IMPLEMENTATION OF FLOATING POINT AND LOGARITHMIC NUMBER SYSTEM ARITHMETIC UNITS AND THEIR COMPARISON FOR FPGA

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

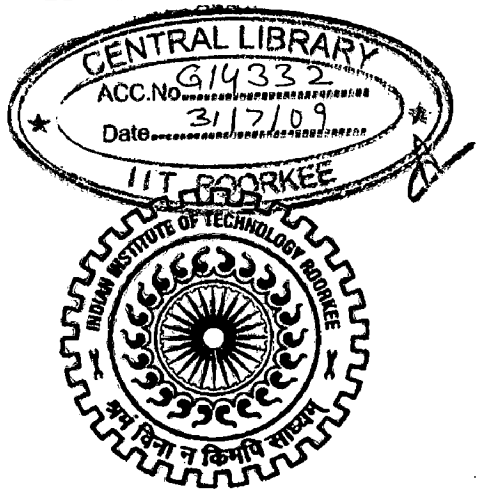
in

ELECTRONICS AND COMPUTER ENGINEERING

(With Specialization in Semiconductor Devices & VLSI Technology)

By

AMIT KUMAR



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE -247 667 (INDIA)

JUNE, 2008

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being in this dissertation report, entitled **"Implementation of Floating Point and Logarithmic Number System Arithmetic Units and their Comparison for FPGA"**, is being submitted in partial fulfillment of the requirements for the award of the degree of **Master of Technology** in **Semiconductor Devices and VLSI Technology**, in the Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee is an authentic record of my own work, carried out from June 2007 to June 2008, under guidance and supervision of **Dr. A.K.Saxena**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology, Roorkee.

The results embodied in this dissertation have not submitted for the award of any other Degree or Diploma.

Date : June 20, 2008

Place : Roorkee

Amit

Amit Kumar

CERTIFICATE

This is to certify that the statement made by the candidate is correct to best of my knowledge and belief.

Dr. A.K. Saxena

Professor

ACKNOWLEDGEMENT

At the outset, I express my heartfelt gratitude to Dr. A.K. Saxena, Professor, Department of Electronics and Computer Engineering at Indian Institute of Technology Roorkee, for his valuable guidance, support, encouragement and immense help. I consider myself extremely fortunate for getting the opportunity to learn and work under his able supervision. I have deep sense of admiration for his innate goodness and inexhaustible enthusiasm. It helped me to work in right direction to attain desired objectives. Working under his guidance will always remain a cherished experience in my memory and I will adore it throughout my life.

My sincere thanks are also due to rest of the faculty in the Department of Electronics and Computer Engineering at Indian Institute of Technology Roorkee, for the technical knowhow and analytical abilities they have imbibed in us which have helped me in dealing with the problems I encountered during the dissertation.

I am greatly indebted to all my friends, who have graciously applied themselves to the task of helping me with ample morale support and valuable suggestions. Finally, I would like to extend my gratitude to all those persons who directly or indirectly helped me in the process and contributed towards this work.

Amit Kumar

M. Tech. (SDVT)

ABSTRACT

Floating point (FP) representation is commonly used to represent real numbers. Some papers have suggested the use of logarithmic number system (LNS) in addition to floating point. In LNS, a real number is represented as a fixed point logarithm. Therefore, multiplication and division in LNS are much simpler in comparison to that in FP, so the LNS can be beneficial if addition and subtraction can be performed with speed and accuracy equivalent to FP. LNS addition and subtraction requires interpolation technique for which some vales are stored in read only memory (ROM). In this dissertation, different sizes of ROMs are used for addition and subtraction, and their performances are compared to themselves and also to the floating point. To obtain good accuracy in LNS addition and subtraction, more values should be stored in the ROM. As a result, FPGA utilization increases. FP addition and subtraction are simple and does not require ROM. The problem is more aggravate in subtraction because the value of $\log_2 x$ varies from -1 to -∞ as x varies from 0.5 to 0. So, more values are stored in ROM for x variation between 0.5 and 0. These values of x do not occur during addition. This is the reason that in LNS subtraction while increasing the ROM size, the values are added for the variation of x from 0.5 to 0 and keeping rest of the ROM same. One more problem with LNS addition and subtraction is that same ROM can not be used for both the operations. In this dissertation, LNS addition and subtraction are also performed using the fixed size ROM (46 values) without using interpolation and the advantage of this method is that same ROM can be used for both LNS addition and subtraction.

CONTENTS

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
FP	Floating Point
LNS	Logarithmic Number System
VHDL	Very high speed integrated circuits Hardware Description Language
FPGA	Field Programmable Gate Array
ROM	Read Only Memory
NaN	Not a Number
LUT	Look Up Table
MSB	Most Significant Bit

Chapter 1
Introduction

1.1 Background

The floating point (FP) [1] and logarithmic number system (LNS) [2,3] are arithmetic number systems used for representing large real numbers in computer and digital hardware. In recent years, there has been a lot of work with the logarithmic number system as a possible alternative to floating point. There has even been a proposal for a logarithmic microprocessor. Most of this work though has been algorithms for LNS addition/subtraction because these are complicated operations in LNS. Most of the implementations use single (32-bit) or double (64-bit) precision for representing floating point and LNS. Using an FPGA gives us the liberty to use a precision and range that best suits an application, which may lead to better performance.

The dynamic ranges of FP and LNS come at the cost of lower precision and increased complexity over fixed point. LNS provide a similar range to FP but may have advantages that multiplication and division in LNS are simplified to fixed-point addition and subtraction. But the disadvantage of LNS is that addition and subtraction are very difficult to perform in hardware description language (HDL) [4] and the accuracy depends upon the size of read only memory (ROM). In this dissertation, arithmetic operations (addition, subtraction, multiplication and division) are implemented in HDL and synthesis results for both are compared to find which number system will suit better to field programmable gate array (FPGA) [5] for real numbers representation.

1.2 Dissertation Contributions

The key contributions of this dissertation are:

1. The algorithms for FP and LNS arithmetic units having operations addition, subtraction, multiplication and division are given.

2. The LNS addition and subtraction operations are given in detail for different sizes of the ROMs and how the accuracy is increased as the size of ROM is increased.

3. LNS addition and subtraction algorithms are also given for which size of the ROM is fixed. Finally, the synthesis results of different operations for FP and LNS are compared.

1.3 Dissertation Organization

The dissertation organization is as follows:

Chapter 2 deals FP and LNS 32 bits representations and their ranges.

Chapter 3 deals with FP arithmetic unit. The algorithms for the FP addition, subtraction, multiplication and division are presented.

Chapter 4 deals with LNS arithmetic unit. The algorithms for the LNS addition, subtraction, multiplication and division are presented.

In chapter 5, the VHDL codes for both FP and LNS arithmetic units are given.

In chapter 6, the simulation results of both FP and LNS arithmetic units are given. The synthesis results for FP and LNS are also presented in this chapter.

Chapter 7 concludes the dissertation.

In the end, reference details are given.

Chapter 2

Number Systems

2.1 Floating Point [1,6]

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard to define floating-point representation and arithmetic. The standard brought out by the IEEE come to be known as IEEE 754 [1]. The single precision format uses a sign bit, 8-bit biased exponent bits, and 23-bit mantissa bits as shown in figure 2.1. The mantissa part has a binary point to the left, and a hidden '1' to the left of the point. The storage layout for single-precision is shown below:

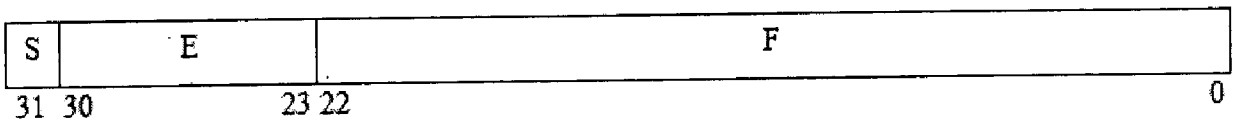


Fig. 2.1. 32 bit floating point format.

The most significant bit starts from the left. The format of numbers represented by the single-precision representation is:

$$\text{Value} = (-1)^S \times 2^{E-127} \times (1.F)$$

where $F = (b_{22}^{-1} + b_{21}^{-2} + \ldots + b_i^n + \ldots + b_0^{-23})$.

b_i^n =1 or 0

S = sign bit (0 is positive, 1 is negative)

E = biased exponent

e = unbiased exponent = E – 127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit, for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27 (100 – 127). The extreme exponents (0 and 255) are used to represent special cases, thus this format has range from - $1.0 * 2^{-126}$ to $1.11111.... * 2^{127}$ i.e. from 1.2E-38 to 3.4E+38.

e_{max} and e_{min} are not used for representing ordinary numbers, they are used for representing special values. The actual minimum and maximum values are $e_{min} = -126$ and $e_{max} = 127$. So that the range of represented numbers is - 2×2^{127} to 2×2^{127} i.e. from - 2^{128} to 2^{128}. The minimum positive represented number is 1×2^{-126}. The table 2.1 gives the sign, exponent and fraction bits for some specific numbers:

Table 2.1. Special Values Supported by IEEE 754 Floating Point Standards [1].

Sign	Exponent	Fraction	Value
0	00000000	00000000000000000000000	+0 (positive zero)
1	00000000	00000000000000000000000	-0 (negative zero)
0	11111111	00000000000000000000000	+ infinity
1	11111111	00000000000000000000000	- infinity
0	11111111	Any combination other than all zero	Not a Number(NaN)
1	11111111	Any combination other than all zero	Not a Number(NaN)

2.1.1 Floating Point Exceptions

The IEEE standard defines the following types of exceptions that should be signaled through a one bit status flag when encountered.

a) Invalid Operation

The invalid operations are:

1) Any operation on a NaN.

2) Addition or subtraction: $\infty + (-\infty)$.

3) Multiplication: $\pm 0 \times \pm \infty$.

4) Division: $\pm \infty / \pm \infty$ or $\pm 0 / \pm 0$.

5) Square root: if the operand is less than zero.

b) Division by Zero

If the divisor is zero and the dividend is finite non zero number, then the division by zero exception should be signaled.

b) Inexact

This exception should be signaled whenever the result of an arithmetic operation is not exact due to the restricted exponent and/or mantissa range.

d) Underflow

The underflow exception is signaled whenever the result is lower than the minimum value that can be represented due to restricted exponent range.

e) Overflow

The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

f) Infinity

This exception is signaled whenever the result is infinity without regard to how that occurred. This exception was not defined in the initial standard and added later to detect faster infinity results.

g) Zero

This exception is signaled whenever the result is zero without regard to how that occurred. This exception was not defined in the standard and added later to detect faster zero results.

2.1.2 Rounding Methods

The IEEE 754 standard specifies four rounding methods:

a) Round to Nearest Even

This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even number. For example: 5.4 will be rounded to 5, 5.6 to 6 and 5.5 will be rounded to 6.

b) Round-to-Zero

Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.4.

c) Round-Up

The number will be rounded up towards $+\infty$, e.g. 5.4 will be rounded to 6, while -5.4 to -5.

d) Round-Down

The opposite of round-up, the number will be rounded up towards $-\infty$, e.g. 5.4 will be rounded to 5, while -5.4 to -6.

2.2 Logarithmic Number System [2,3]

The format of the logarithm number system is

$$A = (-1)^{S_A} * 2^{E_A}$$

where S_A is the sign bit and E_A is a fixed point number. The sign bit signifies the sign of the whole number. E_A is a signed fixed point number. E_A consists of two parts integer(I) and fraction part(F). The integer part is of 8 bits and the fraction part is of 23 bits as shown in figure 2.2. To represent the very small numbers E_A is negative.

Fig. 2.2. 32 bit logarithmic number system format.

The real numbers represented by this format are in the range $\pm 2^{-128}$ to 2^{128} i.e. ±2.9E-39 to 3.4E+38. In order to represent zero, infinity, not a number etc. bits are added to the above format whose combinations will represent the special values.

Chapter 3

Floating Point Arithmetic Unit

The Floating point arithmetic unit has four parts - addition, subtraction, multiplication and division. The operations are performed on operands A and B and the result of the operation will be saved in Z, where A, B and Z are given as

$$A = (-1)^{S_A} * 2^{E_A-127} * (1.M_A)$$

$$B = (-1)^{S_B} * 2^{E_B-127} * (1.M_B)$$

$$Z = (-1)^{S_Z} * 2^{E_Z-127} * (1.M_Z)$$

3.1 FP Addition Algorithm [7,8,9]

Floating point addition involves the following steps:

Result = $Z = A + B = ((-1)^{S_A} * 2^{E_A-127} * 1.M_A) + ((-1)^{S_B} * 2^{E_B-127} * 1.M_B)$

1. Separate the sign, exponent and mantissa bits A and B.

   $A = S_A \ \& \ E_A \ \& \ M_A$

   $B = S_B \ \& \ E_B \ \& \ M_B$

2. Compare |A| and |B|. If |B| is greater than |A|, then swap A and B.
3. Set the exponent of result E_Z equals to E_A and sign of result S_Z equals to S_A.
4. Compute the difference, $d = E_A - E_B$. Shift $(1.M_B)$ to the right by d times and fill the leftmost bits with zeros.
5. Compute the mantissa of result M_Z by adding $(1.M_A)$ and $(1.M_B)$.
6. Normalize the result obtained in step 5.
7. Round off the above result.
8. Check resultant exponent for overflow/underflow:
   - If E_Z is larger than maximum allowed exponent, then set the overflow flag.
   - If E_Z is smaller than minimum allowed exponent, then set the underflow flag.
9. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard, to give the multiplication output.

   $Z = S_Z \ \& \ E_Z \ \& \ M_Z$

Explanation:

In the above algorithm, steps 1 to 5 are self explanatory. In step 6, normalization is done. Normalization means that decimal point should be before the most significant bit (MSB) and the MSB should be one. So after step 5, if carry is generated then right shift M_Z and increase the exponent by one. In step 7, the result is rounded to the nearest value by checking the bits which are eliminated from the result to fit the result in IEEE 754 format. In step 8, overflow/underflow conditions are checked. If the exponent is greater than 254, then overflow flag is set and if exponent is less than 1, then underflow flag is set. The final step is to pack sign, exponent and mantissa bits according to IEEE 754 format.

3.2 FP Subtraction Algorithm [7,8,9]

Floating point subtraction involves the following steps:

Result = $Z = A - B = ((-1)^{S_A} * 2^{E_A-127} * 1.M_A) - ((-1)^{S_B} * 2^{E_B-127} * 1.M_B)$

1. Separate the sign, exponent and mantissa bits of the both operands

   $A = S_A \ \& \ E_A \ \& \ M_A$

   $B = S_B \ \& \ E_B \ \& \ M_B$

2. Compare |A| and |B|. If |B| is greater than |A|, then swap A and B.
3. Set the exponent of result E_Z equals to E_A and sign of result S_Z equals to S_A.
4. Compute the difference, $d = E_A - E_B$. Shift $(1.M_B)$ to the right by d times and fill the leftmost bits with zeros.
5. Compute the mantissa of result M_Z by subtracting $(1.M_B)$ from $(1.M_A)$.
6. Normalize the result obtained in the above step.
7. Round off the above result.
8. Check resultant exponent for overflow/underflow:
   - If E_Z is larger than maximum allowed exponent, then set the overflow flag.
   - If E_Z is smaller than minimum allowed exponent, then set the underflow flag.
9. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard, to give the multiplication output.

