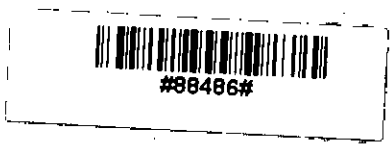
BASE TWO LOGARITHMIC NUMBER SYSTEM IN DIGITAL COMPUTERS

BY

Q. M. M. ARIFEEN

A THESIS SUBMITTED TO THE DEPARTMENT
OF COMPUTER SCIENCE AND ENGINEERING, BUET, DHAKA
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY, DHAKA
APRIL, 1995

This thesis titled "BASE TWO LOGARITHMIC NUMBER SYSTEM IN DIGITAL COMPUTERS", submitted by Q. M. M. Arifeen, Roll no. 891803P, Regd. no. 87391(87-88) of M.Sc. in Engineering has been accepted as satisfactory in partial fulfillment of the requirement for the degree of

Master of Science in Computer Science and Engineering

Examination held on the 11th April, 1995

Approved as to the style and contents by

1. M. S. Alam. 11-4-95
(Dr. Md. Shamsul Alam) (Supervisor) Chairman
Professor and Head
Dept. of CSE.,BUET.

2. _____
(Dr. Md. Kaykobad) Member
Associate Professor
Dept. of CSE.,BUET

3. _____ Member
(Md. Enamul Hamid)
Assistant Professor
Dept. of CSE.,BUET

4. _____
(Dr. Md. Abdul Mottalib) Member(Ext.)
Associate Professor
Dept. of Computer Science
University of Dhaka.

ACKNOWLEDGEMENT

I wish to express my sincere gratitude and profound indebtedness to my supervisor Dr. Md. Shamsul Alam, Professor as well as Head of the department of Computer Science and Engineering, BUET for his constant guidance, helpful advice, invaluable assistance, and endless patience throughout the progress of the work, without which this work could not have been completed.

I gracefully acknowledge the kind support and encouragement extended to me by Dr. A. B. M. Siddique Hossain, Professor and Ex. Head of the department of Computer Science and Engineering, BUET during the course of the work.

I also acknowledge with thanks the valuable suggestions offered to me by Dr. Md. Kaykobad, Associate Professor, Department of Computer Science and Engineering, BUET for improving the contents of the thesis.

My sincere gratitude is for Dr. Md. Abdul Mottalib, Associate Professor, Department of Computer Science, University of Dhaka, Dhaka for kindly acting as the external member. My sincere thanks and gratitude is also for the personnel of the department of Computer Science and Engineering, BUET particularly to the teachers without their valuable advice I could find myself in a plight.

Finally I acknowledge with thanks the all out cooperation of my other friends, other departments such as the Library, BUET, DHAKA whose cooperation enabled me to bring out this work into a fruition.

ABSTRACT

Although logarithmic number system (LNS) has been used considerably in many fields, the storage of numbers in computer memory under LNS format is still not specified, so commercial use of LNS processor is not generally seen. In spite of many unsolved problems, there are interests on LNS processors due to its fast multiplication, division and some arithmetic operations.

This thesis establishes IEEE floating point (FLP) numbers into logarithmic numbers in such a way that the conversion accuracy is high and the process needs a small size of ROM. Both IEEE single and double precision numbers are considered . At first base numbers are generated and suitable correction factor is added to generate the correct LNS number. Anti-conversion is also done in a similar process.

An attempt is made to increase the accuracy by using the non-linear correction factor which in fact tries with new values of correction factors to reduce the value of conversion error. Although non_linear - non_linear conversion technique increased accuracy (table 4-6), its effectiveness is largely dependent on mantissa length (ML). A higher bit configuration leads to lesser accumulated error. Conversion and anti-conversion using an offset value was also done that produced the best result.

ROM size of 212 Kbits (RAL 12 ML 23) produced MCE of 1.07462E-08 and ROM size of 2528 Mbits (RAL 25 ML 52) produced MCE less than the truncation error of double precision floating point number (2.22E-16). Simulated results of conversion and accumulated errors are given in article 4.4. LNS numbers obtained from this process is used in several arithmetic processing and their respective errors are also calculated.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF GRAPHS

LIST OF SYMBOLS

FLP	Floating point number as per IEEE std. 754-1985.
LNS	Logarithmic number system
ln	logarithm base e
lg	logarithm base two
lgA	approximate of logarithm base two
m	mantissa of any IEEE floating number
lg(m)	log base 2 of m where m is the mantissa of a floating point number
lgm	log base 2 of m calculated through ROM table
Δlgm	correction factor for FLP to LNS conversion
Δm	correction factor for LNS to FLP conversion
antilg	antilogarithm of base two
antilg(lgm)	antilogarithm of base two of lgm antilg(lgm) = $2^{lgm} - 1.0$
mx	mantissa of FLP number x
RAL	ROM address length (in bits)
ML	Mantissa length
MCE	maximum conversion error
CF	compensating factor

CHAPTER 1

AN INTRODUCTION TO THE LOGARITHMIC NUMBER SYSTEM

1.1 Introduction

The logarithmic number system (LNS) can draw considerable attention in favour of its use in digital computer due to its faster processing capability for many arithmetic operations. The use of LNS in decimal system is frequent and many arithmetic operations such as multiplication, division, square-root, power etc. are frequently done by using logarithmic number. Due to its faster processing capability, study [1;2,3,10,18] has been done so that LNS can replace the established floating point (FLP) number system as a general use in digital computer. Considerable studies have been done recently in the various field of applications e.g. digital signal processing, fast Fourier transformation etc. that uses LNS [4,5]. The main advantage of using the LNS in a digital computer is evident from the fact that complicated and time consuming multiplication and division processes can be replaced by simple addition and subtraction. In a digital computer, real numbers are represented in floating point (FLP) format and here the main problem becomes to find a suitable method to convert FLP number into LNS number format and vice-versa which is fast enough with no loss or little loss of accuracy.

Various research papers have been published on logarithmic number system and prototype processor has been fabricated that directly deals with logarithmic number [6]. Hybrid type of processors [3] has been proposed that deals with both FLP and LNS numbers. The

processor depicted in [3] uses FLP numbers for addition and subtraction and LNS numbers for multiplication and division intensive arithmetic operations.

An alternate design [1,6] is to convert all the floating point numbers into logarithmic numbers and to perform arithmetic operations (including addition and subtraction) in logarithmic number system. Once the arithmetic operation is completed, the resultant LNS number is converted back into FLP number system.

The organization of the thesis is as follows :

Chapter 1 focuses on the contents of each chapter and the object of this thesis. It discusses arithmetic operations in LNS in general. Here data representation in FLP format and a possible way of data representation in LNS format are shown [3].

Chapter 2 deals with the basic conversion technique of data into LNS format. A discussion on how the error develops and several other methods of conversion are given here. An analysis of different possible formats of LNS data representation and a comparison of these formats with those of IEEE std. 754-1985 are presented in this chapter.

Chapter 3 discusses method of conversion using linear correction factor and presents hardware models for 32 bits and 64 bits of representation for both the ways of data conversion. Simulation model programs with linear correction factors are discussed. Different ROM configurations with their corresponding conversion errors are given and methods of reducing the ROM size are discussed. A mathematical model proving the validity of simulation results is presented here. A model only with ROM and without any multiplier is also cited at the last article of this chapter. Various graphs are added to illustrate the experimental findings.

Chapter 4 concentrates on simulation model programs that apply various correction factors to achieve the minimum value of the maximum conversion error. This chapter shows the effect of non-linear conversion factor. It also shows the effect of non-linear correction factor on errors and compares the result with the errors those have been produced by applying linear correction factor. Arithmetic processing, accumulated errors for format conversion and several other processing results are also discussed.

Chapter 5 discusses addition and subtraction of LNS number. Table construction technique using Taylor's series first order approximation is also described here. In the article 5.3 the processing involving denormalized numbers is also discussed.

Chapter 6 discusses the various sources of error during FLP to LNS and LNS to FLP conversion. Discussion about speed improvement with reasonable assumptions are given in article 6.2. Application of LNS number in the field of evaluating trigonometrical identities, geometric co-ordinate transformation and fast Fourier transformation are also given.

Chapter 7 is the last chapter which contains conclusion and suggestions for further research.

1.2 **Objective of the thesis**

This thesis has manifold purposes. As the interest in favour of using LNS is increasing day by day, general properties of LNS and various mathematical operations in this number system shall be studied.

At present the numbers inside the memory of a computer system are stored in a fixed point or floating point representation. So suitable methods for converting numbers into LNS format shall be

studied. This includes hardware methods [2] so that a very fast conversion can be accomplished or use of a ROM [1,3,10,18] so that much longer word length can be converted. Special attention has been paid to the conversion technique that uses a ROM due to the fact that the packing density of ROM chips are generally increasing with further reduction of access time.

Accuracy calculation for 32 bits and 64 bits conversion of FLP numbers into LNS numbers and vice versa shall be studied. Suitable ROM configuration shall be determined to achieve a prestated accuracy level. Partial correction shall be done by adding a suitable correction factor and hence the effect of the multiplicand bit length over the accuracy shall be studied.

A trial and error method shall be used to determine the new value of the correction factor so that the error can be minimized to the smallest possible value. This method of minimizing the value of the maximum conversion error is termed as non-linear correction factor method.

The way of evaluating mathematical functions shall be explained.

It is also an objective to calculate the speed improvement for elementary mathematical operations. The possibility of using LNS system in the geometric coordinate transformation and fast Fourier transformation shall be included as a part of this thesis work.

1.3 **Arithmetic operation in LNS**

In the LNS system a number x can be represented as

$$x = r^{Ex}$$

where both x and Ex can be a positive or negative number. The radix point of Ex can be varied so that the number represented by x under

LNS can have a wide range with high accuracy.

If we assume 3 numbers x, y and z all represented in LNS and z is the result of the arithmetic operations on x and y then multiplication and division processes can be implemented by binary adder and subtractor respectively, and an XOR gate to decide the sign of the operation so if we allocate Sz as the sign of the result z then the following can be summarized.

1) Multiplication

$z = x * y,\ Ez = Ex + Ey,\ Sz = sign(x) \oplus sign(y)$

2) Division

$z = x / y,\ Ez = Ex - Ey,\ Sz = sign(x) \oplus sign(y)$

Addition and subtraction of two numbers represented in LNS is more difficult to perform. These two operations can be expressed by the following mathematical relationship

3) Addition

$z = x + y,\ Ez = Ex + Fa(v),\ v = Ex - Ey$

$x >= y,\ Fa(v) = \log_r(1 + r^{-v})$

4) Subtraction

$z = x - y,\ Ez = Ex + Fb(v),\ v = Ex - Ey$

$x >= y,\ Fb(v) = \log_r(1 - r^{-v})$

Square or square-root calculations are very simple in LNS system and can be performed by mere left or right shifting. Log_e and log_{10} operations can be done by dividing with the respective constants [3]. Evaluation of exponents and power can be made with some prespecified steps[3].

As it can now be seen that the addition and subtraction processes

are not very convenient in LNS system and development of a suitable process for faster addition and subtraction of two LNS number is still a major obstacle in the development of LNS arithmetic processing system.

During the addition and subtraction processes we see that the exponent is either increased or decreased depending on the value of Ex and Ey and the corresponding Fa(v) or Fb(v) is derived from a ROM look-up table.

Once the basic four operations (addition, subtraction, multiplication and division) procedures are set under LNS environment, it can also perform many other mathematical operations like square root or squaring in an elegant way.

1.4 **Floating point data format**

IEEE std. 754-1985 specifies single and double format of floating point representation with their extensions [7]. Single format 32 bits floating point number comprises a 23 bits mantissa field, a 8 bit exponent field and a sign bit and for double format 64 bits floating point number comprises a 52 bits mantissa, a 11 bits exponent field and a sign bit. The mantissa is always normalized and the leading 1 is not stored so that it remains as a hidden bit and the mantissa is effectively extended by one bit more. The exponent is biased by 127 for single precision and is biased by 1023 for double precision floating point format so their actual exponent (e) values are e-127 and e-1023 respectively. Therefore for 32 bits representation a real number N can be represented by

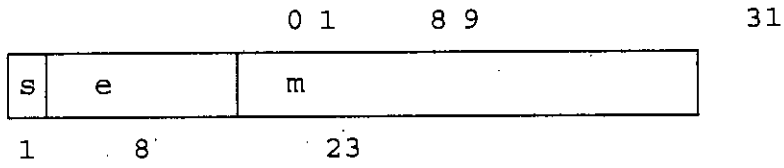
$$N = (-1)^s * 2^{(e-127)} * (1.m)$$

where s represents the sign bit and m represents the mantissa and e is the exponent ($0 < e < 255$). For example the number N = 22.625

is represented by

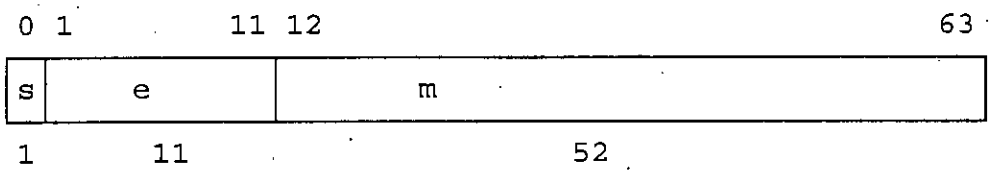
$$N = 0\text{-}10000011\text{-}01101010000000000000000$$

Non-zero floating point numbers in the 32 bit format have magnitude ranging from $2^{-126} * (1.0)$ to $2^{127} * (2 - 2^{-23})$ i.e. from $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ approximately.



e	m	value	name
255	not 0	none	NaN (not a number)
255	0	$(-1)^s$*infinity	infinity
1..254	any	$(-1)^s * 2^{e-127} * (1.m)$	Normalized number
0	0	$(-1)^s * 0$	Zero
0	not 0	$(-1)^s * 2^{-126} * (0.m)$	Denormalized number

Fig. 1-1 : IEEE standard 754-1985 (data type-single precision)



e	m	value	name
2047	not 0	none	NaN (not a number)
2047	0	$(-1)^s$*infinity	infinity
1..2046	any	$(-1)^s*2^{e-1023}*(1.m)$	Normalized number
0	0	$(-1)^s*0$	Zero
0	not 0	$(-1)^s*2^{-1022}*(0.m)$	Denormalized number

Figure 1-2 : IEEE standard 754-1985 (data type-double precision)

1.5 **Logarithmic number system**

A proposal for logarithmic number system (LNS) representation follows the same floating point data format with different interpretation for each section. The left most bit still represents the sign bit s. The e portion defined as the exponent in the floating point data format, indicates the integer bits of the base two logarithm number representation. The portion e can also be coded as 8 bit- excess 127 code with the actual value computed as e - 127. The mantissa[1] portion M is interpreted as the fraction portion of this 32 bit word of e and M with binary point located

[1]When an FLP number represented in IEEE format is changed to LNS format only the mantissa is changed. This change can be marked by changing the small letter m to its capital form M indicating that the same number N has been changed from FLP format to LNS format.

between bit 8 and bit 9. The non-zero number x is actually coded as

$$N = (-1)^{s} * 2^{(e - 127) + 0.M}$$

The basic difference between the LNS and the FLP representation is that the M portion in LNS is the exponent to the base 2, while in FLP it is the binary number with the hidden bit 1. The normalized hidden bit 1 will make the FLP to LNS and LNS to FLP conversion very simple. The non-zero and non-negative data range covered by this 32 bit LNS data format is from $2^{(-126 + 2\hat{}-23)}$ to $2^{127 + (1 - 2\hat{}-23)}$ i.e. from $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ approximately.

CHAPTER 2

CONVERSION TECHNIQUE AND LOGARITHMIC NUMBER SYSTEM

2.1 **Basic conversion technique**

The base 2 format of a real number can be converted to its approximate logarithmic number as described in [8]. Here we represent the fixed form representation of the logarithm to the base 2 of a number N by lg(N) and the approximate logarithm of the same number by lgA(N) then the approximate logarithm of the number N can be calculated as given below.

1) The characteristics of lgA(N) is equal to the number of bits between the left most 1 (the most significant bit) and the radix point, thus 3 for the numbers 8 through 15 and 5 for the numbers 32 through 63.

2) The approximate fraction of lgA(N) is formed by the bits following the left most 1-bit of N.

The necessary conversion can be implemented by using a shift register R and a counter C (fig. 2-1) in which the exponent value can be evaluated from the contents of the C once the conversion operation is completed and R contains the approximate fraction [8].

The binary number whose base two logarithm is to be calculated is loaded into the register R. The counter C is always initialized

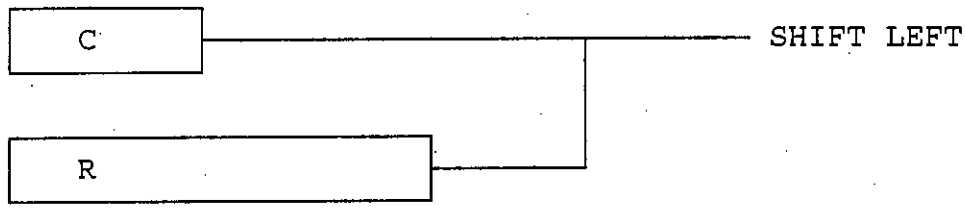


Fig. 2-1 : Basic circuit for LNS conversion using a register R and a counter C

with the register length of R and is decremented by 1 with each left shift of R. If k is the initial position of the left most 1 bit in R, then after making necessary left shifts when the left most 1 bit is exited from R, the contents of the counter register C gives the value of the exponent of the base two logarithm number and the approximate fraction is available in the register R.

A few numbers N and their related lgA(N) are given in table 2-1.

Table 2-1 : Approximate logarithm representation of binary numbers

N (binary)	N (decimal)	lgA(N)
00001000	8	11.00
00001001	9	11.001
.	.	.
.	.	.
00011000	24	100.1000
.	.	.
.	.	.
11111110	254	111.1111110
11111111	255	111.1111111

Thus for the number N = 110110.101 the approximate logarithm is lgA(N) = 101.10110101. This approximate method gives us considerable error as shown below.

$$N = 110110.101 = 54.625$$
$$lgA(N) = 101.10110101 = 2^{5.70703125} = 52.23812646$$

So the error equals to (54.625 - 52.23812646) 2.38687354 which has been produced due to the approximate logarithmic representation of the binary number N. Of course the error produced is large and cannot be accepted for most of the computing applications. The conversion error produced in term of the exponent error can be calculated as follows :

N = number to be converted
then X = ln(N)/ln(2.0)
A = floor(X)

the error can be calculated by the expression

$$error = (X-A) - \left(\frac{N}{2^A} - 1.0\right)$$

The first term X-A calculates the fractional part of the exponent. So if we are investigating N = 54.625 then $N = 2^{5.771489469}$. Therefore X-A = 5.771489469 - 5.0 = 0.771489469. While calculating the second portion, we assume that the number N is loaded into a register having integer and fractional parts. We then begin shifting the whole register to right until the l.s.b. of the integer part is 1 and all of its other bits are zero. The bits configuration of the fractional part gives us $\frac{N}{2^A} - 1.0$ and it is clear that the total no of right shifts necessary is equal to A. This gives us the maximum value of error = 0.0860529

when X - A = 0.5360529.
A list is given below for further clarification.

Number (N)	X-A	error (as exponent)
32.0	0.0	0.0
32.8	0.03562391	0.01062391
40.0	0.321928095	0.071928095
46.4	0.5360529	0.0860529
54.625	0.771489469	0.064458219
56.0	0.807354922	0.057354922
64.0	0.0	0.0

Therefore the total conversion error produced in representing 54.625 becomes $54.625 - 2^{(5.771489469 - 0.064458219)} = 2.386873537$ as we have seen above.

A reasonable amount of correction can be achieved by dividing the whole range of the fraction portion of number into several divisions and applying necessary correction factor[8].

2.2 **Common conversion techniques**

Methods that convert a non-LNS number into LNS number and vice-versa can be broadly classified into two groups :-

a) Techniques that apply complex hardware devices for conversion are known as hardware conversion technique.

b) Techniques that employ either firmware based data/instruction or a complete software based method are known as software method.

The hardware method uses programmable logic array (PLA) [2] to generate precise logarithmic conversion. Although this method is accurate for small word size but is unsuitable for word size of 32 bits and higher. The bit requirement of the PLA can be reduced by arranging the same magnitude of errors in one group i.e. by differential grouping arrangement [2]. This method is faster than the software method.

An alternate approach is to use a ROM look-up table [3] to convert the mantissa which is particularly promising due to the recent advancement in packing density and reduced access time. Here we have considered our binary data in IEEE floating point format and used a ROM look-up table for data conversion. The sign bit and the exponent portion of the FLP number does not change during the conversion process but the mantissa is converted according to the ROM look-up table.

While converting FLP numbers into LNS numbers the mantissa of the FLP number is divided into two portions, the first portion consists of the most significant bits (msb) and the other portion consists of the remaining least significant bits (lsb). The msb portion decides the base value of the logarithm number and the remaining portion decides the correction factor that is to be added with the base value of the logarithm number to get the corrected logarithm value of the mantissa of the floating-point number.

The output of the ROM can be divided into two portions or two separate ROMs of which the first ROM is for the base value and the second ROM for the correction value can be used.

To calculate the correction factor the second portion of the ROM output is multiplied with the lsb portion of the mantissa (multiplier) of the FLP number. The length of the multiplication process can be reduced by increasing the ROM address bits which means the reduction of the multiplier bits.

A reduction in multiplication process increases the accuracy but requires a larger ROM size. If the multiplier bits are zero then the logarithmic conversion is exact and the first portion of the ROM output exactly represents the LNS value. In this way a large number of points are available (e.g. 1024 points if address bits are 10) which gives the exact logarithmic conversion value. As the ROM output value is limited so there will be always some truncation error even the multiplier value is zero.

Another method uses high capacity ROM only and converts FLP to LNS and vice-versa without any correction factor. This method although much faster but less accurate than other method and requires high capacity ROM. This will be investigated at a latter stage (in chapter 3).

2.3 **IEEE std. 754 - 1985 and Logarithmic Number System**

Although the concept and use of LNS is common, there is no standard that clearly specifies the LNS number format as the IEEE std. 754-1985 does for FLP number system.

A real value x can be represented in a sign/logarithm representation X by

$$x = r^{X_l} \cdot (-1)^s$$

of which X is composed of two parts: X_l that has F bits after the radix point which is n-1 bit representation of the base r logarithm of the absolute value of X and s is the sign bit, 0 when x is positive and 1 when x is negative. Numbers when represented in this fashion can never represent zero, denormalized values, infinities or NaN.

IEEE std. 754 completely defines the location of the sign bit,

width of the exponent and mantissa portion, method of representation of the exponent but for LNS there is disagreement whether sign should be on the left or right of the word, disagreement on the representation of X_l whether it should be represented in one's complement, two's complement or offset binary, disagreement about the representation of zero.

In this article some proposed LNS number formats [9] and its denormalized representation shall be discussed so that very small numbers can be represented.

Different formats of LNS numbers as proposed in [9] have four layers. In all layers the sign bit s is placed on the right most bit, utilize 2's complement binary to encode the signed fixed point value for X_l and use base r=2. The width of the exponent and mantissa are kept identical to the IEEE suggestions. IEEE std. 754 does not provide any layer but proposal for various layers exists in LNS system to meet individual demand or application requirement.

Among the four layers, layer 0 does not provide the representation for zero, denormalized values, infinities or NaNs. Underflow and zero in layer 0 are represented with the smallest X_l value. if Δ_i is the smallest normalized X_l permitted in layer i then in the single precision system $\Delta_0 = -128.0$ and so the worst possible error is 2^{-128}. Overflow is treated by the largest $\Delta_1 = 127.9999998808\ (\ 127 + 1 - 2^{-23}\ )$.

Layer 1 treats the minimum X_l value (-128.0 for single precision system) as an exact representation for zero so the X_l value that represents the minimum positive value is $\Delta_1 = \Delta_0 + 2^{-F}$. For a single precision system $\Delta_1 = -127.9999998808$ and the worst possible error due to underflow is 2^{-129}. Layer 1 does not represent denormalized values, infinities or NaN. Overflow is treated as in layer 0.

Denormalized values in LNS system is represented in layer 2. The smallest possible X_1 is still used to represent 0 which is 2^{-128}. The smallest normalized value is represented by $X_1 = \Delta_2$ which is -127.0 in a single precision system. Denormalized values are represented by any values between 0 ($2^{-128.0}$) and $2^{-127.0}$ so X_1 = -127.9999992847 is a denormalized value. Layer 2 does not represents infinities and NaNs and so overflow is treated as in layer 0.

Layer 3 is enhanced by the representation of NaNs and infinities. The smallest normalized number in layer 3 is twice the size of the smallest normalized value in layer 2 so $\Delta_3 = \Delta_2 + 1$.

In IEEE 754 the 2^{23} denormalized points are between 0 and 2^{-126}. This concept is also applied while evaluating LNS denormalized numbers but the numbers are represented between 0 and 2^{-127} and hence the lowest possible value that can be represented in FLP and LNS denormalized form are nearly same (1.4013E-45 vs. 4.8565E-46).

The process of evaluating the denormalized value begins with the evaluation of X_d which equals to $X_1 + 1$. If it is found that $-127.0 < X_1 < -128.0$ then the value of the denormalized number is evaluated by the expression $2^{Xd} - 2^{-127.0}$. The subtrahend is constant ($2^{-127.0}$) but $X_d = f(X_1) = X_1 + 1$. So this maps each denormalized representation into a unique encoding in the range of the bit pattern reserved for the denormalized values, known as the forbidden zone $\Delta_0 \leq X_1 < \Delta_2$. Therefore a value of X_1 = -127.9999992847 is treated as a denormalized value and its magnitude is evaluated by $2^{(-127.9999992847 + 1.0)} - 2^{-127.0} = 2.91695E-45$.

CHAPTER 3

CONVERSION USING LINEAR CORRECTION FACTOR

3.1 Techniques for transferring FLP to LNS format and vice-versa

As the arithmetic operands are first stored in the FLP format, our main concern is to transfer FLP numbers into LNS format before doing the arithmetic operations and once the arithmetic operations are completed the data from LNS format is to be converted back to the FLP format. Both the operations should be made fast and accurate enough. To demonstrate what happens during conversion we have selected two ROMs each having a capacity of 524288 bits [$2^{14}*(23+9)$=16 Kwords] with 14 address lines and the output word length is 32 bits long. The output bits are divided into two portions, the first 14 bits give the base value (logarithm or anti-logarithm) and the remaining 9 bits gives us the value of the correction factor (∆lgm in table 3-1 and ∆m in table 3-2). The correction factor takes the role of multiplicand during the multiplication process. The first 14 bits from 23 bits mantissa is selected to generate the base logarithm number and the remaining 9 bits are used as the multiplier to generate the total value of correction factor (known as *add_factor*[2] in simulation model program m_lgm.c and lgm_m.c). Correction factor is the product of multiplier and multiplicand that is to be added with the base value. While transferring the LNS number to FLP number the same process is adopted but with the second ROM. Correction process is

---

[2]*add_factor is the total value of additions that is to be made with the base values. This is also a variable mentioned in simulation program.* Variables or expressions used in simulation programs whenever mentioned in the text, is mentioned in *italics*

simple because always positive correction factors are added to the base number in both the cases.

