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

Amit Kumar[1], Saxena .A.K[2], Dasgupta .S[3]
Solid State Device &VLSI Technology Group , Dept. of Electronics and Computer Engineering,
Indian Institute of Technology, Roorkee, Uttrakhand, India
E-mail : [1] iitr.amitkumar@gmail.com, [2] kumarfec@iitr.ernet.in, [3] sudebfec@iitr.ernet.in

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 equal to FP. LNS addition and subtraction requires interpolation technique for which some vales are stored in read only memory (ROM). In this paper, different sizes of ROM are used for addition and subtraction and their performances are compared to the floating point.

Key words: FPGA, Logarithmic Number Systems, ROM.

I. INTRODUCTION

The floating point [1] and logarithmic number system [2,3] are the arithmetic number systems used for representing real numbers in computer and digital hardware. Most of the implementations use single (32-bit) or double (64-bit) precision for representing floating point and LNS.

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 have advantages that multiplication and division in LNS are simplified to fixed-point addition and subtraction. But thedisadvantage of LNS is that addition and subtraction are very difficult to perform in hardware descriptive language (HDL) [4] and the accuracy depends upon the size of only memory (ROM). In this paper, arithmetic operations (addition, subtraction, multiplication and division) for FP and LNS are implemented in HDL and synthesis results for both are compared to find which number system will suit better for field programmable logic array (FPGA) [5] real number representation.

II. NUMBER SYSTEMS

A. Floating Point

The IEEE introduced a standard IEEE 754 [1] to define floating-point representation and arithmetic. The single precision format uses 1 sign bit (S), 8 bits biased exponent bits (E), and 23 bits mantissa (F). The mantissa part has binary point to the left, and a hidden '1' to the left of the point. The storage layout for single-precision is shown below:

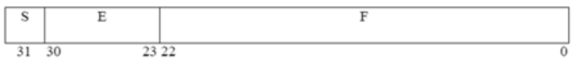


Fig. 1. Floating Point Format.

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

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

where $F = (b_{22}+b_{21} + \ldots\ldots\ldots\ldots +b_i + \ldots\ldots + b_{20})$.

b_i = 1 or 0.

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

E = biased exponent.

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

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.

B. Logarithmic Number System [2,3]

The format of the logarithm number system is

$$A = (-1)^{SA} * 2^{EA}. \quad (2)$$

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

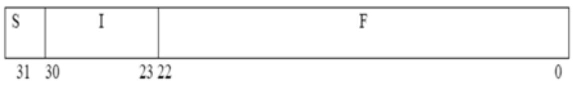


Fig. 2. Logarithmic Number System Format

The real numbers represented by this format are in the range ± 2^{-128} to 2^{-128} i.e. ± 2.9E-39 to 3.4E+38.

III. FLOATING POINT ARITHMETIC UNIT

The Floating point arithmetic unit has four operations : 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

$$A = (-1)^{SA} * 2^{EA-127} * (1.M_A).$$

$$B = (-1)^{SB} * 2^{EB-127} * (1.M_B).$$

$$Z = (-1)^{SZ} * 2^{EZ-127} * (1.M_Z). \qquad (3)$$

A. FP Addition Algorithm [6,7,8]

Floating point addition involves the following steps:

1. Separate the sign, exponent and mantissa bits of the both operands
2. Compare |A| and |B|. If |B| is greater than |A|, then swap A and B.
3. Set the exponent of result $E_Z E_A S_Z S_A$.
4. Compute the difference d = E_B - E_A. 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. Normalization step : If carry is generated in step 5, shift ($1.M_Z$) right by one and increase the exponent E_Z by one.
7. Check resultant exponent for overflow / underflow :

   If E_Z is larger than maximum emponent allowed, then set the overflow flag.

   If E_Z is Smaller than minimum exponent allowed, then set the underflow flag.
8. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard.

B. FP Subtraction Algorithm [6,7,8]

Floating point subtraction involves the following steps:

1. Separate the sign, exponent and mantissa bits of the both operands.
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_B - E_A. 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 ($1.M_B$) and ($1.M_A$).
6. Normalization step : If carry is generated in step 5, shift ($1.M_Z$) right by one and increase the exponent E_Z by one.
7. Check resultant exponent for overflow / underflow :

   If E_Z is larger than maximum exponent allowed, then set the overflow flag.