   $Z = S_Z \ \& \ E_Z \ \& \ M_Z$

Explanation:

Steps 1 to 5 are self explanatory. In step 6, normalization is done. Normalization means that decimal point should be before most significant bit (MSB) and the MSB should be one. So after step 5, if carry is generated then right shift M_Z and increase the exponent by one. In step 7, the result is rounded to the nearest value by checking the bits which are eliminated from the result to fit the result in IEEE 754 format. In step 8, overflow/underflow conditions are checked. If the exponent is greater than 254, then overflow flag is set and if exponent is less than 1, then underflow flag is set. The final step is to pack sign, exponent and mantissa bits according to IEEE 754 format.

3.3 FP Multiplication Algorithm [7,8,10]

Floating point multiplication involves the following steps:

Result = $Z = A * B = ((-1)^{S_A} * 2^{E_A-127} * 1.M_A) * ((-1)^{S_B} * 2^{E_B-127} * 1.M_B)$

1. Separate the sign, exponent and mantissa bits of the both operands

   $A = S_A \ \& \ E_A \ \& \ M_A$

   $B = S_B \ \& \ E_B \ \& \ M_B$

2. Compute the sign of the result: $S_Z = S_A$ XOR S_B
3. Compute the exponent of the result:
   - Result exponent = $E_A + E_B$ - "01111111".
4. Compute the mantissa of the result [10]:
   - Multiply the mantissas: $(1.M_A) * (1.M_B)$. The calculated mantissa will be in the 48 bits [11].
5. Normalize the result if needed.
6. Round the above result to the allowed number (24 bits) of mantissa bits.
7. Check resultant exponent for overflow/underflow:
   - If E_Z is larger than maximum allowed exponent, then set the overflow flag.
   - If E_Z is smaller than minimum allowed exponent, then set the underflow flag.
8. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard, to give the multiplication output.

   $Z = S_Z \ \& \ E_Z \ \& \ M_Z$

Explanation:

The sign bit, exponent bits and mantissa bits for both the operands are separated and stored in different variables. The sign bit of the result is calculated taking the XOR of the sign bit of A and B. If both the operands represent the valid floating point number, then the result exponent is calculated as shown in the step 3.

Multiplication of the mantissas i.e. $(1.M_A) * (1.M_B)$ in step 4 is done using the shift and add algorithm [11]. Let $1.M_A$ and $1.M_B$ be stored in registers X and Z as shown in figure 3.1.

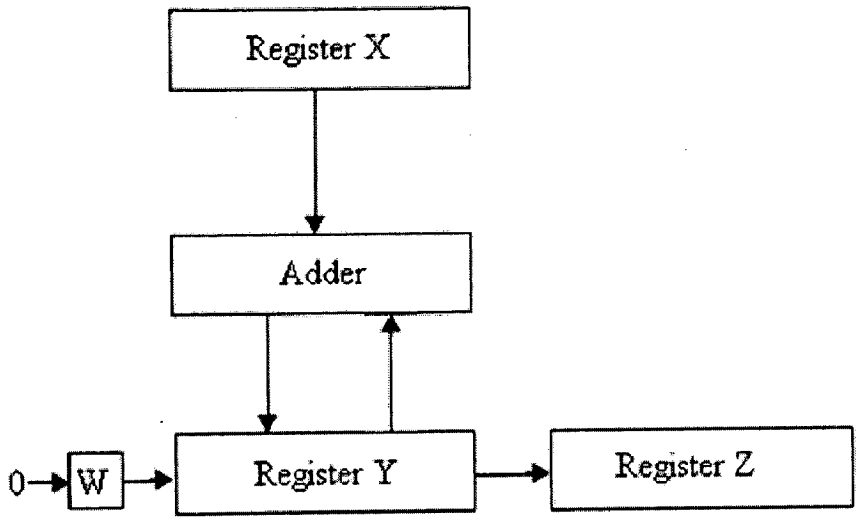


Fig. 3.1. Hardware for multiplication of mantissas [11].

The sum of X and Y forms a partial product which is transferred to the WY register. Both partial product and Z can be shifted to the right by instruction shr WYZ in the algorithm below. The detailed algorithm is given below.

1. Save multiplicand and multiplier in register X and Z.

   $X = 1.M_A$

   $Z = 1.M_B$

2. Set W = 0 and Y = 0.
3. Set sequence counter, SC=24.

4. If Z(0) = 1, then
   WY = WY + X.
5. Shr WYZ.
6. Decrease SC by one.
7. If SC is not equal to 0, then go to step 4, otherwise end and result will be in YZ.

The result of the above multiplication will be in 48 bits stored in the registers Y and Z. The result is now normalized by adjusting the exponent. As mantissa part of A and B is of 24 bits (including the implicit 1), so the result of the multiplication of the mantissas of A and B will be of 48 bits. But floating point 32 bit format requires 23 mantissa bits (excluding the implicit '1'), so the extra bits of the YZ after normalization have to be rounded. The various methods of rounding are given below:

1. The truncation method is accomplished by dropping the extra digits
2. Round towards plus infinity or minus infinity.
3. The round to nearest.

Third method is used for rounding. After rounding, the overflow and underflow conditions are checked. For overflow, the resultant exponent must be greater than 254 and for the underflow, the resultant exponent must be less than 1. The overflow and underflow conditions are shown by making the output signals overflow and overflow equal to '1'.

The final step in the multiplication is to pack sign bit, exponent bits and mantissa bits according to IEEE 754 format.

All the above steps are implemented in VHDL and the simulation results are seen using ModelSim.

3.4 FP Division Algorithm [7,8,10]

Floating point division involves the following steps:

Result = Z = A / B = $((-1)^{S_A} * 2^{E_A-127} * 1.M_A) / ((-1)^{S_B} * 2^{E_B-127} * 1.M_B)$

1. Separate the sign, exponent and mantissa bits of the both operands.

   $A = S_A \ \& \ E_A \ \& \ M_A$

$B = S_B \ \& \ E_B \ \& \ M_B$

2. Compute the sign of the result: $S_Z = S_A$ XOR S_B.
3. Compute the exponent of the result:
   - Result exponent = E_A - E_B + "01111111".
4. Compute the mantissa of the result [10]:
   - Divide the mantissas: $(1.M_A) / (1.M_B)$.
5. Normalize the result if needed.
6. Round the above result to the allowed number (24 bits) of mantissa bits.
7. Check result exponent for overflow/underflow:
   - If E_Z is larger than maximum allowed exponent, then set the overflow flag.
   - If E_Z is smaller than minimum allowed exponent, then set the underflow flag.
8. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard, to give the multiplication output. i.e.

$Z = S_Z \ \& \ E_Z \ \& \ M_Z$

Explanation:

The sign bit, exponent bits and mantissa bits for both the operands are separated and stored in different variables. The sign bit of the result is calculated taking the XOR of the sign bit of A and B. If both the operands represent the valid floating point number, then the result exponent is calculated as shown in the step 3.

To divide the mantissa i.e. $(1.M_A) / (1.M_B)$, the following algorithm is followed

1. Set remainder, rem = $1.M_A$.
2. Difference, D = rem - $(1.M_B)$
3. If(D<0) then quotient = 0, rem = 2 * rem
   Else quotient = 1, rem = D, rem = 2* rem.

Note that rem = 2 * rem will shift the remainder bits left with least bit filled with zero.

Steps 2 to 3 are repeated to get one more bit of the quotient. Generally, the division does not provide a fixed length result, as in the case of the multiplication. The number of algorithmic cycles will therefore depend on the desired accuracy.

The normalization, rounding, checking for overflow/underflow and final packing of the sign, exponent and mantissa bits are done in the same way as in the case of the FP multiplication.

All the above steps for floating point division are implemented in VHDL and the simulation results are seen using ModelSim.

Chapter 4

Logarithmic Number System Arithmetic Unit

The LNS arithmetic unit has four parts - addition, subtraction, multiplication and division. The operations are performed on operands A and B and the result of the operation will be saved in Z, where A, B and Z are given as

$$A = (-1)^{S_A} * 2^{E_A}$$

$$B = (-1)^{S_B} * 2^{E_B}$$

$$Z = (-1)^{S_Z} * 2^{E_Z}$$

4.1 LNS Addition

LNS addition algorithms are divided in to two categories. One for which sizes of ROMs are not fixed and accuracy depends on the size of the ROM and the other for which size of the ROM is fixed. For the first type, algorithm is given in section 4.1.1 and for the second type, algorithm is given section 4.1.2.

4.1.1 LNS Addition Algorithm 1 [12,13,14,15]

The algorithm has following steps:

1. Separate the sign and fixed point exponent of both operand

   $A = S_A$ & E_A

   $B = S_B$ & E_B

2. Generate the ROM values for the function, $f(d)= \log_2(1 + 2^{-d})$ of suitable size using C++ and store them in a constant two dimensional array.
3. If $|A| < |B|$, then swap the numbers A and B.
4. Sign of result: $S_Z = S_A$.
5. Calculate difference, $x = E_A - E_B$.
6. Use the second order polynomial interpolation method to obtain the value of $\log_2(1 + 2^{-x})$ [16,17].
7. Add E_A and $\log_2(1 + 2^{-x})$ to get the result exponent E_Z.
8. Check for overflow/underflow.

9. Assemble the result in to 32 bit.

$Z = S_Z$ & E_Z

Explanation:

$Z = A + B.$

$(-1)^{S_Z} * 2^{E_Z} = (-1)^{S_A} * 2^{E_A} + (-1)^{S_B} * 2^{E_B}$

If $|A| > |B|$, then swap the numbers A and B. So the sign of Z will be equal to the sign of A. The value of E_Z can be calculated as [18,19,20]:

$$E_Z = \log_2|(A + B)| = \log_2\left|A\left(1 + \frac{B}{A}\right)\right|$$

$$E_Z = \log_2|A| + \log_2\left|\left(1 + \frac{B}{A}\right)\right| = E_A + f(E_A - E_B) \quad \ldots\ldots(1)$$

where $f(E_A - E_B) = \log_2\left|\left(1 + \frac{B}{A}\right)\right| = \log_2\left|1 + 2^{-(E_A - E_B)}\right|$

Let $x = (E_A - E_B)$, therefore $f(x) = \log_2(1 + 2^{-x})$. The values of f(x) are calculated for different x and stored in a ROM. The value of f(x) varies from 0 to 1 as x varies from ∞ to 0 [21]. If values are stored in the ROM for all possible values of x, then size of the ROM will be 2^{30} . But it is impossible to store so many values. So interpolation is used to calculate the value of f(x) i.e. instead of storing all values only subset of values will be stored in the ROM and interpolation will be used to value of f(x) for required x. The above algorithm uses the polynomial interpolation. The polynomial interpolation has following steps:

1. Calculate i = integer(x/h), where h is the difference between two consecutive indices of ROM.
2. $u = x/h - i.$
3. $a_0 = f(i).$
4. $a_1 = [\, f(i + 1) - f(i - 1)] /2.$
5. $a_2 = [\, f(i + 1) - 2 * f(i) + f(i - 1)] /2.$
6. $\log_2(1 + 2^{-x}) = a_0 + u * (a_1 + u * a_2).$

In the above steps, multiplication and division by 2 are performed by shifting operand left and right respectively.

After calculating $\log_2(1 + 2^{-x})$, E_A is added to it. Then the result is checked for overflow/underflow. And finally S_Z and E_Z are packed in to 32 bit LNS format.

4.1.2 LNS Addition Algorithm 2 [22]

The algorithm has following steps:

1. Separate the sign and fixed point exponent of both operand

   $A = S_A$ & E_A

   $B = S_B$ & E_B

2. Generate the ROM values for the function $f(d) = \sqrt[i]{2}$ where i = 2, 4, 8, 16, 32.....8388608 i.e. of size 23 using C++ and store that in a constant two dimensional array rom1.

3. Generate the ROM values for the function $f(d) = \sqrt[i]{2^{-1}}$ where i = 2 ,4, 8, 16, ...8388608 i.e. of size 23 using C++ and store that in a constant two dimensional array rom2.

4. If |A| < |B|, then swap the numbers A and B.

5. Set sign of result: $S_Z = S_A$.

6. Calculate difference, $x = E_A - E_B$. x is of 31 bits out of which 8 bits(from most significant bit) will be in integer part and 23 bits(from least significant bit) will be fraction part. i.e. x = I.F

7. Calculate the value of 2^{-x} using lut1 following the steps given below.

   7.1 Add one to the integer part and subtract the fraction part from one.

   7.2 Initialize the variable of type std_logic_vector with name rega and set it equal to 1.

   7.3 For k starting from 0 to 22 repeat the steps from 7.4 to 7.5

   7.4 If F(k) ='1' then rega = rega * rom1(k)..

   7.5 Increment k.

   7.6 Right shift the rega by I times. Save the final result in a variable m. i.e. $m = 2^{-x} = 2^{-I.F}$

8. Calculate the value of $\log_2(1 + m)$ following the steps given below.

   8.1 Add 1 to m.

   8.2 Initialize the variable of type std_logic_vector with name regb and set it equal to b.

   8.3 For k starting from 0 to 22 repeat the steps from 8.4 to 8.6

   8.4 If rega >= rom1(22-i), then Set regb(22-i) = '1' and rega = rega * rom2(22-i).

   8.5 If rega < rom1(22-i), then Set regb(22-i) = '0'.

   8.6 Increment k.

9. Add E_A and $\log_2(1 + m)$ to get the result exponent E_Z.

10. Check for overflow/underflow.

11. Assemble the result in to 32 bit.

$Z = S_Z$ & E_Z

Explanation:

From equation 1

$$E_Z = \log_2|A| + \log_2\left|\left(1+\frac{B}{A}\right)\right| = E_A + f(E_A - E_B)$$

where $f(E_A - E_B) = \log_2\left|\left(1+\frac{B}{A}\right)\right| = \log_2\left|1 + 2^{-(E_A - E_B)}\right|$

So to calculate $f(E_A - E_B)$, first $2^{-(E_A - E_B)}$ is calculated and then $\log_2\left|1 + 2^{-(E_A - E_B)}\right|$ will be calculated.

Let $x = E_A - E_B$ and $m = 2^{-(E_A - E_B)}$

$f(E_A - E_B) = \log_2\left|1 + 2^{-x}\right| = \log_2(1 + m)$

Let x consists of integer part (I) and fractional part (F). i.e. x = I.F

$2^{-x} = 2^{-I.F} = 2^{-I} * 2^{-0.F} = 2^{-(I+1)} * 2^{(1-0.F)} = 2^{-(I+1)} * 2^{Z}$

where $Z = 1 - 0.F$

Multiplication by $2^{-(I+1)}$ can be performed easily by right shifting the result by (I + 1) times.

Let $Z = a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + a_4 * 2^{-4} + \ldots\ldots + a_{23} * 2^{-23}$

where $a_i = 0$ or 1.

$$2^Z = 2^{(a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + a_4 * 2^{-4} + \ldots\ldots + a_{23} * 2^{-23})}$$

$$= 2^{a_1 * 2^{-1}} \times 2^{a_2 * 2^{-2}} \times 2^{a_3 * 2^{-3}} \times 2^{a_4 * 2^{-4}} \times \ldots\ldots \times 2^{a_{23} * 2^{-23}}$$

$$= \sqrt[2]{2^{a_1}} \times \sqrt[4]{2^{a_2}} \times \sqrt[8]{2^{a_3}} \times \sqrt[16]{2^{a_4}} \times \ldots\ldots \times \sqrt[2^{23}]{2^{a_{23}}}$$

$$= \prod_{i=1}^{23} \sqrt[2^i]{2^{a_i}} \quad \ldots\ldots \quad (2)$$

A ROM is needed to store the values of $\sqrt[2^i]{2}$ for i varying from 1 to 23. A program is written in C++ to create the ROM values. First decimal value of $\sqrt[2^i]{2}$ is obtained and then it is converted in to binary equivalent and these binary equivalent are stored in the ROM rom1. To calculate 2^Z, values of rom1 are multiplied to each other for which $a_i = 1$. After multiplication, result is right shifted by (I + 1) times to calculate 2^{-x} [23].