3.2 **32 bits transformation analysis**

Tables 3-1 and 3-2 together with their associated calculations in sheets 3-1 and 3-2 respectively show the transformation of FLP number to LNS number and from LNS number to FLP number when the numbers are represented in 32 bits format. Table 3-1 and sheet 3-1 give the details for FLP to LNS conversion. Various segments have been selected in column 1 so that the value of m = 0.1977539063 is the initial value of segment no. 3240. Total number of segments are determined by the length of the bits covered in the ROM. So when ROM bit length is selected as 14 as in this case, total no. of segments become (2^{14}) 16384 ranging from 0 to 16383. Similarly m = 0.9998168946 is the initial value of segment no. 16381. Column 2 is the binary representation of column 1 showing the address generated which corresponds to the first 14 bits of the mantissa of the floating point number. In the third column lg(m) represents the base 2 logarithm of the first columns and their corresponding binary representation. The FLP number is stored in IEEE format so the hidden 1 is added with m. The values represented in Δlgm (column 4) represents the correction factor for adding each bit to the base number. The number Δlgm is very small so only last significant 12 out of 32 bit configuration needs to be saved. All of its other preceding bits are zero and need not to be saved. Column 5 gives us the binary representation of six random numbers generated. These numbers are stored in a 23 bit mantissa as per IEEE single precision number system. Only the least significant 12 bits are mentioned. The complete bit pattern can be found by cascading the corresponding bit pattern in column 2. For example the first element of column 6 is 010-111111111 so the complete bit pattern generated is 00000000000010111111111. This also implies that the mantissa of the random number we are investigating is 1.182896259E-04 as can be seen in sheet 3-1. Column 6 gives us the

bit pattern that we have produced after converting the value of column 5 by using the ROM table data as available in column 3 and 4. Here we also like to mention that lg(m) means that log base 2 of m is calculated through a computer with maximum possible precision (m is enclosed in parenthesis) whereas lgm is calculated through ROM table (m is not enclosed in parenthesis). Column 7 gives us the difference between the bit pattern of lg(m) and lgm. This is more explained in sheet 3-1. The average loss of bits is less than 1 because it is seen that in many cases there is no error i.e. no bit has been dropped.

For 32 bits LNS to FLP conversion the reverse process is adopted as shown in table 3-2 and sheet 3-3. The numbers are assumed existing in LNS format and are given in column 1 under lgm. Column 2 represents the address produced against each value of lgm and their corresponding representation in binary. Only first 14 bits are mentioned and all the other following bits are zero. In the third column antilg(lgm) represents the base 2 antilogarithm value of the column 1. The value of Δm indicates the correction factor each bit that is to be added with the value of antilg(lgm) under column 3 for each bit increment of the address generated for the value under investigation. For example the value under investigation is 0.0001449584962 which has produced the address of 00000000000010011000000 so we have to multiply 0.8263832031E-07 by $(2^8 + 2^7)$ and have to add the product with the base value of 0.00008461627. As we can see that the multiplicand bit is represented by a bit configuration of relatively smaller size (only by 9 bits) still the conversion accuracy is relatively high (this excludes the truncation error). Here we also see that the average loss of bits is less than one because it is seen that such case exists where no bits has been dropped.

A block diagram showing the details of conversion and anti-conversion is given in fig. 3-1.

Table 3-1

m to $\log_2 m$ conversion data table (RAL 14 ML 23)

m	address of m first 14 bits	lg(m) = $\log_e(1.0 + m)/\log_e(2.0)$	
0.0	00000000000000	0.0	0--------------------0
0.6103515625E-05	001	8.805242111E-05	00000000000001011100010
1.2207031250E-04	010	1.760994828E-04	00000000000010111000101
1.8310546880E-04	011	2.641411570E-04	00000000000100010100111
2.4414062510E-04	00000000000100	3.521774731E-04	00000000000101110001010
0.1977539063	00110010101000	0.26033152	01000010101001010001011
0.1978149415	001	0.26040503	01000010101010011110011
0.1978759766	010	0.26047854	01000010101011101011100
0.1979370118	011	0.26055205	01000010101100111000101
0.1979980469	00110010101100	0.26062556	01000010101110000101101
0.400390625	01100110100000	0.48582931	01111100010111110100111
0.4004516602	001	0.4858921863	01111100011000110110111
0.4005126953	010	0.4859550611	01111100011001111000110
0.4005737305	011	0.48601793	01111100011010111010101
0.4006347656	01100110100100	0.48608080	01111100011011111100101
0.6005859375	10011001110000	0.67860014	10101101101110001011110
0.6006469727	001	0.67865515	10101101101111000101100
0.6007080078	010	0.67871016	10101101101111111111001
0.600769043	011	0.67876517	10101101110000111000110
0.6008300781	10011001110100	0.67882018	10101101110001110010100
0.80078125	11001101000000	0.84862294	11011001001111110101101
0.8008422852	001	0.84867184	11011001010000101000111
0.8009033203	010	0.84872073	11011001010001011100001
0.8009643555	011	0.84876963	11011001010010001111011
0.8010253906	11001101000100	0.84881852	11011001010011000010101
0.9997558594	11111111111100	0.99982388	11111111111101000111010
0.9998168946	11111111111101	0.9998679113	11111111111101110101011
0.9998779297	11111111111110	0.99991194	11111111111110100011101
0.9999389649	11111111111111	0.9999559718	11111111111111010001110

cont.

Table 3-1 (cont.)

Δlgm*E-07 ls 12 of 32	address generated	lgm	Error
1.71977385 001011100010 1.719669174 001011100010 1.719563949 001011100010 1.719459299 001011100010	010-111111111	00000000000100010100110	nil
1.435742188 001001101000 1.435742188 001001101000 1.435742188 001001101000 1.435742188 001001101000	001-100000000	01000010101011000101000	nil
1.228125 001000001111 1.227929688 001000001111 1.227929688 001000001111 1.227929688 001000001111	001-110000000	01111100011001101000010	nil
1.074414063 000111001101 1.074414063 000111001101 1.074414063 000111001101 1.074414063 000111001101	000-000001111	10101101101110001101011	1 bit
0.955078125 000110011010 0.954882812 000110011010 0.955078125 000110011010 0.954882812 000110011010	010-000001100	11011001010001011101011	1 bit
0.859986328 000101110001 0.859935546 000101110001 0.859996093 000101110001	101-000100000	11111111111101111000010	1 bit

Table 3-2

$\log_2 m$ to m conversion data table (RAL 14 ML 23)

lgm	address of lgm	antilg(lgm) = $2^{lgm} - 1.0$	
0.0	00000000000000	0.0	0--------------------0
0.00006103515625	001	0.00004230724	00000000000000101100010
0.0001220703125	010	0.00008461627	00000000000001011000101
0.0001831054688	011	0.00012692709	00000000000010000101000
0.0002441406251	00000000000100	0.0001692397	00000000000010110001011
0.1977539063	00110010101000	0.146911368	00100101100110111111101
0.1978149415	001	0.146959891	00100101100111110010100
0.1978759766	010	0.147008416	00100101101000100101011
0.1979370118	011	0.147056942	00100101101001011000011
0.1979980469	00110010101100	0.147105471	00100101101010001011010
0.400390625	01100110100000	0.319865230	01010001111000101011000
0.4004516602	001	0.319921070	01010001111001100101100
0.4005126953	010	0.319976912	01010001111010100000000
0.4005737305	011	0.320032757	01010001111011011010101
0.4006347656	01100110100100	0.320088604	01010001111100010101001
0.6005859375	10011001110000	0.516332286	10000100001011100101101
0.6006469727	001	0.516396438	10000100001100101000111
0.6007080078	010	0.516460539	10000100001101101100001
0.600769043	011	0.516524750	10000100001110101111011
0.6008300781	10011001110100	0.51658891	10000100001111110010101
0.80078125	11001101000000	0.742044225	10111101111101101001110
0.8008422852	001	0.742117926	10111101111110110111000
0.8009033203	010	0.74219630	10111110000000000100010
0.8009643555	011	0.742265338	10111110000001010001100
0.8010253906	11001101000100	0.742339048	10111110000010011110111
0.999755859	11111111111100	0.999661577	11111111111010011101001
0.999816894	11111111111101	0.999746178	11111111111011110101110
0.999877929	11111111111110	0.999830781	11111111111101001110100
0.999938964	11111111111111	0.999915389	11111111111110100111010

cont.

Table 3-2 (cont.)

Δm*E-07	ls 12 of 32	address generated	antilglgm	error
0.8263132813	000101100010			
0.8263482422	000101100010			
0.8263832031	000101100010	010-011000000	00000000000001101001001	1 bit
0.8264181641	000101100010			
0.94771484	000110010111	000-100000001	00100101100111011001001	nil
0.94775391	000110010111			
0.94777344	000110010111			
0.94783203	000110010111			
1.090625	000111010100			
1.09066406	000111010100			
1.09072265	000111010100	010-000010100	01010001111010100010010	nil
1.09076172	000111010100			
1.25296875	001000011010			
1.25197265	001000011001	001-100100000	10000110001101001110110	1 bit
1.25412109	001000011010			
1.253125	001000011010			
1.43947265	001001101010	000-001000001	10000100001101001110110	nil
1.43953125	001001101010			
1.43960937	001001101010			
1.43964844	001001101010			
1.65236328	001011000101	100-110000000	11111111111011011111001	1 bit
1.65240234	001011000101			
1.6525	001011000101			

Sheet 3-1

```
1) value investigating      = m = 1.82986259E-04
   address generated        = 0000 0000 0000 1011 1111 111
       delta lgm            = 0000 0000 0000 0000 0000 0010 1110 0010
   multiplier               =                           1 1111 1111
   add_factor               = 0000 0000 0000 0101 1100 001
   base lgm                 = 0000 0000 0000 1011 1000 101
       lgm                  = 0000 0000 0001 0001 0100 110
   lg(m) = 2.639692048E-04  = 0000 0000 0001 0001 0100 110
              (no error, lgm and lg(m) both have same bit pattern)

2) value investigating      = m = 0.1978454591
   address generated        = 0011 0010 1010 0110 0000 000
       delta lgm            = 0000 0000 0000 0000 0000 0010 0110 1000
                                                     1 0000 0000
    add_factor              = 0000 0000 0000 0010 0110 1000 0000 0000
    base lgm                = 0100 0010 1010 1001 1110 011
       lgm                  = 0100 0010 1010 1100 0100 111
    lg(m) = 0.26044179      = 0100 0010 1010 1100 0100 111 (no error)

3) value investigating      = m = 0.400497436
   address generated        = 0110 0110 1000 0111 0000 000
       delta lgm            = 0000 0000 0000 0000 0000 0010 0000 1111
                                                     1 1000 0000
    add_factor              = 0000 0000 0000 0011 0001 0110 1000 0000
    base lgm                = 0111 1100 0110 0011 0110 111
       lgm                  = 0111 1100 0110 0110 1000 010
   lg(m) = 0.48593934       = 0111 1100 0110 0110 1000 010 (no error)

4)  value investigating     = m = 0.6005877
    address generated       = 1001 1001 1100 0000 0001 111
       delta lgm            = 0000 0000 0000 0000 0000 0001 1100 1101
                                                     0 0000 1111
    add_factor              = 0000 0000 0000 0000 0001 1011 0000 0011
    base lgm                = 1010 1101 1011 1000 1011 110
       lgm                  = 1010 1101 1011 1000 1101 011
    lg(m) = 0.67860175      = 1010 1101 1011 1000 1101 100 (error on last bit)

5)  value investigating     = m = 0.8009047508
    address generated       = 1100 1101 0000 1000 0001 100
       delta lgm            = 0000 0000 0000 0000 0000 0001 1001 1010
                                                     0 0000 1100
    add_factor              = 0000 0000 0000 0000 0001 0011 0011 1000
    base lgm                = 1101 1001 0100 0101 1100 001
       lgm                  = 1101 1001 0100 0101 1101 010
    lg(m) = 0.8487218796    = 1101 1001 0100 0101 1101 011 (error on last bit)
```

The last calculation is dropped to avoid mere repetition.

Sheet 3-2

```
1) value investigating          = lgm = 0.0001449584962
   address generated            = 0000 0000 0000 1001 1000 000
        delta m                 = 0000 0000 0000 0000 0000 0001 0110 0010
                                                            0 1100 0000
    add_factor                  = 0000 0000 0000 0001 0000 1001 1000 0000
    base antilglgm              = 0000 0000 0000 0101 1000 101
        antilglgm               = 0000 0000 0000 0110 1001 001
   antilg(lgm) = 1.0048262E-04  = 0000 0000 0000 0110 1001 010 (on last bit)
2) value investigating          = lgm = 0.1977845
   address generated            = 0011 0010 1010 0010 0000 001
        delta m                 = 0000 0000 0000 0000 0000 0001 1001 0111
                                                       1 0000 0001
    add_factor                  = 0000 0000 0000 0001 1001 1000 1001 0111
    base antilglgm              = 0010 0101 1001 1011 1111 101
        antilglgm               = 0010 0101 1001 1101 1001 001
    antilg(lgm) = 0.14693569    = 0010 0101 1001 1101 1001 001 (no error)
3)  value investigating         = lgm = 0.400515
    address generated           = 0110 0110 1000 1000 0010 100
        delta m                 = 0000 0000 0000 0000 0000 0001 1101 0100
                                                            0 0001 0100
    add_factor                  = 0000 0000 0000 0000 0010 0100 1001 0000
    base antilglgm              = 0101 0001 1110 1010 0000 000
        antilglgm               = 0101 0001 1110 1010 0010 010
    antilg(lgm) = 0.3199790206  = 0101 0001 1110 1010 0010 010 (no error)
4)  value investigating         = lgm = 0.6006813049
    address generated           = 1001 1001 1100 0110 0100 000
        delta m                 = 0000 0000 0000 0000 0000 0010 0001 1001
                                                       1 0010 0000
    add_factor                  = 0000 0000 0000 0010 0101 1100 0010 0000
    base antilglgm              = 1000 0100 0011 0010 1000 111
        antilglgm               = 1000 0100 0011 0100 1110 101
    antilg(lgm) = 0.5164325245  = 1000 0100 0011 0100 1110 110 (on last bit)
5)  value investigating         = lgm = 0.8007889986
    address generated           = 1100 1101 0000 0000 1000 001
        delta m                 = 0000 0000 0000 0000 0000 0010 0110 1010
                                                            0 0100 0001
    add_factor                  = 0000 0000 0000 0000 1001 1100 1110 1010
    base antilglgm              = 1011 1101 1111 0110 1001 110
        antilglgm               = 1011 1101 1111 0111 0011 100
    antilg(lgm) = 0.7420535816  = 1011 1101 1111 0111 0011 100 ( no error)
6)  value investigating         = lgm = 0.9998016357
    address generated           = 1111 1111 1111 0011 0000 000
        delta m                 = 0000 0000 0000 0000 0000 0010 1100 0000
                                                       1 1000 0000
    add_factor                  = 0000 0000 0000 0100 0010 0111 1000 0000
    base antilglgm              = 1111 1111 1110 1001 1101 001
        antilglgm               = 1111 1111 1110 1101 1111 100
    antilg(lgm) = 0.9997250276  = 1111 1111 1110 1101 1111 101 ( on last bit)
```

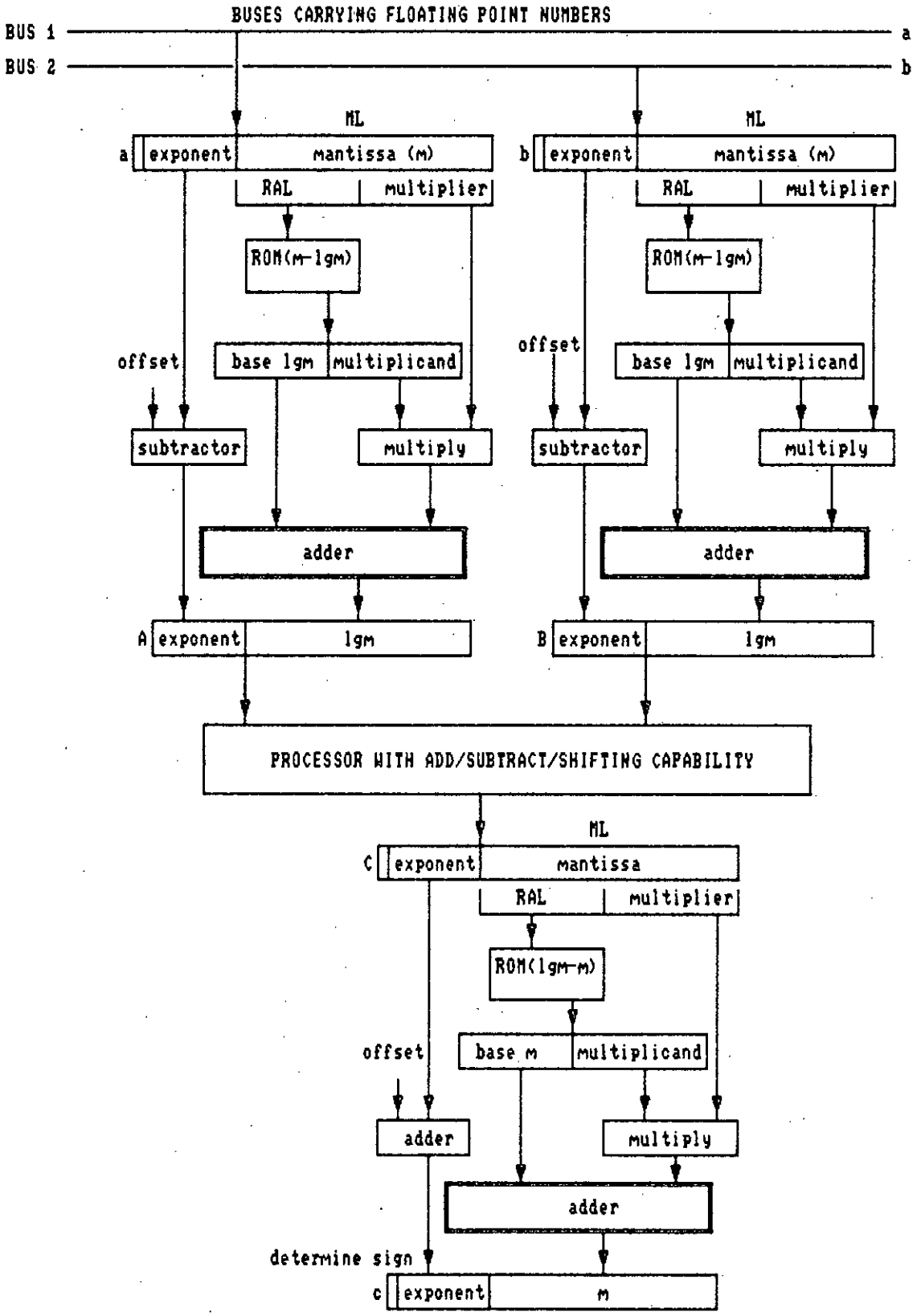


Fig 3-1: Block diagram of LNS processing

(WITH DETAILS OF FLP TO LNS AND LNS TO FLP CONVERSION)

3.3 **Simulation model program**

We have also written a software routine as given in the listing of program m_lgm.c (the listing of all programs those are mentioned here are given in the Appendix) to process the FLP to LNS conversion of numbers under simulation environment. Numbers can be represented by any mantissa bit length (defined as ML in the program) but we have done exhaustive checking up to 64 bits length. To find the *add_factor* we have multiplied the individual bits (ML - RAL no of bits) by the suitable multiplying factor (*mul_factor*) and finally accumulating the result as shown in the function *lin_int_values()* of m_lgm.c.

The program runs through external number of loops as decided by the ROM size as may be stated under the definition RAL. On each external loop the program calculates the higher log base 2 value by *new.lgm* and the lower log base 2 value by *old.lgm*. Next the variable *difference* is calculated which must be added for each bit of increment. The *old_m* is incremented by one bit value each time in the function *lin_int_values()* to cover all possible number configuration from 0.0 to 1.0. To calculate the intermediate log base 2 values each bit is treated separately to simulate the hardware multiplication effect instead of adding the whole value of correction at one time.

To calculate the *add_factor* it is necessary to multiply the second portion of the binary representation of old_m (first portion is covered by RAL) with Δlgm so the second portion is treated as the integer and as the multiplier in the multiplication process.

Next *error* is calculated and *max_con_error* (MCE) is replaced by *error* if it is found higher than the *max_con_error*. The whole process is repeated until the most exterior loop is exhausted.

A typical value of 2.50E-13 was achieved for 64 bits FLP to LNS

conversion with 1 Mbytes of ROM (RAL 20). The other results are also stimulating e.g. when RAL and ML values were set for 12 and 23 (for 32 bit configuration) respectively, the MCE was reported 1.07E-08 which occurred at input value of m = 0.000122070312. Error pattern can be seen in graph 3-1.

We have also studied the effect of ROM output bits length (choice 2 of m_lgm.c) for FLP to LNS conversion. When we set RAL 12, ML 23 and DL 20, MCE was reported to 1.09471E-08 but when DL was set to 30 or higher the MCE reduces to its minimum value as expected. Table 3-3 gives us the value of other MCEs for various DL settings.

Table 3-3 : Effect of multiplicand bit length on MCE
(m-lgm conversion, RAL 12 ML 23)

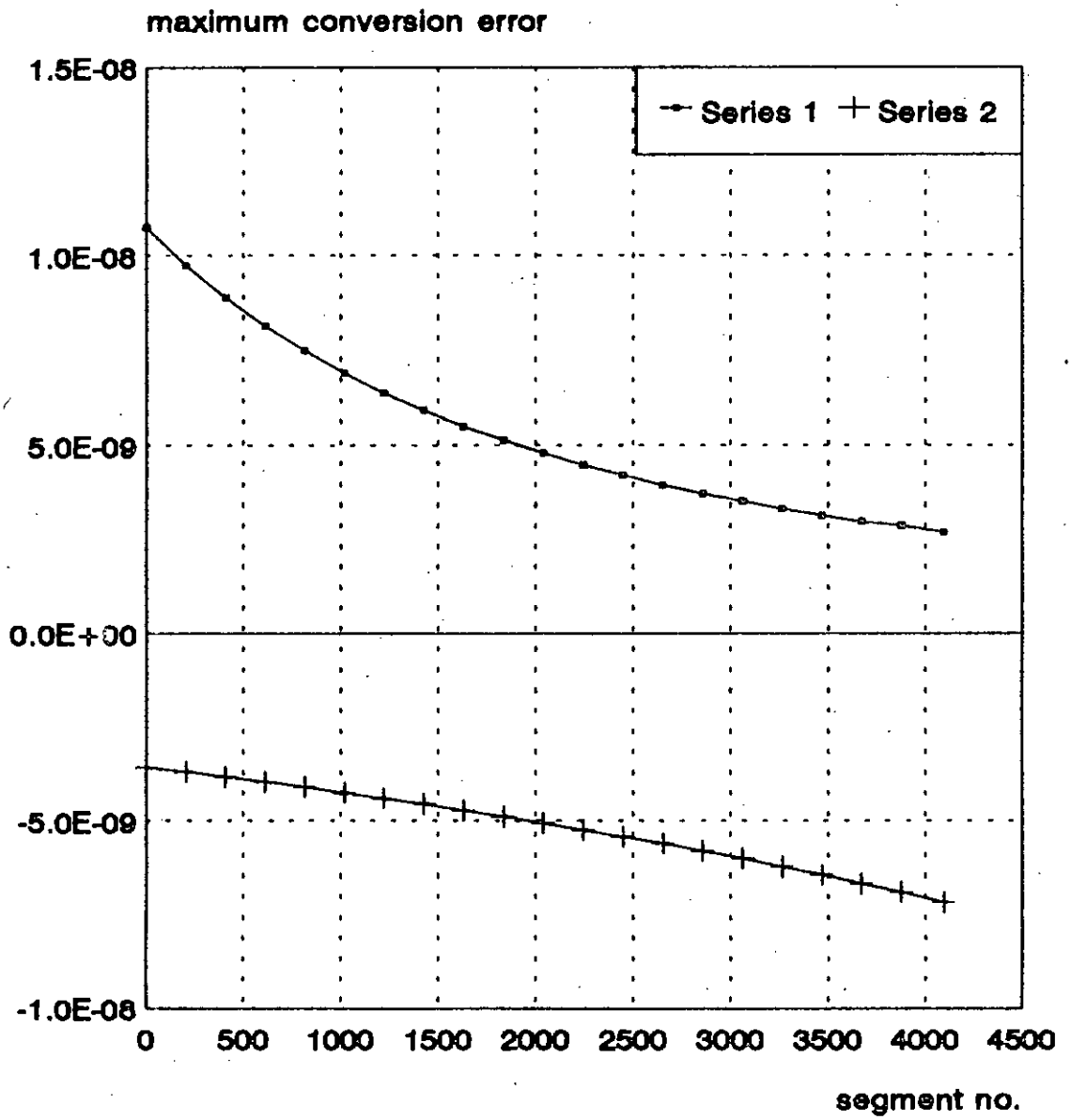
RAL = 12 ML = 23

DL	MCE	m	ROM size (in Kbits)
10	4.53295E-07	0.00244140625	132
20	1.09471E-08	0.0008554458681	172
30	1.07461E-08	0.0001220703125	212
40	1.07461E-08	0.0001220703125	252
50	1.07461E-08	0.0001220703125	292

The effect of correction factor bit length can be visualized when compared the simulation results with our hardware model. If we set RAL 14 and ML 23 (in m_lgm.c) we get the maximum conversion error(MCE) = $6.71766 * 10^{-10}$ at m = $3.0517578 * 10^{-5}$ but if we convert the above value of m by table 3-1 where the correction factor is represented by only 9 bits we get the MCE (4.402687511-4.398822784)$*10^{-5}$ = $3.864727 * 10^{-8}$

The LNS to FLP conversion is also done by a software routine lgm_m.c which is similar to m_lgm.c and as listed in the Appendix.

m to lgm and lgm to m conversion error graph when linear correction factor is applied (RAL 12 ML 23)



series 1 for m to lgm conversion
series 2 for lgm to m conversion

graph 3-1 : Both way conversion error pattern (32 bit)

A typical value of MCE = -1.09428E-13 was achieved for 64 bits LNS to FLP conversion with 1 Mbytes of ROM (RAL 20). The other results are also stimulating e.g. when RAL and ML values are set for 12 and 23 respectively, the absolute max_con_error was reported -7.16E-09 occurred at input value of m = 0.9998779296875 (error pattern in graph 3-1). Fast locating the max_con_error is achieved by executing the choice 7 of lgm_m.c which traces down the numbers starting from 1.0 instead of 0.0.