   If E_Z is Smaller than minimum exponent allowed, then set the underflow flag.
8. Pack the sign bit, exponent bits and mantissa bits according to the IEEE 754 floating point standard.

C. FP multiplication algorithm [6,7,9]

Floating point multiplication involves the following steps:

1. Separate the sign, exponent and mantissa bits of the both operands
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. Calculate the mantissa of the result [10]:

   Multiply the mantissas : ($1.M_A$) * ($1.M_B$) The calculated mantissa will be in the 48 bits.
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 exponent allowed, then set the overflow flag.

   If E_Z is larger than minimum exponent allowed, 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.

D. FP division algorithm [6,7,9]

Floating point division involves the following steps:

1. Separate the sign, exponent and mantissa bits of the both operands
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. Calculate the mantissa of the result [10]:

   Multiply 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 resultant exponent for overflow/underflow:

   If E_Z is larger than maximum exponent allowed, then set the overflow flag.

   If E_Z is larger than minimum exponent allowed, 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.

IV. LNS 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

$$A = (-1)^{SA} * 2^{EA}.$$

$$B = (-1)^{SB} * 2^{EB}.$$

$$Z = (-1)^{SZ} * 2^{EZ}. \qquad (4)$$

A. LNS Addition

1) LNS Addition Algorithm 1 [11,12,13,14]:

1. Separate the sign and fixed point exponent bits of both operands.
2. Generate the ROM values for the function f(d)= log2 $(1+^{2-d})$ 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 $(1 + 2^{-x})$ [15,16].
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 LNS format.

2) LNS Addition Algorithm 2 [17]:

1. Separate the sign and fixed point exponent bits of both operands.
2. Generate the ROM values for the function f (d) = $\sqrt[i]{2}$ where i = 2, 4, 8, 16,………..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.

7. Calculate the value of 2^{-x} 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 $= 2^{-x} = 2^{-1.F}$

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

   8.1 Add 1 to m and store the result in variable rega.

   8.2 Initialize the variable of type std_logic_vector with name regb.

   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 $\mathrm{Log}_2(1 + m)$ to get the result exponent E_Z.

10. Check for overflow/underflow.
11. Assemble the result in to 32 bit.

B. LNS Subtraction

1) LNS Subtraction Algorithm 1 [11,12,14,18]:

1. Separate the sign and fixed point exponent bits of both operands.
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 $\log_2 (1-2^{-X})$ [15,16].
7. Add E_A and $\log_2 (1-2^{-d})$ to get the result exponent E_Z.
8. Check for overflow/underflow.
9. Assemble the result in to 32 bit LNS format.

2) LNS Subtraction Algorithm 2 [17]:

1. Separate the sign and fixed point exponent bits of both operands.
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, 32…..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).

7. Calculate the value of 2^{-x} following the same steps as in step number 7 of LNS addition algorithm 2.
8. Calculate the value of log (1 - m) following the same steps as in step number 8 of LNS addition algorithm 2.
9. Add E_A and (1 - *m*) to get the result exponent E_Z.
10. Check for overflow/underflow.
11. Assemble the result in to 32 bit LNS format.

C. LNS Multiplication Algorithm [11,18]

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

D. LNS Division Algorithm [11,18]

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

V. RESULTS

Tables I – IV show the synthesis results for different FP and LNS operations. Tables V & VI show the variation of accuracy for LNS addition and subtraction as the size of ROM varies. In tables V and VI, all results are shown in decimal number format after conversion from LNS for easy comparison.