Calculating $\log_2(1 + m)$ is considered as inverse operation of 2^z. For example to calculate the value $\log_2 x$, the following steps will be followed

If $x \geq \sqrt[2]{2}$, then $x = x/\sqrt[2]{2}$ and $a_1 = 1$ otherwise $a_1 = 0$.

If $x \geq \sqrt[4]{2}$, then $x = x/\sqrt[4]{2}$ and $a_2 = 1$ otherwise $a_2 = 0$.

If $x \geq \sqrt[8]{2}$, then $x = x/\sqrt[8]{2}$ and $a_3 = 1$ otherwise $a_3 = 0$.

.

.

.

Division by $\sqrt[2^i]{2}$ is equivalent to the multiplication by $\sqrt[2^i]{2^{-1}}$. A ROM is needed to store the values of $\sqrt[2^i]{2^{-1}}$ for i varying from 1 to 23. A program is written in C++ to create the ROM values. First decimal value of $\sqrt[2^i]{2^{-1}}$ is obtained and then it is converted in to binary equivalent and these binary equivalent are stored in the ROM rom2. There are also other algorithm [24] for calculating $\log_2(1 + m)$, but for them number of iterations will be very large, hence they are not used.

After calculating $\log_2(1 + m)$, E_A and $\log_2(1 + m)$ are added, the result is checked for overflow/underflow. Finally the result is packed in to 32 bit logarithmic number system format.

4.2 LNS Subtraction

LNS subtraction algorithms are divided in to two categories. One for which sizes of ROMs are not fixed and accuracy depends on the size of the ROM and the other for which size of the ROM is fixed. For the first type, algorithm is given in section 4.2.1 and for the second type, algorithm is given section 4.2.2.

4.2.1 LNS Subtraction Algorithm 1 [12,13,15,25]

The algorithm has following steps:

1. Separate the sign and fixed point exponent of both operand

   $A = S_A \ \& \ E_A$

   $B = S_B \ \& \ E_B$

2. Generate the ROM values for the function $f(d) = \log_2(1 - 2^{-d})$ of suitable size using C++ and store that in a constant two dimensional array.

3. If $|A| < |B|$, then swap the numbers A and B.

4. Sign of result: $S_Z = S_A$.
5. Calculate difference, $x = E_A - E_B$
6. Use the second order polynomial interpolation method to obtain the value of $\log_2(1 - 2^{-x})$ [16,17].
7. Add E_A and $\log_2(1 - 2^{-x})$ to get the result exponent E_Z.
8. Check for overflow/underflow.
9. Assemble the result in to 32 bit.

$$Z = S_Z \,\&\, E_Z$$

Explanation:

$Z = A - B.$

$(-1)^{S_Z} * 2^{E_Z} = (-1)^{S_A} * 2^{E_A} - (-1)^{S_B} * 2^{E_B}.$

If $|A| > |B|$, then swap the numbers A and B. So the sign of Z will be equal to the sign of A. The value of E_Z can be calculated as [18,19]:

$$E_Z = \log_2|(A - B)| = \log_2\left|A\left(1 - \frac{B}{A}\right)\right|$$

$$E_Z = \log_2|A| + \log_2\left|\left(1 - \frac{B}{A}\right)\right| = E_A + f(E_A - E_B) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(3)$$

where $f(E_A - E_B) = \log_2\left|\left(1 - \frac{B}{A}\right)\right| = \log_2\left|1 - 2^{-(E_A - E_B)}\right|$

Let $x = (E_A - E_B)$, therefore $f(x) = \log_2(1 - 2^{-x})$. The values of f(x) are calculated for different x and stored in a ROM. If values are stored in the ROM for all possible values of x, then size of the ROM will be 2^{30}. But it is impossible to store so many values. So interpolation is used to calculate the value of f(x) i.e. instead of storing all values only subset of values will be stored in the ROM and interpolation will be used to value of f(x) for required x. The above uses the polynomial interpolation. The polynomial interpolation has following steps:

1. Calculate i =integer(x /h), where h is the difference between two consecutive indices of ROM.
2. $u = x/h - i.$
3. $a_0 = f(i).$
4. $a_1 = [\, f(i + 1) - f(i - 1)] /2.$
5. $a_2 = [\, f(i + 1) - 2 * f(i) + f(i - 1)] /2.$
6. $\log_2(1 - 2^{-x}) = a_0 + u * (a_1 + u * a_2).$

In the above steps, multiplication and division by 2 are performed by shifting operand left and right respectively

After calculating $\log_2(1 - 2^{-x})$, E_A is added to it. Then the result is checked for overflow/underflow. And finally S_Z and E_Z are packed in to 32 bit LNS format.

The value of f(x) varies from 0 to $-\infty$ as x varies from infinity to 0. The transition in the value of f(x) is very rapid as x varies from 0.5 to 0. For these variations f(x) varies from -1 to $-\infty$ [26]. So ROM will be divided in to 2 parts one for x € $(0.5,\infty)$ and other for x € (0,0.5). The numbers of values in ROM are increased to enhance the accuracy [27]. The number of entries are increased for x € (0,0.5) because in this region variation in f(x) is large.

4.2.2 LNS Subtraction Algorithm 2 [22]

The algorithm has following steps:

1. Separate the sign and fixed point exponent of both operand

   $A = S_A$ & E_A

   $B = S_B$ & E_B

2. Generate the ROM values for the function $f(d) = \sqrt[i]{2}$ where i = 2, 4, 8, 16, 32.....8388608 i.e. of size 23 using C++ and store that in a constant two dimensional array rom1.

3. Generate the ROM values for the function $f(d) = \sqrt[i]{2^{-1}}$ where i = 2,4, 8, 16, ...8388608 i.e. of size 23 using C++ and store that in a constant two dimensional array rom2.

4. If |A| < |B|, then swap the numbers A and B.

5. Sign of result: $S_Z = S_A$.

6. Calculate difference, $x = E_A - E_B$. x is 31 bits out of which 8 bits(from most significant bit) will be in integer part(I) and 23 bits(from least significant bit) will be fraction part(F). i.e x = I.F

7. Calculate the value of 2^{-x} using lut1 following the steps given below.

   7.1 Add one to the integer part and subtract the fraction part from one.

   7.2 Initialize the variable of type std_logic_vector with name rega and set it equal to 1.

   7.3 For k starting from 0 to 22 repeat the steps from 7.4 to 7.5

   7.4 If F(k)='1' then rega = rega * rom1(k)..

   7.5 Increment k.

   7.6 Right shift the rega by I times. Save the final result in a variable m. i.e. $m = 2^{-x} = 2^{-I.F}$

8. Calculate the value of log (1 - m) following the steps given below.

8.1 Subtract m from 1.

8.2 Initialize the variable of type std_logic_vector with name regb and set it equal to b.

8.3 For k starting from 0 to 22 repeat the steps from 8.4 to 8.6

8.4 If rega >= rom1(22-i), then Set regb(22-i):='1' and rega = rega * rom2(22-i).

8.5 If rega < rom(22-i), then Set regb(22-i):='0'.

8.6 Increment k.

9. Add E_A and $\log_2(1 - m)$ to get the result exponent E_Z.

10. Check for overflow/underflow.

11. Assemble the result in to 32 bit.

$Z = S_Z$ & E_Z

Explanation:

From equation 3

$$E_Z = \log_2|A| + \log_2\left|\left(1 - \frac{B}{A}\right)\right| = E_A + f(E_A - E_B)$$

where $f(E_A - E_B) = \log_2\left|\left(1 - \frac{B}{A}\right)\right| = \log_2\left|1 + 2^{-(E_A - E_B)}\right|$

So to calculate $f(E_A - E_B)$, first $2^{-(E_A - E_B)}$ is calculated and then $\log_2\left|1 - 2^{-(E_A - E_B)}\right|$ will be calculated.

Let $x = E_A - E_B$ and $m = 2^{-(E_A - E_B)}$

$f(E_A - E_B) = \log_2\left|1 - 2^{-x}\right| = \log_2(1 - m)$

Let x consists of integer part (I) and fractional part (F). i.e. x = I.F

$2^{-x} = 2^{-I.F} = 2^{-I} * 2^{-0.F} = 2^{-(I+1)} * 2^{(1-0.F)} = 2^{-(I+1)} * 2^{Z}$

where $Z = 1 - 0.F$

Multiplication by $2^{-(I+1)}$ can be performed easily by right shifting the result by (I + 1) times.

Let $Z = a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + a_4 * 2^{-4} + \ldots\ldots\ldots\ldots\ldots\ldots + a_{23} * 2^{-23}$

where $a_i = 0$ or 1.

$2^{Z} = 2^{(a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + a_4 * 2^{-4} + \ldots\ldots\ldots\ldots\ldots + a_{23} * 2^{-23})}$

$= 2^{a_1 * 2^{-1}} \times 2^{a_2 * 2^{-2}} \times 2^{a_3 * 2^{-3}} \times 2^{a_4 * 2^{-4}} \times \ldots\ldots\ldots\ldots\ldots\ldots \times 2^{a_{23} * 2^{-23}}$

$$= \sqrt[2]{2^{a_1}} \times \sqrt[4]{2^{a_2}} \times \sqrt[8]{2^{a_3}} \times \sqrt[16]{2^{a_4}} \times \ldots\ldots\ldots\ldots\ldots\ldots \times \sqrt[2^{23}]{2^{a_{23}}}$$

$$= \prod_{i=1}^{n} \sqrt[2^i]{2^{a_i}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

A ROM is needed to store the values of $\sqrt[2^i]{2}$ for i varying from 1 to 23. A program is written in C++ to create the ROM values. First decimal value of $\sqrt[2^i]{2}$ is obtained and then it is converted in to binary equivalent and these binary equivalent are stored in the ROM rom1. To calculate 2^Z, values of rom1 are multiplied to each other for which $a_i = 1$. After multiplication, result is right shifted by (I + 1) times to calculate 2^{-x} [23].

Calculating $\log_2(1 - m)$ is considered as inverse operation of 2^Z. First $(1 - m)$ is converted in to value in between 1 to 2 by multiplying $(1 - m)$ by 2 and decreasing E_A by one. Iterate this if required.

To calculate the value $\log_2 x$, the following steps will be followed

If $x \geq \sqrt[2]{2}$, then $x = x/\sqrt[2]{2}$ and $a_1 = 1$ otherwise $a_1 = 0$.

If $x \geq \sqrt[4]{2}$, then $x = x/\sqrt[4]{2}$ and $a_2 = 1$ otherwise $a_2 = 0$.

If $x \geq \sqrt[8]{2}$, then $x = x/\sqrt[8]{2}$ and $a_3 = 1$ otherwise $a_3 = 0$.

.

.

.

Division by $\sqrt[2^i]{2}$ is equivalent to the multiplication by $\sqrt[2^i]{2^{-1}}$. A ROM is needed to store the values of $\sqrt[2^i]{2^{-1}}$ for i varying from 1 to 23. A program is written in C++ to create the ROM values. First decimal value of $\sqrt[2^i]{2^{-1}}$ is obtained and then it is converted in to binary equivalent and these binary equivalent are stored in the ROM rom2.

After calculating $\log_2(1 - m)$, E_A and $\log_2(1 - m)$ are added, the result is checked for overflow/underflow. Finally the result is packed in to 32 bit logarithmic number system format.

4.3 LNS Multiplication Algorithm [12,25]

LNS multiplication follows the following steps:

Result = $Z = A * B = ((-1)^{S_A} * 2^{E_A}) * ((-1)^{S_B} * 2^{E_B})$

1. Separate the sign and fixed point exponent bits of both operands.

$A = S_A \; \& \; E_A$

$B = S_B \ \& \ E_B$

2. Compute the sign of the result: $S_Z = S_A$ XOR S_B
3. $E_Z = E_A + E_B$.
4. Check for overflow/underflow.
5. Assemble the result in to 32 bit LNS format.

$Z = S_Z \ \& \ E_Z$

4.4 LNS Division Algorithm [12,25]

LNS division follows the following steps:

Result = $Z = A / B = ((-1)^{S_A} * 2^{E_A})/((-1)^{S_B} * 2^{E_B})$

1. Separate the sign and fixed point exponent of both operand

$A = S_A \ \& \ E_A$

$B = S_B \ \& \ E_B$

2. Compute the sign of the result: $S_Z = S_A$ XOR S_B.
3. $E_Z = E_A - E_B$.
4. Check for overflow/underflow.
5. Assemble the result in to 32 bit format.

$Z = S_Z \ \& \ E_Z$

All the above algorithms of LNS arithmetic unit are implemented in VHDL and the simulation results are seen using ModelSim.

Chapter 5

VHDL Codes

5.1) Floating Point Arithmetic Unit

5.1.1) Addition

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ad is
Port (   a : in std_logic_vector(31 downto 0);
         b : in std_logic_vector(31 downto 0);
         z : out std_logic_vector(31 downto 0);
         nan,infinity,zero,overflow,underflow : out std_logic );
end ad;

architecture Behavioral of ad is
--signal exp1,exp2,exp,menti1,menti2,menti,diff:integer;
begin
process(a,b)
variable mentz,menta, mentb ,temp : std_logic_vector(23 downto 0);
variable addab : std_logic_vector(24 downto 0);
variable e,e1,e2: std_logic_vector(7 downto 0);
variable d,n:integer;
variable sa,sb,ment1: std_logic;
 begin
   underflow<='0';
   overflow<='0';
```

```
nan<='0';
infinity<='0';
zero<='0';
menta:='1' & a(22 downto 0);
mentb:='1' & b(22 downto 0);
e1:= a(30 downto 23);
e2:=  b(30 downto 23);
sa:=a(31);
sb:=b(31);
--exp1<=conv_integer(e1) ; exp2<=conv_integer(e2);  menti1<=conv_integer(menta);
menti2<=conv_integer(mentb);
if(e1="11111111") then
   if(menta="10000000000000000000000") then
        if(e2="11111111" and mentb >"10000000000000000000000")  then
             e:="11111111" ;
             mentz:="10000000000000000000001";
             nan<='1';
        else
             e:="11111111" ;
             mentz:="10000000000000000000000";
             infinity<='1';
        end if;
   else
         e:="11111111" ;
         mentz:="10000000000000000000001";
          nan<='1';
   end if;
elsif(e2="11111111") then
   if(mentb="10000000000000000000000") then
        if(e1="11111111" and menta >"10000000000000000000000") then
             e:="11111111" ;
```

```
                mentz:="10000000000000000000001" ;
                nan<='1';
            else
                e:="11111111" ;
                mentz:="10000000000000000000000" ;
                infinity<='1';
            end if;
        else
            e:="11111111" ;
            mentz:="10000000000000000000001";
            nan<='1';
        end if;
else
    if(menta="10000000000000000000000" and e1="00000000" and
            mentb="10000000000000000000000" and e2="00000000" ) then
            mentz:="00000000000000000000000";
            e:="00000000";
            zero<='1';
  elsif(menta="10000000000000000000000" and e1="00000000")then
            mentz:=mentb;
            e :=e2;
  elsif(mentb="10000000000000000000000" and e2="00000000")then
            mentz:=menta;
            e :=e1;
  else

  if((e2>e1)or ((e1=e2) and (mentb>menta)))then
          temp:=menta;
          menta:=mentb;
          mentb:=temp;
          e2:=  a(30 downto 23);
```