We have also studied the effect of ROM output bits length (choice 2 or 8 of lgm_m.c) for LNS to FLP conversion as shown in table 3-4. When we set RAL 12, ML 23 and DL 10 MCE was reported to 3.44519E-07 but when DL was set to 30 or higher the MCE reduces to its minimum value as expected. Here we also like to mention that as the *max_con_error* for LNS to FLP conversion is determined by the expression *error = antilg_old_lgm - (old_m + add_factor)* and as it is found to be negative, a lower value of *add_factor* can give us a better value of max_con_error, It may be noted here that a slightly better value has been achieved for DL 20. Table 3-4 gives us the value of other *max_con_errors* for various DL settings.

Table 3-4 : Effect of multiplicand bit length on MCE
(lgm-m conversion, RAL 12 ML 23)

RAL = 12 ML = 23

DL	MCE	lgm	ROM size (in Kbits)
10	3.44593E-07	1.0	132
20	-7.15789E-09	0.9998779296875	172
30	-7.15857E-09	0.9998779296875	212
40	-7.15857E-09	0.9998779296875	252
50	-7.15857E-09	0.9998779296875	292

3.4 **64 bits transformation analysis**

The scope of transformation of floating point numbers into 64 bits logarithm numbers is the transformation of 52 bits mantissa of a floating point number into a logarithm number having a 52 bits mantissa. We have done 64 bits transformation in a way similar to 32 bits transformation as shown in table 3-5.

As the magnitude of the multiplicand is very small and we have taken the least significant 40 bits of its 90 bits configuration, a final 50 bits SHR operation has become necessary to adjust the result. The details of multiplication (Booth recorded multiplier scheme) result for row no 2 and 6 are also attached in sheet 3-3 and sheet 3-4.

It has been found that when 6 random numbers are generated and their base two logarithm numbers are calculated the typical error lies between 2.65E-10 and 9.10E-13. While simulating the situation with m_lgm.c (RAL 14 ML 40 DL 38) the MCE was reported to be equals to $6.7176 * 10^{-10}$ at $m = 3.0517578125 * 10^{-5}$ which is higher than $2.65 * 10^{-10}$ as may be expected. When we converted the above mentioned value of m into lgm by table 3-5 we got an error equals to $(4.40268711 - 4.402621055) * 10^{-5} = 6.6456 * 10^{-10}$ which is well compatible with the simulation result. It should be also mentioned here that the typical error for 32 bits conversion process (table 3-1 and table 3-2) can be reasonably approximated to $((2^{-23}/2 + 2^{-24}/2)/2)$ $4.47 * 10^{-8}$.

Table 3-5

m to $\log_2 m$ conversion data table (RAL = 14 ML = 52)

m	address	lg(m)	Δlgm	<-50 bits->	<- L.S. 40 of 90 bits ->
0.0	00000000000000	0.0	3.2033284198-16	0<- ->0	
0.00006103515625	001	0.00008805242111	3.203133445E-16		
0.00012207031250	010	0.00017609948280	3.202937449E-16		0101110001010001100000011100100000111101
0.00018310546880	011	0.00026414115700	3.202742522E-16		
0.00024414062510	100	0.0003521774731		0<- ->0	
0.1977539063	00110010101000	0.2603315186			
0.1978149415	001	0.2604050336	2.67446012E-16	0<- ->0	0100110100010101000100001111000101010000
0.1978759767	010	0.260478545			
0.1979370119	011	0.2605520525			
0.1979980471	100	0.2606255563			
0.400390625	01100110100000	0.4858293087			
0.4004516602	001	0.4858921863			
0.4005126954	010	0.4859550612			
0.4005737306	011	0.4860179334	2.2871744488-16	0<- ->0	0100000111101100011000000100010101001101
0.4006347658	100	0.4860808028			
0.6005859375	100110011110000	0.6786001391			
0.6006469727	001	0.6786551523			
0.6007080078	010	0.6787101635	2.0012194E-16	0<- ->0	0011100110101110011001001001000101010101
0.600769043	011	0.6787651726			
0.6008300781	100	0.6788201796			
0.80078125	110011010000000	0.8486229404			
0.8008422852	001	0.8486718379	1.778822488-16	0<- ->0	0011001101000101011001001010100111010010
0.8009033204	010	0.8487207338			
0.8009643556	011	0.8487696279			
0.8010253908	100	0.8488185205			
0.9997558594	111111111111100	0.999823879	1.601885724E-16	0<- ->0	0010111000101011110101001001010011000010
0.9998168946	101	0.9998679113			
0.9998779297	110	0.9999119422			
0.9999389649	111	0.9999559718			
1.0		1.0			

Table 3-5 (cont.)

value of m under study	address produced	lg(m)	
0.0001226067543	00000000000010000000100100000000000000000000000000000	0.00001768733075	00000000000001011100101110111100001
0.1978165863	00110010101001000010000000100000000000000000000000000	0.2604073399	01000010101010100000111000110000
0.4006042474	01100110100011100000000000000000000000000000000000000	0.4860493676	01111100011011011011101100111010
0.6007232666	10011001110010010000000000000000000000001000000000000	0.678723917	10101101110000001101100110111110
0.8008732796	11011101000001010000010000000000000000000000000000000	0.848696668	11011001010001000010111101010001
0.9997560382	11111111111100000000001100000000000000000000000000000	0.999824008	11111111111101000111011101011000

Table 3-5 (cont.)

	lgm (from ROM)	Error	The first discrepancy noted
0001001000111101010	00000000000001011100101110111000011111000111111110110	1.57E-11	on 32nd bit
1000001001000100011	01000010101010100000111000110000000101110100110011001	9.79E-11	on 33rd bit
0001010101011111 0001	0111110001101101101110110011101100111000111110110	2.65E-10	on 32nd bit
0011101110101001 0011	1010110111000000110110011011110101000010000000000000	2.27E-10	on 31st bit
0100100010111111 0101	1101100101000100001011110101000101001000101000001	1.52E-10	on 32nd bit
0001110110011010101 1	1111111111110100011101110101100000011100100110101011	9.09E-13	on 40th bit

Example for row 2

address = 00110010101001000010000001000000000000000000000000

value of m = 0.1978168563

lg(m) = 0.2604073399

= 01000010101010100000111000110000100000100100010001000110

```
                multiplicand = Algm   = |01001101000101010001000011111000101010 0
                          multiplier = |00001000000010000000000000000000000000
    ---------------------------------------|----------------------------------------
    |00000000000000000000000000000000000000|00000000000000000000000000000000000000
    |11111111111111011001011101010111011110|0001110101100                      2's complement of
    |00000000000001001101000101010001000011|110001010100                        multiplicand
    |11111101100101111010101110111111000011101|01100
    |00000100110100010101000100001111000101|0100
    ---------------------------------------------------------------------------------
    |00000010011010110001000100110000000100|10000010101000000000000000000000000000
50 SHR|                                     |00000000000000000010011010110001000100   final shifted product
```

```
base    lgm = 01000010101010011111001110111111110000011010011001100 1 ( value at the beginning of the segment )
add_factor  = 00000000000000000001001101011000100010                    ( total correction to be added )
              ---------------------------------------------------------
      lgm = 01000010101010100000111000110000000101110100110011001
```

lg(m) = 01000010101010100000111000110000100000100100010001000110

lgm = 01000010101010100000111000110000000101110100110011001

0<- ->01101011101010101101

Absolute error = 2^-34 + 2^-35 + 2^-37 + 2^-39 + 2^-40 + 2^-41 +
2^-43 + 2^-45 + 2^-47 + 2^-49 + 2^-50 + 2^-52

= 9.7222781E-11

Example for row 6

address = 11111111111100000000001100000000000000000000000000

value of m = 0.9997560382

lg(m) = 0.999824008

= 11111111111101000111011101011000000111011001101010 11

```
              multiplicand = Algm   = |00101110001010111101010010010100110000
                           multiplier = |00000000110000000000000000000000000000
     ------------------------------------|--------------------------------------
     |00000000000000000000000000000000000000|00000000000000000000000000000000000000
     |11111111111101000111010100001010110110|1011010000                          2's complement of
     |00000000001011100010101111010100100101|00110000                            multiplicand
     ------------------------------------------------------------------------------
     |00000000001000101010000011011111011011|11100100000000000000000000000000000000
50 SHR|                                      |00000000000000000000010001010100000011  final shifted product
```

```
base  lgm   = 11111111111101000111010100101110000100001001101010 11 ( value at the beginning of the segment )
add_factor  = 00000000000000000000000100010101000011                 ( total correction to be added )
              ----------------------------------------------------
       lgm = 11111111111101000111011101011000000111001001101010 11
```

lg(m) = 11111111111101000111011101011000000111011001101010 11

lgm = 11111111111101000111011101011000000111001001101010 11

0<- ->01000000000000

Absolute error = 2^{-40}

= 9.094947018E-13

To bring down the conversion error to the level of the 64 bit FLP truncation error (2.22E-16) we have selected a ROM with 25

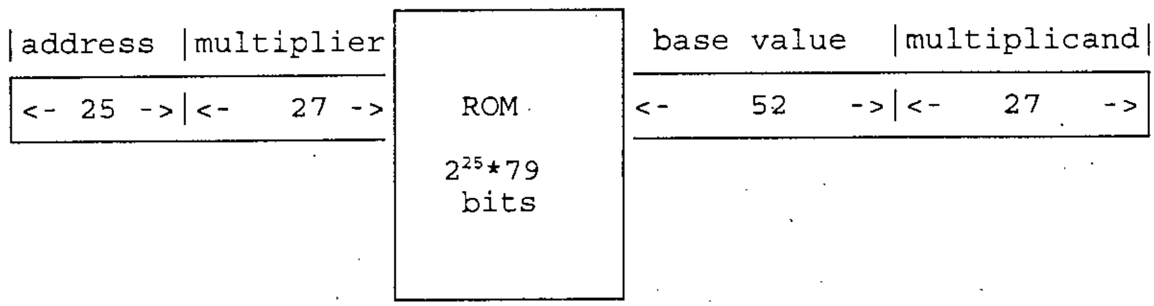
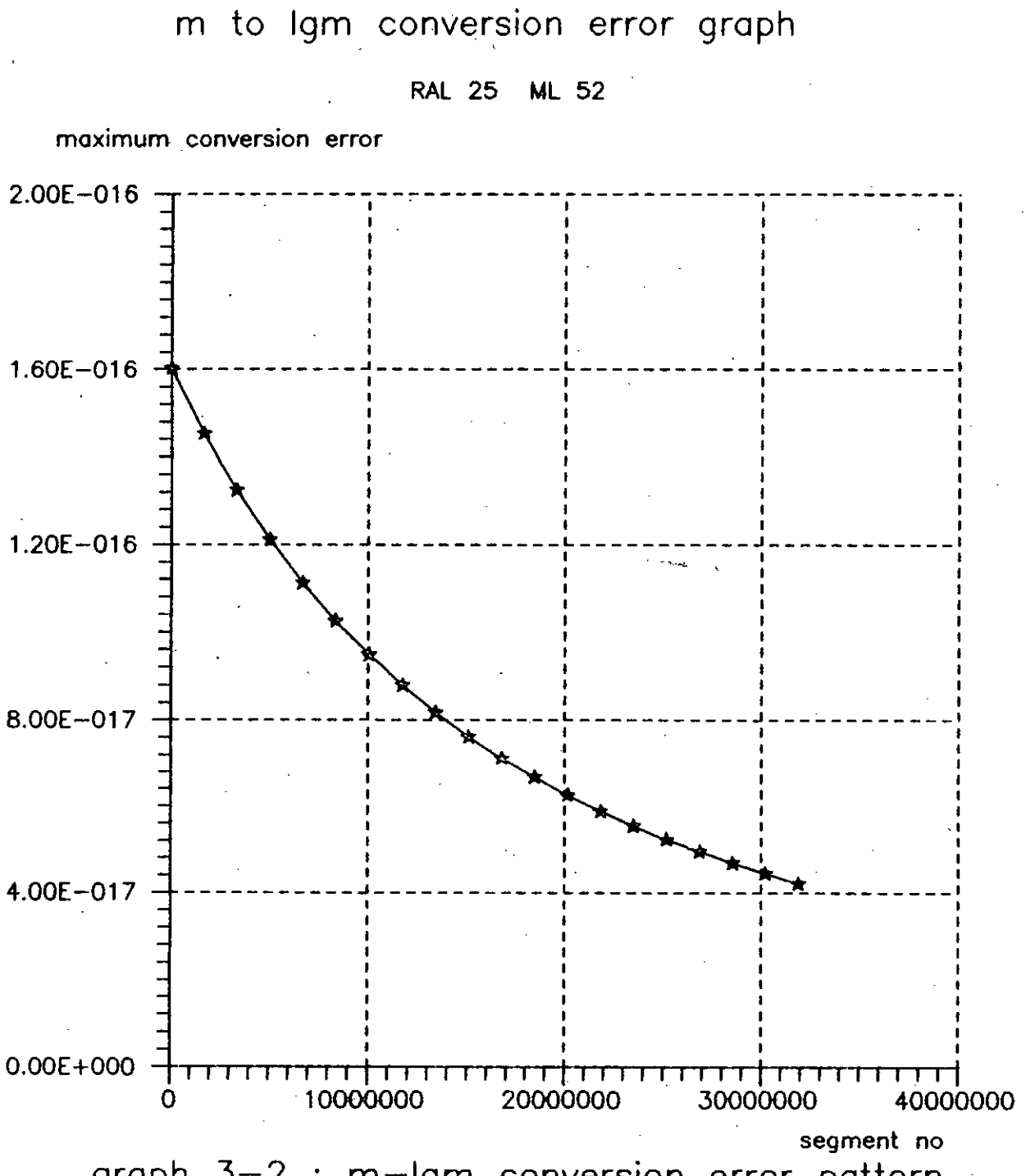


Fig 3-2 : A typical ROM configuration for 52 bit one end conversion (RAL 25 ML 52)

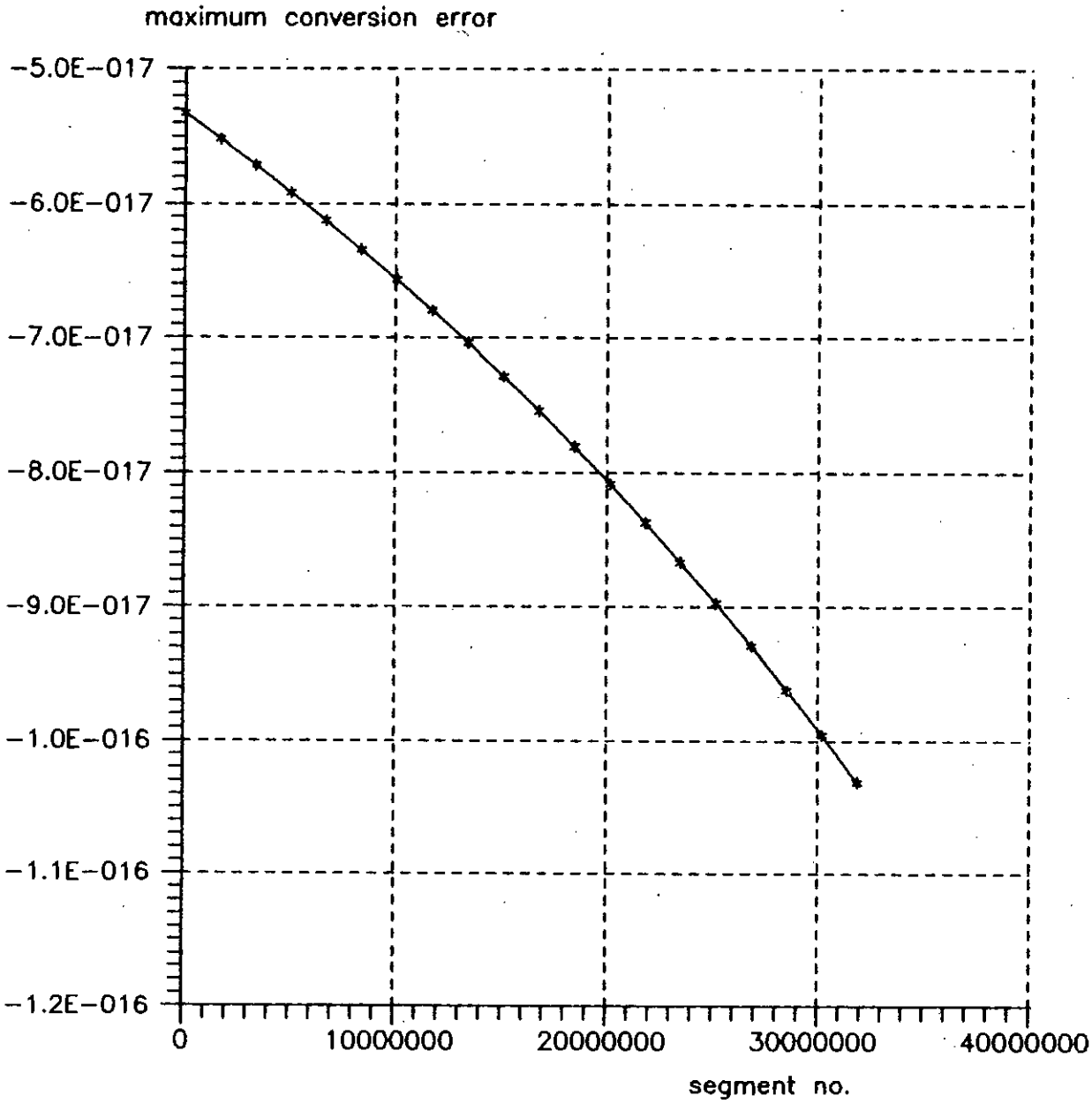
address lines and having 79 output bits as shown in fig. 3-2. By using the choice 1 of m_lgm.c and choice 7 of lgm_m.c we can find out a configuration such that the conversion error shall not exceed that of the truncation error of 64 bit floating point numbers. Graphs 3-2 and 3-3 shows the error pattern for 64 bit FLP - LNS and LNS - FLP conversion.

The ROM capacity needed for each ROM/(block of ROM) is 2528 Mbits. Two such capacity of ROM are necessary for FLP to LNS conversion and a third one is necessary for LNS to FLP conversion. So the total capacity of ROM necessary = (2528 * 3)/8 = 948 Mbytes.

We have also studied the effect of ROM output bits length for 64 bits FLP to LNS and vice versa conversion. Table 3-6 and table 3-7 show us the effect of ROM output bit lengths on maximum conversion error produced. Table 3-6 is for FLP to LNS (choice 1 of m_lgm.c) and table 3-7 for LNS to FLP (choice 7 of lgm_m.c) conversion.



graph 3-2 : m-lgm conversion error pattern



graph 3-3 : lgm-m conversion error pattern

Table 3-6 : Effect of multiplicand bit length on MCE
(m-lgm conversion, RAL 25 ML 52)

RAL = 25 ML[3] = 52

DL	MCE	m	ROM size(Mbits)
10	3.84038E-11	0.000000029802322388	1984
20	5.68176E-14	0.000010192394256529	2304
30	1.89122E-16	0.000003532841219567	2624
40	1.60198E-16	0.000000169312773132	2944
50	1.60171E-16	0.000000014901161194	3262

Table 3-7 : Effect of multiplicand bit length on MCE
(lgm-m conversion, RAL 25 ML 52)

RAL = 25 ML = 52

DL	MCE	lgm	ROM size (Mbits)
10	4.55594E-11	1.0	1984
20	2.79499E-14	0.999999999956344254	2304
30	-2.68882E-16	0.999999985564500093	2654
40	-2.68882E-16	0.999999985564500093	2944
50	-2.68882E-16	0.999999985564500093	3264

[3] While preparing tables 3-6 and 3-7, ML was set to 40 instead of 52. This saves huge computing time with little or no loss of accuracy. If ML is decreased while keeping RAL constant, the value of correction factor increases but multiplier decreases thus keeping the product (multiplier * multiplicand) constant.

While executing the simulation program for 64 bits analysis we took the help of data file for temporarily storage of data so that searches throughout the whole segment can be done which requires considerable long period of time.

3.5 **Reduction of ROM size**

We also like to mention here the several possibilities which can be used to reduce the amount of ROM.

Firstly some registers those contain fixed value of differences and these can be used instead of various difference values for each segment. If we observe table 3-5 we can conclude that the value of difference varies little with segment number and hence an intermediate or other suitable value of difference is loaded into a register. It can reduce the ROM requirement by 33% with little loss of accuracy.

For example the range 0.0-1.0 can be divided into various numbers (numbers those are powers of 2). Several registers can be selected accordingly e.g. if we divide 0.0 - 1.0 in four ranges namely 0.0-0.125, 0.125-0.5, 0.5-0.75 and 0.75-1.0, we have to select four registers those will contain a suitable value of difference for that segment. Selection can be done through a 2 to 4 decoder using the first two bits of the address generated.

This mode of selection not only decreases a significant amount of ROM size but also decreases the overall conversion time. If the difference value is embedded in the ROM, multiplication can begin only after accessing the ROM but the use of registers can allow the beginning of the multiplication process in parallel to the ROM access as the value of the multiplier is composed of the second

portion of the address generated and the multiplicand is already available in one of the registers. Thus the delay of conversion due to the use of multiplication can be avoided.

Embedding of differences values in registers decreases overall accuracy but as the range of accuracy may vary according to requirement we have also investigated the maximum conversion error as given below. Here correction factor may have different values e.g. value of the first segment or value of the last segment. In our simulation we accepted the correction factor applicable to the mid-segment i.e. at the middle of the each range.

RAL = 12 ML = 23

TOTAL REGISTER	maximum conversion error	m
64	2.667E-06	0.000244140625
32	5.2949E-06	0.000244140625
16	1.03189E-05	0.000244140625

It is worthy to mention here that if the difference values are kept in the ROM, MCE becomes 1.07E-08 at m = 0.000122

Secondly use of limited multiplication may reduce ROM size without significant loss of accuracy. In fig. 3-1 the multiplicand bit length can be reduced to a lower value say 16 if the accuracy limit does not fall below the pre-specified requirement. This process also reduces ROM and also reduces overall conversion time.

3.6 **Development of a mathematical model**

So far we have seen that the MCEs occur approximately at the middle of each segment and the absolute value of the MCE is the largest in the first segment (segment 0) for FLP to LNS conversion. While converting LNS to FLP number system we may

observe that although MCEs have occurred at the middle of the each segment its absolute maximum value is in the highest segment (segment no. $2^{RAL} - 1$).

The occurrence of the MCEs at the middle of the each segment is natural. The FLP to LNS conversion curve is a convex curve and on the other hand the LNS to FLP conversion curve is a concave one with maximum deviation approximately at the middle. So the MCE always occurs at a point close to the segment mid value with a positive value for FLP to LNS conversion and a negative value for LNS to FLP conversion.

To prepare the mathematical model for the above analysis, we proceed as follows :

We can calculate the FLP to LNS conversion error by

$$error = lg\_old\_m - (old\_lgm + add\_factor) \quad ..(3.1)$$

where $lg\_old\_m = \dfrac{\ln(1.0 + C*i + m)}{\ln 2}$ [$i = 0,1,2 \ldots (2^{RAL}-1)$]

[ln implies $\log_e$ and i implies ith segment]
and C = CONSTANT = $1/2^{RAL}$, the value of m cannot exceed the range of each segment value so that $0.0 \le m \le C$. Here old_lgm gives us the base value of log base 2 of the starting value of each segment. Therefore

$$old\_lgm = \ln(1 + C*i)/\ln 2$$

$$difference_i = d_i = \frac{old\_lgm_{(i+1)} - old\_lgm_i}{FACTOR}$$

$$d_i = \frac{\ln[1 + C*(i + 1)] - \ln(1 + C*i)}{2^{(ML - RAL)} * \ln 2}$$

To calculate the add_factor we have to calculate the integer value (multiplier) returned from m. The process is described fully in article 3.3.

The integer value I becomes

$$I = \frac{m}{\frac{C}{2^{(ML - RAL)}}}$$

replacing C by $1/2^{RAL}$, we get

$$I = m * 2^{ML}$$

As add_factor = I * d_i, hence add_factor becomes

$$\text{add\_factor} = d_i * m * 2^{ML}$$

$$\text{add\_factor} = \frac{\ln[1 + C*(i + 1)] - \ln(1 + C*i)}{2^{(ML - RAL)} * \ln 2} * m * 2^{ML}$$

$$= \frac{\ln[1 + C*(i+1)] - \ln(1 + C*i)}{\ln 2} * m * 2^{RAL}$$

so (3.1) becomes

$$error = \frac{\ln(1+C*i+m)}{ln2} - [\frac{\ln(1+C*i)}{ln2} + \frac{\ln[1+C*(i+1)] - \ln(1+C*i)}{ln2} *m*2^{RAL}] ..(3.2)$$

differentiating error with respect to m and setting the result to zero, we get

$$\frac{d(error)}{dm} \overset{0}{=} \frac{1}{\ln 2*(1+C*i+m)} - 2^{RAL}*\frac{\ln[1+C*(i+1)]-\ln(1+C*i)}{\ln 2}$$

$$0 = \frac{1}{1 + C*i + m} - 2^{RAL}*[\ln[1 + C*(i+1)] - \ln(1 + C*i)]$$

$$\frac{1}{1+C*i+m} = 2^{RAL}*[\ln[1+C*(i+1)]-\ln(1+C*i)] \quad ..(3.3)$$

This gives us the expression of m in terms of segment no. i. If i = 0 (first segment) the expression reduces to

$$\frac{1}{1 + m} = \frac{\ln (1 + C)}{C} \qquad [\ C * 2^{RAL} = 1\ ]$$

Upon expanding ln(1 + C) and neglecting the terms containing the powers of C, we get

$$\frac{1}{1 + m} = 1 - \frac{C}{2}$$

As $C*m \leq C^2$, the expreassion further reduces to

$$m = \frac{C}{2}$$

As the error pattern is same for each segment it can be concluded that the MCE occurs at the mid point of each segment. Evaluation of m for various values if i from equation (3.3) also confirms the above as given in table 3-8.

Table 3-8 : Location of MCE in any segment during m - lgm conversion (RAL 12 ML 23).

RAL 12 ML 23	
segment no (i)	m
0	1.220858291E-04
1	1.220146858E-04
2	1.220264856E-04
.	.
115	1.220443059E-04
.	.
4095	1.219975490E-04

We next try to find out the segment no in which the MCE occurs. Let us replace m by C/2 in equation (3.2) and after simplification, we get

$$error = \frac{\ln(1+\frac{C}{2}+C*i)}{\ln} - \frac{1}{2}*\frac{\ln(1+C*i)}{ln2} - \frac{1}{2}*\frac{\ln[1+C*(i+1)]}{ln2} \quad ..(3.4)$$

Upon differentiating with respect to i and setting the result to zero, we get

$$0 = \frac{C}{\ln2*(1+C/2+C*i)} - \frac{1}{2}*\frac{C}{\ln2*(1+C*i)} - \frac{1}{2}*\frac{C}{\ln2*[(1+C)+C*i]}$$

Upon rearranging and simplification it becomes

$$\frac{4}{2 + C + C*i} = \frac{2 + C + 2C*i}{1 + C + 2C*i}$$

which further reduces to

$$2C * i = 0 \text{ or } i = 0$$

Which confirms that for FLP to LNS conversion the MCE always occurs at the first segment.

