB);

rangeshifter : rs_region port map
(s_addsub,r,i,j,F1_addr_rs,F1_rs,F2_addr_rs,F2_rs,F3_addr_rs,F3_rs,r1a,i1,i
2,rs_cu,rs_infi,ResFromF3,val_near_zero,val_near_modtwo);

control_unit_int : cu_int port map
(clk,s_addsub,val_near_modtwo,rs_cu,rs_infi,en_busA,sel_busA,en_busB);

r1a_us <= std_logic_vector(r1a(37 downto 0));--change bits to unsigned
mux_busA : busA port map
(clk,s_addsub,en_busA,sel_busA,rs_infi,r1a_us,i1,i2,r2_new,r_int,i_int);

LUT_lookup : partlookup2 port map
(r_int,s_addsub,F1a_addr,D1a_addr,E1a_addr,F2a_addr,D2a_addr,E2a_addr,F4a_a
ddr,D4a_addr,E4a_addr,F8a_addr,D8a_addr,E8a_addr,F16a_addr,D16a_addr,E16a_a
ddr,F32a_addr,D32a_addr,E32a_addr,F2s_addr,D2s_addr,E2s_addr,F4s_addr,D4s_a
ddr,E4s_addr,F8s_addr,D8s_addr,E8s_addr,F16s_addr,D16s_addr,E16s_addr,F32s_
addr,D32s_addr,E32s_addr,P_addr,delta);

i_int_us <= std_logic_vector(i_int(37 downto 0));--change bits to unsigned
LUT_addmul : addmul_WT2 port map
(i_int_us,r_int,s_addsub,F1a_LUT,D1a_LUT,E1a_LUT,F2a_LUT,D2a_LUT,E2a_LUT,F4
a_LUT,D4a_LUT,E4a_LUT,F8a_LUT,D8a_LUT,E8a_LUT,F16a_LUT,D16a_LUT,E16a_LUT,F3
2a_LUT,D32a_LUT,E32a_LUT,F2s_LUT,D2s_LUT,E2s_LUT,F4s_LUT,D4s_LUT,E4s_LUT,F8
s_LUT,D8s_LUT,E8s_LUT,F16s_LUT,D16s_LUT,E16s_LUT,F32s_LUT,D32s_LUT,E32s_LUT
,P,delta,result_tmp);

i2_new_us <= std_logic_vector(i2(37 downto 0));
sub_r2 : subR2 port map (i2_new_us,result_tmp,r2_new);

mux_busB : busB port map (en_busB,result_tmp,result);

Final_result : resultStatus port map
(s_addsub,result,j,i,ResFromF3,val_near_zero,val_near_modtwo,Azero,Bzero,NE
G,AltB,AeqB,sq,q,oflow,uflow);

--ROMs

F1table : LUT128 port map (clk,F1_addr_rs,F1_rs);

F2table : LUT256a port map (clk,F2_addr_rs,F2_rs);

F3table : LUT256b port map (clk,F3_addr_rs,F3_rs);

F1int_add : f1a port map (clk,F1a_addr,F1a_LUT);

D1int_add : d1a port map (clk,D1a_addr,D1a_LUT);

E1int_add : e1a port map (clk,E1a_addr,E1a_LUT);

F2int_add : f2a port map (clk,F2a_addr,F2a_LUT);

D2int_add : d2a port map (clk,D2a_addr,D2a_LUT);
```

```
E2int_add : e2a port map (clk,E2a_addr,E2a_LUT);
F4int_add : f4a port map (clk,F4a_addr,F4a_LUT);
D4int_add : d4a port map (clk,D4a_addr,D4a_LUT);
E4int_add : e4a port map (clk,E4a_addr,E4a_LUT);
F8int_add : f8a port map (clk,F8a_addr,F8a_LUT);
D8int_add : d8a port map (clk,D8a_addr,D8a_LUT);
E8int_add : e8a port map (clk,E8a_addr,E8a_LUT);
F16int_add : f16a port map (clk,F16a_addr,F16a_LUT);
D16int_add : d16a port map (clk,D16a_addr,D16a_LUT);
E16int_add : e16a port map (clk,E16a_addr,E16a_LUT);
F32int_add : f32a port map (clk,F32a_addr,F32a_LUT);
D32int_add : d32a port map (clk,D32a_addr,D32a_LUT);
E32int_add : e32a port map (clk,E32a_addr,E32a_LUT);
F2int_sub : f2 port map (clk,F2s_addr,F2s_LUT);
D2int_sub : d2 port map (clk,D2s_addr,D2s_LUT);
E2int_sub : e2 port map (clk,E2s_addr,E2s_LUT);
F4int_sub : f4 port map (clk,F4s_addr,F4s_LUT);
D4int_sub : d4 port map (clk,D4s_addr,D4s_LUT);
E4int_sub : e4 port map (clk,E4s_addr,E4s_LUT);
F8int_sub : f8 port map (clk,F8s_addr,F8s_LUT);
D8int_sub : d8 port map (clk,D8s_addr,D8s_LUT);
E8int_sub : e8 port map (clk,E8s_addr,E8s_LUT);
F16int_sub : f16 port map (clk,F16s_addr,F16s_LUT);
D16int_sub : d16 port map (clk,D16s_addr,D16s_LUT);
E16int_sub : e16 port map (clk,E16s_addr,E16s_LUT);
F32int_sub : f32 port map (clk,F32s_addr,F32s_LUT);
D32int_sub : d32 port map (clk,D32s_addr,D32s_LUT);
E32int_sub : e32 port map (clk,E32s_addr,E32s_LUT);
Ptab : ptable port map (clk,P_addr,P);
end rtl;
```