```
        e1:= b(30 downto 23);
        sa:=b(31);
        sb:=a(31);
else null;
end if;  --e1 becomes greater them e2
e:=e1;
d:=conv_integer(e1) - conv_integer(e2);
--diff<=d;
for i in 24 downto 1 loop
        if(i<=d) then
            ment1:=mentb(0);
            mentb:='0' & mentb(23 downto 1);
            --e2:=e2 +1;
        else null;
        end if;
end loop;
if(d=24) then
        ment1:='1';
elsif(d>24)then
        ment1:='0';
end if;
addab:=('0' & menta)+('0' & mentb);
if(addab(24)='1') then
      if(addab(0)='1') then
            addab:=('0' & addab(24 downto 1))+1;
      else
            addab:='0' & addab(24 downto 1);
      end if;
      e1:=e1+1;
elsif(ment1='1') then
      addab:=addab + 1;
```

```
    else null;
    end if;
    e:=e1;
    mentz:=addab(23 downto 0);
    if(e1<1)then
            e:="11111111" ;
            mentz:="10000000000000000000001" ;
            underflow<='1';
     elsif(e1>254)then
            e:="11111111" ;
            mentz:="10000000000000000000001" ;
            overflow<='1';
     else null;
   end if;
 end if;
 z<= sa & e   & mentz(22 downto 0) ;
 end if;
end process;
end Behavioral;
```

5.1.2) Subtraction

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sub is
Port (    a : in std_logic_vector(31 downto 0);
          b : in std_logic_vector(31 downto 0);
          z : out std_logic_vector(31 downto 0);
          nan,infinity,zero,overflow,underflow : out std_logic );
```

```
end sub;

architecture Behavioral of sub is
--signal exp1,exp2,exp,menti1,menti2,menti,diff:integer;
begin
process(a,b)
variable mentz,menta, mentb ,temp : std_logic_vector(23 downto 0);
variable subab : std_logic_vector(24 downto 0);
variable e,e1,e2: std_logic_vector(7 downto 0);
variable d,n:integer;
variable sa,sb,sz,ment1: std_logic;
 begin
   underflow<='0';
   overflow<='0';
   nan<='0';
   infinity<='0';
   zero<='0';

   menta:='1' & a(22 downto 0);
   mentb:='1' & b(22 downto 0);
   e1:= a(30 downto 23);
   e2:=  b(30 downto 23);
   sa:=a(31);
   sb:=b(31);
   sz:='0';
   --exp1<=conv_integer(e1) ; exp2<=conv_integer(e2);  menti1<=conv_integer(menta);
  menti2<=conv_integer(mentb);
  if(e1="11111111") then
     if(menta="10000000000000000000000") then
           if(e2="11111111" and mentb >="10000000000000000000000")  then
                e:="11111111" ;
```

```
                mentz:="10000000000000000000001";
                nan<='1';
            else
                e:="11111111" ;
                mentz:="10000000000000000000000";
                infinity<='1';
            end if;
        else
            e:="11111111" ;
            mentz:="10000000000000000000001";
            nan<='1';
        end if;
    elsif(e2="11111111") then
        if(mentb="10000000000000000000000") then
            if(e1="11111111" and menta >="10000000000000000000000") then
                e:="11111111" ;
                mentz:="10000000000000000000001" ;
                nan<='1';
            else
                e:="11111111" ;
                mentz:="10000000000000000000000" ;
                infinity<='1';
                sz:='1';
            end if;
        else
            e:="11111111" ;
            mentz:="10000000000000000000001";
            nan<='1';
        end if;
    else
        if(a=b ) then
```

```
        mentz:="00000000000000000000000";
        e:="00000000";
        zero<='1';
elsif(menta="10000000000000000000000" and e1="00000000")then
        mentz:=mentb;
        e:=e2;
        sz:=not(sb);
        --z<=(not(sb)) & b(30 downto 0);
 elsif(mentb="10000000000000000000000" and e2="00000000")then
         mentz:=menta;
         e:=e1;
        sz:=sa;
 else
        if((e2>e1)or ((e1=e2) and (mentb>menta)))then
                temp:=menta;
                menta:=mentb;
                mentb:=temp;
                e2:= a(30 downto 23);
                e1:= b(30 downto 23);
               sz:=not(b(31));
               sa:=b(31);
               sb:=a(31);
        else null;
        end if;  --e1 becomes greater them e2
 --e:=e1;
 d:=conv_integer(e1) - conv_integer(e2);
 --diff<=d;
temp :="00000000000000000000000";
ment1:='0';
for i in 24 downto 1 loop
        if(i<=d) then
```

```
            ment1:=mentb(0);
            temp(i-1):= ment1;
            mentb:='0' & mentb(23 downto 1);
            --e2:=e2 +1;
        else null;
        end if;
    end loop;
    if(ment1='0')then
        subab:=('0' & menta)+ not('0' & mentb) + 1;
    else
        subab:=('0' & menta)+ not('0' & mentb);
    end if;
    for i in 23 downto 0 loop
        if(subab(23)='0') then
            subab:= subab(23 downto 0) & temp(i) ;
            e1:=e1-1;
        else null;
        end if;
    end loop;
    e:=e1;
    mentz:=subab(23 downto 0);
    if(e1<1)then
        e:="11111111" ;
        mentz:="10000000000000000000000" ;
        underflow<='1';
    elsif(e1>254)then
        e:="11111111" ;
        mentz:="10000000000000000000000" ;
        overflow<='1';
  else null;
 end if;
```

```
 end if;
 z<= sz & e & mentz(22 downto 0) ;
end if;
end process;
end Behavioral;
```

5.1.3) Multiplication

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity mul is
   Port (  a : in std_logic_vector(31 downto 0);
           b : in std_logic_vector(31 downto 0);
           z : out std_logic_vector(31 downto 0);
           nan,infinity,zero,dividebyzero,overflow,underflow : out std_logic);
 end mul;
architecture Behavioral of mul is
 --signal exp1,exp2 ,exp,menti1,menti2,menti: integer;
--signal ment: std_logic_vector(47 downto 0);
signal num1:integer;
--signal num3:real;
begin
process(a,b)
variable mentz,menta, mentb,temp : std_logic_vector(23 downto 0);
variable mentd: std_logic_vector(24 downto 0);
variable mentia, mentib,r,diff: std_logic_vector(24 downto 0);
--variable quot : std_logic_vector(30 downto 0);
variable carry: std_logic_vector(24 downto 0);
--variable extrament: std_logic_vector(22 downto 0);
--variable pp : std_logic_vector(47 downto 0);
```

```
variable e,e1,e2: std_logic_vector(7 downto 0);
variable d,n,expt:integer;
variable sa,sb,s: std_logic;
 begin
   underflow<='0';
   overflow<='0';
   nan<='0';
   infinity<='0';
   zero<='0';
   dividebyzero<='0';
   menta:='1' & a(22 downto 0);
   mentb:='1' & b(22 downto 0);
   e1:= a(30 downto 23);
   e2:=  b(30 downto 23);
   sa:=a(31);
   sb:=b(31);
   mentia:='0'& menta;
   mentib:='0' & mentb;
   --exp1<=conv_integer(e1) ; exp2<=conv_integer(e2);  menti1<=conv_integer(menta);
   menti2<=conv_integer(mentb);
   -- e:= e1 + e2 - "01111111" ;
   s:= sa xor sb;
   if(e1="11111111") then
      if(menta="10000000000000000000000") then
           if(e2="11111111" and mentb >="10000000000000000000000")  then
                e:="11111111" ;
                mentz:="10000000000000000000001";
                nan<='1';
           else
                e:="11111111" ;
                mentz:="10000000000000000000000";
```

```
            infinity<='1';
        end if;
    else
        e:="11111111" ;
        mentz:="10000000000000000000001";
        nan<='1';
    end if;
elsif(e2="11111111") then
    if(mentb="10000000000000000000000") then
        if(e1="11111111" and menta >="10000000000000000000000") then
            e:="11111111" ;
            mentz:="10000000000000000000001" ;
            nan<='1';
        else
            e:="11111111" ;
            mentz:="10000000000000000000000" ;
            infinity<='1';
        end if;
    else
        e:="11111111" ;
        mentz:="10000000000000000000001";
        nan<='1';
    end if;
else
    if((e1="00000000" and menta="10000000000000000000000") or (e2="00000000"
    and     mentb="10000000000000000000000")) then
        e:="00000000";
        mentz:="00000000000000000000000";
        zero<='1';
    else
        expt:=conv_integer(e1)+ conv_integer(e2);
```

```
if(expt<127)then
        e:="11111111" ;
        mentz:="10000000000000000000000001" ;
        underflow<='1';
else
        e:=e1 +e2 - "01111111" ;
       expt:=expt - 127;
       if(expt>255)then
            e:="11111111" ;
            mentz:="10000000000000000000000001" ;
            overflow<='1';
       else
             mentd:="00000000000000000000000000";
             for i in 0 to 23 loop
                  if(mentb(0)='1') then
                        --re:=rd;
                        mentd:= mentd + ("0" & menta);
                  else null;
                  end if;
                  mentb:=mentd(0) & mentb(23 downto 1);
                  mentd:='0' & mentd(24 downto 1);
             end loop;
            --rd:=rd(22 downto 0) & rb(23);
            if(mentd(23)='0')then
                 --mentz:=pp(46 downto 23);
                 mentz:=mentd(22 downto 0) & mentb(23);
                 if(mentb(22)='1') then
                        mentz:=mentz + 1;
                else null;
                end if;
            else
```

```
                    mentz:=mentd(23 downto 0);
                    e:=e+"00000001";
                    expt:=expt+1;
                   if(mentb(23)='1') then
                        mentz:=mentz + 1;
                  else null;
                  end if;
                  if(expt<1)then
                        e:="11111111" ;
                        mentz:="10000000000000000000001" ;
                        underflow<='1';
                  elsif(expt>254)then
                        e:="11111111" ;
                        mentz:="10000000000000000000001" ;
                        overflow<='1';
                  else null;
                  end if;
              end if;
          end if;
       end if;
    end if;
  end if;
  z<= s & e & mentz(22 downto 0);
end process;
end Behavioral;
```

5.1.4) Division

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity div is
  Port (      a : in std_logic_vector(31 downto 0);
              b : in std_logic_vector(31 downto 0);
              z : out std_logic_vector(31 downto 0);
              nan,infinity,zero,dividebyzero,invalid,overflow,underflow : out std_logic);
end div;

architecture Behavioral of div is
signal exp1,exp2 ,exp,menti1,menti2,menti: integer;
--signal ment: std_logic_vector(47 downto 0);
begin
process(a,b)
variable mentz,menta, mentb,temp : std_logic_vector(23 downto 0);
variable mentia, mentib,nmentib,r,diff: std_logic_vector(24 downto 0);
--variable quot : std_logic_vector(30 downto 0);
variable carry: std_logic_vector(24 downto 0);
variable extrament: std_logic_vector(22 downto 0);
variable pp : std_logic_vector(24 downto 0);
variable e,e1,e2: std_logic_vector(7 downto 0);
variable d,n,expt:integer;
variable sa,sb,s: std_logic;
 begin
   underflow<='0';
   overflow<='0';
   invalid<='0';
   infinity<='0';
   zero<='0';
   dividebyzero<='0';
   menta:='1' & a(22 downto 0);
   mentb:='1' & b(22 downto 0);
```

```
e1:= a(30 downto 23);
e2:=  b(30 downto 23);
sa:=a(31);
sb:=b(31);
s:= sa xor sb;
mentia:='0'& menta;
mentib:='0' & mentb;
nmentib:=not(mentib) + 1;
exp1<=conv_integer(e1) ;
exp2<=conv_integer(e2);
menti1<=conv_integer(menta);
menti2<=conv_integer(mentb);
if(e1="11111111") then
    if(menta="10000000000000000000000") then
        if(e2="11111111" and mentb >="10000000000000000000000") then
            e:="11111111" ;
            mentz:="10000000000000000000001";
            nan<='1';
        elsif(e2="00000000" and mentb ="10000000000000000000000" )then
            e:="11111111" ;
            mentz:="10000000000000000000000";
            dividebyzero<='1';
            infinity<='1';
        else
            e:="11111111" ;
            mentz:="10000000000000000000000";
            infinity<='1';
        end if;
    else
        e:="11111111" ;
        mentz:="10000000000000000000001";
```

```
            invalid<='1';
        end if;
    elsif(e2="11111111") then
        if(mentb="10000000000000000000000") then
            if(e1="11111111" and menta >="10000000000000000000000")  then
                e:="11111111" ;
                mentz:="10000000000000000000001" ;
                nan<='1';

        else
                e:="00000000" ;
                mentz:="10000000000000000000000" ;
                -- infinity<='1';
            end if;
        else
            e:="11111111" ;
            mentz:="10000000000000000000001";
            invalid<='1';
        end if;
    else
        if(e1="00000000" and menta="10000000000000000000000") then
            if(e2="00000000" and mentb="10000000000000000000000")then
                e:="11111111" ;
                mentz:="10000000000000000000001";
                invalid<='1';
            else
                e:="00000000";
                mentz:="00000000000000000000000";
                zero<='1';
            end if;
        elsif(e2="00000000" and mentb="10000000000000000000000")then
```

```
        e:="11111111" ;
        mentz:="10000000000000000000000";
        dividebyzero<='1';
        infinity<='1';
else
      expt:=conv_integer(e1)- conv_integer(e2);
      expt:=expt + 127;
      e:=e1-e2 + "01111111" ;
r:=mentia;
for i in 24 downto 0 loop
        diff:=r + nmentib;
        --d:=conv_integer(r)- conv_integer(mentib);
        if(diff(24)= '1') then
                pp(i):='0';
        else
                pp(i):='1';
                  r:=diff;
                  --r:=r -mentib;
      end if ;
        r:=r (23 downto 0) &'0';
 end loop;
 --ment<=pp;
  if(pp(24)='0') then
            --mentz:=pp(46 downto 23);
            e:=e-1;
            expt:=expt-1;
            pp:=pp(23 downto 0) & '0';
  else  null;
             --mentz := pp(47downto 24);
  end if;
  mentz := pp(24 downto 1);
```

```
        if(pp(0)='1') then
                mentz:=mentz+1;
        else null;
        end if;
        if(expt<1)then
                e:="11111111" ;
                mentz:="10000000000000000000001" ;
                underflow<='1';
        elsif(expt>254)then
                 e:="11111111" ;
                 mentz:="10000000000000000000001" ;
                 overflow<='1';
         else null;
         end if;
      end if;
   end if;
 z<= s & e & mentz(22 downto 0);
end process;
end Behavioral;
```

5.2) LNS Arithmetic Unit

5.2.1) Addition

5.2.1.1) Algorithm 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

package lns_mul184 is
function mul(a, b:  std_logic_vector(23 downto 0))
        return std_logic_vector;
end package lns_mul184;
```

```
package body lns_mul184 is
        function mul(a,b: std_logic_vector(23 downto 0))
        return std_logic_vector is
        variable rb,rd,re,ra: std_logic_vector(23 downto 0);
        begin
                rd:="000000000000000000000000";
                rb:=b;
                ra:=a;
                for i in 0 to 23 loop
                        if(rb(0)='1') then
                                re:=rd;
                                rd:= re + ra;
                        else null;
                        end if;
                        rb:=rd(0) & rb(23 downto 1);
                        rd:='0' & rd(23 downto 1);
                end loop;
                rd:=rd(22 downto 0) & rb(23);
                return rd;
        end function mul;
end package body lns_mul184;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.lns_mul184.all;

entity addinterpolation184 is
port(       a : in std_logic_vector(31 downto 0);
```