To figure out the necessary conditions during LNS to FLP conversion let us first define the conversion error by

$$error = antilg\_old\_lgm - (old\_m + add\_factor) \quad ..(3.5)$$

$$antilg\_old\_lgm = 2^{old\_lgm} - 1.0$$

$$= 2^{(C*i + lgm)} - 1.0 \qquad [\ 0 \le lgm \le C\ ]$$

$$old\_m = 2^{C*i} - 1.0$$

$$difference_i = d_i = \frac{[\ 2^{C*(i+1)} - 1.0\ ] - [\ 2^{C*i} - 1.0\ ]}{2^{ML - RAL}}$$

$$= \frac{2^{C*(i+1)} - 2^{C*i}}{2^{ML - RAL}}$$

integer value returned from lgm is therefore

$$I = \frac{lgm * 2^{ML-RAL}}{C}$$

$$I = lgm * 2^{ML}$$

$$add\_factor = Id_i = \frac{2^{C*(i+1)} - 2^{C*i}}{2^{ML - RAL}} * lgm * 2^{ML}$$

$$= [\ 2^{C*(i+1)} - 2^{C*i}\ ] * lgm * 2^{RAL}$$

Therefore equation (3.5) becomes

$$error = 2^{(C*i+lgm)} - 1.0 - [2^{C*i} - 1.0 + 2^{C*(i+1)}*lgm*2^{RAL} - 2^{C*i}*lgm*2^{RAL}]$$

$$error = 2^{(C*i+lgm)} - 2^{C*i} - 2^{C*(i+1)}*lgm*2^{RAL} + 2^{C*i}*lgm*2^{RAL} \quad ..(3.6)$$

$$\frac{d(error)}{dlgm} = 0$$

$$= 2^{(C*i+lgm)}*\ln 2 - 2^{C*(i+1)}*2^{RAL} + 2^{C*i}*2^{RAL}$$

$$2^{(C*i + lgm)} * \ln 2 = 2^{C*(i+1)+R} - 2^{C*i+R}$$

which finally gives us

$$lgm = \frac{\ln[2^{C*(i+1)+RAL} - 2^{C*i+RAL}] - \ln(\ln(2.0))}{ln2} - C*i \quad ..(3.7)$$

This gives us the expression of lgm in terms of segment no i. Any selection of $0 \le i \le 2^{RAL}$ will give us the value of lgm which is approximately half of the length of each segment. This has been verified for various ranges of C and a table like table 3-8 can be formed easily. So this can be concluded that for LNS to FLP conversion the MCEs also occur approximately at the middle of each segment. So we can continue our expedition by replacing lgm by C/2 in equation (3.6), differentiating it with respect to i, setting the result to zero and solving for i yield an impossible result

$$2^{C*(i+1)} = 2^{C*i}$$

This happens due to the fact that LNS to FLP conversion error vs. segment no. curve (graph 3-3) does produce more minimum values if the segment nos are allowed to increase (it may be mentioned here

that for FLP to LNS conversion a theoretical value of i lower than zero satisfies the error expression (3.2 with m replaced by C/2) and produces a lesser error) so we replace the lgm by C/2 in equation (3.6) and get

$$error = 2^{C*i}*[2^{\frac{C}{2}}-1] - \frac{2^{C*(i+1)}}{2} + \frac{2^{C*i}}{2} \text{..(3.8)}$$

and finally

$$error = -\frac{2^{C*(i+1)}}{2} + 2^{C*i}*[2^{\frac{C}{2}}-\frac{1}{2}] \text{..(3.9)}$$

As C is very small the second portion of (3.9) reduces to $1/2*2^{C*i}$. So the error expression reduces to

$$error = -\frac{2^{C*(i+1)}}{2} + \frac{2^{C*i}}{2}$$

Whose magnitude continuously decreases as i increases. Therefore for LNS to FLP conversion the absolute magnitude of the largest value of the MCE occurs at the highest segment value.

3.7 **Use of 1Mbyte of ROM for direct conversion**

This concept uses 1 Mbytes of ROM (although other configuration can be used) for FLP to LNS and LNS to FLP conversion without using a multiplier for determining the correction factor. Base 2 logarithm of numbers 0.0 to 1.00 with an increment of $1/2^{20}$ which equals to 9.536743164E-07 are stored in the ROM. While simulating (choice 5

of m_lgm.c) for error calculation only the mid-values between two addresses are considered because any other value within this range will produce lesser error. Thus the maximum error produced is 6.88E-07 at m = 0.000000476837158. The choice 5 of lgm_m.c is simulated for calculating LNS to FLP conversion and the maximum error produced is 6.6103633578E-07 at lgm = 0.999999523162842.

CHAPTER 4

NON-LINEAR CONVERSION AND ERROR ANALYSIS

4.1 Non-linear correction factor

We have modified the basic conversion technique of converting FLP numbers into base 2 logarithm number system as described in the choice 1 of m_lgm.c but the scope in choice 3 has been extended by including the possibility of improving the maximum conversion error. This improvement is done in the *optimum_difference_error()* procedure by applying a search procedure which resembles binary search technique.

We did not apply pure binary search technique because searches at both directions may be felt necessary particularly at the beginning of the search procedure. However by guessing the right direction of search we have also applied binary search technique which yielded the identical results.

A significant improvement has been noted e.g. maximum conversion error has been reduced to 7.39E-09 from 1.07E-08 for 32 bits configuration with 4 Kbytes of ROM (RAL 12 ML 23) while FLP to LNS conversion is done.

The non-linear search is done in the *optimum_difference_error()* procedure. The procedure tries to reduce the maximum conversion error for each segment (each block of ROM is a segment). At first *ideal_difference* is calculated for which the *max_con_error* turns to zero but it produces new values of *max_con_error* in the segment. To select the optimum value of correction factor (here known as *segment.difference*) searches in both directions are made by

allowing *left_search* and *right_search* as *ALLOWED*. If a search in any direction (left or right) produces an error higher than the present *max_con_error* search in that direction is made prohibited by setting it *NOT_ALLOWED*. When searches in both direction produces higher error than the present *max_con_error*, the while loop is exited.

The Pseudo-code for *optimum_difference_error()* is given below.

```
/* Pseudo-code for optimum_difference_error() */
/* for mx to lgmx conversion */
/* segment.max_con_error and segment.difference for the segment is already
 /* calculated by linear search technique   */

  right_search = left_search <-- ALLOWED
  calculate ideal_difference  /* find_ideal_difference()  */
/* as explained in article 4.1 */
  high_end_difference = ideal_difference  /* U.L. in fig. 4-2 */
  mid_difference = ideal_difference
  low_end_difference = segment.difference  /* L.L. in fig. 4-2 */
  calculate temp_new_con_error at ideal_difference
/* temp_new_con_error is the temporary value of the MCE in the segment */
  if fabs(temp_new_con_error) > segment.max_con_error
                                  and temp_new_con_error < 0.0
     right_search <-- NOT_ALLOWED
     left_search <-- ALLOWED
    else print error message and exit
/* now swing to lower side because any search to higher side will produce more
/* error so right_search is set to NOT_ALLOWED */
/* henceforth all tried new values of difference (Δlgm) is ideal_difference
/* Here trials are made to find the best  value of difference  so that the
/* absolute value of MCE in the segment is minimum. Therefore without
/* increasing the number of variables by introducing a new name, the previous
/* ideal_difference is used */
 ideal_difference = (mid_difference + low_end_difference) * 0.5

  while ( right_search  =  ALLOWED or left_search = ALLOWED )
    if ( right_search )
      ideal_difference <-- (mid_difference + high_end_difference) * 0.5
      calculate right_new_con_error
    if ( left_search )
      ideal_difference <-- (low_end_difference + mid_difference) * 0.5
      calculate left_new_con_error
        if right_new_con_error or left_new_con_error < segment.max_con_error
```

```
segment.max_con_error <-- min{right_new_con_error, left_new_con_error}
    segment.difference <-- ideal_difference
    left_search = right_search <-- ALLOWED
  else right_search = left_search = NOT_ALLOWED
/* end of optimum_difference_error() */
```

Fig 4-1 : Pseudo-code for optimum_difference_error()

To illustrate the matter further we can proceed as follows. Let us set RAL 3 and ML 8 so that we can throughout study the first segment out of total eight segments and can find out how the *optimum_difference_error()* procedure finds out the minimum value of the *max_con_error* in any segment.

When the program starts executing the instructions of the *optimum_difference_error()* procedure the global variable *difference* (Δlgm) is already assigned with a value of 0.005310156295072 and the *segment.max_con_error* is assigned with the corresponding value of 0.00250034. This error occurs at the middle of the first segment at the value of *old_m* = 0.0625 and this also equals to *segment*.m (segment.m indicates the value of m at which the MCE occours). Left_search and right_search are assigned with *ALLOWED* so that searches to both higher and lower sides can be made if found necessary.

In the procedure *optimum_difference_error()* our intention is to find a new value of the global variable *difference* so that the *segment.max_con_error* turns to its minimum possible value. To find the minimum value let us set the present value of *error* as zero (if the absolute value of the *max_con_error* turn to zero for every input value it is the most desirable condition but it cannot be zero for every input value, so let us set it to zero for one value

and try the result for other input values). So calculate the value of *ideal_difference* (so that MCE at the present value of segment.m turns to zero) from the equation[4]

ideal_difference = (log(1.0+segment.m)/log(2.0)-lgm)/divisor

from which we get the value of the ideal_difference = 0.005466427578. As the error is determined from the equation[5] *error = lg_old_m - (old_lgm + add_factor)* we can see from the table 4-1 that a new value of *error* = - 0.0050006811 has been produced at another value of *old_m* but the error has turned to

Table 4-1 : The first swing during non-linear search

ideal_difference	new_m	old_m	lg_old_m	error
0.005466427578	0.125	0.00390625	0.005624549194	0.000158121616
		0.03125		0.000662698736
		0.0625	0.08746284125	0.0
		0.09375		-0.00191124492
		0.125	0.16992500014	-0.0050006811

zero for *old_m* = 0.0625 (at the middle of the segment under study) as expected. As the value of the *error* is negative, it is clear that any further increase of the variable *difference* will

---

[4] *ideal_difference* is calculated through the procedure find_ideal_difference() so segment.m has been replaced by m.

[5] *error* in *find_ideal_difference()* is calculated through the procedure get_error(). Here the more known form of the equation of the *error* is mentioned.

increase the absolute value of the *error* i.e. a search for a different value towards high must be not made so we accordingly set *right_search = NOT ALLOWED* in the expression *if (fabs (temp_new_con_error) > segment.max_con_error && temp_new_con_error < 0.0) right_search = NOT_ALLOWED*. So the lower limit (L.L.) is set by the *segment.difference* and the upper limit (U.L.) is set by the *ideal_difference*. Now execution begins from *if (left_search)* and swing is made to the left direction. All left and right

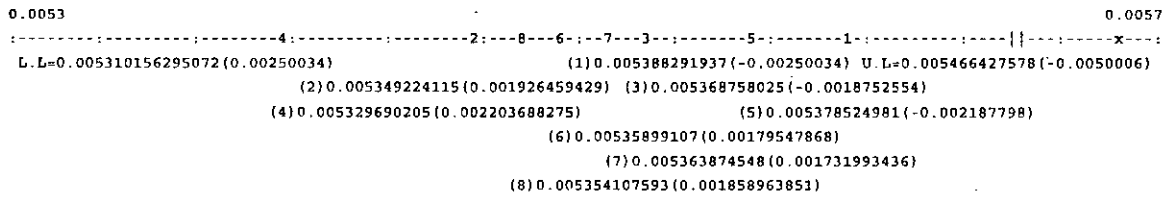


Fig 4-2 : left and right swings in the non-linear conversion

swings are shown in the fig. 4-2. Here the MCEs (*max_con_errors)* are given inside the parenthesis together with *difference* (Δlgm) and search no., so it is clear that on the search no 7 the minimum value of the *max_con_error* = 0.001731993436 is achieved at the value of *difference* = 0.005363874548. Two more searches 9 and 10 (not shown) are also further made for the values *difference* = 0.005366316287 and 0.005361432809 but as they produce errors higher than 0.001731993436. Therefore the searches in the first segment is completed and the searches for the other segments are continued. So it is clear that the procedure *optimum_difference_error()* has find out a new value of *difference* = 0.005363874548 (instead of 0.005310156295072) that produces a lesser value of *segment.max_con_error* = 0.001731993436 (instead of 0.00250034)

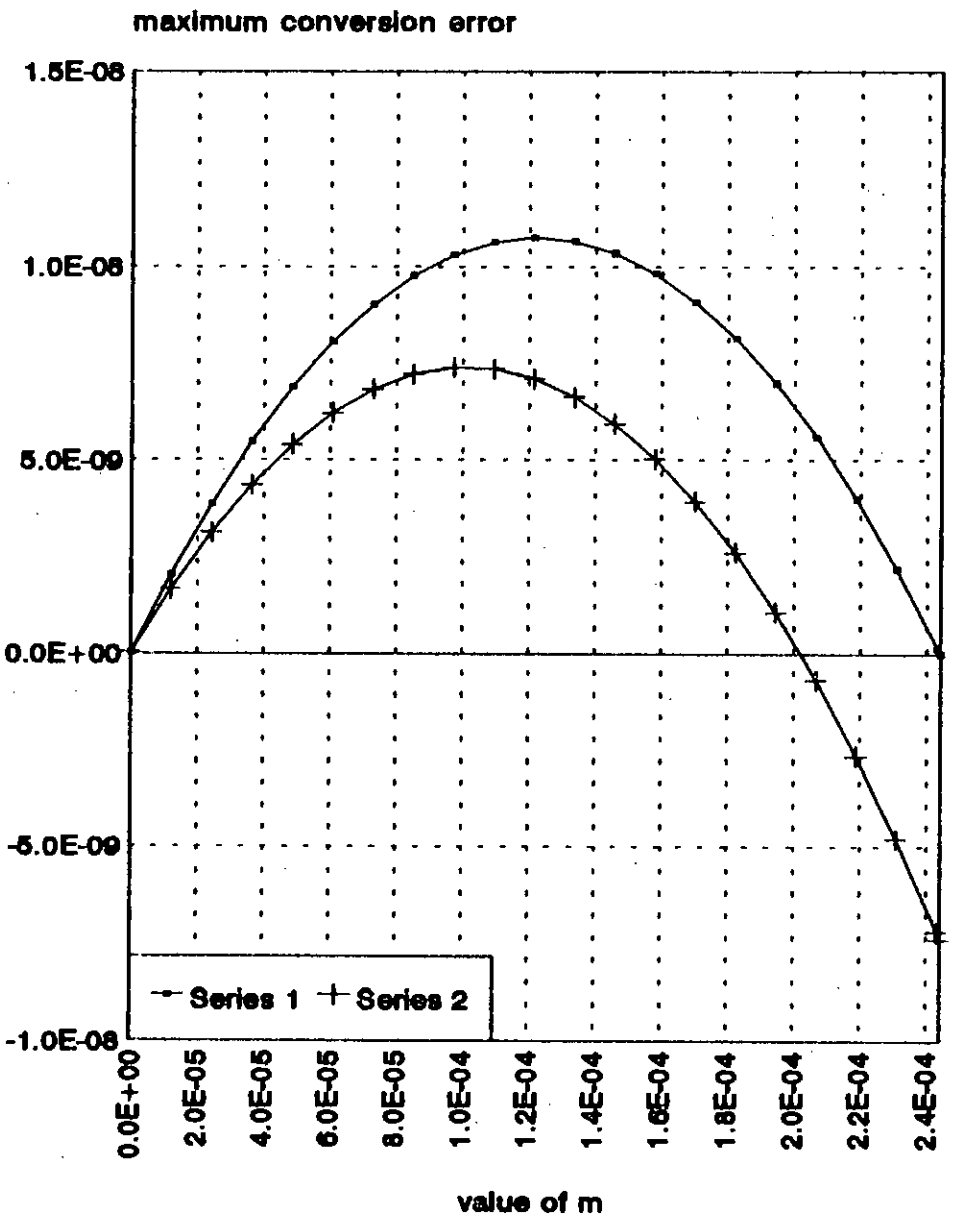
The error pattern when linear and non-linear correction factors are applied are given in graph no 4-1 and 4-2.

Once the searches in one segment finishes, similar searches for the other segments in this fashion continues through all the segments.

The non-linear correction for LNS to FLP conversion has done by choice 3 of lgm_m.c as listed in the Appendix . Here we have also noted significant improvement e.g. maximum conversion error has been reduced to 4.9E-09 from -7.15E-09. for 32 bits configuration with 4 Kbytes of ROM (RAL 12 ML 23).

Linear and non-linear error curve for 32 bit FLP to LNS conversion is given in graph 4-3 and that of LNS to FLP conversion is given in graph 4-4 respectively.

linear and non_linear error pattern
in the first segment for RAL 12 ML 23
for m to lgm conversion (segment no 0)

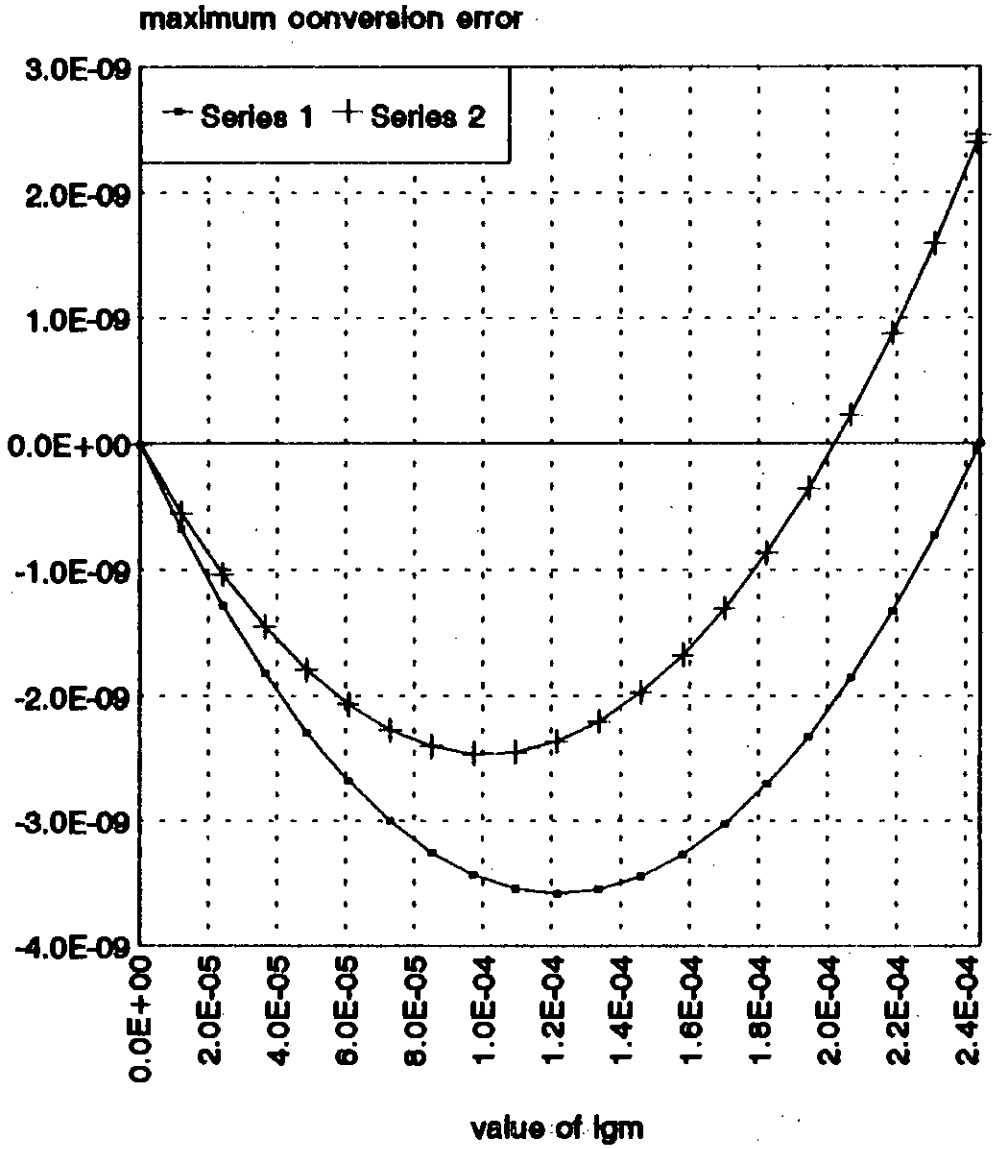


series 1 when linear correction factor applied
series 2 when non_linear correction factor applied

graph 4-1 : linear and non_linear error pattern in a segment for m to lgm conversion

linear and non_linear error pattern

in the first segment for RAL 12 ML 23
for lgm to m conversion (segment no 0)

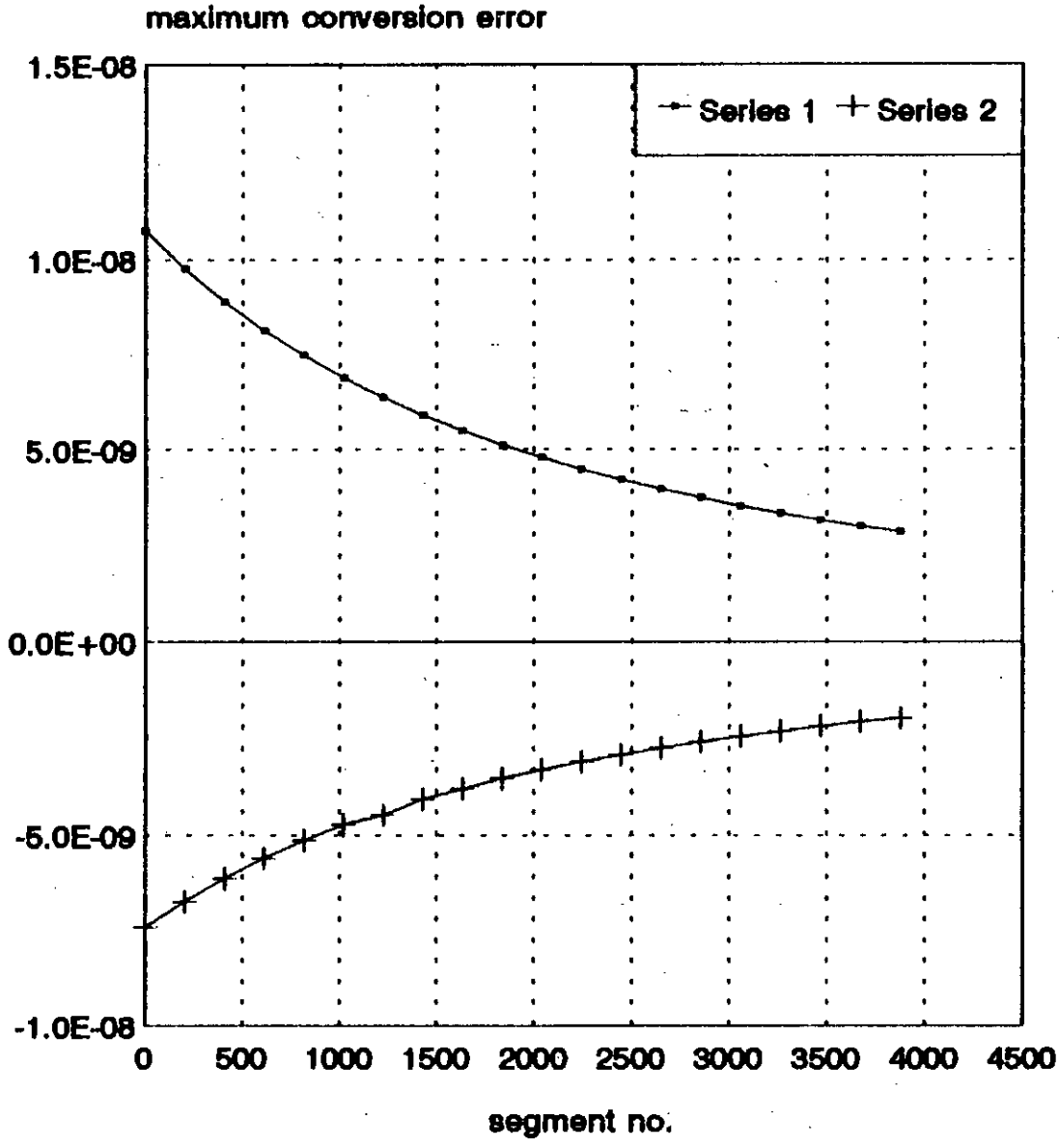


series 1 when linear correction factor applied
series 2 when non_linear correction factor applied

graph 4-2 : linear and non_linear error pattern in a segment for lgm to m conversion

m to lgm conversion error graph

RAL 12 ML 23

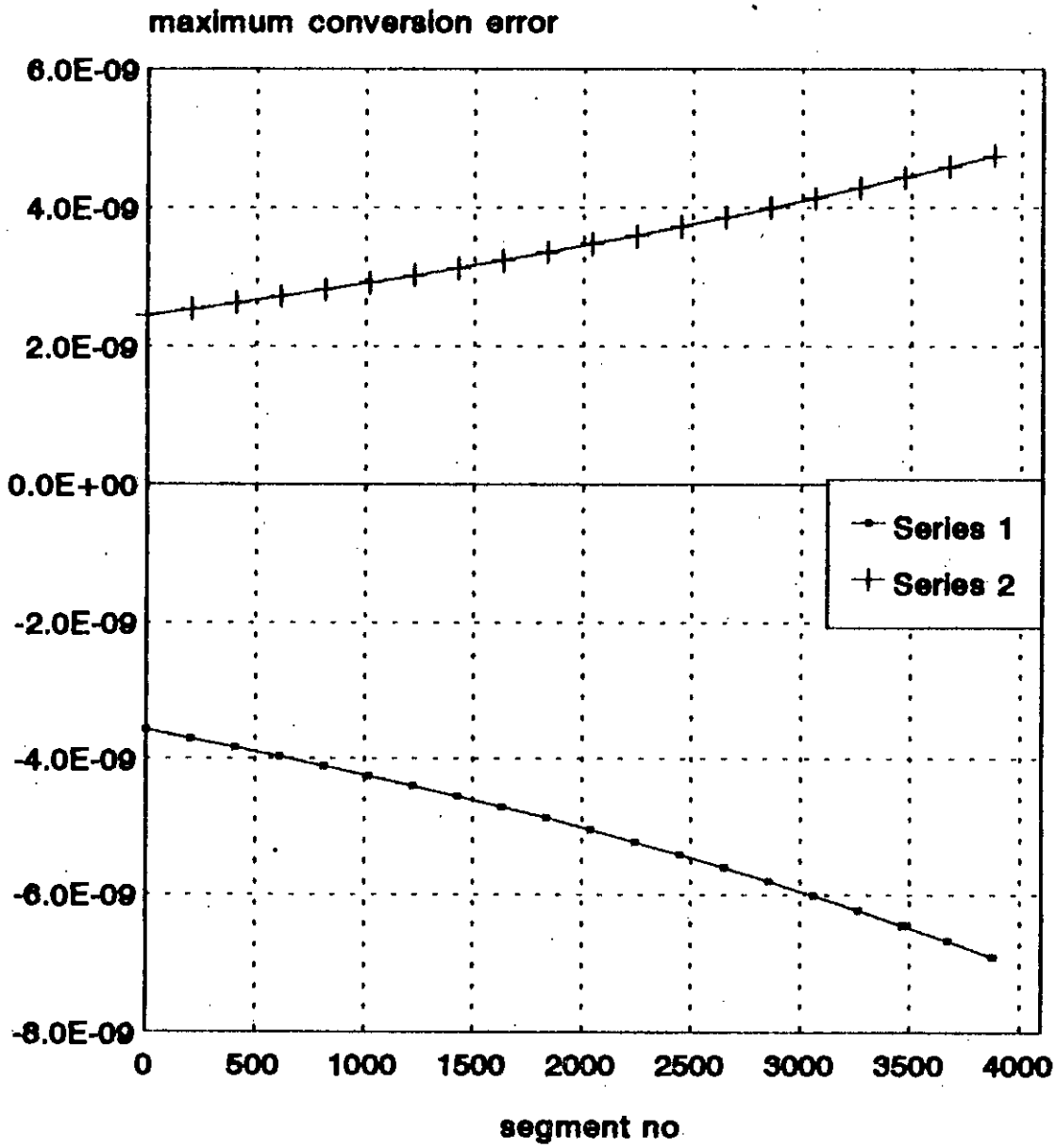


series 1 for linear correction factor
series 2 for non-linear correction factor

graph 4-3 : m to lgm linear and non linear conversion error pattern (RAL 12 ML 23)

lgm to m conversion error graph

RAL 12 ML 23



series 1 for linear correction factor
series 2 for non-linear correction factor

graph 4-4 : lgm to m linear and non_linear conversion error pattern
(RAL 12 ML 23)