Table I. 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	476	841	53.77	76740
LNS	Algorithm 1	92	2959	3904	224.52	136196
		184	3173	4302	224.68	143364
	Algorithm 2	46	26691	43902	2998.07	1111860

Table II. 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	477	851	54.14	77828
LNS	Algorithm 1	184	6894	9096	219.29	169988
		192	6720	9007	223.84	168964
		208	6632	9007	224.53	167940
		240	6714	9189	224.93	171012
		304	6791	9462	223.63	174084
		432	7447	10577	222.87	188804
	Algorithm 2	46	27519	45263	3047.80	1137524

Table III. 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	672	1232	81.14	78852
LNS	0	56	64	37.37	64452

Table IV. 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	646	1197	171.36	76804
LNS	0	71	64	37.42	65404

Table V. LNS Addition Examples

A	B	Exact result	Algorithm 1		Algorithm 2
			Size of ROM		
			92	184	
57.88618	16.92117	74.80735	74.80672	74.80718	74.80734
54.24420	48.25241	102.49661	108.48840	102.49658	102.49658
57.88618	48.25241	106.13859	106.13848	106.13855	106.13855

Table VI. LNS Subtraction Examples

A	B	Exact result	Algorithm 1						Algorithm 2
			Size of ROM						
			184	192	208	240	304	432	
57.88618	16.92117	40.96501	40.96594	40.96594	40.96594	40.96594	40.96594	40.96594	40.96500
54.24420	48.25241	5.99179	7.44002	6.07557	5.99701	5.99233	5.99188	5.99180	5.99180
57.88618	48.25241	9.63377	9.67958	9.64351	9.63584	9.63399	9.63381	9.63378	9.63378

VI. CONCLUSION

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 infinity as x varies from 0.5 to 0. These values of x do not occur during addition. So more values are stored in ROM for x variation between 0.5 and 0. This is the reason that in LNS subtraction while increasing theROM 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 V, 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 and have good accuracy, but they require so much number of FPGA slices that their FPGA implementation is of no use. So the final choice left is to use the floating point representation to represent the large values of real numbers.

ACKNOWLEDGMENT

This work was supported by Special Man Power Development in VLSI & Related Softwares, Phase-II (SMDP-II), Ministry of Information Technology, and Government of India.

REFERENCES

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

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

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

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

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

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

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

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

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

[10] M. Morris Mano, 2002. “Computer System Architecture,” Pearson Education Inc., 3rd edition.

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

[12] Michael Haselman, Michael Beauchamp, Aaron Wood, Scott Hauck, Keith Underwood and K. Scott Hemmert April 2005. “A Comparison of Floating Point and Logarithmic Number Systems on FPGAs,” 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp 180-190.

[13] Y. Wan and C.L. Wey, May 1999. “Efficient algorithms for binary logarithmic conversion and addition,” IEE Proceedings on Computers and Digital Techniques, vol. 146, no. 3, pp 168 – 172.

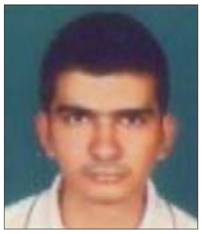
[14] Sheng-Chieh Huang, Liang-Gee Chen and Thou-Ho Chen June 1994. “The chip design of a 32-b logarithmic number system,” IEEE International Symposium on Circuits and Systems, vol. 4, pp 167-170.

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

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

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

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



Amit Kumar received B.Tech degree in Electronics and Instrumentation Control Engg. from YMCA Institute of Engineering in 2006. He is currently pursuing M. Tech degree at Indian Institute of Technology Roorkee in specialisation Semiconductor Devices and VLSI Technology. He worked on the Floating Point and Logarithmic Number system for M. Tech Dissertation.

Dr. Saxena obtained Ph.D. from Department of Electronics and Electrical Engg.,UMIST/Sheffield University (UK) in 1975 and 1978, respectively as one of the two Government of India National Scholars.He is a Professor in Solid State Electronics and VLSI Technology in IIT - Roorkee. The discovery of a level in GaAlAs is christened as 'Saxena's Deep Donor' by Philips Research Laboratory, Eindhoven (Netherlands). He is also a winner of INSA Young Scientist, Roorkee University Khosla Award Gold Medal, Kothari Scientific Research Institute Award, S. K. Mitra Memorial Awards (twice) of IETE and Bharat Excellence Award. He has published about 175 research papers in international journals and conference proceedings with very high citation index of about 775 so far..Dr. Saxena has supervised many Ph.D./M.E./M.Tech./M.Phil. theses in the area of VLSI design, metal-semiconductor ohmic and non-ohmic contacts, band structure and deep energy levels of GaAs, GaAlAs, GaP, InP, etc and quantum wells under pressure. He has also written AICTE sponsored nine volumes on the related subjects for working professionals.