```
        b : in std_logic_vector(31 downto 0);
        z : out std_logic_vector(31 downto 0));
end addinterpolation184;

architecture Behavioral of addinterpolation184 is
signal dsig: std_logic_vector(30 downto 0);
signal logxsig, usig, a1sig, a2sig: std_logic_vector(23 downto 0);
begin
process(a,b)
variable diff: std_logic_vector(30 downto 0);
variable regt: std_logic_vector(31 downto 0);
subtype word24 is std_logic_vector(23 downto 0);
variable logx,a1,a2,u : word24;
variable index:integer;
type data is array(0 to 183) of word24;
constant lut : data:= (
"100000000000000000000000","011110000010110001011000","011100001011000100111001",
"011010011000111000100010","011000101100001001000010","010111000100110001111000",
......"000000000000000000000001");

begin
    if(a(30)='0' and b(30)='0') then
        if(a(29 downto 0) >= b(29 downto 0))then
            diff:= ('0'& a(29 downto 0)) - ('0' & b(29 downto 0));
            regt:=a;
        else
            diff:=('0' & b(29 downto 0)) - ('0' & a(29 downto 0));
            regt:= b;
        end if;
    elsif(a(30)='0' and b(30)='1') then
        diff:= ('0' & b(29 downto 0)) + ('0' & a(29 downto 0));
```

```
        regt:=a;
elsif(a(30)='1' and b(30) ='0') then
        diff:=('0' & a(29 downto 0)) + ('0' & b(29 downto 0));
        regt:=b;
elsif(a(30) ='1' and b(30)='1') then
        if(a(29 downto 0) >= b(29 downto 0))then
        diff:= ('0'& a(29 downto 0)) - ('0' & b(29 downto 0));
        regt:=b;
 else
        diff:=('0' & b(29 downto 0)) - ('0' & a(29 downto 0));
        regt:= a;
end if;
else null;
end if;
dsig<=diff;
if(diff>"00010110000000000000000000000000") then
         logx:="00000000000000000000000";
else
        index:=conv_integer(diff(27 downto 20));
        u:="0" & diff(19 downto 0) & "000" ;
        usig<=u;
        a1:=lut(index+1) - lut(index - 1);
        --a1:= '0' & a1(23 downto 1);
        a2:=lut(index + 1) + lut(index - 1);
        a2:='0' & a2(23 downto 1);
        a2:=a2 - lut(index);
        a2sig<=a2;
        if(a1(23)='0')then
                a1:= '0' & a1(23 downto 1);
                a1sig<=a1;
                logx:= lut(index) + mul(u,(a1+ mul(u,a2)));
```

```
            else
                    a1:= not(a1);
                    a1:=a1 + 1;
                    a1:= '0' & a1(23 downto 1);
                    a1sig<=a1;
                    logx:= lut(index)- mul(u,(a1 - mul(u,a2)));
                    --lut(index)-mul(u,(a1 - mul(u,a2)));
            end if;
        end if;
        logxsig<=lut(index);
        z<=regt + ("00000000" & logx);
end process;
end Behavioral;
```

5.2.1.2) Algorithm 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add1 is
port(   a : in std_logic_vector(31 downto 0);
        b : in std_logic_vector(31 downto 0);
        overflow,underflow: out std_logic;
        z : out std_logic_vector(31 downto 0));

end add1;

architecture Behavioral of add1 is
signal uint : std_logic;
signal regsig,regonex : std_logic_vector(25 downto 0);
```

```
function mulfun(a,b: std_logic_vector(25 downto 0))
        return std_logic_vector is
        variable rb,rd,re,ra: std_logic_vector(25 downto 0);
        begin
                rd:="00000000000000000000000000";
                rb:=b;
                ra:=a;
                for i in 0 to 25 loop
                        if(rb(0)='1') then
                                re:=rd;
                                rd:= re + ra;


                        else null;
                        end if;
                        rb:=rd(0) & rb(25 downto 1);
                        rd:='0' & rd(25 downto 1);
                end loop;
                rd:=rd(23 downto 0) & rb(25 downto 24);
                if(rb(23)='1') then
                        rd:=rd +1;
                else null;
                end if;
                return rd;
end function mulfun;
begin
process(a,b)
subtype word25 is std_logic_vector(25 downto 0);
type data is array(22 downto 0) of word25;
constant lut : data:=
("0101101010000010011110011 0","0100110000011011111110000 1","0100010111001010111 0
000100","0100001011010101011000011 1","0100000101100110110000110 1","0100000010110
```

```
0100110100100","0100000001011000111101101l","0100000000101100011011000","010000
0000010110001100l000","0100000000001011000110000","0100000000000101100010l111",
"0100000000000010110001011l","0100000000000001011000l100","0100000000000000l011
000110","0100000000000000010110001l","0100000000000000010110001","0100000000000
0000001011001","0100000000000000000101100","0100000000000000000010110","010000
0000000000000000l011","0100000000000000000000110","0100000000000000000000011",
"0100000000000000000000001");
constant lutlog : data:=
("0010110101000001001111001l","0011010111010001001111l101","0011101010110000001l
000111","0011110101001001010111l101","0011111010100000111011001l","0011111101001
1111000001100","0011111110100111100001000l","0011111111010011101100l011","001111
1111101001110101011O","0011111111110100l110100111","0011111111111010011101001l",
"0011111111111101001110100l","0011111111111110100111010l","0011111111111111010
111010","0011111111111111110100l1101","0011111111111111111010011l1","0011111111111
1111110100111","0011111111111111111110101O0","0011111111111111111101010","001111
1111111111111110101","0011111111111111111111010","0011111111111111111111101",
"0011111111111111111111111");
variable dvar : std_logic_vector(30 downto 0) ;
variable regc : std_logic_vector(31 downto 0);
variable rega,regb,regk,regd : std_logic_vector(25 downto 0);
variable d23:std_logic;
variable pp:std_logic_vector(51 downto 0);
variable k,x : integer;
begin
        if(a(30 downto 0)>b(30 downto 0)) then
                regc:=a;
                dvar:=a(30 downto 0)- b(30 downto 0) ;
        else
                 regc:=b;
                 dvar:=b(30 downto 0)- a(30 downto 0);
        end if;
```

```
if(dvar(22 downto 0)/=0)then
        dvar(30 downto 23):=dvar(30 downto 23) + 1;
        d23:=dvar(23);
        dvar(23 downto 0):="10000000000000000000000"-dvar(22 downto 0);
        dvar(23):=d23;
else
        dvar:=dvar;
end if;
rega:="01000000000000000000000000";
for i in 22 downto 0 loop
        if(dvar(i)='1') then
                regb:=lut(i);
                regd:=mulfun(regb,rega);
                rega:=regd;
        else null;
        end if;
end loop;
x:=conv_integer(dvar(30 downto 23));
--uint:=x;
for i in 0 to 25 loop
        if(x>0) then
                rega:='0' & rega(25 downto 1);
                 x:=x-1;
         else null;
        end if;
end loop;
rega:= "01000000000000000000000000" + rega;
regsig<=rega;
x:=0;
--uint<=x;
regb:="00000000000000000000000000";
```

G 14332

```
        for i in 0 to 22 loop
                if(rega>=lut(22-i)) then
                        regb(25-i):='1';
                        regd:=lutlog(22-i);
                        regk:=mulfun(regd,rega);
                        rega:=regk;
                else regb(25-i):='0';
                end if;
        end loop;
        regonex<=regb;

        z<=regc+( "000000000" & regb(25 downto 3));
end process;
end Behavioral;
```

5.2.2) Subtraction

5.2.2.1) Algorithm 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

package lns_muls7 is
function mul(a, b:  std_logic_vector(23 downto 0))
        return std_logic_vector;
end package lns_muls7;
package body lns_muls7 is
        function mul(a,b: std_logic_vector(23 downto 0))
         return std_logic_vector is
        variable rb,rd,re,ra: std_logic_vector(23 downto 0);
                begin
                rd:="000000000000000000000000";
```

```
            rb:=b;
            ra:=a;
            for i in 0 to 23 loop
                  if(rb(0)='1') then
                        re:=rd;
                        rd:= re + ra;
                  else null;
                  end if;
                  rb:=rd(0) & rb(23 downto 1);
                  rd:='0' & rd(23 downto 1);
            end loop;
            rd:=rd(22 downto 0) & rb(23);
            return rd;
      end function mul;
end package body lns_muls7;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.lns_mul7.all;

entity subinter7 is
port(     a : in std_logic_vector(31 downto 0);
          b : in std_logic_vector(31 downto 0);
          z : out std_logic_vector(31 downto 0));
end subinter7;

architecture Behavioral of subinter7 is
signal dsig: std_logic_vector(30 downto 0);
```

```
signal logxsig, usig, a1sig, a2sig: std_logic_vector(23 downto 0);
signal indexint: integer;
begin
process(a,b)
variable diff: std_logic_vector(30 downto 0);
variable regt: std_logic_vector(31 downto 0);
subtype word24 is std_logic_vector(23 downto 0);
variable logx,a1,a2,u : word24;
variable index:integer;
type data1 is array(0 to 255) of word24;
constant lut1 : data1:=(
"000011111111111111111111","100010000111110111010011","011110001000010111010001",
"011011110011000111001101","011010001001010111001100","011000110111011100101010",
......"000100000001000000001011");

type data2 is array(0 to 175) of word24;
constant lut2 : data2:= (
"100000000000000000000000","011100010100011011000100","011001001011110000011010",
"010110011111000000010001","010100001001000000010011","010010000101110101111011",
.....000000000000000000000001");

begin
        if(a(30)='0' and b(30)='0') then
                if(a(29 downto 0) >= b(29 downto 0))then
                        diff:= ('0'& a(29 downto 0)) - ('0' & b(29 downto 0));
                        regt:=a;
                else
                        diff:=('0' & b(29 downto 0)) - ('0' & a(29 downto 0));
                        regt:= '1' &  b(30 downto 0);
                end if;
        elsif(a(30)='0' and b(30)='1') then
```

```
        diff:= ('0' & b(29 downto 0)) + ('0' & a(29 downto 0));
        regt:=a;
elsif(a(30)='1' and b(30) ='0') then
        diff:=('0' & a(29 downto 0)) + ('0' & b(29 downto 0));
        regt:=b;
elsif(a(30) ='1' and b(30)='1') then
        if(a(29 downto 0) >= b(29 downto 0))then
                diff:= ('0'& a(29 downto 0)) - ('0' & b(29 downto 0));
                regt:=b;
        else
                diff:=('0' & b(29 downto 0)) - ('0' & a(29 downto 0));
                regt:= '1' &  a(30 downto 0);
        end if;
else null;
end if;
dsig<=diff;
if(diff>"00010110000000000000000000000000") then
        logx:="00000000000000000000000";
elsif(diff>="00000001000000000000000000000000") then
        index:=conv_integer(diff(27 downto 20))-  8;
        indexint<=index;
        logxsig<=lut2(index);
        u:="0" & diff(19 downto 0) & "000" ;
        usig<=u;
        a1:=   lut2(index - 1)- lut2(index+1);
         a1:= '0' & a1(23 downto 1);
        a1sig<=a1;
        --a1:=lut(index+1) - lut(index - 1);
        --a1:= '0' & a1(23 downto 1);
        a2:= lut2(index + 1) + lut2(index - 1);
        a2:='0' & a2(23 downto 1);
```

```
        a2:=lut2(index) - a2;
        if(a2(23)='0')then
                --a2:= '0' & a2(23 downto 1);
                a2sig<=a2;
                logx:=  lut2(index)- mul(u,(a1+ mul(u,a2))) ;
        else
                a2:= not(a2);
                 a2:=a2 + 1;
                --a2:= '0' & a2(23 downto 1);
                a2sig<=a2;
                logx:= lut2(index) - mul(u,(a1 - mul(u,a2)));
                 --lut(index)-mul(u,(a1 - mul(u,a2)));
        end if;
        z<=regt - ("00000000" & logx);
elsif(diff>="00000000001000000000000000000") then
        index:=conv_integer(diff(27 downto 15));
        indexint<=index;
        logxsig<=lut1(index);
        u:="0" & diff(14 downto 0) & "00000000" ;
        usig<=u;
        a1:=  lut1(index - 1)- lut1(index+1);
        a1:= '0' & a1(23 downto 1);
        a1sig<=a1;
        --a1:=lut(index+1) - lut(index - 1);
        --a1:= '0' & a1(23 downto 1);
        a2:= lut1(index + 1) + lut1(index - 1);
        a2:='0' & a2(23 downto 1);
        a2:=lut1(index) - a2;
        if(a2(23)='0')then
                --a2:= '0' & a2(23 downto 1);
                 a2sig<=a2;
```

```
                logx:=  lut1(index)- mul(u,(a1+ mul(u,a2))) ;
            else
                a2:= not(a2);
                a2:=a2 + 1;
                --a2:= '0' & a2(23 downto 1);
                a2sig<=a2;
                logx:= lut1(index) - mul(u,(a1 - mul(u,a2)));
            end if;
            z<=regt - ("00000" & logx & "000");
        else null;
        end if;
end process;
end Behavioral;
```

5.2.2.2) Algorithm 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sublut2 is
port(   a : in std_logic_vector(31 downto 0);
        b : in std_logic_vector(31 downto 0);
        overflow,underflow: out std_logic;
        z : out std_logic_vector(31 downto 0));
end sublut2;

architecture Behavioral of sublut2 is
signal uint : std_logic;
signal regsig,regonex : std_logic_vector(25 downto 0);
function mulfun(a,b: std_logic_vector(25 downto 0))
```

```
        return std_logic_vector is
        variable rb,rd,re,ra: std_logic_vector(25 downto 0);
        begin
        rd:="00000000000000000000000000";
        rb:=b;
        ra:=a;
        for i in 0 to 25 loop
                if(rb(0)='1') then
                        re:=rd;
                        rd:= re + ra;

                else null;
                end if;
                rb:=rd(0) & rb(25 downto 1);
                rd:='0' & rd(25 downto 1);
        end loop;
        rd:=rd(23 downto 0) & rb(25 downto 24);
        if(rb(23)='1') then
                rd:=rd +1;
        else null;
        end if;
        return rd;
end function mulfun;
begin
process(a,b)
subtype word25 is std_logic_vector(25 downto 0);
type data is array(22 downto 0) of word25;
constant lut : data:= (
"01011010100000100111100110","01001100000110111111100001","01000101110010101110
000100","01000010110101010110000111","01000001011001101100001101","0100000010110
0100110100100","01000000010110001111011011","01000000001011000110110000","010000
```

```
00000101100011001000","01000000000010110001100000","01000000000001011000101111",
"01000000000000101100010111","01000000000000010110001100","01000000000000001011
000110","01000000000000000101100011","01000000000000000010110001","0100000000000
0000001011001","01000000000000000000101100","01000000000000000000010110","010000
0000000000000001011","01000000000000000000000110","01000000000000000000000011",
"01000000000000000000000001");
constant lutlog:
data:=("00101101010000010011110011","00110101110100010011111101","001110101011000
00011000111","00111101010010010101111101","00111110101000001110110011","001111110
10011111000001100","00111111101001111000010001","00111111110100111011001011","001
11111111010011101010110","00111111111101001110100111","00111111111110100111010011
1","00111111111111010011101001","00111111111111101001110100","00111111111111110100111010","00111111111111111010011101","00111111111111111101001111","0011111111111
1111110100111","00111111111111111111010100","00111111111111111111101010","0011111
11111111111110101","00111111111111111111111010","00111111111111111111111101","0
01111111111111111111111");

variable dvar : std_logic_vector(30 downto 0) ;
variable regc : std_logic_vector(31 downto 0);
variable rega,regb,regk,regd : std_logic_vector(25 downto 0);
variable d23:std_logic;
variable pp:std_logic_vector(51 downto 0);
variable k,x : integer;
begin
        if(a(30 downto 0)>b(30 downto 0)) then
                regc:=a;
                dvar:=a(30 downto 0)- b(30 downto 0) ;
        else
                regc:='1' & b(30 downto 0);
                dvar:=b(30 downto 0)- a(30 downto 0);
        end if;
```

```
if(dvar(22 downto 0)/=0)then
        dvar(30 downto 23):=dvar(30 downto 23) + 1;
        d23:=dvar(23);
        dvar(23 downto 0):="10000000000000000000000"-dvar(22 downto 0);
         dvar(23):=d23;
else
        dvar:=dvar;
end if;

rega:="01000000000000000000000000";
for i in 22 downto 0 loop
        if(dvar(i)='1') then
                regb:=lut(i);
                regd:=mulfun(regb,rega);
                rega:=regd;
        else null;
        end if;
end loop;
x:=conv_integer(dvar(30 downto 23));
--uint:=x;
for i in 0 to 25 loop
if(x>0) then
         rega:='0' & rega(25 downto 1);
        x:=x-1;
else null;
end if;
end loop;
rega:= "01000000000000000000000000" - rega;
regsig<=rega;
x:=0;
for i in 0 to 25 loop
```

```
            if(rega(24)='0')then
                rega:=rega(24 downto 0) & '0';
                regc(30 downto 23):=regc(30 downto 23) - "00000001";
            else null;
            end if;
        end loop;
        regb:="00000000000000000000000000";
        for i in 0 to 22 loop
            if(rega>=lut(22-i)) then
                regb(25-i):='1';
                regd:=lutlog(22-i);
                regk:=mulfun(regd,rega);
                rega:=regk;
            else regb(25-i):='0';
            end if;
        end loop;
        regonex<=regb;
        z<=regc+( "000000000" & regb(25 downto 3));
end process;
end Behavioral;
```