4.2 **Arithmetic processing**

We have decided to cite a multiplication procedure by LNS number system which will clearly depict the conversion, processing and anti-conversion process. Let us select two numbers x and y as

x = 0.7061309815 and y = 12.3330078125 so that
z = x * y = 8.708718911

We assume that the numbers are represented in a computer system by IEEE format so they will be

x = 0-01111110-01101001100010100000000
y = 0-10000010-10001010101010000000000
z = 0-10000010-00010110101011011101001

Our first job is to convert these FLP numbers into the corresponding LNS numbers. To accomplish this we run choice 4 of m_lgm.c (which is the interactive version of m_lgm.c) and get the ROM contents of the address bits 011010011000 (the first 12 bits corresponds to count = 1688). The contents of the ROM is as given below.

m_initial	*lg_m_initial*	*difference*
0.412109375	0.4978518369511178	1.2178078307E-07
0.412109375	0.4978518369511178	1.2178259231E-07

The first line gives the values for linear conversion and the second line gives the data for non-linear conversion. So the LNS value of the mantissa of the FLP number x becomes

0.4978518369511178 + 1.2178259231E-07 * 1280 = 0.4980077187

Here we multiply the difference with the value of the remaining 11

bits (10100000000 = 1280) and add the product with the base lgm value which is available under the column *lg_m_initial*. So the representation of x after converting it to LNS format (FLP representation in small letter and LNS representation in capital letter) becomes

X = 0-01111110-01111111011111010110111

We next convert y into LNS format. The contents of the ROM for this conversion is

m_initial	*lg_m_initial*	*difference*
0.54150390625	0.624338545312985	1.1155925875E-07
0.54150390625	0.624338545312985	1.1156077703E-07

After following the above mentioned procedure, we get the arithmetic expression of 0.624338545312985 + 1.1156077703E-07 * 1024 = 0.6244527835 and thus y becomes

Y = 0-10000010-10011111110111000010001

After converting both the numbers into LNS format we have to add them because the multiplication procedure is now replaced by addition. Now we have to subtract 127 from both the exponents of X and Y and to add them. The extraction of the hidden 1s are not necessary[3] because these hidden 1s in the mantissa of the FLP numbers make the conversion easier as $2.0^{(e - 127)} * 1.m$ in FLP is equals to $2.0^{(e - 127 + 0.M)}$ in LNS system . So we get

X = 0-11111111-01111111011111010110111
Y = 0-00000011-10011111110111000010001

Z = X + Y = 0-00000011-00011111010110011001000

Now we have the result in LNS format. Our next job is to convert Z into FLP format. We run the interactive version of lgm_m.c (which

is choice 4 of lgm_m.c) and get the following ROM chunk.

lgm_initial	*antilg_lgm_initial*	*difference*
0.122314453125	0.0884796633448476	8.994823118E-08
0.122314453125	0.0884796633448476	8.994692308E-08

If we follow the linear difference we get the expression of 0.0884796633448476 + 8.994823118E-08 * 1224 = 0.08858975998 and

z = 0-00000011-00010110101011011101000 or
z = 8.70871808

If we follow the non-linear difference we get the expression of 0.0884796633448476 + 8.994692308E-08 * 1224 = 0.08858975838 and

z = 0-00000011-00010110101011011101000
z = 0-10000010-00010110101011011101000 (in IEEE format)
z = 8.70871808

We get the same result because the difference cannot be represented within the limited 23 bit mantissa format but obviously $2.0^{3.0} * 1.08858975998$ is not equal to $2.0^{3.0} * 1.08858975838$.

We have also converted and reconverted a single number e.g. m = 1.0001438856125. The selection of m is intentional so that we can work in the ROM area containing maximum conversion error. Applying linear correction factor we get lgm = 0.0 + 1.719616603E-07 * 1207 = 2.07557724E-04 and by applying non- linear correction factor we get lgm = 0.0 + 1.7196526776E-07 * 1207 = 2.075620782E-04. Now $2.0^{2.07557724E-04}$ = 1.000143878 and $2.0^{2.075620782}$ = 1.000143881 and the result obtained by non-linear conversion closes to the actual value of 1.0001438856125 but when represented in a 23-bit mantissa both lgms are to be represented under the same bit configuration and hence the difference cannot be stored for further processing.

Next we shall evaluate y^x by using LNS system. As we have clearly described the conversion process from ROM, here the converted values in IEEE format shall be shown without giving any further details of addition process.

Let y = 4.125, x = 0.796875 so that y^x = 3.093256701

Let us first represent the real values of x and y in IEEE format, so that

y = 0-10000001-00001000000000000000000
x = 0-01111110-10011000000000000000000

Next we convert y into lgy by using ROM look-up table. We already know that changing FLP to LNS value or vice versa shall only bring changes in the mantissa, so we get

Y′ = lgy = 0-10000001-00001011010111010110100

However it is again necessary to calculate the base 2 logarithm of Y′ [3] and hence it is necessary to convert Y′ into lgY′. We know that the ROM can only accurately convert a number to its base 2 logarithm value if the number is represented in FLP format. The conversion is done by a software routine as outlined in [3]. It is not also difficult to visualize the conversion technique by writing lgy = 2.04439401627 = $1.00000101101011101011010 * 2^{1.0}$ so that the number Y′ is represented in real number format shall be

Y′ = 0-00000001-00000101101011101011010

We next convert x and Y′ to their respective log base 2 and get

X′ = lgx = 0-01111110-10101100001001000001000
Y" = lgY′ = 0-00000001-00001000000110101011111

Now it is necessary to add X′ and Y" to get X′ + Y"

```
X′   =    0-11111111-10101100001001000001000
Y"   =    0-00000001-00001000000110101011111
-------------------------------------------------
X′+Y" = 0-00000000-10110100001111101100111
```

Now it is necessary to find the antilogarithm of X′ + Y". Through the ROM that converts LNS to FLP, we get

$z = 2.0^{(X' + Y")} = 0\text{-}00000000\text{-}10100001000011000110010$

To find y^x, it is again necessary to get the antilogarithm of z. We take care of the fact that to find the antilogarithm of any number we must represent the number in LNS format and then access the ROM.

The conversion from FLP to LNS can be done by another software routine [3]. The procedure is rather simple. To convert into the LNS format we insert the hidden bit and adjust the exponent . Therefore, after converting z into LNS format, we get

$z' = 0\text{-}00000001\text{-}10100001000011000110010$

The final conversion of z′ to $2.0^{z'}$ gives the value of

$2^{z'} = 0\text{-}00000001\text{-}10001011111011011010001$

$= 3.093189478$

Obviously y^{-x} can be calculated by first finding out y^x and then inverting the result.

4.3 **Effect of non-linear correction factor on Error**

Non-linear conversion minimizes the absolute magnitude of the maximum conversion error [MCE] in any segment. Once the absolute magnitude is decreased, the average value of conversion errors decreases for any input value. We have studied a few evenly spaced numbers for FLP to LNS conversion in one segment and the results are shown in Table 4-2. While converting these numbers from its FLP format to LNS format we used the following ROM contents (vide choice 4 of m_lgm.c).

RAL = 12 ML = 23

m_initial	*lg_m_initial*	*difference*	*max_con_error*
4.8828125E-04	7.042690112466E-04	1.7187774587E-07	+1.07358E-08
4.8828125E-04	7.042690112466E-04	1.7188134980E-07	-7.38087E-09

Table 4-2 : m to lgm linear and non-linear conversion error for RAL 12 ML 23

RAL 12 ML 23					
m (segment no =2)	lg(m)	lgm		error	
		linear	non-linear	linear	non-linear
4.884004539E-04	7.044408966E-04	7.04440889E-04	7.044408926E-04	7.60000E-12	4.0000E-12
5.216598511E-04	7.523998473E-04	7.523947801E-04	7.523957892E-04	5.06720E-09	4.0581E-09
5.866289138E-04	8.460784765E-04	8.460681516E-04	8.460711248E-04	1.03249E-08	7.3517E-09
6.320476532E-04	9.115639647E-04	9.115535728E-04	9.115579191E-04	1.03919E-08	6.0456E-09
6.799697876E-04	9.806556592E-04	9.806484266E-04	9.806542217E-04	7.23260E-09	1.4375E-09
7.323026657E-04	1.05610277E-03	1.056102757E-03	1.056110134E-03	1.30000E-11	-7.3640E-09

As we see that in most of the cases the error produced is higher when linear correction factor has been employed but there is instance that the use of non-linear correction factor has increased the absolute value of error. This is expected because when we select a new value of difference, we concentrate on reducing the absolute of the MCE and pay no notice in keeping the error value less than that produced during linear correction factor. Thus we

see that the average value of non-linear conversion error becomes 4.37848E-09 (negative values are added as positive errors) which is lesser than the average value of linear correction error which is 5.5062E-09.

We have also reconverted all these LNS numbers into the FLP format. While making the LNS to FLP conversion we have used the following ROM contents (choice 4 of lgm_m.c). We used the lgm values previously achieved in FLP to LNS conversion and by using non-linear correction factor.

RAL = 12 ML = 23

lgm_initial	*antilg_lgm_initial*	*difference*	*max_con_error*
4.88281250E-04	3.385080526823E-04	8.266454800E-08	-3.58117E-09
4.88281250E-04	3.385080526823E-04	8.266334583E-08	+2.46205E-09
7.32421875E-04	5.078050469876E-04	8.267853812E-08	-3.58177E-09
7.32421875E-04	5.078050469876E-04	8.267733575E-08	+2.46247E-09
9.76556250E-04	6.771306930664E-04	8.269253062E-08	-3.58238E-09
9.76556250E-04	6.771306930664E-04	8.269132803E-08	+2.46289E-09

Table 4-3 : lgm to m linear and non-linear conversion error

lgm	lgm (neglecting truncation)	$m=2^{lgm}-1.0$	m (from ROM)		error	
			linear	non-linear	linear	non-linear
7.044408926E-04	7.04407692E-04	4.8837742E-04	4.883788728E-04	4.883766987E-04	-1.4528E-09	7.213E-10
7.523957892E-04	7.523298264E-04	5.2161128E-04	5.216123629E-04	5.216121621E-04	-1.0829E-09	-8.821E-10
8.460711248E-04	8.460283279E-04	5.8659412E-04	5.865976938E-04	5.86596548E-04	-3.5738E-09	-2.428E-09
9.115579191E-04	9.114742279E-04	6.319854E-04	6.319882112E-04	6.319864053E-04	-2.8112E-09	-1.0053E-09
9.806542217E-04	9.806156158E-04	6.79942E-04	6.799422391E-04	6.799421982E-04	-2.391E-09	-1.982E-10
1.056110134E-03	1.056075096E-03	7.3228346E-04	7.32286611E-04	7.322858089E-04	-3.151E-09	-2.3489E-09

From table 4-3 we see that if we apply non-linear correction factor for LNS to FLP conversion, we get the minimum conversion error as expected. The error calculations given in table 4-2 and 4-3 do not consider truncation error. Truncation error develops due to the fact that when we convert a FLP number into a LNS number, the

converted number cannot exactly represented by a limited number of storage bits. Conversion error that takes the effect of truncation error is shown in table 4-6.

We have further delved into the matter and reconverted all these numbers into the FLP format. We used the lgm values previously achieved in FLP to LNS conversion and by using non-linear correction factor.

Table 4-4: Partial compensation effect using non_linear correction factor during lgm-m conversion

RAL 12 ML 23				
m (segment no. 2)	value returned using correction factor		Error produced	
	linear	non-linear	linear	non-linear
4.884004593E-04	4.883788782E-04	4.883766987E-04	2.15811E-08	2.37606E-08
5.216598511E-04	5.216123629E-04	5.216121621E-04	4.74882E-08	4.76890E-08
5.866289139E-04	5.865976938E-04	5.865965480E-04	3.12201E-08	3.23689E-08
6.320476532E-04	6.319882112E-04	6.319864053E-04	5.94420E-08	6.12479E-08
6.799697876E-04	6.799422391E-04	6.799421982E-04	2.75485E-08	2.75894E-08
7.323026657E-04	7.322866110E-04	7.322858089E-04	1.60547E-08	1.68568E-08

From the table 4-4 we see that if we apply non-linear correction factor for FLP to LNS conversion and then apply linear correction factor for LNS to FLP conversion we get minimum conversion errors in all the cases. Although only LNS to FLP conversion by applying non-linear correction factor will give a less average value of conversion error but if the values are already converted from FLP to LNS, then linear correction factor will compensate some portion of the error which is already made during FLP to LNS conversion since it has a higher value of correction factor.

4.4 **Accumulated error**

Conversion error can be defined as an error that equals to the difference of the double precision calculation in the computer system in use minus the result obtained from the algorithm we have developed. So if we have randomly generate a FLP number x and first convert it to the LNS system then reconverted it back to FLP format and suppose after conversion and reconversion the new number is x′, then the conversion error = x - x′

We have converted mantissas of 10000 FLP numbers into LNS numbers by randomly generating FLP numbers (between 0.0 and 1.0) and reconverted them back into their previous FLP format with the help of data previously generated and stored in hard disk. Accumulated errors for such conversion have been calculated and the summarized result is given below in table no. 4-5 and 4-6. The first table i.e. table no. 4-5 gives us one way

Table 4-5 : One way accumulated error, m-lgm or lgm-m conversion

RAL 12 ML 23		
conversion	accumulated error	
	linear	non-linear
m - lgm	3.56E-05	1.713E-05
lgm - m	-3.423E-05	-1.641E-05

Table 4-6 : Accumulated error for m-lgm and lgm-m conversion

RAL 12 ML 23		
conversion		accumulated error
m - lgm	lgm - m	
linear	linear	6.17806E-04
linear	non-linear	6.36292E-04
non-linear	linear	5.98879E-04
non-linear	non-linear	6.18523E-04

conversion error. The one way accumulated error produced for RAL 12 and ML 23 was found to be compatible with the figures given in the graphs 4-1 and 4-2. This also indicates that the accumulated error generated while using linear correction factor is approximately twice larger than the accumulated error generated while using non-linear correction factor. The difference in accumulated error is not negligible when one way conversion is considered.

The next table i.e. table 4-6 gives us the accumulated error when a FLP number x is first converted into LNS number and then reconverted back into FLP number again. Various combinations of conversion techniques have been used as shown in the table 4-6. Here the accumulated error is higher than that of Table 4-5 due to the truncation error for internal storing of 23 bit format after conversion. It can be mentioned here that if we convert mantissa of a FLP number into the corresponding LNS number and store it in a 23 bit format, then only due to the limited storage bit capacity the truncation error can be as high as 2^{-23} = 1.1921E-07.

4.5 **Accumulated error for multiplication, squaring and division**

Conversion error as defined in article 4.4 has been also calculated for multiplication and other operations for 10000 operations on randomly generated numbers. Conversion data for 23 bit mantissa length and 12 bit ROM length has been generated and stored in a hard disk. Their summarized result is given below:

Table 4-7: Accumulated error for multiplication

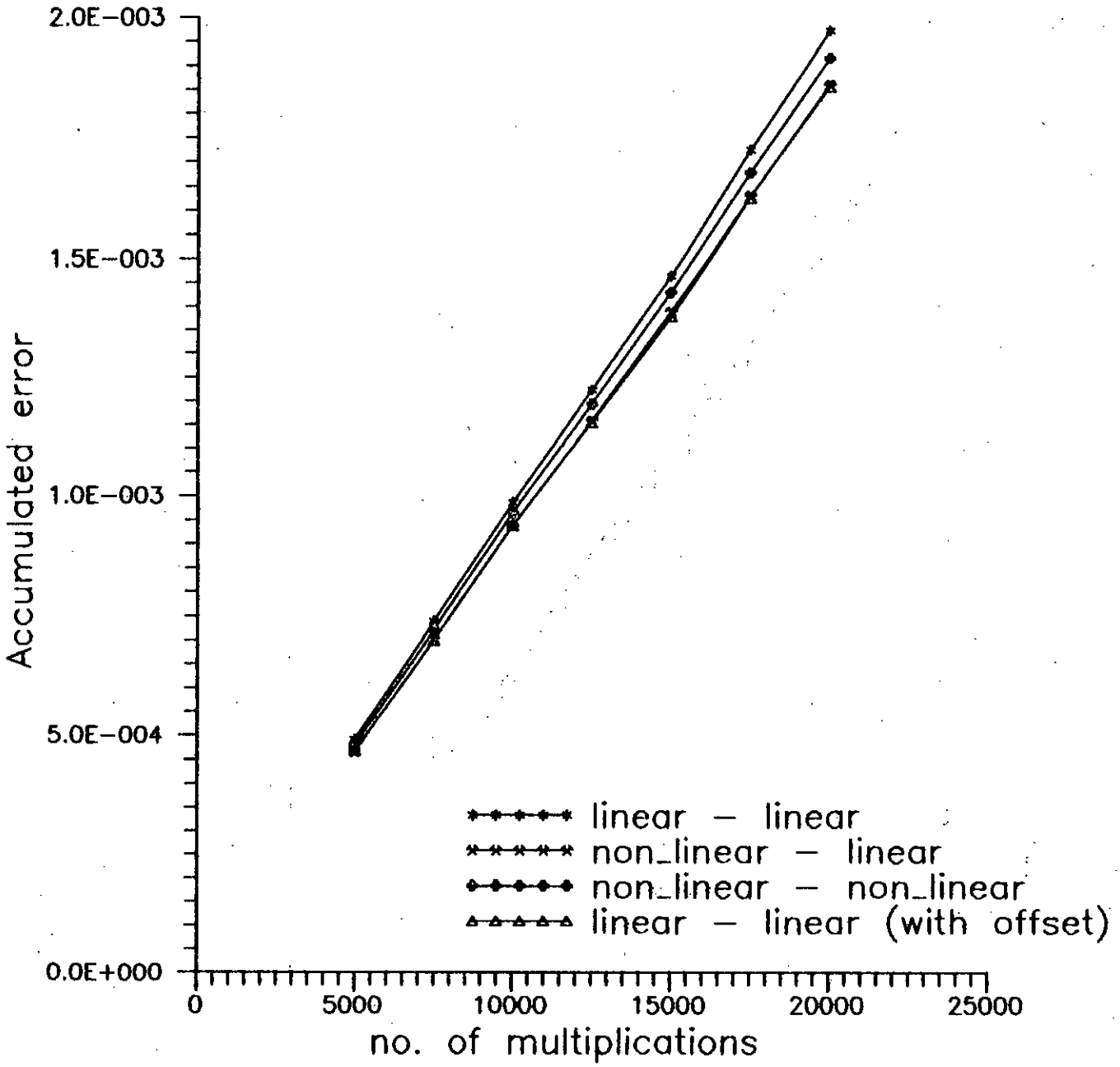
RAL 12 ML 23		
conversion		accumulated error
linear	linear	9.84775E-04
non-linear	linear	9.35826E-04
non-linear	non_linear	9.63621E-04
linear	linear (with offset)	9.34800E-04

Here we see that the accumulated error is higher than the both way conversion (Table 4-6) accumulated error due to the multiplication effect. Accumulated error for linear-non_linear is disregarded because it has already the highest accumulated error in table 4-6. Accumulated error with offset means that the factor (segment.max_con_error/2.0) is subtracted from the individual number conversion error during m-lgm conversion and added during lgm-m conversion.

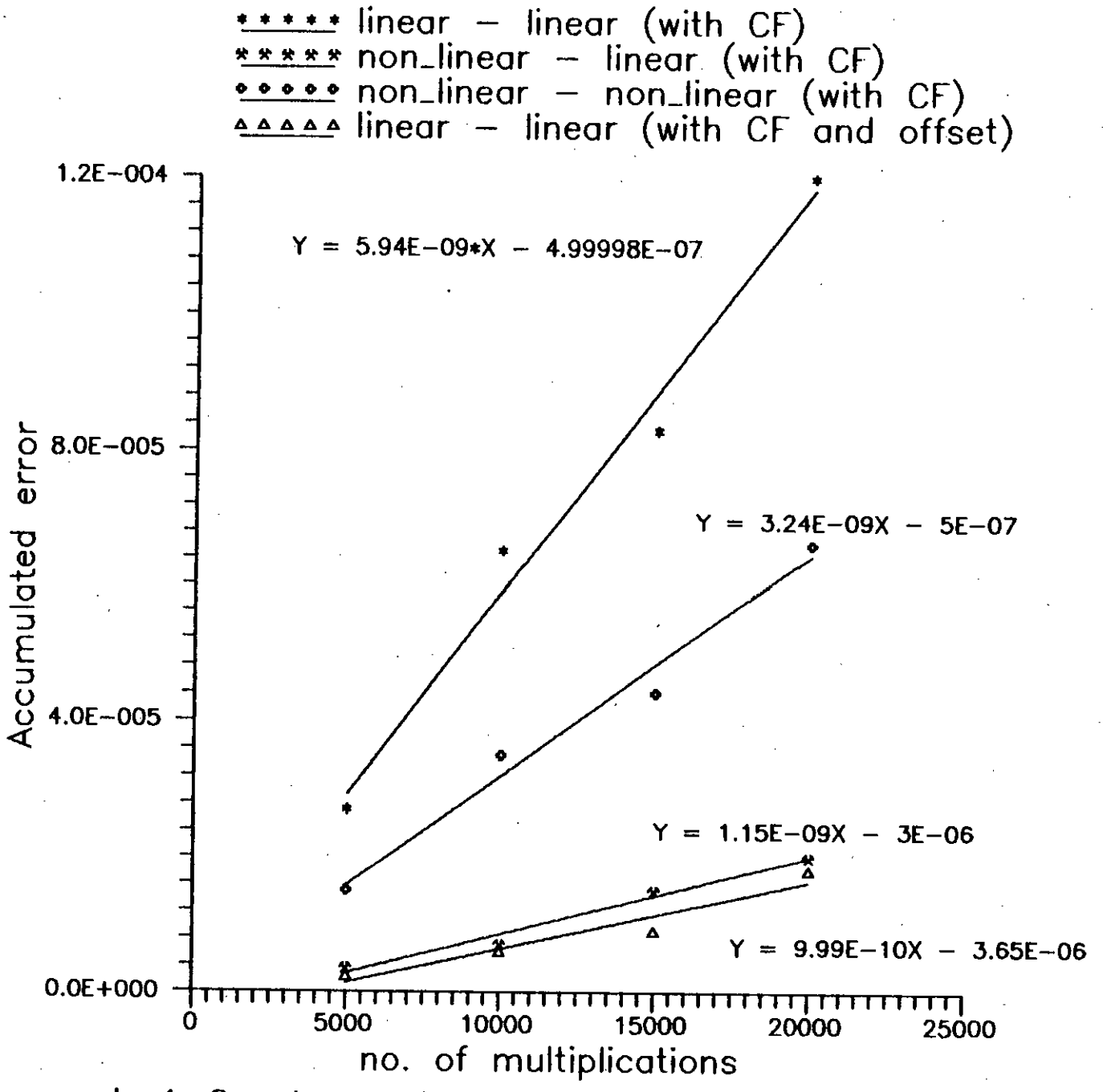
While converting numbers we noted that a significant part of the accumulated error is contributed by the truncation error produced due to the limited storage bit length. As this error always reduces the number while storing, we can increase the value of the correction factor (Δm) so that the new higher value of correction factor will produce a higher value of x' while performing LNS-FLP conversion thus producing less conversion error. A lesser conversion error during one conversion process will eventually reduce the total accumulated error. A selection of compensation

factor of 1.000165 can reduce the accumulated conversion error by more than 100 times as may be seen from the graphs 4-5 and 4-6 for non_linear-linear conversion. The whole process means that we have to multiply the correction factor (only during LNS-FLP conversion) by 1.000165 and to perform all operations with the augmented value of the correction factor. Numerical examples show that the value of CF varies from segment to segment but a uniform selection of CF = 1.000165 produced the best result for non_linear-linear conversion.