5.2.3) Multiplication

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multi is
port(   a : in std_logic_vector(31 downto 0);
        b : in std_logic_vector(31 downto 0);
        overflow,underflow: out std_logic;
```

```
        z : inout std_logic_vector(31 downto 0));
end multi;

architecture Behavioral of multi is
begin
process(a,b)
variable rega,regb,regt : std_logic_vector(29 downto 0);
variable regd:  std_logic_vector(30 downto 0);
variable se, sz: std_logic ;
begin
        underflow<='0';
        overflow<='0';
        rega:=a(29 downto 0);
        regb:=b(29 downto 0);
        se:=a(30);
        if(rega < regb)then
                regt:=rega;
                rega:=regb;
                regb:=regt;
                se:=b(30);
         else null;
        end if;
        sz:= a(31) xor b(31);
        if(((a(30)='0' and b(30)='1'))or ((a(30)='1' and b(30)='0'))) then
                 regd:=("0" & rega) - ("0" & regb);
                 z<= sz & se & regd(29 downto 0);
        elsif((a(30)='0' and b(30)='0')) then
                regd:=("0" & rega) + ("0" & regb);
                if(regd(30)='1') then
                        overflow<='1';
                        z<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
```

```
                else
                        z<= sz &  regd(30 downto 0);
                end if;
        elsif((a(30)='1' and b(30)='1')) then
                regd:=("0" & rega) + ("0" & regb);
                if(regd(30)='1') then
                        underflow<='1';
                        z<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                else
                        z<= sz & '1' & regd(29 downto 0);
                 end if;
        else null;
        end if;
end process;
end Behavioral;
```

5.2.4) Division

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity div is
port(   a : in std_logic_vector(31 downto 0);
        b : in std_logic_vector(31 downto 0);
        overflow,underflow: out std_logic;
        z : inout std_logic_vector(31 downto 0));
end div;

architecture Behavioral of div is
begin
```

```
process(a,b)
variable rega,regb,regt : std_logic_vector(29 downto 0);
variable regd:  std_logic_vector(30 downto 0);
variable se, sz: std_logic ;
begin
        underflow<='0';
        overflow<='0';
        rega:=a(29 downto 0);
       regb:=b(29 downto 0);
       se:=a(30);
       if(rega < regb)then
              regt:=rega;
             rega:=regb;
             regb:=regt;
             se:=not(b(30));
       else null;
       end if;
       sz:= a(31) xor b(31);
       if(((a(30)='0' and b(30)='0')) or (a(30)='1' and b(30)='1')) then
             regd:=("0" & rega) - ("0" & regb);
              z<= sz & se & regd(29 downto 0);
       elsif((a(30)='0' and b(30)='1')) then
              regd:=("0" & rega) + ("0" & regb);
              if(regd(30)='1') then
                    overflow<='1';
                     z<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
             else
                     z<= sz & '0' & regd(29 downto 0);
             end if;
        elsif((a(30)='1' and b(30)='0')) then
             regd:=("0" & rega) + ("0" & regb);
```

```
                if(regd(30)='1') then
                        underflow<='1';
                        z<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
                else
                        z<= sz & '1' & regd(29 downto 0);
                end if;
        else null;
        end if;
end process;
end Behavioral;
```

Chapter 6

Results and Discussion

6.1 FP Simulation Results

6.1.1 Addition

1) A = 01000010011001111000101101110010

B = 01000001100001110101111010001011

Then calculated Z = A + B = 01000010100101011001110101011100, as shown in fig. 6.1.

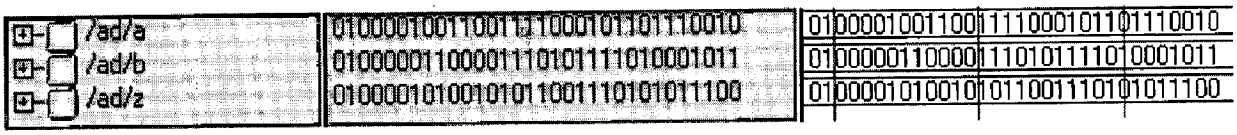


Fig. 6.1. Simulation of FP addition example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 74.80734311

2) A = 01000010010000010000001001110110

B = 01000010010110001111101000001111

Then calculated Z = A + B = 01000010110011001111111001000011, as shown in fig. 6.2.

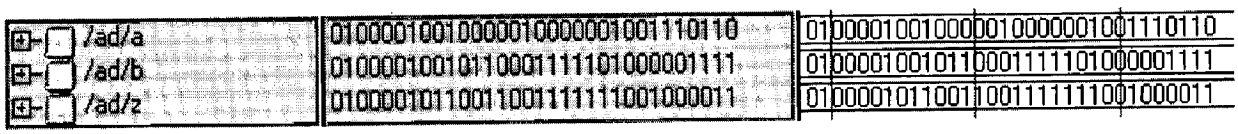


Fig. 6.2. Simulation of FP addition example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 102.4966075

3) A = 01000010011001111000101101110010

B = 01000010010000010000001001110110

Then calculated Z = A + B = 01000010110101000100011011110100, as shown in fig. 6.3.

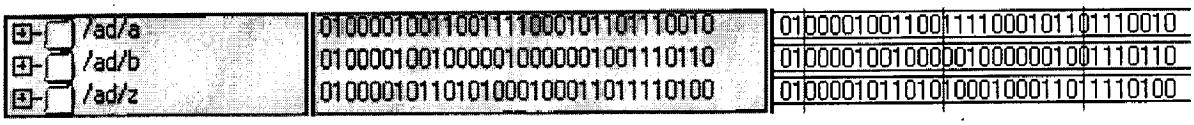


Fig. 6.3. Simulation of FP addition example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 106.1385849

6.1.2 Subtraction

1) A = 01000010011001111000101101110010

B = 01000001100001110101111010001011

Then calculated Z = A - B = 01000010001000111101110000101100, as shown in fig. 6.4.

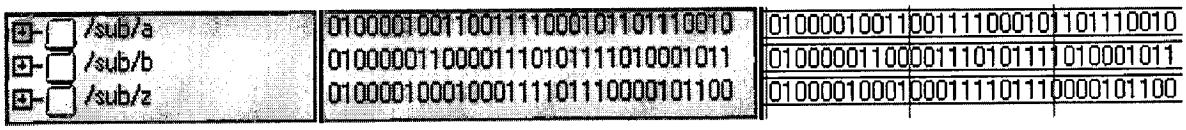


Fig. 6.4. Simulation of FP subtraction example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.96501297

2) A = 01000010010000010000001001110110

B = 01000010010110001111101000001111

Then calculated Z = A - B = 11000000101111111011110011001000, as shown in fig. 6.5.

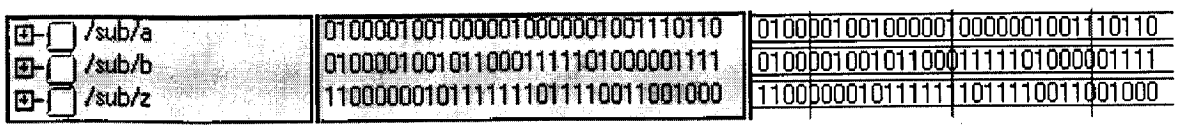


Fig. 6.5. Simulation of FP subtraction example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -5.99179458

3) A = 01000010011001111000101101110010

B = 01000010010000010000001001110110

Then calculated Z = A - B = 01000001000110100010001111110000, as shown in fig. 6.6.

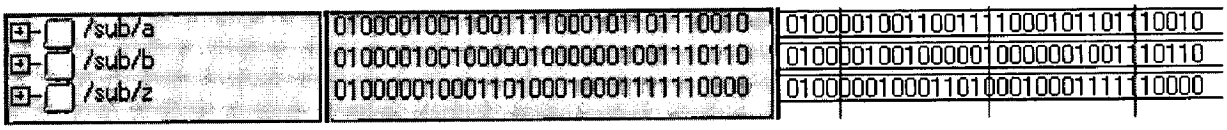


Fig. 6.6. Simulation of FP subtraction example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.63377380

6.1.3 Multiplication

1) A = 01000010011001111000101101110010

B = 01000001100001110101111010001011

Then calculated Z = A * B = 01000100011101001110000000011000, as shown in fig. 6.7.

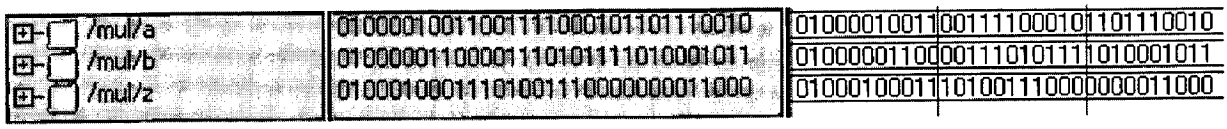


Fig. 6.7. Simulation of FP multiplication example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 979.5014648

2) A = 01000010010000010000001001110110

B = 01000010010110001111101000001111

Then calculated Z = A * B = 01000101001000111001011010011011, as shown in fig. 6.8.

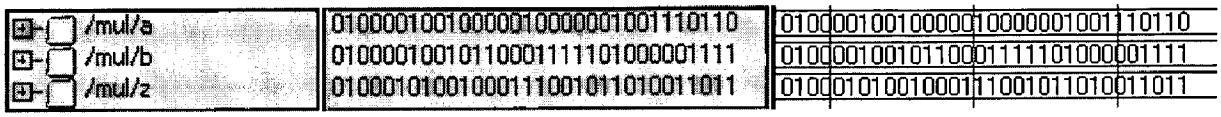


Fig. 6.8. Simulation of FP multiplication example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 2617.412842

3) A = 01000010011001111000101101110010

B = 01000010010000010000001001110110

Then calculated Z = A * B = 01000101001011101001001001011011, as shown in fig. 6.9.

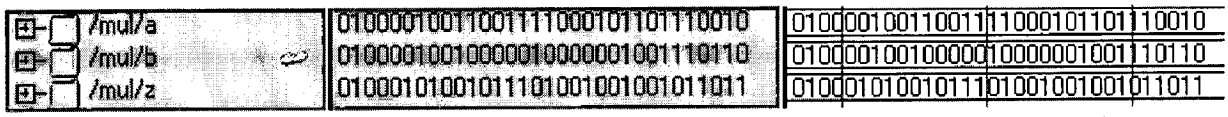


Fig. 6.9. Simulation of FP multiplication example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 2793.147411

6.1.4 Division

1) A = 01000010011001111000101101110010

B = 01000001100001110101111010001011

Then calculated Z = A / B = 01000000010110101111000010010100, as shown in fig. 6.10.

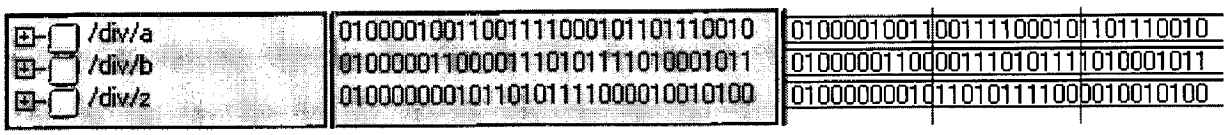


Fig. 6.10. Simulation of FP division example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 3.420933723

2) A = 01000010010000010000001001110110

B = 01000010010110001111101000001111

Then calculated Z = A / B = 00111111011000111011100011101010, as shown in fig. 6.11.

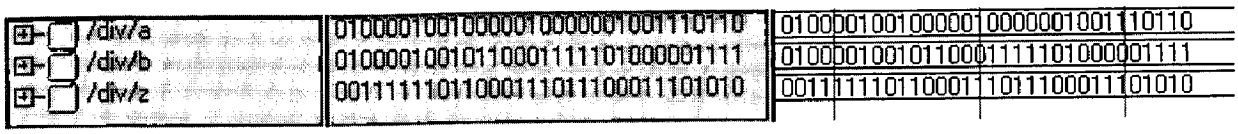


Fig. 6.11. Simulation of FP division example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 0.889540315

3) A = 01000010011001111000101101110010

B = 01000010010000010000001001110110

Then calculated Z = A / B = 00111111100110011000111001000001, as shown in fig. 6.12.

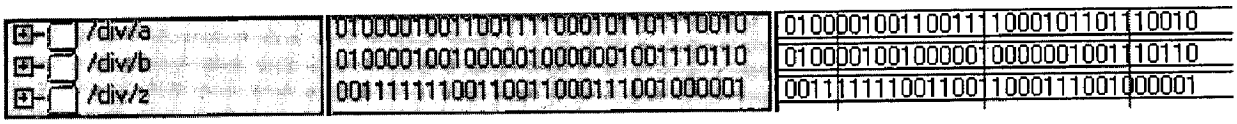


Fig. 6.12. Simulation of FP division example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 1.199653745

6.2 LNS Simulation Results

6.2.1 Addition

6.2.1.1 Algorithm 1

A) Size of ROM= 92

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A + B = 00000011000111001100111111110110, as shown in fig. 6.13.

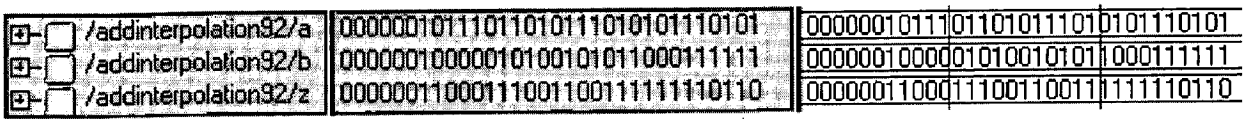


Fig. 6.13. Simulation of LNS addition algorithm 1 with ROM size 92 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 74.80672154

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A + B = 00000011010110011100111000010001, as shown in fig. 6.14.

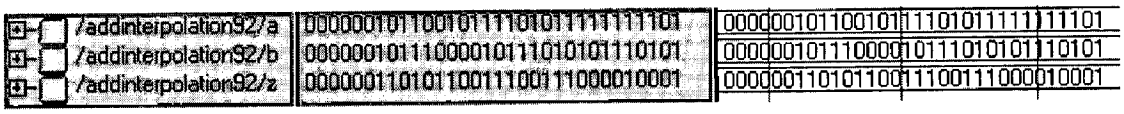


Fig. 6.14. Simulation of LNS addition algorithm 1 with ROM size 92 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 104.0837474

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A + B = 00000011010111010110101000111000, as shown in fig. 6.15.

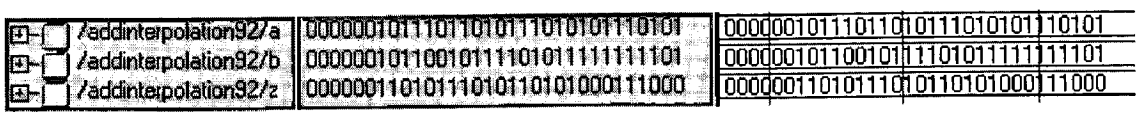


Fig. 6.15. Simulation of LNS addition algorithm 1 with ROM size 92 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 106.1384812

B) Size of ROM = 184

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A + B = 00000011000111001101000000111111, as shown in fig. 6.16.

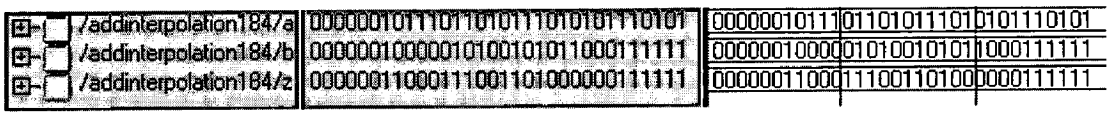


Fig. 6.16. Simulation of LNS addition algorithm 1 with ROM size 184 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 74.80718821

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A + B = 00000011010101101111011110011110, as shown in fig. 6.17.

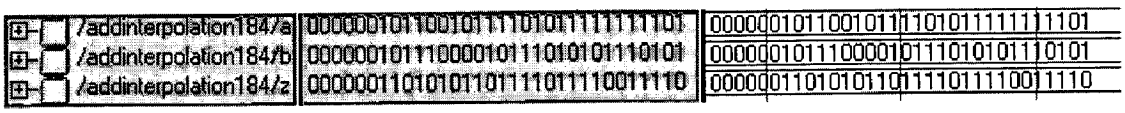


Fig. 6.17. Simulation of LNS addition algorithm 1 with ROM size 184 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 102.4965827

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A + B = 00000011010111010110101001000001, as shown in fig. 6.18.

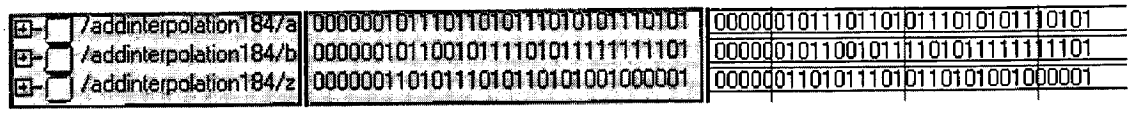


Fig. 6.18. Simulation of LNS addition algorithm 1 with ROM size 184 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 106.1385548

6.2.1.2 Algorithm 2

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A + B = 00000011000111001101000001010110, as shown in fig. 6.19.

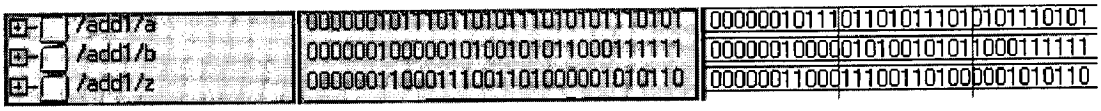


Fig. 6.19. Simulation of LNS addition algorithm 2 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 74.80734377

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A + B = 00000011010101101111011110100010, as shown in fig. 6.20.

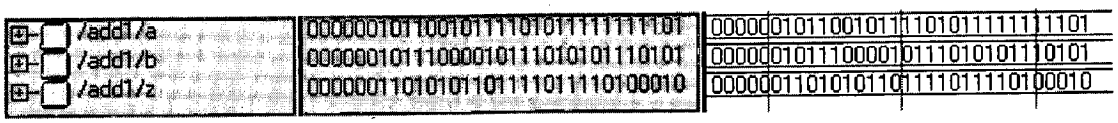


Fig. 6.20. Simulation of LNS addition algorithm 2 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 102.4965827

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A + B = 00000011010111010110101001000001, as shown in fig. 6.21.

Fig. 6.21. Simulation of LNS addition algorithm 2 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 106.1385548

6.2.2 Subtraction

6.2.2.1 Algorithm 1

A) Size of ROM = 184

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.22.

Fig. 6.22. Simulation of LNS subtraction algorithm 1 with ROM size 184 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001011100101001100101101101, as shown in fig. 6.23.

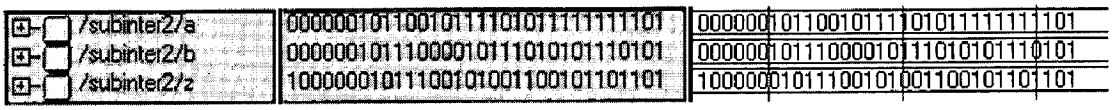


Fig. 6.23. Simulation of LNS subtraction algorithm 1 with ROM size 184 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -7.440022581

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000110011000101100101, as shown in fig. 6.24.

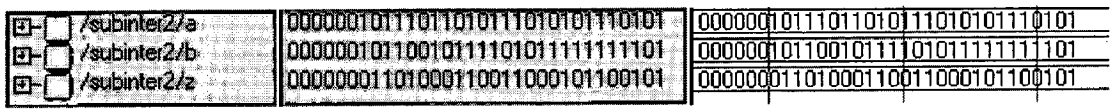


Fig. 6.24. Simulation of LNS subtraction algorithm 1 with ROM size 184 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.679583691

B) Size of ROM = 192

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.25.

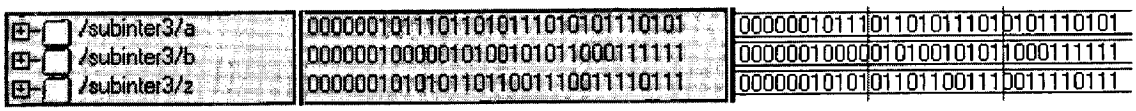


Fig. 6.25. Simulation of LNS subtraction algorithm 1 with ROM size 192 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010011010010111111000101, as shown in 6.26.

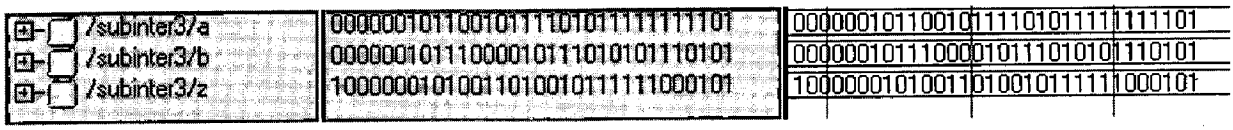


Fig. 6.26. Simulation of LNS subtraction algorithm 1 with ROM size 192 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -6.075570984

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000101000000011100101, as shown in fig. 6.27.

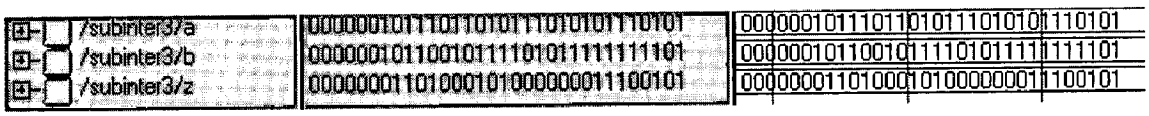


Fig. 6.27. Simulation of LNS subtraction algorithm 1 with ROM size 192 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.643514362

C) Size of ROM = 208

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.28.

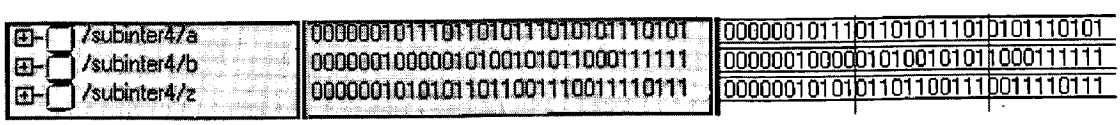


Fig. 6.28. Simulation of LNS subtraction algorithm 1 with ROM size 208 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010010101100100010000101, as shown in fig. 6.29.

Fig. 6.29. Simulation of LNS subtraction algorithm 1 with ROM size 208 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -5.997012583

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000100101101101000101, as shown in fig. 6.30.

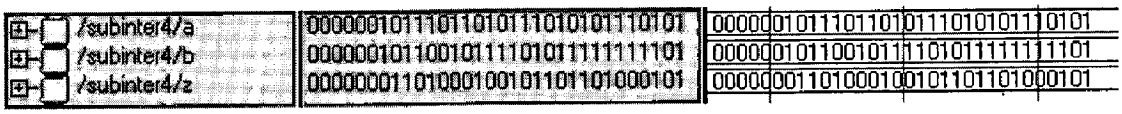


Fig. 6.30. Simulation of LNS subtraction algorithm 1 with ROM size 208 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.635837073

D) Size of ROM = 240

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.31.

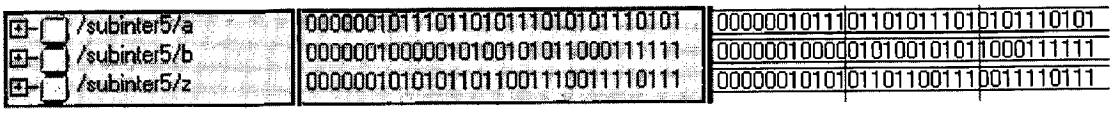


Fig. 6.31. Simulation of LNS subtraction algorithm 1 with ROM size 240 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010010101010001110010101, as shown in fig. 6.32.

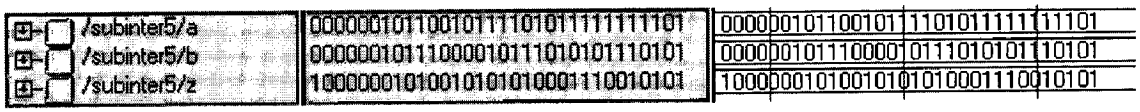


Fig. 6.32. Simulation of LNS subtraction algorithm 1 with ROM size 240 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -5.992329685

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000100101001000101101, as shown in fig. 6.33.

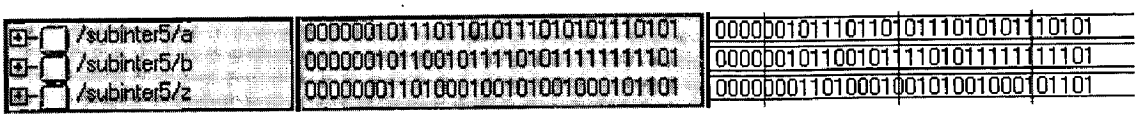


Fig. 6.33. Simulation of LNS subtraction algorithm 1 with ROM size 240 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.633987152

E) Size of ROM = 304

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.34.

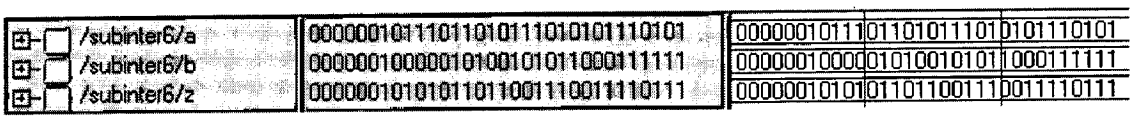


Fig. 6.34. Simulation of LNS subtraction algorithm 1 with ROM size 304 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010010101010000000000101, as shown in fig. 6.35.

Fig. 6.35. Simulation of LNS subtraction algorithm 1 with ROM size 304 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -5.991876964

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000100101000101001101, as shown in fig. 6.36.

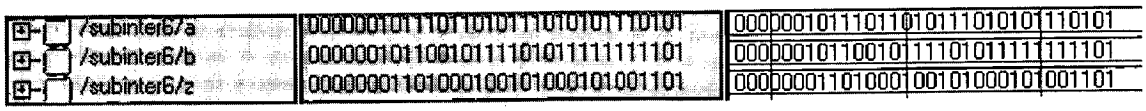


Fig. 6.36. Simulation of LNS subtraction algorithm 1 with ROM size 304 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.633806854

F) Size of ROM = 432

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001110011110111, as shown in fig. 6.37.

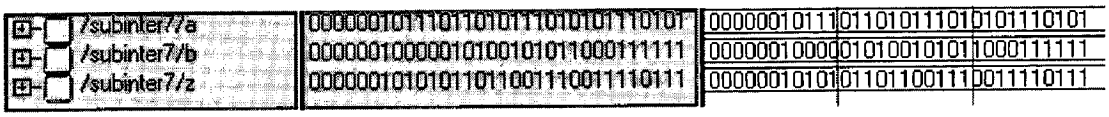


Fig. 6.37. Simulation of LNS subtraction algorithm 1 with ROM size 432 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.9659396

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010010101001111101100101, as shown in fig. 6.38.

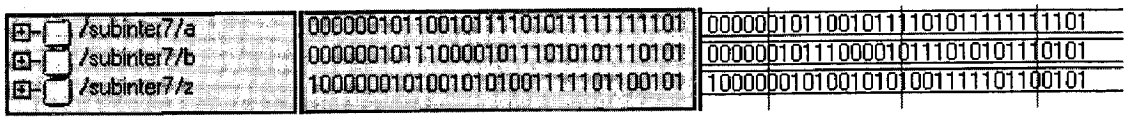


Fig. 6.38. Simulation of LNS subtraction algorithm 1 with ROM size 432 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = - 5.991798052

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000100101000100100101, as shown in fig. 6.39.

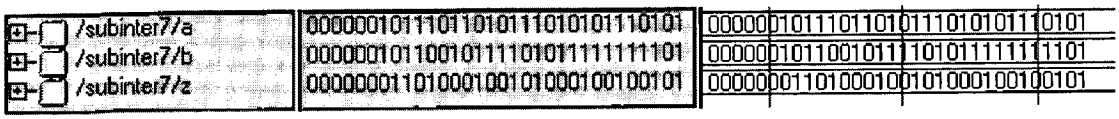


Fig. 6.39. Simulation of LNS subtraction algorithm 1 with ROM size 432 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.633773466

6.2.2.2 Algorithm 2

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A - B = 00000010101011011001101111101000, as shown in fig. 6.40.

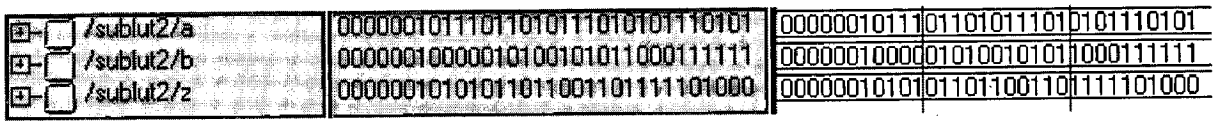


Fig. 6.40. Simulation of LNS subtraction algorithm 2 example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 40.96500256

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A - B = 10000001010010101001111101100000, as shown in fig. 6.41.