Graph 4-5 shows accumulated errors during multiplications for various selection of correction factors and the effect of offset on the other hand graph 4-6 shows the effect of compensating and offset factor for various combinations.



graph 4–5 : Accumulated error for multiplication without CF for RAL 12 and ML 23



graph 4–6 : Accumulated error for multiplication showing compensating (CF=1.000165) and offset effect when RAL 12 and ML 23

The pseudo code for performing multiplication is given below.

```
/* Pseudo code for multiplication */

generate mx and my between 0.0 and 1.0  /* generate_two_randoms() */
com_product = (1.0 + mx) * (1.0 + my) /* product of two FLP numbers */
bin_mx[ML]  <-- mx  /* via in_binary() */
bin_my[ML] <-- my  /* via in_binary() */
rom_mx <-- long returned by RAL bit configuration /* equ_integer() */
rom_my  <-- long returned by RAL bit configuration /* equ_integer() */

for bin_mx[ML]
  open m - lgm conversion data file ("a:m_lg????.dat") and read
    segment value as pointed by rom_mx /* m_lgm_data() */
  calculate mX_linear        /* convert() */
  calculate mX_non_linear  /* convert() */

for bin_my[ML]
  if ( rom_my <> rom_mx )
    open m - lgm conversion data file ("a:m_lg????.dat") and read
      segment value as pointed by rom_my
  calculate mY_linear
  calculate mY_non_linear

mZ_lin_linear  <-- ( mX_linear + mY_linear )
if ( mZ_lin_linear ≥ 1.0 )
  lin_carry = 1.0
  mZ_lin_linear -= 1.0
mZ_non_lin_non_linear  <-- ( mX_non_linear + mY_non_linear )
if ( mZ_non_lin_non_linear ≥ 1.0)
  non_lin_carry = 1.0
  mZ_non_lin_non_linear -= 1.0

bin_mZ[ML] <-- mZ_lin_linear
for bin_mZ[ML]  /* as got from mZ_lin_linear */
  rom_mZ_linear  <-- long returned by RAL bit configuration
  open lgm - m conversion data file ("a:lg_m????.dat") and read
    segment value as pointed by rom_mZ_linear
  calculate mz_lin_linear
  if ( lin_carry = 1.0 )
    lin_lin_product = (1.0 + mz_lin_lin) * 2^1.0
   else lin_lin_product = (1.0 + mz_lin_lin)
  error = com_product - lin_lin_product
  lin_lin_acc_error += error

bin_mZ[ML] <-- mZ_non_lin_non_linear
```

```
for bin_mZ[ML] /*as got from mZ_non_lin_non_linear */
 rom_mZ_non_linear  <-- long returned by RAL bit configuration
 if ( rom_mZ_non_linear <> rom_mZ_linear )
     open lgm - m conversion data file ( "a:lg_m????.dat") and read
segment value as pointed by rom_mZ_non_linear
 calculate mz_non_lin_non_linear
 if ( non_lin_carry = 1.0 )
  non_lin_non_lin_product = (1.0 + mz_non_lin_non_linear) * 2^1.0
 else non_lin_non_lin_product = (1.0 + mz_non_lin_non_linear)
 error = com_product - non_lin_non_lin_product
 non_lin_non_lin_acc_error += error
```

Fig 4-3 : Pseudo-code for multiplication

A table is given below showing the accumulated error for performing squaring operation on 10000 randomly generated numbers.

Table 4-8: Accumulated error for squaring

RAL 12 ML 23		
conversion		accumulated error
linear	linear	2.82925E-03
non-linear	linear	2.77373E-03
non-linear	non-linear	2.80248E-03

A table is also given next showing the accumulated error for performing 10000 divisions on randomly generated numbers.

Table 4-9: Accumulated error for division

RAL 12 ML 23		
conversion		accumulated error
linear	linear	4.00242E-04
non-linear	linear	4.03167E-04
non-linear	non-linear	4.15061E-04

CHAPTER 5

ADDITION SUBTRACTION IN LNS AND DENORMALIZED NUMBER

5.1 Addition and subtraction of long word length in LNS.

Addition and subtraction of long words e.g. 64 bits are feasible by using ROM and a multiplier as may be seen in table 3-5 but multiplication using a large no of bits may slow down the system. A proposal of using Taylor's series [10] uses lesser ROM but also slows down the system due to several access to ROM for logarithmic and anti- logarithmic conversion.

When two real numbers x and y are added the processor receives lgx and lgy as the augend and addend. The augend is augmented by $lg(1 + 2^r)$ where $r = \min\{(lgx - lgy), (lgy - lgx)\}$ when x and y are represented within a small range, a small ROM look-up table can be constructed that accepts r as the input and returns $lg(1 + 2^r)$ as output. If the address input to the ROM is small, the ROM can have $2^{(address\ bits)}$ words. The volume of ROM can be further reduced by noting that when the value of r grows to higher numbers not all the least significant bits of r have any effect on $f(r) = lg(1 + 2^r)$ and as some of the l. s. bits of r can be dropped, the size of the ROM table can be significantly reduced [1].

The proposal [10] uses Taylor's series approximation to reduce the f(r) table. Lewis has partitioned the binary representation of r into several parts which has F fractional bits as shown in fig. 5-1 in which r_i represents the integer portion and $r_h+r_l+r_e$

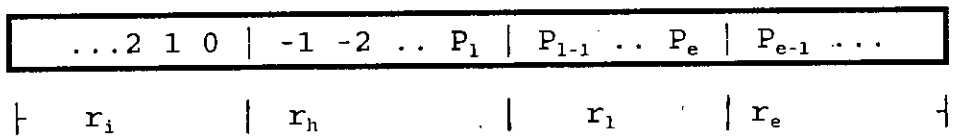


Fig 5-1 : Division of r bits for evaluating Taylor,s series first order approximation.

represents the fractional part and $r_t = r_i + r_h$.

The Taylor's series approximation for constructing the f(r) table can be done by implementing the series by [10]

$$f(r) = f(r_t) + r_l * \frac{df(r_t)}{dr} \text{ ..(3.6 of [10])}$$

where

$$\frac{df(r_t)}{dr} = \frac{2^{r_t}}{1 + 2^{r_t}}$$

while constructing the reduced f(r) table the least significant portion r_e is completely neglected. The hardware structure is given in fig. 5-2 shall realize equation 3.6 of [10] as given

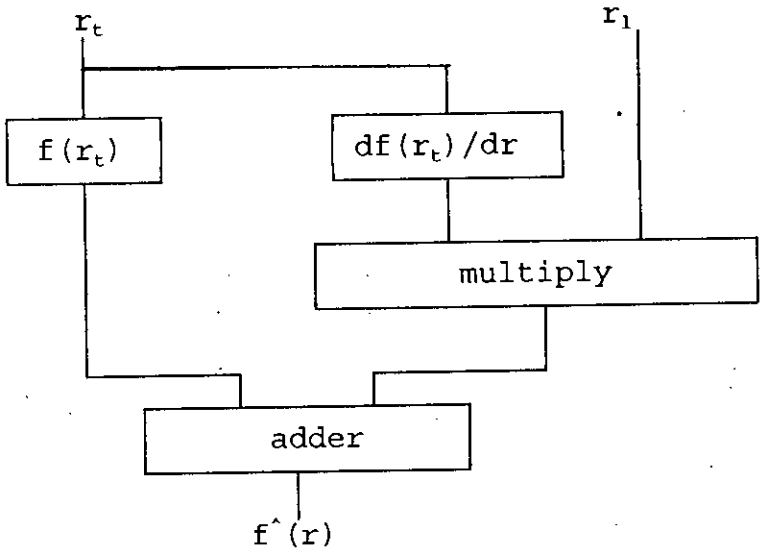


Fig 5-2 : Taylor's series approximation hardware

above. The design uses two look-up tables for $f(r_t)$ and $\frac{df(r_t)}{dr}$ and a multiplier to get the Taylor's series approximation of $f(r)$.

Multiplier less hardware is also described in [10] that converts r_1 into $lg(r_1)$ and another log base 2 conversion of $df(r_t)/dr$ replaces the multiplier by an adder. An antilogarithm table is still necessary but the logarithm table necessary for $lg[df(r_t)/dr]$ can be eliminated because

$$lg[\, df(r_t)/dr] = r_t - lg(\ 1 + 2^{rt}).$$

The error calculation has been determined by the second order approximation of the Taylor's series which includes the effect of r_e and subtracting the second order approximation from the first order approximation and limiting the error to 2^{-F} the limiting value has been calculated to [4.36 of 10]

$$P_1 = \lfloor (-F-r_i-1)/2 \rfloor$$

where P_1 is an integer that indicates mantissa bit position of r such that $P_e \leq P_1$ and $P_1 < 0$ (fractional bit position indicates zero). The limiting value of P_e is given by the expression [4.39 of 10]

$$P_e = \lfloor -F - r_i - 5 \rfloor$$

Analysis in [10] shows that for a 22 bit mantissa conversion while the error is less than 2^{-22} = 2.384E-07 requires a ROM table of 2660 Kbits of total size which is much smaller than the ROM size as may be found necessary after applying the memory reduction technique described in [1].

It has been also shown in [15] that the memory requirement of [10] can be further reduced by 25%.

5.2 **Simulation of Taylor's series approximation**

Taylor's series approximation as depicted in [10] with a multiplier has been simulated in a model program. IEEE single precision number system has been selected so the PRECISION is set to 23. To make f to zero all the 23 bits of its mantissa should be zero and as

$$fa = \log (1.0 + 2^{-rt})/\log(2.0)$$
$$\text{and } fs = \log (1.0 - 2^{-rt})/\log(2.0)$$

a value of r equals to or greater than 23.5 will set it to zero. This can be seen in the flow charts (fig.5-3 and fig.5-4) that r is to be stored in a 28 bit storage length. The integer portion i.e. 23 can be fully represented by a 5 bit configuration so r_i takes the value of 4 as per 3.5 of [10] and fig. 5-1. Fig 5-1 indicates that the integer portion of r_i is counted from 0,1,2 .. so a 5 bit configuration of r_i gives the value of r_i = 4. Once r_i is settled to

Fig 5-3 : Flow chart for logarithmic addition

(using Taylor's series first order approximation)

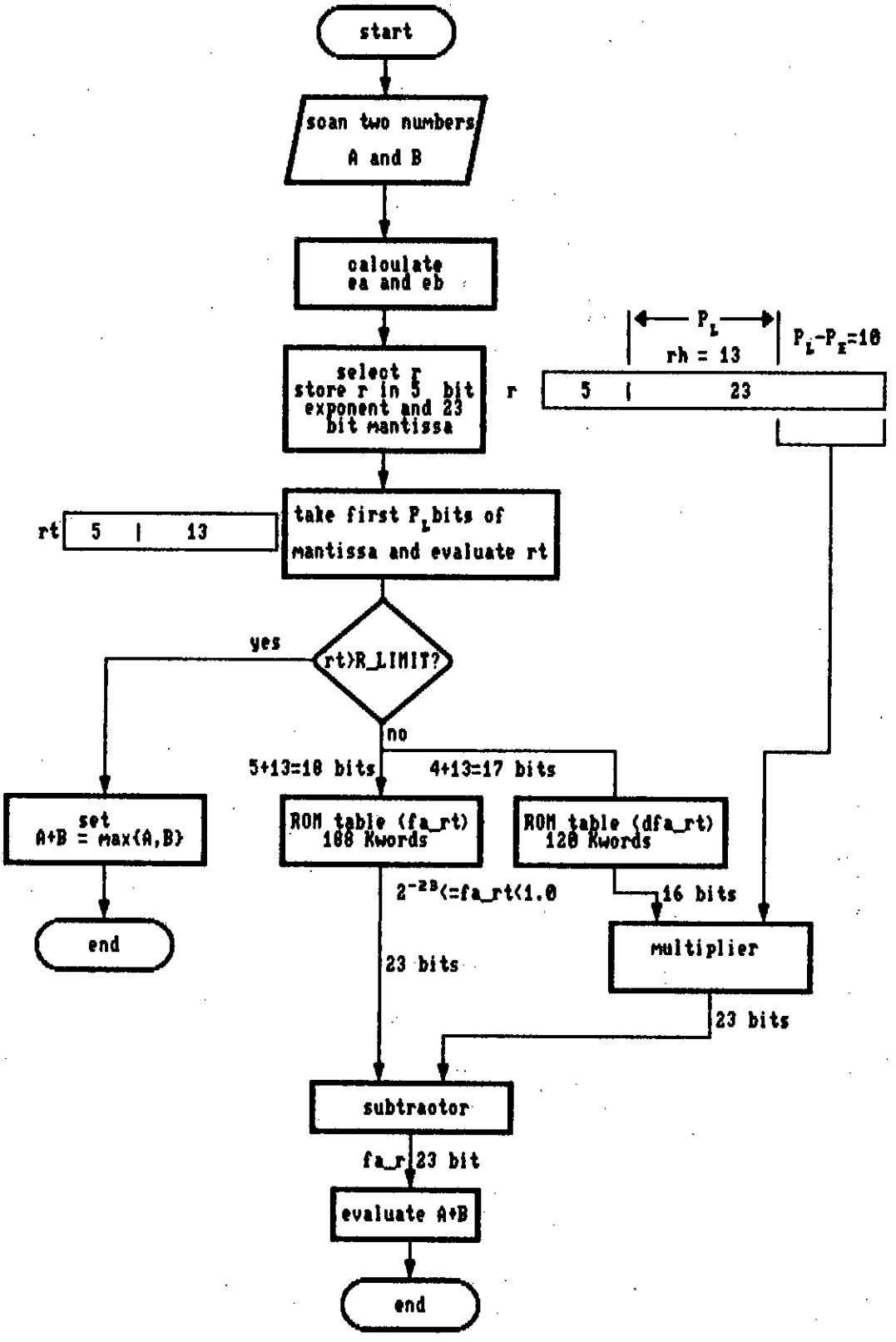
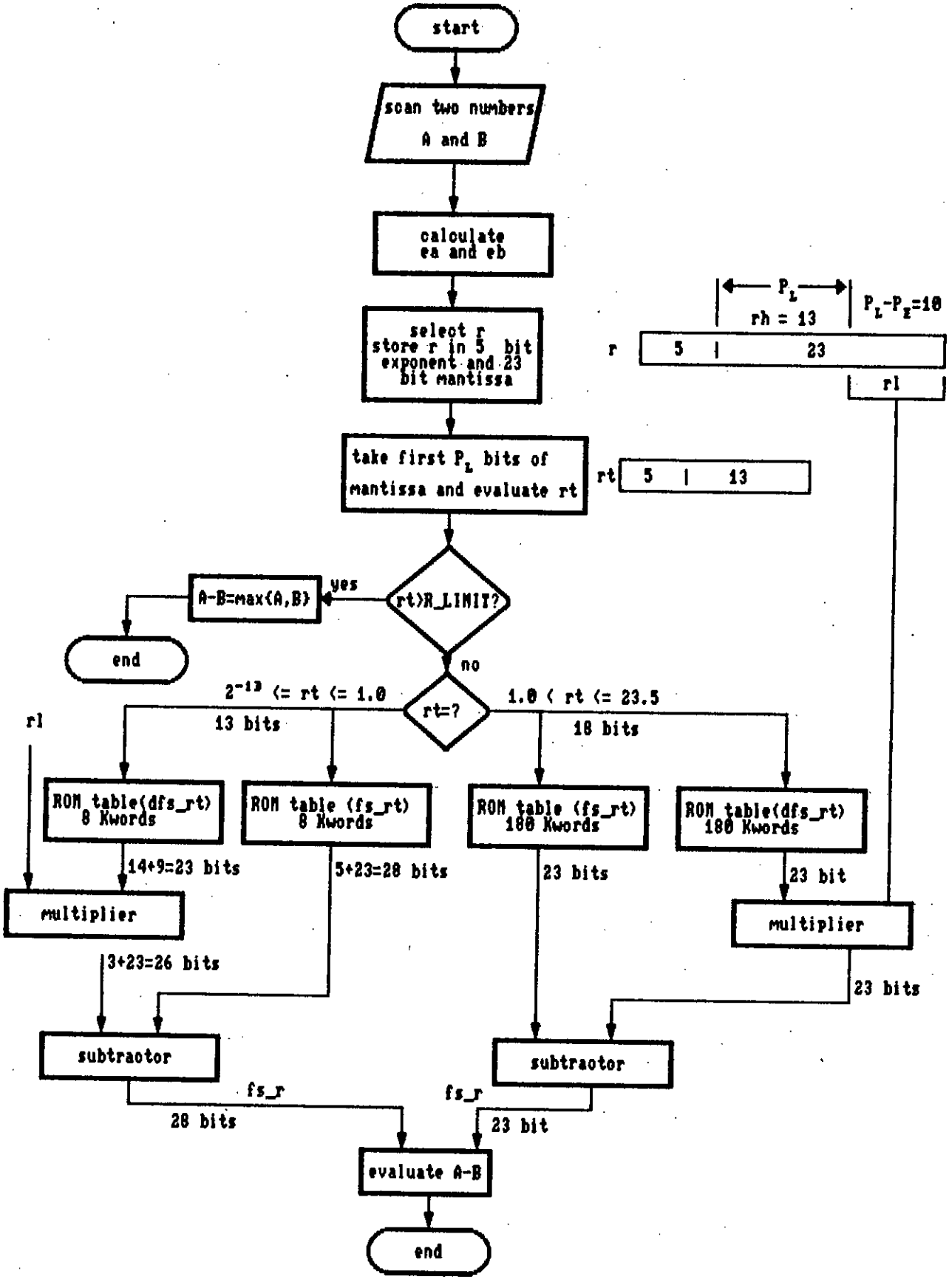


Fig 5-4 : Flow chart for logarithmic subtraction

(using Taylor's series first order approximation)



4, PL becomes 14 as per 4.36 of [10] and article 5.1. PE is set to the last limit i.e. to 23 and r_e is completely neglected.

While evaluating the addition of two numbers r_t is stored in a 5 + 13 = 18 bit format. Although PL has been fixed to 14 we reduced it to 13 and still found the error limit as acceptable. The ROMs necessary for calculating fa_r_t and dfa_r_t with their input address bits of 18 and 17 respectively. The address bits of dfa_r_t ROM has been reduced by one because $dfa\_r_t = 2^{-rt}/(1+2^{-rt})$ turns to zero (for 16 bit representation) at a value of r_t = 15.0 which can be represented by a 4 bit integer instead of 5 bits. From table 5-1 the above mentioned conclusion can also be derived.

Table 5-1 : dfa_r_t for various values of r_t

r_t	dfa_r_t
1.220703E-04	0.4990788
.	.
.	.
0.1	0.4826782552
.	.
.	.
15.0	3.051664683E-05

To calculate the total ROM necessary for addition process we proceeded as follows: It has been stated earlier that if r_t exceeds 23.5 fa_r_t turns to zero when represented in a 23 bit format. r_t is itself represented by a 5 bit integer part and a 13 bit mantissa part. So it is clear that a bit configuration of 10111-1000000000000 is the maximum limit of the r_t. So the ROM necessary for building fa_r_t equals to $2^{17}+2^{15}+2^{14}+2^{13}+2^{12}$ = 188 Kwords. Any bit configuration higher than 23.5 do not need a output from ROM (output is zero). So ROM contents for those configurations are not necessary. As explained above dfa_r_t turns to zero when r_t becomes 15.0. As 15.0 can be represented by 4 bits in the integer portion the maximum upper level of bit configuration of

1111-0000000000000 is to be taken care. So the ROM necessary for building dfa_r_t is $2^{16}+2^{15}+2^{14}+2^{13}$ = 120 Kwords. It is to be mentioned here that both fa_r_t and dfa_r_t shall turn to zero for any higher bit configuration than their maximum limit. So if r_t > R_LIMIT, the sum of the two numbers becomes to equal to the larger of the two numbers as shown in fig. 5-3.

Therefore to complete the addition process total ROM necessary are (188+120) 308 Kwords.

To perform the subtraction process r_t is again stored in a 18 bit format. As we see that both fs_r_t and dfs_r_t are less than -1.0 when r_t becomes less than 1.0 else it is less than zero. So two sets of fs_r_t and dfs_r_t ROMs have been selected. The first one is set for $2^{-23} \leq r_t \leq 1.0$ and the second ROM is set for $1.0 < r_t \leq$ 23.5 (fig. 5-4). The first set has an address lines of 13 bits and the second set has an address line of 18 bits. Total ROM necessary for subtraction process is (8*2 + 180*2) 376 Kwords.

Therefore to perform both addition and subtraction process total ROM necessary is (308 + 376) 684 Kwords.

While simulating Taylor's series approximation we found that the error produced during addition and subtraction process is always less than the error produced when a bit is dropped/added at the lease significant position of the higher number of the pair once the boundary conditions are met. For further clarification we like to process two numbers

A = 80100.0 and

B = 80090.0

The representation of A in IEEE format is

A = 0-10001111-00111000111001000000000

If one bit is added at the least significant position then

A'= 0-10001111-00111000111001000000001

A'= 80100.0078

Therefore the error produced is A' - A = 0.0078. So an error of ±0.0078 is acceptable. When the Taylor's series is simulated with PL = 13 error produced was -6.46E-03 for the addition process. Subtraction requires higher setting of PL to guarantee minimum error condition. Error generated was also checked with very large random number pairs and the maximum error produced was found within the allowed limit.

If we use ROM tables of (5 + 23) 28 bits wide address bus with input range varies between 0.0 and 23.5, then for only addition, two ROM tables with combined capacity of (188 * 2) 376 Mwords are needed.

We have also studied the ROM requirement for other bit configuration of the IEEE number system. The result is summarized in a tabular form as given below. The ROM size is necessary for both addition and subtraction

Accuracy level	ROM size
2^{-20}	512 Kwords
2^{-23}	608 Kwords
2^{-30}	45 Mwords

While calculating R_LIMIT in our model program, we first noted that fa_r < $2^{-PRECISION}$. So

$$fa\_r = \frac{\ln(1.0 + 2^{-r})}{\ln 2} < 2^{-PRECISION}$$

$$\ln(1.0 + 2^{-r}) < 2^{-PRECISION} * \ln 2$$

$$1.0 + 2^{-r} < \exp(2^{-PRECISION} * \ln 2)$$

$$2^{-r} < \exp(2^{-PRECISION} * \ln 2) - 1.0$$

$$-r = \log_2(\exp(2^{-PRECISION} * \ln 2) - 1.0)$$

$$r = -\frac{\ln(\exp(2^{-PRECISION} * \ln 2) - 1.0)}{\ln 2}$$

Therefore R_LIMIT = $r = -\frac{\ln(\exp(2^{-PRECISION} * \ln 2) - 1.0)}{\ln 2}$

5.3 **Denormalized number**

IEEE std. 754 allows the use of very small numbers in the denormalized form (fig. 1-1 and fig. 1-2) so that in the single precession representation the largest positive denormalized number is 1.175494211E-38 (which is just below the smallest positive normalized number of 1.175494491E-38) and the smallest positive denormalized number is 1.401298464E-45. This thesis also includes the way of converting FLP denormalized numbers by using the same ROM table used for normalized number conversion but we do not try to specify the special LNS format for denormalized number because the details of representing denormalized numbers under LNS is still unspecified.

A single precision denormalized FLP number x can be expressed by

$$x = (-1)^s * 2^{-126} * (0.m) \text{ where m is the mantissa}$$

The above number x can also be represented by

$$x = (-1)^s * 2^{-126} * 2^{-1} * (1.m)$$

which can be converted into a LNS number by the same ROM that was used for converting normalized FLP numbers. The portion 2^{-126} can be processed separately with the exponent of the LNS/FLP number to get the final result.

An algorithm is given next (fig 5-5) for LNS multiplication when denormalized FLP numbers are involved. All examples mentioned in this algorithm follows next.

```
/* multiplication algorithm involving denormalized FLP numbers */
ex = exponent of x in FLP
mx = mantissa of x in FLP
Ex = exponent of x in LNS
Mx = mantissa of x in LNS
/* ey, my, Ey, My have the similar meaning for number y */

case 1: /* one number say x is normalized and the other number y is denormalized
FLP number. Convert the denormalized number y into a normalized number by
hiding the -126 of its exponent so that the number can be converted into a LNS
number by using the ROM look- up table */

(1) do
      SHL my with left most bit to carry
      decrement ey
    until ( carry = 1 )
(2) Mx <-- log base 2 of mx /* by ROM table */
    My <-- log base 2 of my /* by ROM table */
    Z <-- X + Y /* add two LNS numbers X and Y to produce Z */
(3) z <-- Z  /* convert LNS number into FLP number by ROM table */
(4) if ( ez is negative ) /* example 1 */
      carry <-- 1
      SHR mz with carry
      increment ez
(5)   while ( ez is not equal to zero )
        SHR mz
        increment ez
      END.
/* z now represents x*y in denormalized form */
(6)   else if ( ez is positive ) /* example 2 */
      increment ez /* increment of ez =  adding (-126+127) */
    END.
/* z now represents x*y in normalized form */

case 2: /* both x any y are denormalized numbers */
   UNDERFLOW /* product is too small */
   END.
```

Fig 5-5 : Multiplication algorithm for denormalized FLP numbers

The following examples associated with multiplication algorithm show error free operation of the algorithm.

The examples given here are associated with article 5.3. The numbers represented by an alphabet can have two forms namely, FLP and LNS among which small letter representation is for FLP and capital letter representation is for LNS.

example 1

let x = 0.7061309815 = 0-01111110-01101001100010100000000
y = 7.714181677E-39 = 0-00000000-10101000000000000000000

therefore z = x*y = 5.447222679E-39

(1) y = 0-11111111-01010000000000000000000
(2) X = 0-11111111-01111111011111010110111
Y = 0-11111111-01100100011011101110101

Z = 0-11111110-11100011111011000101100
(3) z = 0-11111110-11011010100001010001111
(4) z = 0-11111111-11101101010000101000111
(5) z = 0-00000000-01110110101000010100011 = 5.447221278E-39

example 2

let x = 2.764898077E+38 = 0-11111110-10100000000001000000000
y = 7.714181677E-39 = 0-00000000-10101000000000000000000

therefore z = x*y = 2.132892608

(1) y = 0-11111111-01010000000000000000000
(2) X = 0-01111111-10110011010100111001000
Y = 0-11111111-01100100011011101110101

Z = 0-01111111-00010111110000100111101 (Z = X + Y)
(3) z = 0-01111111-00010001000000101001111
(6) z = 0-10000000-00010001000000101001111 = 2.13289237

CHAPTER 6

SOURCES OF ERROR AND APPLICATION OF LNS PROCESSOR

6.1 Various sources of error

When we convert a number x represented in IEEE floating-point format first into LNS format and then reconvert it back into its previous FLP format with the help of ROM tables, we do not get back the original number x but get a new number x′. Although for single number conversion the conversion error (x - x′) is very small and negligible but this conversion error can be accumulated for arithmetic processing that requires large numbers for conversion and anti-conversion.

Here we shall discuss about the error generated at the various stages of conversion and anti-conversion. There are four major sources of error generation and those occurs at two stages namely (1) x to lgx conversion and (2) lgx to x conversion. Among four sources of errors two are contributed at stage (1) and these two sources are (a) conversion error from m to lgm that generates due to the storing Δlgm in limited storage bit length in ROM and (b) truncation error that develops when we store the Δlgm value in a limited storage bit length (ML). The stage (2) contributes the other two sources of error and these two sources are (c) conversion error from lgm to m that generates due to the storing Δm in limited storage bit length in ROM and (d) truncation error that develops when we store the reconverted value of m in a limited storage bit length (ML).

Thus the algebraic sum of (a), (b), (c) and (d) produces the net error for one FLP to LNS to FLP conversion process.

To clarify the matter further we illustrate the whole process with an example. This example has been generated selecting RAL 12 and ML 23.

Let m = 4.884004593E-04 be the mantissa of the number generated. While generating this random number we face no truncation error as this number can be represented fully within a 23 bit format. Thus the binary representation of m should be

m = 4.884004593E-04 = 00000000001000000000001

without any truncation error. The value of Δlgm (correction factor) is equal to 1.718813489E-07. So the value of lgm becomes

lgm = base value of lg(m) + Δlgm * multiplier
= 7.042690112466E-04 + 1.718813489E-07*1
= 7.04440926E-04

Now we are in stage (1) and error (a) is already generated. This error is generated due to the multiplicand bit length. The more the multiplicand bit length, the more preciously the value of Δlgm can be represented thus producing lesser error from source (a).