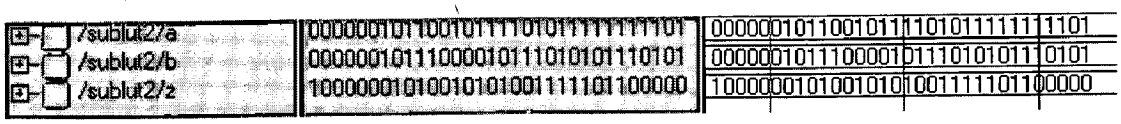


Fig. 6.41. Simulation of LNS subtraction algorithm 2 example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = -5.991798052

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A - B = 00000001101000100101000100101010, as shown in fig. 6.42.

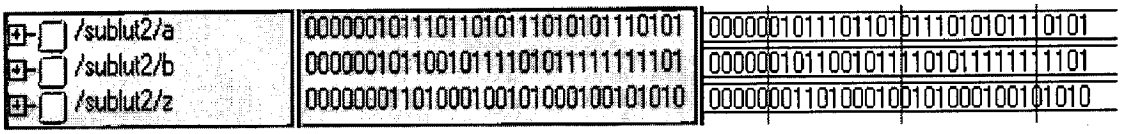


Fig. 6.42. Simulation of LNS subtraction algorithm 2 example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 9.633780144

6.2.3 Multiplication

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A * B = 00000100111101111100101110110100, as shown in fig. 6.43.

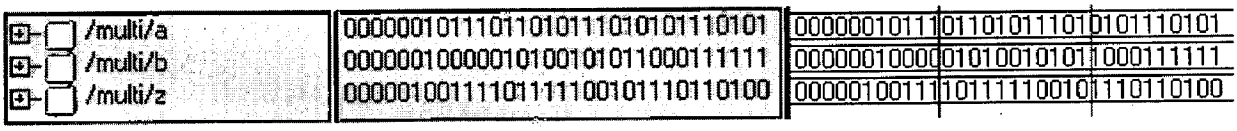


Fig. 6.43. Simulation of LNS multiplication example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 979.5015736

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A * B = 00000101101011010100110101110010, as shown in fig. 6.44.

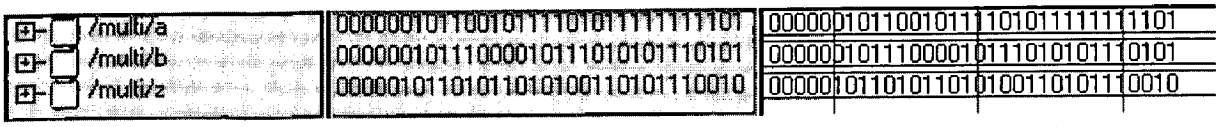


Fig. 6.44. Simulation of LNS multiplication example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 2617.413238

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A * B = 00000101101110010100110101110010, as shown in fig. 6.45.

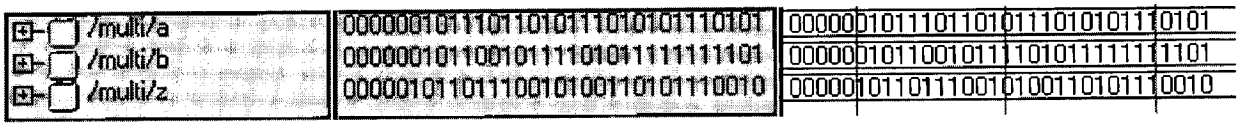


Fig. 6.45. Simulation of LNS multiplication example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 2793.147411

6.2.4 Division

1) A = 00000010111011010111010101110101

B = 00000010000010100101011000111111

Then calculated Z = A / B = 00000000111000110001111100110110, as shown in fig. 6.46.

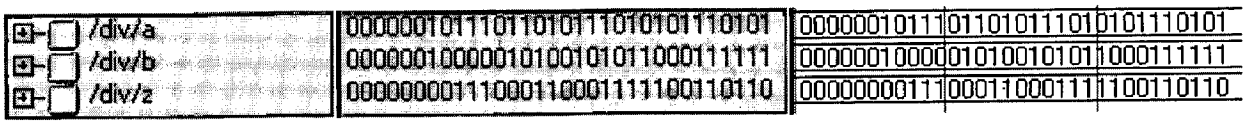


Fig. 6.46. Simulation of LNS division example 1.

Values in decimal number system are

A = 57.88617804

B = 16.92116507

Z = 3.420933359

2) A = 00000010110010111101011111111101

B = 00000010111000010111010101110101

Then calculated Z = A / B = 01000000000101011001110101111000, as shown in fig. 6.47.

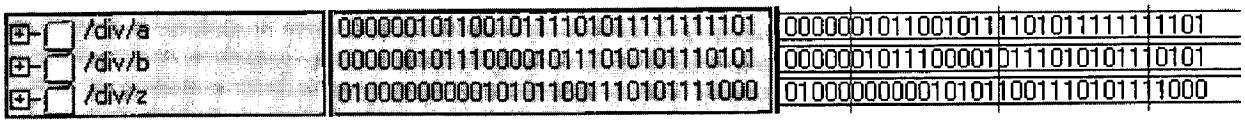


Fig. 6.47. Simulation of LNS division example 2.

Values in decimal number system are

A = 48.25240681

B = 54.24420067

Z = 0.889540378

3) A = 00000010111011010111010101110101

B = 00000010110010111101011111111101

Then calculated Z = A / B = 00000000001000011001110101111000, as shown in fig. 6.48.

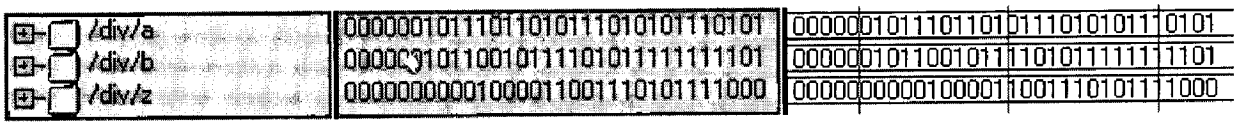


Fig. 6.48. Simulation of LNS division example 3.

Values in decimal number system are

A = 57.88617804

B = 48.25240681

Z = 1.199653693

Tables 6.1 – 6.4 show the synthesis results for different FP and LNS operations obtained by using Xilinx (for Spartan 3). Tables 6.5 & 6.6 show the variation of accuracy for LNS addition and subtraction as the size of ROM varies. From tables 6.3 and 6.4, it is clear that LNS multiplication and division can be done very efficiently for FPGA in comparison to the FP multiplication and division. Tables 6.1 and 6.2 show that FP addition and subtraction FPGA implementations are much better than LNS addition and subtraction. LNS addition algorithm 2

and LNS subtraction algorithm 2 results are more accurate and less values to be stored in the ROM in comparison to LNS addition algorithm 1 and LNS subtraction algorithm 1. But LNS addition algorithm 2 and LNS subtraction algorithm 2 implementation on FPGA requires so many slices and 4 input LUTs that their implementation is not a good idea in comparison to the FP.

In tables 6.5 and 6.6, all results are shown in decimal number format after conversion from LNS for easy comparison. Tables 6.5 & 6.6 show that output approaches the exact output as the size of the ROM is increased. In LNS subtraction algorithm 1, while increasing the ROM size, the values are added for the variation of x from 0.5 to 0 and keeping rest of the ROM same. As a result in table 6.6, in the first example result is same for all sizes of ROM for algorithm 1 because for that x is 1.77439.

Table 6.1. Addition Synthesis Results

Number system		Size of ROM	Number of Slices	Number of 4 input LUTs	Total delay (in ns)	Total memory usage (in KB)
FP		0	411	729	42.312	86020
LNS	Algorithm 1	92	2956	3898	224.311	137732
		184	3173	4302	224.684	144900
	Algorithm 2	46	27976	46310	2896.787	1142532

Table 6.2. Subtraction Synthesis Results

Number system		Size of ROM	Number of Slices	Number of 4 input LUTs	Total delay (in ns)	Total memory usage (in KB)
FP		0	1091	1878	118.207	96772
LNS	Algorithm 1	184	6889	9097	221.284	174084
		192	6720	9007	223.841	170500
		208	6629	9003	224.022	170500
		240	6714	9189	224.933	172548
		304	6779	9468	222.753	177668
		432	7447	10577	222.878	190340
	Algorithm 2	46	28773	47615	2921.622	1142532

Table 6.3. Multiplication Synthesis Results

Number system	Size of ROM	Number of Slices	Number of 4 input LUTs	Total delay (in ns)	Total memory usage (in KB)
FP	0	696	1271	81.549	84420
LNS	0	142	251	38.441	71108

Table 6.4. Division Synthesis Results

Number system	Size of ROM	Number of Slices	Number of 4 input LUTs	Total delay (in ns)	Total memory usage (in KB)
FP	0	716	1324	183.159	83460
LNS	0	143	252	38.441	71108

Table 6.5. LNS Addition Examples

A	B	Exact result	Algorithm 1		Algorithm 2
			Size of ROM		
			92	184	
57.886178	16.921165	74.80734	74.80672	74.80718	74.80734
54.244201	48.252407	102.4966	104.0837	102.4966	102.4966
57.886178	48.252407	106.1386	106.1385	106.1386	106.1386

Table 6.6. LNS Subtraction Examples

A	B	Exact result	Algorithm 1						Algorith 2
			Size of ROM						
			184	192	208	240	304	432	
57.886178	16.921165	40.96501	40.96594	40.96594	40.96594	40.96594	40.96594	40.96594	40.965
54.244201	48.252407	5.991794	7.440023	6.075571	5.997013	5.992330	5.991877	5.991798	5.9917
57.886178	48.252407	9.633771	9.679584	9.643514	9.635837	9.633987	9.633807	9.633773	9.6337

Chapter 7

Conclusions

LNS has very efficient implementation of multiplication and division operations in comparison to the floating point. But LNS main disadvantage is its addition and subtraction operations. To obtain good accuracy in LNS addition and subtraction, more values should be stored in the ROM. As a result, FPGA utilization increases. LNS addition and subtraction require different set of values to be stored in ROM i.e. same ROM can not be used for both operations. FP addition and subtraction are simple and does not require ROM. The problem is more aggravate in subtraction because the value of $\log_2 x$ varies from -1 to $-\infty$ as x varies from 0.5 to 0. These values of x do not occur during addition. Therefore, more values are stored in ROM for x variation between 0.5 and 0. This is the reason that in LNS subtraction while increasing the ROM size, the values are added for the variation of x from 0.5 to 0 and keeping rest of the ROM same. As a result in table 6.6, in the first example result is same for all sizes of ROM for algorithm 1 because for that x is 1.77439. There are algorithms for LNS addition and subtraction (LNS addition algorithm 2 and LNS subtraction algorithm 2) for which number of values in the ROM are fixed, have good accuracy and same ROM can be used for both LNS addition & subtraction, but they require so many numbers of FPGA slices that their FPGA implementation is not a good idea in comparison to the FP. So, the final choice left is to use the floating point representation to represent the large values of real numbers.

References

[1] IEEE Standards Board, "IEEE Standard for Binary Floating-point Arithmetic," Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985

[2] E. Swartzlander and Aristides G. Alexopoulos, "The Sign/Logarithm Number System," IEEE Transactions on Computers, vol. C-24, no. 12, pp. 1238- 1242, December 1975

[3] I. Koren, "Computer Arithmetic Algorithms", A.K. Peters Ltd., 2nd edition, 2002

[4] Douglas L. Perry, "VHDL programming by example," Tata McGraw Hills publisher, Fourth edition, 2002

[5] Wayne Wolf, "FPGA- Based System Design," Pearson education, First edition, 2005

[6] David Goldberg, "What Every Scientist should know About Floating-point Arithmetic," ACM computing surveys, vol. 23, no. 1, pp. 171-264, March 1991

[7] Loucas Louca, Todd A. Cook and William H. Johnson, " Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs, " IEEE Symposium on FPGAs for Custom Computing Machines, pp. 107 – 116, 17-19 April 1996

[8] N. Shirazi, A. Walters and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," IEEE Symposium on FPGAs for Custom Computing Machines, pp. 155 – 162, 19-21 April 1995

[9] Jian Liang, Russell Tessier and Oskar Mencer, "Floating Point Unit Generation and Evaluation for FPGAs," IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 185- 194, 9-11 April 2003

[10] Jean- Pierre Deschamps, Jean Antoine Bioul and Gustavo D. Sutter, "Synthesis of arithmetic circuits – FPGA, ASIC and Embedded System," John Wiley & Sons, Inc., publication, 2006

[11] M. Morris Mano, "Computer System Architecture," Pearson Education Inc., 3rd edition, 2002

[12] J.N. Coleman, E.I. Chester, "A 32-Bit Logarithmic Arithmetic Unit and its Performance Compared to Floating-Point," 14th IEEE Symposium on Computer Arithmetic (ARITH-14 '99), pp. 142-151, 14-16 April 1999

[13] Michael Haselman, Michael Beauchamp, Aaron Wood, Scott Hauck, Keith Underwood and K. Scott Hemmert, "A Comparison of Floating Point and Logarithmic Number

Systems on FPGAs," 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp 180-190, 18-20 April 2005

[14] Y. Wan and C.L. Wey, "Efficient algorithms for binary logarithmic conversion and addition," IEEE International Symposium on Circuits and Systems, vol. 5, pp 233 – 236, 31 May-3 June 1998

[15] Sheng-Chieh Huang, Liang-Gee Chen and Thou-Ho Chen, "The chip design of a 32-b logarithmic number system," IEEE International Symposium on Circuits and Systems, vol. 4, pp 167-170, 30 May-2 June 1994

[16] D.M. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator," Proc. 11th IEEE Symposium Computer Arithmetic, pp. 2-9, 29 June-2 July 1993

[17] D. M. Lewis, "Interleaved Memory Function Interpolators with Applications to an Accurate LNS Arithmetic Unit," IEEE Transactions on Computers, vol. 43, no. 8, pp. 974-982, August 1994

[18] D. M. Lewis, "An architecture for addition and subtraction of long word length numbers in the logarithmic number system", IEEE Transactions on Computers, vol. 32, no. 11, pp. 1325-1336, November 1990

[19] J. Detrey, F. Dinechin, "A VHDL Library of LNS Operators", Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, vol. 2, pp. 2227 – 2231, 9-12 November 2003

[20] S. Collange, J. Detrey and F. de Dinechin, "Floating Point or LNS: Choosing the Right Arithmetic on an Application Basis," 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, pp. 197-203, 30 August- 1 September 2006

[21] B.R. Lee, N. Burgess, "A parallel look-up logarithmic number system addition/subtraction scheme for FPGA", IEEE International Conference on Field-Programmable Technology (FPT), pp. 76 – 83, 15-17 December 2003

[22] Demetrios K. Kostopoulos, "An Algorithm for the Computation of Binary Logarithms," IEEE transactions on Computers, vol. 40, no. 11, pp. 1267-1270, November 1991

[23] J. Detrey and F. de Dinechin, "A parameterized floating-point exponential function for FPGAs," IEEE International Conference on Field-Programmable Technology, pp. 27 – 34, 11-14 December 2005

[24] J.C. Majithia, D. Levan, "A note on base-2 logarithm computations," Proceedings of the IEEE, vol. 61, no. 10, pp. 1519-1520, October 1973

[25] Lawrence K. Yu and David M. Lewis, "A 30-b Integrated Logarithmic Number System processor," IEEE journal of Solid-state Circuits, vol. 26, no. 10, pp. 1433-1440, October 1991

[26] F. J. Taylor, "A 20 Bit Logarithmic Number System Processor," IEEE Transactions on Computers, vol. 37, no. 2, pp. 190-200, February 1988

[27] P.T.P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," 10th IEEE Symposium on Computer Arithmetic, pp. 232-236, 26-28 Jun 1991