The number lgm = 7.04440926E-04 must be stored in a 23 bit format so the storage bit configuration becomes

lgm --> 00000000001011100010101

which means that we have actually stored a number whose magnitude is lesser than 7.04440926E-04 and in fact the value of lgm actually represented is 7.04407692E-04. This is the truncation error due to the limited storage bit length as mentioned in (b).

Now we have to convert lgm value represented in LNS format to m value in FLP format. Errors those may be contributed in stage (1) have been accumulated and our next job is to calculate the total error that may be accumulated at stage (2).

We next perform the LNS to FLP conversion process. The value of Δm = 8.2664548E-08. So the value of m′ becomes

m′ = base value of antilg(lgm) + Δm * multiplier
m′ = 3.385080526823E-04 + 8.2664548E-08*1813
m′ = 4.883788728E-04

Now we are stage (2) and error (c) has been generated. The type of this error can be compared with that produced in (a).

The value of m′ as stored in a 23 bit format gives us the exact value of m′ and here the error at source (d) has been generated. If we store m′ in a 23 bit format, then

m′ = 00000000001000000000000
= 4.8828125125E-04

so the conversion error is

m - m′ = (4.884004593 - 4.8828125125) * E-04
= 1.192093E-07

Therefore for 10000 random numbers generated the accumulated error can be as high as 1.1921E-07*10000 = 0.0011921. This result is well compatible with table 4-6.

Other sources of error can be due to the limited representation of the base values of m and lg(m) which we have neglected during our simulation process.

6.2 **Speed improvement**

The most important point of switching to LNS system over the present established form of FLP system is its speed improvement. As the LNS operations are done mainly in addition/subtraction a very fast adder is the most desirable criterion of a viable LNS processor.

Unit gate delay (Δ) of 3 nanosecond (ns) is achievable with readily available commercial TTL chips [11] so the total add time for the 32 bit adder is 24 ns [12]. Using commercial available technology the achievement of 12 ns CLA add time [1] has been also reported which is the outcome of using faster ISL (Integrated Schottky Logic) that gives us unit gate delay of 1.5 ns[1].

For speed improvement a fast ROM is essential. TTL ROM has a typical access time of 150 ns [11] which is not fast enough. ISL ROM gives us a delay of 20 ns [1] so that a 32 bit multiplication speed improvement in LNS system is approximately (200 : 20+10+20) 4 : 1. A division process is 5-10 times slower than multiplication even on coprocessors. As LNS multiplication and division process takes the same period of time 20 : 1 speed improvement is expected under LNS system. For square and square-roots an improvement of 20 : 1 has been reported [1].

6.3 **Application**

Evaluation of trigonometrical functions are also well suited under LNS environment. Bounded elementary functions e.g. sinθ, cosθ etc. can be expressed in the form of an infinite power series. So we can express

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} \cdots$$

that always converges for $0 \leq \theta \leq 90^{\circ}$. Once θ (in radians) is calculated θ^5, θ^7 and others can be obtained by merely shifting the bits. The divisor integers can be stored in the ROM table [12,13] to perform the division process.

As LNS system can perform multiplication/division oriented arithmetic operations much better than the FLP number system it has significant advantage over other in multiplier intensive geometric coordinate transformation and digital signal processing.

In geometric coordinate transformation scaling (changing the size of an image) and rotation is done extensively [14]. If we like to rotate a line between origin and a point $P_1(x_1,y_1)$ by an angle θ so that the new location of the lines is from the origin to a new point $P_2(x_2,y_2)$. To rotate the line to its new location we must have to calculate the length of the line segment and sine and cosine of the angle θ to form the rotation matrix [14].

The length of a line segment having end points between (0,0) and (x,y) can be calculated by

$$L = (x^2 + y^2)^{1/2}$$

and the angles

$$\sin\theta = y / L$$
$$\cos\theta = x / L$$

The operations like calculating the length L can be done faster by using LNS number system. To calculate L we proceed as follows.

At first convert x and y into LNS number system. x^2 and y^2 can be made by merely shifting x and y by one bit to left. x^2 and y^2 are passed through the antilogarithm process and added to the floating

point adder/subtractor unit to get $x^2 + y^2$. A further access to ROM gives us the log base 2 of $x^2 + y^2$ and a shift right by one bit gives us the required length L in LNS system. To get the answer in FLP system a further conversion from LNS to FLP is necessary.

Once the length L and sine and cosines of the angle θ have been calculated the rotation matrix can be constructed and it is post-multiplied with the point P(x,y) which gives us the location of the new coordinates of P.

Suppose we wish to rotate the point P(3,2) counter clockwise by an angle of 30 degrees. Then the rotation matrix will be

$$= \begin{vmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{vmatrix}$$

and the rotated point will be P(1.098, 2.232) as per

$$\begin{vmatrix} 3 & 2 \end{vmatrix} \begin{vmatrix} 0.866 & 0.5 \\ -0.5 & 0.866 \end{vmatrix} = \begin{vmatrix} 1.098 & 2.232 \end{vmatrix}$$

In scaling transformation [14] we usually post multiply a point $P_1 = [x_1, y_1]$ by the transformation matrix. If the transformation matrix is an unit matrix then there is no change in the size of the image but the transformation matrix

$$T = \begin{vmatrix} 0.5 & 0 \\ 0 & 1 \end{vmatrix}$$

will shrink the x co-ordinates to one-half of their original sizes. Similarly the transformation matrix

$$T = \begin{vmatrix} 2 & 0 \\ 0 & 1 \end{vmatrix}$$

will make all x coordinates twice as large as the old value.

Matrix multiplication is well suited under LNS environment and if desired the conversion of the elements of the matrix can be done during DMA transfer further enhancing the performance.

The well known complex FFT algorithm is often used in most of the digital signal processing area to do digital filtering or frequency spectrum analysis. The FFT is implemented by recursive application of the "butterfly" equation [16].

$$X_m[p] = X_{m-1}[p] + W^r_N X_{m-1}[q],$$
$$X_m[q] = X_{m-1}[p] + W^r_N X_{m-1}[q].$$

where W^r_N is the appropriate power of W i.e. $W^r_N = e^{-j2\pi r/N}$ and Xm-1[p], Xm[p-1], Xm-1[q], Xm[q-1] are all complex numbers.

The most difficult for implementing the FFT is that instead of real multiplication, complex multiplication are to be performed. A complex multiplication such as $W^r_N X_{m-1}[q]$ can be decomposed into four real multiplications and two real additions which is well suited under LNS system. It has been shown [3] that it takes up to eight pipeline cycles to complete one butterfly cycle and the performance improvement is 10:6 [3].

The LNS can also minimize computations when the transformation is done as per the Cooley-Tukey FFT algorithm described in [17]. The Cooley-Tukey algorithm can be considered as a method of factoring N x N matrix by n nos. N x N matrices in such a way that the new matrices reduces nos of complex multiplications and additions by

suitably bringing zero terms in the factored matrices. As an example if a 4-point complex exponential W^{nk} matrix is partitioned into two factored matrices then a total 4 nos of complex multiplications and 8 complex additions are to be performed for each factored matrix. Each complex multiplication requires 4 nos. of multiplications and 2 nos. of additions. According to our multiplication addition speed gain ratio of 4:1, an overall speed improvement of 2.5:1 is expected.

CHAPTER 7

CONCLUSIONS AND SUGGESTIONS

7.1 Conclusions

So far we have seen that conversion from FLP to LNS and vice-versa can be done with reasonable size of ROM, the size of which depends on the prestated accuracy. The higher the accuracy, the bigger the ROM is necessary. For example 16 Kwords (512 kbits, word size = 32 bit) of ROM is sufficient for one way conversion if the accuracy level is approximately 6.0E-08. This is applicable for 32 bit word length. The requirement of ROM is very high (2528 Mbits, fig. 3-2) when 64 bit word length is considered and the accuracy is equal to the truncation error of 64 bit word length (2.22E-16).

The conversion from FLP to LNS numbers and vice-versa is straightforward. The absolute value of maximum conversion error produced is encouraging in both way conversion. A single number conversion time may be adequate but for numbers represented by large mantissa, the time to find out the maximum conversion error of all segments is prohibitively large, however search in only one segment is necessary to find the maximum conversion error. A shortcut method exists that will only calculate error at the mid-points of each segment (as well as the difference) which is also linear maximum conversion error for that segment but for non-linear technique larger search time is necessary than linear method. However once difference (Δlgm or Δm) for all the segments are generated conversion and anti-conversion can be done with many choices (table 4-6).

The concept of non-linear correction is to find out a new value of difference so that it will reduce the absolute magnitude of MCE in any segment. While doing this it has been found that a new value of difference can produce the optimum value of MCE so that its average magnitude is always less than that of the linear correction method. In linear correction method the error always lies on one side e.g. linear correction error for FLP to LNS conversion is always positive so it always lies over the horizontal axis in the error graph but non-linear correction factor (difference) produces both positive and negative error.

Shifting of error curve in the upper or lower direction by half of its maximum value has been done by adding or subtracting an offset value. This method gives the lowest accumulated error (table 4-7) but requires one more adder and one more subtractor and the size of the ROM have to be increased by at least 25% due to the storing of segment.max_con_error. The difference of accumulated error obtained by offset and that obtained by non_linear - linear method is negligible as may be seen in graphs 4-5 and 4-6. Non_linear - linear does not require additional hardware and increment of ROM size is not necessary. However long search time is necessary to find out the non-linear value of *difference* (atleast for Δm).

The non_linear - non_linear correction may not be found effective when the ROM is large and the multiplication is performed on numbers represented by fewer number of bits. However when the multiplication plays a much more significant part in the addition process (this means a small value of RAL and comparatively larger value of ML) non-linear conversion can improve overall conversion process accuracy. The selection of non-linear correction gives us more choice and hence the error produced in each conversion may be selected in such a way that they can be compensating at least partially. It has been found in table 4-4 and 4-6 that while converting LNS to FLP, the use of linear correction factor has some self compensating effect. The self compensating effect may further

be increased by pre-multiplying the difference by a suitable compensation factor (CF) and then performing the multiplication process. In general the use of linear and non-linear correction factor in each stage may lead us to many combinations so that the best choice can be selected.

The application of non-linear correction factor obviously reduces total conversion error for one way conversion as we have seen in table 4-5. While we converted x into $\log_2 x$ and then reconvert $\log_2 x$ into x, we noted in table 4-6 that non-linear conversion still reduces the accumulated error. The accumulated error difference has increased in table 4-7 due to the multiplication effect.

The embedding of correction factors in different registers can be very effective in reducing the size of ROM when some lower level of accuracy is acceptable as we have seen in article 3.5. This method also reduces conversion time that makes the overall process much faster. Limited multiplication can be also promising as explained in the same article. In fact tables 3-1,3-2 and 3-5 all have been generated on limited multiplication mode which requires smaller ROM sizes. ROM conversion tables may also lead to some ROM reduction. Table 3-1,3-2 indicates that m-lg(m) and lgm-antilg(lgm) have the same bit at the msb position. Therefore the msb from lg(m) in table 3-1 and from antilg(lgm) in table 3-2 can be dropped.

The accuracy of the conversion depends on the storage bit capacity and due to this reason the accuracy as achieved by using limited hardware multiplier (table 3-1 and table 3-2) is much less than software simulation results. The ROM access time can also become a major stumbling block in developing LNS system. Typical TTL devices are not fast enough for LNS to FLP and FLP to LNS conversion but much faster devices of ISL family can make the LNS processor a practical device. An alternate choice can ECL family chips which can give a unit gate delay of less than 1 ns [12,13].

The complete contents of the ROM for any value of RAL and ML can be printed out at any time. We do not include the complete contents to avoid excessive volume.

While we progressed through our work we have noted that the Turbo C compiler ver. 2.0 from Borland does produce unreliable $\log_2 m$ - m conversion results when the value of $\log_2 m$ is very small (e.g. RAL 26 ML 40). This produced lot of trouble in evaluating the correct results but we continued.

7.2 **Suggestions for further research**

The result of any arithmetic operation (e.g. multiplication/division) using random numbers of double precision FLP number and the corresponding converted LNS numbers is to be studied. This may be conveniently done by generating necessary data for all segments and first storing them on a hard disk and then recalling the data for the particular segment for which the random number has been generated. In this thesis data have been generated and tested through arithmetic simulation up to 23 bit mantissa length (IEEE single precision FLP number) and noted that generation of necessary of data for higher mantissa length requires very long time. Therefore access to a very fast computing system is essential.

As we have mentioned in article 4.4 that accumulation of truncation error contributes a large portion of the accumulated error. Therefore, suitable method is to be investigated that reduces the effect of truncation error. Use of CF as explained in article 4.5 can reduce the effect of truncation error but it should be carefully investigated that after pre-multiplying the difference by the CF (whose value is very close to unity) how much improvement is achievable when the augmented difference is stored in a limited bit storage. The effect of truncation can be minimized by storing

lgm in a longer bit sequence. As may be seen in article 4.2, the numbers 2.07557724E-04 and 2.075620782E-04 have different storage bit value beginning from 26th bit position which means that lgm obtained after applying non_linear correction factor can return a higher value of multiplier which can also lead to partial compensation.

Memory requirement for 64 bit conversion is very high. A suitable method is to be invented so that the memory requirement reduces to an acceptable level.

While doing addition and subtraction using LNS numbers, although we have found that Taylor's first order approximation greatly reduces memory requirement, still the memory requirement is large particularly when the accuracy level is less than IEEE single precision number system. Further study is necessary so that logarithmic addition and subtraction can be done with smaller size memory without reducing the accuracy level.

Interleaved memory interpolators that uses a ROM [18] shows an arithmetic unit with worst case relative error better than the worst case relative error of single precision FLP number. This should be further studied for performing LNS addition and subtraction with a small ROM size.

REFERENCES

1) Fred J. Taylor, R. Gill, Jim Joseph and Jeff Radke " A 20 bit logarithmic number system processor", IEEE trans. on Computer vol. 37 No 2 February 1988.

2) Hao Y Lo and Y Aoki "Generation of a precise binary logarithm with difference grouping programmable logic array", IEEE trans. on Computers vol. C-34, No 8 August 1985.

3) Fang S Lai and Ching F Eric "A hybrid number system processor with geometric and complex arithmetic capabilities", IEEE trans. on Computers vol 40 No 8 August 1991.

4) Ernest L. Hall, David D. Lynch and Samuel J. Dwyer " Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications" IEEE trans on Computer Vol. C-19, No. 2, February 1970.

5) Earl E. Swartzlander, D. V. Satish Chandra, H. Troy Nagle and Scott A. Starks " Sign/Logarithm Arithmetic for FFT Implementations" IEEE trans on Computers, Vol. C-32, No. 6, June 1983.

6) Jerry H. Lang, Charles A. Zukowski, Richard O. Lamaire and Chae H. An " Integrated-Circuit Logarithmic Arithmetic Units", IEEE trans. on Computers vol C-34 No 5 May 1985.

7) IEEE std. 754 - 1985 for binary floating point Arithmetic.

8) M Combet, H V Zonneveld and L Verbeek "Computation of the base two logarithm of binary number",IEEE trans. on Electronic Computers vol. EC-14, No 6 December 1965.

9) Mark G. Arnold, Thomas A. Bailey, John A. Cowles and Mark D.Winkel " Applying Features of IEEE 754 to Sign/Logarithmic Arithmetic" IEEE trans. on Computer Vol. 41 No. 8 August 1992.

10) Devid M. Lewis "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System" IEEE trans. on Computers Vol 39 No 11 November 1990.

11) Fairchild Semiconductor " TTL Data Book" 1982.

12) Kai Hwang " Computer Arithmetic Principles, Architecture and Design" John Wiley & Sons.

13) Joseph J. F. Cavanagh "Digital Computer arithmetic - Design and Implementation" Mc-Graw Hill, 2nd printing 1986.

14) Steven Harrington " Computer Graphics - A Programming Approach",Second Edition, McGraw-Hill, Second Edition.

15) M. Arnold, T. Bailey and J. Cowles " Comments on An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System" IEEE trans on Computer Vol 41 No. 6 June 1992.

16) Alan V. Oppenheim, Ronald W. Schafer " Discrete Time Signal Processing" Prectice-Hall of India (Pvt.) Ltd., 1992.

17) E. O. Brigham, R. E. Morrow "The fast Fourier Transform" - IEEE spectrum December 1967.

18) David M. Lewis " Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic unit" IEEE trans on Computer, Vol 43, No. 8, August 1994.

APPENDIX

PROGRAMS

This Appendix lists two programs i.e. m_lgm.c and lgm_m.c those are mentioned in the text of the preceding chapters. A large number of other programs have been also written but not mentioned in this Appendix for the sake of volume.

```
/* m_lgm.c */

/* converts FLP numbers to LNS numbers. The FLP number is assumed to
represented in IEEE format of which the representation of the  mantissa
portion is controlled by the ML. To convert  FLP mantissa to its
corresponding LNS mantissa  help  of a ROM table should be taken, no of
address bits of which are indicated by the  RAL . To perform the FLP to
LNS conversion using linear  correction factor we have to set the  RAL
for ROM portion  of the mantissa and the ML to represent the mantissa
 portion  of  the  floating  point  number.  If  we  want  to  limit  the
multiplicand bit length DL should be set accordingly,  otherwise a
setting between 32-42 is adequate. The effect of the  multiplicand bit
length can be observed by selecting the second  choice from the main
menu. Linear search may not produce the  optimum value of maximum
conversion error in each segment so the third choice should  be selected
to find the absolute value of the MCE in each segment by non_linear
method.  Selection 4 will produce the same result of 3 except that the
search  in any segment ( count no. represents the segment no. ) can be
done  selectively. Selection 5 represents simulation with ROM without
any  multiplication for correction. Selection 6 to quit  the program.
*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

#define RAL  20        /* msb covered in the ROM,max. value=31 */
#define ML   35        /* maximum length of binary bits */
#define DL   40        /* multiplicand bit length */
#define HIGH_VALUE  7  /* limits no of choices */
#define MAXVAL (unsigned long)pow(2.0,(double)RAL)
#define CONSTANT 1.0/pow(2.0,(double)RAL)
#define FACTOR  CONSTANT/pow(2.0,(double)(ML - RAL))
#define DIVIDEND  pow(2.0,(double)(ML - RAL))
#define MULTIPLIER pow(2.0,(double)(ML - RAL - 1))
#define ALLOWED 1
#define NOT_ALLOWED 0

    struct {
      double m_initial;
      double lg_m_initial;
      double difference;
      float  max_con_error;
      double m;
    } segment;
    struct {
      double m;
      double lgm;
      } new, old;

int choice, disp_count = 0;
unsigned long count;
double difference = 0.0, ideal_difference, increment, at_m;
double  max_con_error = 0.0;
char bin_m[ML+1], bin_difference[DL+1];

void basic_difference_error();
```

```
void mid_value_error();
double modify_difference ( double value );
void lin_int_values (double new_m, double old_m, double old_lgm);
void non_lin_int_values (double new_m, double old_m, double old_lgm);
void optimum_difference_error ();
void find_binary ( char *ptr, double value, int n);
void rom_only();

main()
{
     do {
        puts ("");
           puts ("1 - Linear difference without the effect of DL setting ");
        puts ("2 - Linear difference with DL setting effective");
        puts ("3 - Simulation with non-linear correction factor");
        puts ("4 - Interactive version");
        puts ("5 - Simulation with ROM  without multiplication");
        puts ("6 - Quit");
        puts ("7 - Only mid values - a short cut attempt");
        puts ("");
        printf("%s","Press the key of choice... ");
        scanf ("%d",&choice);
        switch (choice)
        {
           case 1 :
           case 2 : clrscr();
                    disp_count = 0;
                    max_con_error = 0.0;
                    old.m = old.lgm = 0.0;
                    increment = CONSTANT;
                    for ( count = 0; count < MAXVAL; count++ )
                   basic_difference_error();
                    break;
           case 3 : old.m = old.lgm = 0.0;
                    increment = CONSTANT;
                    printf("\nm_initial         lgm_initial       ");
                    printf (" difference         max_con_error\n\n");
                    for ( count = 0; count < MAXVAL; count++ )
                    {
                    basic_difference_error ();
                    optimum_difference_error ();
                    }
                    break;
           case 4 : printf ("Enter count no: ");
                    scanf ("%ld",&count);
                    increment = CONSTANT * (count + 1);
                    old.m = increment - CONSTANT;
                    old.lgm = log ( 1.0 + old.m)/log(2.0);
                    printf("\nm_initial        lgm_initial    ");
                    printf("   difference         max_con_error\n\n");
                    for ( ; count < MAXVAL ; count++)
                    {
                   basic_difference_error();
                   optimum_difference_error();
                    }
                    break;
```

```
            case 5 : rom_only();
                     break;
            case 6 : break;
            case 7 : mid_value_error();
                     break;
            default: puts ("Bad choice");
                     break;
          } /* end of switch */
          if ( choice >= 1 && choice <= HIGH_VALUE ) continue;
          printf ("Only 1 through %d are permitted\n",HIGH_VALUE);
        }  while ( choice != 6 );
}
void basic_difference_error ()
{
      segment.m_initial = old.m;
      segment.lg_m_initial = old.lgm;
      new.m = increment;
      new.lgm = log(1.0 + new.m)/log(2.0);
      difference = (new.lgm - old.lgm)/DIVIDEND;
      if ( choice == 2 )
       difference = modify_difference ( difference);
      segment.difference = difference;
      if ( choice == 3 || choice == 4 )
        non_lin_int_values (new.m, old.m, old.lgm);
        else   lin_int_values (new.m, old.m, old.lgm);
      old.m = new.m;
      old.lgm = new.lgm;
      increment += CONSTANT;
}
void mid_value_error ()
{
      unsigned long mul_factor = MULTIPLIER;
      double mid_m, high_value, low_value, lg_mid_m, add_factor;
      double error, max_con_error = 0.0, temp_max_con_error = 0.0;

      increment = CONSTANT;
      old.m = old.lgm = 0.0;
      clrscr();
      for ( count = 0; count < MAXVAL; count++ )
      {
        segment.m_initial = old.m;
        segment.lg_m_initial = old.lgm;
        new.m = increment;
        new.lgm = log(1.0 + new.m)/log(2.0);
        difference = (new.lgm - old.lgm)/DIVIDEND;
        mid_m = (old.m + new.m)/2.0;
        lg_mid_m = log(1.0 + mid_m)/log(2.0);
        add_factor = difference * mul_factor;
        temp_max_con_error = lg_mid_m - (old.lgm + add_factor);
        if ( temp_max_con_error > max_con_error )
        {
       max_con_error = temp_max_con_error;
       gotoxy(8,5);
        printf("Max. error =%E occurred at %18.18f", max_con_error,mid_m);
        }
/* now check higher and lower side to confirm that no other value */
```

```
/* higher or lower than the middle value produces higher error    */
/* first checking the lower side  */
       low_value = mid_m - FACTOR;
       lg_mid_m = log(1.0 + low_value)/log(2.0);
       add_factor -= difference;
       error = lg_mid_m - ( old.lgm + add_factor );
       if ( error > temp_max_con_error ){
         gotoxy(8,7);
         printf("Max. error =%E",temp_max_con_error);
         gotoxy(8,8);
         printf("Error produced at lower  side =%E",error);
       }
/* now checking the higher side   */
       high_value = mid_m + FACTOR;
       lg_mid_m = log(1.0 + high_value)/log(2.0);
       add_factor += 2.0*difference;
       error = lg_mid_m - ( old.lgm + add_factor );
       if ( error > temp_max_con_error ) {
          gotoxy(8,7);
          printf("Max. error =%E",temp_max_con_error);
          gotoxy(8,8);
          printf("Error produced at higher side =%E",error);
       }
        disp_count++;
        if ( disp_count == 500)    {
        gotoxy(20,3);
        printf("%10.10f",mid_m);
        disp_count = 0;          }

        old.m = new.m;
        old.lgm = new.lgm;
        increment += CONSTANT;
     }
}
double modify_difference ( double value )
/* stores a real number < 1.0 in a n-bit binary field and  */
/* recalculates its real value based on its n-bit length   */
{
     int modify_count = 0, no_left_shift = 0;
     double addend = 0.5;
     char *cptr;
     cptr = &bin_difference[0];
/* find the first occurrence of '1' and discard all the leading '0's */
     while ( value <= 0.5 ) {
       value += value;
       no_left_shift++;      }
     while ( modify_count < DL )
     {
     value += value;
     if ( value >= 1.0 ){
         *cptr = '1';
         value -= 1.0;  }
       else *cptr = '0';
     cptr++;
     modify_count++;
     }
```

```
     *cptr = '\x0';
/* recalculate the value of difference based on storage bit length */
     cptr = &bin_difference[0];
     value = 0.0;
     do {
        if ( *cptr == '1' ) {
           value += addend;
           addend *= 0.5;  }
         else addend *= 0.5;
        cptr++;
      } while ( *cptr != '\x0');
/* make necessary right_shifts i.e. division */
     value = value / pow(2.0,(double)(no_left_shift));
     return ( value );
}
void lin_int_values (double new_m, double old_m, double old_lgm)
/* generate all values those can be fully represented by  */
/* ML within one segment                                  */
{
     double lg_old_m, mul_factor, error = 0.0, add_factor = 0.0;
     char *cptr;
     while (old_m < new_m)
     {
     old_m += FACTOR;
     lg_old_m = log(1.0 + old_m)/log(2.0);
     find_binary ( &bin_m[0], old_m, ML );
     cptr = &bin_m[RAL];
     mul_factor = MULTIPLIER;
     do {
          if (*cptr == '1')     add_factor += difference * mul_factor;
          *cptr++;
          mul_factor = mul_factor/2.0;
        } while (*cptr != '\x0');
     if (add_factor == 0.0 || old_m == 1.0)
       add_factor = difference * DIVIDEND;
     error = lg_old_m - (old_lgm + add_factor);
     if ( error > max_con_error )
     {
       max_con_error = error;
       at_m = old_m;
       gotoxy(8,5);
         printf("Max. error =%E occurred at %18.18f", max_con_error,at_m);
     }
     disp_count++;
     if ( disp_count == 1000 )
     {
       gotoxy(8,7);
       printf("scanning %18.18f and above with error=%E",old_m,error);
       disp_count = 0;
     }
     add_factor = 0.0;
     }
}
double get_error (double m, double log2m, double differ);
/* function local to non_lin_int_values function     */
void non_lin_int_values (double new_m, double old_m, double old_lgm)
```

```
{
     float error, temp_max_con_error= 0.0;
     double at_m;

     while ( old_m < new_m)
     {
       old_m += FACTOR;
       error = get_error (old_m, old_lgm, segment.difference);
       if ( fabs(error) > temp_max_con_error) {
        temp_max_con_error = error;
        at_m = old_m;                          }
     }
     segment.max_con_error = temp_max_con_error;
     segment.m = at_m;
     printf ("%15.15f %15.15f %15.15f %+E\n",segment.m_initial,
      segment.lg_m_initial,segment.difference,segment.max_con_error);
}
double find_ideal_difference ( double m );
/* function local to optimum_difference_error function */
void optimum_difference_error ()
/* calculates the new value of segment.difference that produces */
/* the lowest absolute value of the segment.max_con_error       */
{
     float error, temp_new_con_error = 0.0;
     float left_new_con_error, right_new_con_error;
     double temp_new_seg_m;
     double old_m, old_lgm, new_m;
     double low_end_difference, mid_difference, high_end_difference;
     double left_ideal_difference, right_ideal_difference;
     double left_new_seg_m, right_new_seg_m;
     unsigned char left_search, right_search;

     left_search = right_search = ALLOWED;
     old_m = segment.m_initial;
     old_lgm = log( 1.0 + old_m )/log(2.0);
     new_m = old_m + CONSTANT;

     ideal_difference = find_ideal_difference ( segment.m );

     while (old_m < new_m)
     {
       old_m += FACTOR;
      error = get_error (old_m, old_lgm, ideal_difference);
       if ( fabs(error) > fabs(temp_new_con_error)){
      temp_new_con_error = error;
      temp_new_seg_m = old_m;                    }
     } /* while (old_m < new_m)  */
     if ( fabs(temp_new_con_error) < segment.max_con_error) {
/* here ideal_difference cannot produce a value of temp_max_con_error */
/* that is less than segment.max_con_erro. In fact the value of the   */
/* ideal_difference is the higher side limit in the search procedure  */
       puts("ERROR"); exit(0);                              }

     low_end_difference = segment.difference;
     mid_difference = ideal_difference;
     high_end_difference = ideal_difference;
```

```
left_new_con_error = segment.max_con_error;
right_new_con_error = segment.max_con_error;
if ( fabs(temp_new_con_error) > segment.max_con_error &&
   temp_new_con_error < 0.0 )   right_search = NOT_ALLOWED;
  else        {
  puts("Algorithm don't consider such situation for swing.");
  exit(0); }
while (left_search == ALLOWED || right_search == ALLOWED)
{
  if ( right_search )
  {
   temp_new_con_error = 0.0;
   ideal_difference = (mid_difference + high_end_difference)/2.0;
   right_ideal_difference = ideal_difference;
   old_m = segment.m_initial;
   old_lgm = log (1.0 + old_m)/log(2.0);
   new_m = old_m + CONSTANT;

 while ( old_m < new_m )
 {
   old_m += FACTOR;
   error = get_error (old_m, old_lgm, ideal_difference);
   if ( fabs(error) > fabs(temp_new_con_error)) {
     temp_new_con_error = error;
     temp_new_seg_m = old_m;                    }
 } /* while */
 right_new_con_error = temp_new_con_error;
 right_new_seg_m = temp_new_seg_m;
 } /* if ( right_search ) */
 if ( left_search )
 {
temp_new_con_error = 0.0;
ideal_difference = (low_end_difference + mid_difference)/2.0;
left_ideal_difference = ideal_difference;
old_m = segment.m_initial;
old_lgm = log( 1.0 + old_m )/log( 2.0 );
new_m = old_m + CONSTANT;

while ( old_m < new_m )
{
  old_m += FACTOR;
  error = get_error (old_m, old_lgm, ideal_difference);
  if ( fabs(error) > fabs(temp_new_con_error)) {
    temp_new_con_error = error;
    temp_new_seg_m = old_m;                    }
} /* while */
left_new_con_error = temp_new_con_error;
left_new_seg_m = temp_new_seg_m;
 } /* if left_search() */

 if (fabs(left_new_con_error) >= segment.max_con_error
                 &&  left_new_con_error < 0.0 ) {
right_search = NOT_ALLOWED;
left_search  = ALLOWED;
mid_difference = left_ideal_difference;                    }
 else if (fabs(left_new_con_error) < fabs(right_new_con_error)
```

```
          && fabs(left_new_con_error) < fabs(segment.max_con_error)){
          segment.difference = left_ideal_difference;
          segment.max_con_error = left_new_con_error;
          segment.m = left_new_seg_m;
          left_search = right_search = ALLOWED;
          high_end_difference = mid_difference;
          mid_difference = left_ideal_difference;  }
        else if (fabs(right_new_con_error) < fabs(left_new_con_error)
          && fabs(right_new_con_error) < fabs(segment.max_con_error)){
          segment.difference = right_ideal_difference;
          segment.max_con_error = right_new_con_error;
          segment.m = right_new_seg_m;
          left_search = right_search = ALLOWED;
          low_end_difference = mid_difference;
          mid_difference = right_ideal_difference;   }
        else if(fabs(right_new_con_error) > fabs(segment.max_con_error)
          && fabs(left_new_con_error) > fabs(segment.max_con_error)){
          right_search = NOT_ALLOWED;
          left_search = NOT_ALLOWED; }
        else      {
          puts("we don't consider such situation in flow-control");
          exit(0); }

      } /* while ( right_search || left_search == ALLOWED ) */
      printf ("%15.15f %15.15f %15.15f %+E\n\n",segment.m_initial,
       segment.lg_m_initial,segment.difference,segment.max_con_error);
      putch('\a');
} /* optimum_difference_error */

double find_ideal_difference ( double m )
/* calculates a new value of difference that turn the  */
/* segment.max_con_error into zero as calculated in    */
/* the non_lin_int_values function                     */
{
      int divisor = 0, add_factor;
      double lgm;
      char *cptr;

      add_factor = (long)pow(2.0,(double)(ML - RAL))/2.0;
      find_binary (bin_m, m, ML);
      cptr = &bin_m[RAL];
      do {
        if ( *cptr == '1' ) divisor  += add_factor;
        *cptr++;
        add_factor = add_factor/2.0;
      } while ( *cptr != '\x0');
      lgm = log( 1.0 + ( m - FACTOR*divisor ))/log(2.0);
      ideal_difference = ( log( 1.0 + m )/log(2.0) - lgm)/divisor;
      return ( ideal_difference );
}

double get_error (double m,double log2m,double differ)
/* accepts a real value 0.0 < m < 1.0 and calculates its base  */
/* 2 logarithm value, correction factor is added with the base */
/* logarithm value and the error is calculated                 */
{
```

```
        static double temp_lgm;
        double multiplier, deviation, add_factor = 0.0;
        char *cptr;

        difference = differ;
        temp_lgm = log(1.0 + m)/log(2.0);
        find_binary(bin_m, m, ML);
        cptr = &bin_m[RAL];
        multiplier = MULTIPLIER;
        do {
           if ( *cptr == '1')
             add_factor += difference * multiplier;
           *cptr++;
           multiplier = multiplier/2.0;
         } while ( *cptr != '\x0');
        if ( add_factor == 0.0 || m == 1.0)
       add_factor = difference * DIVIDEND;
        deviation = temp_lgm -(log2m + add_factor);
        return ( deviation );
}

void find_binary (char *ptr, double value, int n)
/* accepts a real value < 1.0 and generates  */
/* its n-bit binary representation           */
{
     int in_count = 0;
     while (in_count < n)
     {
       value += value;
       if (value >= 1.0) {
       *ptr = '1';
       value -= 1.0;  }
     else
       *ptr = '0';
      ptr++;
       in_count++;
     }
     *ptr = '\x0';
}

void rom_only()
/* simulation of FLP to LNS conversion using only ROM */
/* the ROM capacity is to be determined by RAL   */
/* neglect the values of ML  and  DL */
{
    int c=0;
    double increment = 1.0/pow(2.0,(double)RAL);
    double m = 0.0, next_m, mid_value;
    double error, max_con_error = 0.0, at_m;
    double lg_m, lg_next_m, lg_mid_value;

    clrscr();
    do
    {
       next_m = m + increment;
       mid_value = (m + next_m)/2.0;
```

```
        lg_next_m = log(1 + next_m)/log(2.0);
        lg_mid_value = log( 1.0 + mid_value ) / log(2.0);
        error = lg_next_m - lg_mid_value;
        if ( error > max_con_error ){
          max_con_error = error;
          at_m = mid_value;
          gotoxy(8,5);
             printf("Max. error =%E occurred at %18.18f", max_con_error,at_m);
        }
        c++;
       if (c == 5000) {
       gotoxy(8,7);
        printf("scanning %18.18f and above with error =%E",
                                                  mid_value,error);
       c = 0;        }
        m = m + increment;
    }  while ( m < 1.0 );
}
```

```
/* lgm_m.c */

/* converts LNS numbers to FLP numbers. The LNS number is assumed to
represented in IEEE format of which the representation of the  mantissa
portion is controlled by ML. To convert  LNS mantissa to its
corresponding FLP mantissa we take the help  of a ROM table, no of
address bits of which are indicated by the RAL . To perform the LNS to
FLP conversion using linear  correction factor we have to set the RAL
for ROM portion  of the mantissa and the ML to represent the mantissa
 portion of the floating point number. If we want to limit the
multiplicand bit length DL should be set accordingly,  otherwise a
setting between 32-42 is adequate. The effect of the  multiplicand bit
length can be observed by selecting the second  choice from the main
menu. Linear search may not produce the optimum value of maximum
conversion error in each segment so the third choice should  be selected
to find the absolute value of the MCE in each segment produced by
non_linear method.  Selection 4 will produce the same result of 3 except
that the search  in any segment ( count no. represents the segment no.)
can be done  selectively. Selection 5 represents simulation with ROM
without any  multiplication for correction. Selection 6 to quit  the
program. */
#include <stdio.h>
#include <conio.h>
#include <math.h>

#define RAL  25         /* msb covered in the ROM */
#define ML   37         /* maximum length of binary bits */
#define DL   40         /* multiplicand bit length */
#define HIGH_VALUE  8   /* limits no of choices */
#define MAXVAL (unsigned long)pow(2.0,(double)RAL)
#define CONSTANT 1.0/pow(2.0,(double)RAL)
#define FACTOR  CONSTANT/pow(2.0,(double)(ML - RAL))
#define DIVIDEND  pow(2.0,(double)(ML - RAL))
#define MULTIPLIER pow(2.0,(double)( ML - RAL -1))
#define ALLOWED 1
#define NOT_ALLOWED 0

struct {
     double lgm_initial;
     double antilg_lgm_initial;
     double difference;
     float max_con_error;
     double lgm;
     } segment;
static struct {
     double lgm;
     double m;
     } new,old;

int choice, disp_count = 0;
unsigned long count;
double difference = 0.0, ideal_difference, increment, at_lgm;
double max_con_error = 0.0;
char bin_lgm[ML+1], bin_difference[DL+1];

void basic_difference_error();
```

```
void reverse_basic_difference_error();
double modify_difference ( double value );
void lin_int_values ( double new_lgm, double old_lgm, double old_m );
void reverse_int_values (double new_lgm,double old_lgm,double old_m);
void non_lin_int_values (double new_lgm,double old_lgm,double old_m);
void optimum_difference_error();
void find_binary ( char *ptr, double value, int n );
void rom_only();

main()
{
     do {
        puts("");
        puts("1 - Linear difference without the effect of DL setting");
        puts("2 - Linear difference with DL setting effective");
        puts("3 - Simulation with non-linear correction factor");
        puts("4 - Interactive version");
        puts("5 - Simulation with ROM without multiplication");
        puts("6 - Quit");
/* In fact selection 7 and 8 do the same function of items 1 and 2
   respectively but instead of starting from the lowest segment ( no
   0 ) number, selection 7 and 8 begins from the highest segment
      no. to accelerate the process of finding the max. conversion error */

        puts("7 - Linear difference without the effect of DL setting
                                                          (1-0)");
        puts("8 - Linear difference with DL setting effective (1-0)");
        puts("");
        printf("Press the key of choice... ");
        scanf("%d",&choice);
        switch ( choice )
        {
           case 1 :
           case 2 : clrscr();
                    max_con_error = 0.0;
                    old.lgm = old.m = 0.0;
                    increment = CONSTANT;
                    for ( count = 0; count < MAXVAL; count++ )
                     basic_difference_error();
                     break;
           case 3 : old.lgm = old.m = 0.0;
                    increment = CONSTANT;
                    printf("\nRAL = %d",RAL);
                    printf("  ML = %d\n",ML);
                    printf("\nlgm_initial       antilg_lgm_initial  ");
                    printf("  difference      max_con_error\n\n");
                    for ( count =0; count < MAXVAL; count++)
                    {
                    basic_difference_error ();
                    optimum_difference_error();
                    }
                    break;
           case 4 : printf( "Enter count no: ");
                    scanf("%ld",&count);
                    increment = CONSTANT * ( count + 1);
                    old.lgm = increment - CONSTANT;
```

```
                old.m = pow(2.0, old.lgm) - 1.0;
               printf("\nlgm_initial      antilg_lgm_initial  ");
               printf("difference       max_con_error\n\n");
                for ( ; count < MAXVAL; count++ )
                {
                basic_difference_error();
                optimum_difference_error();
                }
                break;
          case 5 : rom_only();
                break;
          case 6 : break;
          case 7 :
          case 8 : clrscr();
                max_con_error = 0.0;
                old.lgm = old.m = 1.0;
                increment = 1.0 - CONSTANT;
                reverse_basic_difference_error();
                break;
          default: puts("Bad choice");
                break;
        } /* end of switch */
        if ( choice >= 1 && choice <= HIGH_VALUE ) continue;
        printf( "Only 1 through %d are permitted",HIGH_VALUE);
      } while ( choice != 6 );
}

void basic_difference_error()
{

     segment.lgm_initial = old.lgm;
     segment.antilg_lgm_initial = old.m;
     new.lgm = increment;
     new.m = pow(2.0,new.lgm) - 1.0;
     difference = (new.m - old.m)/DIVIDEND;
     if ( choice == 2 ) difference = modify_difference ( difference );
     segment.difference = difference;
     if ( choice == 3 || choice == 4)
           non_lin_int_values ( new.lgm, old.lgm, old.m);
     else      lin_int_values ( new.lgm, old.lgm, old.m);
     old.lgm = new.lgm;
     old.m = new.m;
     increment += CONSTANT;
}
void reverse_basic_difference_error()
{
     clrscr();
          for (count = MAXVAL - 1; (count >= 0) && (count < MAXVAL); count--)
     {
     new.lgm = increment;
     new.m = pow(2.0,new.lgm) - 1.0;
     difference=(old.m - new.m)/DIVIDEND;
     if ( choice == 8 ) difference = modify_difference (difference);
     reverse_int_values (new.lgm, old.lgm, new.m);
     old.lgm = new.lgm;
     old.m = new.m;
```

```
      increment -= CONSTANT;
     }
}

double  modify_difference ( double value )
/* stores a real value < 1.0 in a n-bit binary field and
   recalculates its real value based on its n bit length */
{
     int modify_count = 0, no_left_shift =0;
     double addend = 0.5;
     char *cptr;

     cptr = &bin_difference[0];
/* find the first occurrence of '1' and discard the leading '0's */
     while ( value < 0.5 )
     {
     value += value;
     no_left_shift++;
     }
     while ( modify_count < DL )
     {
     value += value;
     if ( value >= 1.0 ) {
         *cptr = '1';
         value -= 1.0;   }
       else *cptr = '0';
     cptr++;
     modify_count++;
     }
     *cptr = '\x0';
/* recalculate the value of the difference based on storage bit length
*/
     cptr = &bin_difference[0];
     value = 0.0;
     do {
        if ( *cptr == '1' ) {
          value += addend;
          addend *= 0.5; }
         else addend *= 0.5;
        cptr++;
      }  while ( *cptr != '\x0' );
/* make necessary right_shift i.e. division */
     value = value / pow( 2.0, (double)(no_left_shift));
     return ( value );
}

void lin_int_values ( double new_lgm, double old_lgm, double old_m)
{
     double antilg_old_lgm, mul_factor, error = 0.0, add_factor = 0.0;
     char *cptr;
     while (old_lgm < new_lgm)
     {
     old_lgm += FACTOR;
     antilg_old_lgm = pow(2.0,old_lgm) - 1.0;
     find_binary(&bin_lgm[0], old_lgm, ML);
     cptr = &bin_lgm[RAL];
```

```
      mul_factor = MULTIPLIER;
      do {
           if (*cptr == '1')     add_factor += difference * mul_factor;
           *cptr++;
           mul_factor = mul_factor/2.0;
         } while (*cptr != '\x0');
      if (add_factor == 0.0 || old_lgm == 1.0)
        add_factor = difference * DIVIDEND;
      error = antilg_old_lgm - (old_m + add_factor);
      if ( fabs(error) > fabs(max_con_error) ) {
        max_con_error = error;
        at_lgm = old_lgm;
        gotoxy(8,5);
          printf("Max.    error    =%E    occurred    at    %18.18f",
max_con_error,at_lgm); }
      disp_count++;
      if ( disp_count == 5000 ) {
        gotoxy(8,7);
          printf("scanning %18.18f and above with error=%E",old_lgm,error);
        disp_count = 0;        }
      add_factor = 0.0;
      }
}

void reverse_int_values (double new_lgm, double old_lgm, double new_m)
{
      double antilg_old_lgm, mul_factor, error = 0.0, add_factor = 0.0;
      char *cptr;
      while (old_lgm > new_lgm)
      {
      old_lgm -= FACTOR;
      antilg_old_lgm = pow(2.0,old_lgm) - 1.0;
      find_binary(&bin_lgm[0], old_lgm, ML);
      cptr = &bin_lgm[RAL];
      mul_factor = MULTIPLIER;
      do {
           if (*cptr == '1')     add_factor += difference * mul_factor;
           *cptr++;
           mul_factor = mul_factor/2.0;
         } while (*cptr != '\x0');
      if (add_factor == 0.0 || old_lgm == 0.0) add_factor = 0.0;
      error = antilg_old_lgm - (new_m + add_factor);
      if ( fabs(error) > fabs(max_con_error) ) {
        max_con_error = error;
        at_lgm = old_lgm;
        gotoxy(8,5);
             printf("Max.error =%E occurred at %18.18f",max_con_error,at_lgm);}
      disp_count++;
      if ( disp_count == 5000 ) {
        gotoxy(8,7);
          printf("scanning %15.15f and below with error=%E",old_lgm,error);
        disp_count = 0;          }
      add_factor = 0.0;
      }
}
```

```
double get_error ( double lgm, double anti_lgm, double differ );
void non_lin_int_values (double new_lgm,double old_lgm,double old_m)
{
     float error, temp_max_con_error = 0.0;
     double at_lgm;

     while ( old_lgm < new_lgm)
     {
       old_lgm += FACTOR;
       error = get_error ( old_lgm, old_m, segment.difference);
       if ( fabs(error) > fabs(temp_max_con_error)) {
           temp_max_con_error = error;
           at_lgm = old_lgm;                    }
     }
     segment.max_con_error = temp_max_con_error;
     segment.lgm = at_lgm;
     printf("%15.15f %15.15f %15.15f %+E\n",segment.lgm_initial,
      segment.antilg_lgm_initial,
                    segment.difference, segment.max_con_error);
}

double find_ideal_difference ( double lgm );
void optimum_difference_error()
{

     float error, temp_new_con_error = 0.0;
     float left_new_con_error, right_new_con_error;
     double temp_new_seg_lgm;
     double old_lgm, old_m, new_lgm;
     double low_end_difference, mid_difference, high_end_difference;
     double left_ideal_difference, right_ideal_difference;
     double left_new_seg_lgm, right_new_seg_lgm;
     unsigned char left_search, right_search;

     left_search = right_search = ALLOWED;
     old_lgm = segment.lgm_initial;
     old_m = pow(2.0, old_lgm) - 1.0;
     new_lgm = old_lgm + CONSTANT;

     ideal_difference = find_ideal_difference ( segment.lgm );

     while (old_lgm < new_lgm)
     {
       old_lgm += FACTOR;
       error = get_error ( old_lgm, old_m, ideal_difference);
       if ( fabs(error) > fabs(temp_new_con_error)) {
      temp_new_con_error = error;
      temp_new_seg_lgm = old_lgm;                }
     } /* while (old_lgm < new_lgm) */
     if ( fabs(temp_new_con_error) < fabs(segment.max_con_error)){
       puts("ERROR");
       exit(0);}
     if ( fabs(temp_new_con_error) > fabs(segment.max_con_error)
      && temp_new_con_error > 0.0 )  left_search = NOT_ALLOWED;
       else        {
      puts("Algorithm don't consider such situation for swing.");
```

```
 exit(0);  }
low_end_difference = ideal_difference;
mid_difference = ideal_difference;
high_end_difference = segment.difference;
left_new_con_error = segment.max_con_error;
right_new_con_error = segment.max_con_error;

while (left_search == ALLOWED || right_search == ALLOWED)
{
 if ( right_search )
 {
 temp_new_con_error = 0.0;
 ideal_difference = (mid_difference + high_end_difference)/2.0;
 right_ideal_difference = ideal_difference;
 old_lgm = segment.lgm_initial;
 old_m = pow(2.0, old_lgm) - 1.0;
 new_lgm = old_lgm + CONSTANT;

 while ( old_lgm < new_lgm )
 {
   old_lgm += FACTOR;
   error = get_error ( old_lgm, old_m, ideal_difference );
   if ( fabs(error) > fabs(temp_new_con_error)) {
     temp_new_con_error = error;
     temp_new_seg_lgm = old_lgm;                }
 } /* while */
 right_new_con_error = temp_new_con_error;
 right_new_seg_lgm = temp_new_seg_lgm;
 } /* if ( right_search ) */
 if ( left_search )
 {
   temp_new_con_error = 0.0;
   ideal_difference = (low_end_difference + mid_difference)/2.0;
   left_ideal_difference = ideal_difference;
   old_lgm = segment.lgm_initial;
   old_m = pow(2.0, old_lgm) - 1.0;
   new_lgm = old_lgm + CONSTANT;

 while ( old_lgm < new_lgm )
 {
   old_lgm += FACTOR;
   error = get_error ( old_lgm, old_m, ideal_difference );
   if ( fabs(error) > fabs(temp_new_con_error)) {
     temp_new_con_error = error;
     temp_new_seg_lgm = old_lgm;                }
 } /* while */
 left_new_con_error = temp_new_con_error;
 left_new_seg_lgm = temp_new_seg_lgm;
 } /* if left_search() */

 if (fabs(right_new_con_error) >= segment.max_con_error
                 &&  right_new_con_error > 0.0 ) {
   left_search = NOT_ALLOWED;
   right_search  = ALLOWED;
   mid_difference = right_ideal_difference;  }
```

```
        else if (fabs(left_new_con_error) < fabs(right_new_con_error)
          && fabs(left_new_con_error) < fabs(segment.max_con_error)){
         segment.difference = left_ideal_difference;
         segment.max_con_error = left_new_con_error;
         segment.lgm = left_new_seg_lgm;
         high_end_difference = mid_difference;
         mid_difference = left_ideal_difference;
         left_search = right_search = ALLOWED;  }

        else if (fabs(right_new_con_error) < fabs(left_new_con_error)
          && fabs(right_new_con_error) < fabs(segment.max_con_error)){
         segment.difference = right_ideal_difference;
         segment.max_con_error = right_new_con_error;
         segment.lgm = right_new_seg_lgm;
         low_end_difference = mid_difference;
         mid_difference = right_ideal_difference;
         left_search = right_search = ALLOWED;                    }
        else if(fabs(right_new_con_error) > fabs(segment.max_con_error)
          && fabs(left_new_con_error) > fabs(segment.max_con_error)) {
         right_search = NOT_ALLOWED;
         left_search = NOT_ALLOWED;                }
        else        {
         puts("we don't consider such situation in flow-control");
         exit(0); }

      } /* while ( right_search || left_search == ALLOWED ) */
      printf("%15.15f %15.15f %15.15f %+E\n\n",segment.lgm_initial,
           segment.antilg_lgm_initial,segment.difference,

      putch('\a');
} /* optimum_difference_error */

double find_ideal_difference ( double lgm )
/* calculates a new value of difference that turn the
   segment.max_con_error into zero  */
{
      int divisor = 0, add_factor;
      double anti_lgm;
      char *cptr;

      add_factor = (long)pow(2.0,(double)(ML - RAL))/2.0;
      find_binary (bin_lgm, lgm, ML);
      cptr = &bin_lgm[RAL];
      do {
        if ( *cptr == '1' ) divisor  += add_factor;
        *cptr++;
        add_factor = add_factor/2.0;
      } while ( *cptr != '\x0');
      anti_lgm = pow( 2.0, (lgm - FACTOR*divisor ));
      ideal_difference = ( pow( 2.0, lgm ) - anti_lgm)/divisor;
      return ( ideal_difference );
}
double get_error (double lgm, double anti_lgm, double differ)
/* accepts a real value 0.0 < m < 1.0 and calculates its base
   2 logarithm value, correction factor is added with the base
   logarithm value and the error is calculated     */
```

```
{
        static double temp_m;
        double multiplier, deviation, add_factor = 0.0;
        char *cptr;

        difference = differ;
        temp_m = pow(2.0, lgm) - 1.0;
        find_binary ( bin_lgm, lgm, ML );
        cptr = &bin_lgm[RAL];
        multiplier = MULTIPLIER;
        do {
           if ( *cptr == '1')
             add_factor += difference * multiplier;
           *cptr++;
           multiplier = multiplier/2.0;
         } while ( *cptr != '\x0');
      if ( add_factor == 0.0 || lgm == 1.0)
       add_factor = difference * DIVIDEND;
      deviation = temp_m - (anti_lgm + add_factor);
      return ( deviation );
}

void find_binary (char *ptr, double value, int n)
/* accepts a real value < 1.0 and generates its n-bit
   binary representation   */
{
      int in_count = 0;
      while (in_count < n)
      {
        value += value;
        if (value >= 1.0) {
        *ptr = '1';
        value -= 1.0;  }
      else
        *ptr = '0';
       ptr++;
        in_count++;
      }
      *ptr = '\x0';
}
void rom_only()
/* simulation of LNS to FLP conversion using  ROM table */
/* ROM capacity to be determined by RAL */
/* neglect ML and DIFFERENCE_FACTOR  */

{
    int c = 0;
    double lgm = 0.0, next_lgm, mid_value;
    double error, max_con_error = 0.0, at_lgm;
    double antilg_lgm, antilg_next_lgm, antilg_mid_value;

    increment = 1.0/pow(2.0, (double)RAL);
    clrscr();
    do
    {
       next_lgm = lgm + increment;
```

```
        mid_value = (lgm + next_lgm)/2.0;
        antilg_next_lgm = pow( 2.0, next_lgm) - 1.0;
        antilg_mid_value = pow( 2.0, mid_value ) - 1.0;
        error = antilg_next_lgm - antilg_mid_value;
        if ( fabs(error) > fabs(max_con_error) ){
         max_con_error = error;
         at_lgm = mid_value;
         gotoxy(8,5);
         printf("Max. error =%E occurred at %18.18f",
                                    max_con_error,at_lgm);
        }
        c++;
        if (c == 1000) {
         gotoxy(8,7);
         printf("scanning %18.18f and above with error =%E",
                                                mid_value,error);
         c = 0;        }
        lgm = lgm + increment;
    }   while ( lgm < 1.0 );
}
```