Implementation of a High Speed Single Precision Floating Point Unit using Verilog

Ushasree G
ECE-VLSI, VIT University,
Vellore- 632014, Tamil Nadu, India

R Dhanabal
ECE-VLSI, VIT University,
Vellore- 632014, Tamil Nadu, India

Sarat Kumar Sahoo
ECE-VLSI, VIT University,
Vellore- 632014, Tamil Nadu, India

ABSTRACT
To represent very large or small values, large range is required as the integer representation is no longer appropriate. These values can be represented using the IEEE-754 standard based floating point representation. This paper presents high speed ASIC implementation of a floating point arithmetic unit which can perform addition, subtraction, multiplication, division functions on 32-bit operands that use the IEEE 754-2008 standard. Pre-normalization unit and post normalization units are also discussed along with exceptional handling. All the functions are built by feasible efficient algorithms with several changes incorporated that can improve overall latency, and if pipelined then higher throughput. The algorithms are modeled in Verilog HDL and have been implemented in ModelSim.

Keywords
Floating point number, normalization, exceptions, latency, etc.

1. INTRODUCTION
An arithmetic circuit which performs digital arithmetic operations has many applications in digital coprocessors, application specific circuits, etc. Because of the advancements in the VLSI technology, many complex algorithms that appeared impractical to put into practice, have become easily realizable today with desired performance parameters so that new designs can be incorporated [2]. The standardized methods to represent floating point numbers have been instituted by the IEEE 754 standard through which the floating point operations can be carried out efficiently with modest storage requirements,.

The three basic components in IEEE 754 standard floating point numbers are the sign, the exponent, and the mantissa [3]. The sign bit is of 1 bit where 0 refers to positive number and 1 refers to negative number [3]. The mantissa, also called significand which is of 23bits composes of the fraction and a leading digit which represents the precision bits of the number [3] [2]. The exponent with 8 bits represents both positive and negative exponents. A bias of 127 is added to the exponent to get the stored exponent [2]. Table 1 show the bit ranges for single (32-bit) and double (64-bit) precision floating-point values [2].

The value of binary floating point representation is as follows where S is sign bit, F is fraction bit and E is exponent field.

Value of a floating point number= $(-1)^S$ x val(F) x $2^{val(E)}$

Table 1. Bit Range for Single (32-bit) and Double (64-bit) Precision Floating-Point Values [2]

	Sign	Exponent	Fraction	Bias
Single precision	1[31]	8[30-23]	23[22-00]	127
Double precision	1[63]	11[62-52]	52[51-00]	1023

There are four types of exceptions that arise during floating point operations. The Overflow exception is raised whenever the result cannot be represented as a finite value in the precision format of the destination [13]. The Underflow exception occurs when an intermediate result is too small to be calculated accurately, or if the operation's result rounded to the destination precision is too small to be normalized [13] The Division by zero exception arises when a finite nonzero number is divided by zero [13]. The Invalid operation exception is raised if the given operands are invalid for the operation to be performed [13].In this paper ASIC implementation of a high speed FPU has been carried out using efficient addition, subtraction, multiplication, division algorithms. Section II depicts the architecture of the floating point unit and methodology, to carry out the arithmetic operations. Section III presents the arithmetic operations that use efficient algorithms with some modifications to improve latency. Section IV presents the results that have been simulated in ModelSim. Section V presents the conclusion.

2. ARCHITECTURE AND METHODOLOGY
The FPU of a single precision floating point unit that performs add, subtract, multiply, divide functions is shown in figure 1 [1]. Two pre-normalization units for addition/subtraction and multiplication/division operations has been given[1]. Post normalization unit also has been given that normalizes the mantissa part[2]. The final result can be obtained after post-normalization. To carry out the arithmetic operations, two IEEE-754 format single precision operands are considered. Pre-normalization of the operands is done. Then the selected operation is performed followed by post-normalizing the output obtained .Finally the exceptions occurred are detected and handled using exceptional handling. The executed operation depends on a two bit control signal (z) which will determine the arithmetic operation is shown in table 2.

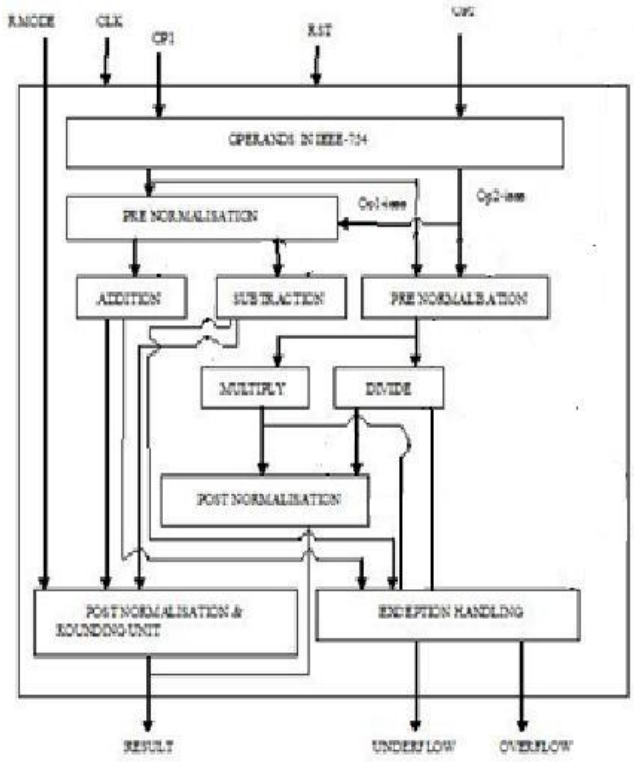

Fig 1: Block diagram of floating point arithmetic unit [1]

Table 2. Floating Point Unit Operations

z(control signal)	Operation
2'b00	Addition
2'b01	Subtraction
2'b10	Multiplication
2'b11	Division

3. 32 BIT FLOATING POINT ARITHMETIC UNIT

3.1 Addition Unit

One of the most complex operation in a floating-point unit comparing to other functions which provides major delay and also considerable area. Many algorithms has been developed which focused to reduce the overall latency in order to improve performance. The floating point addition operation is carried out by first checking the zeros, then aligning the significand, followed by adding the two significands using an efficient architecture. The obtained result is normalized and is checked for exceptions. To add the mantissas, a high speed carry look ahead has been used to obtain high speed. Traditional carry look ahead adder is constructed using AND, XOR and NOT gates. The implemented modified carry look ahead adder uses only NAND and NOT gates which decreases the cost of carry look ahead adder and also enhances its speed also [4].

The 16 bit modified carry look ahead adder is shown in figure 2 and the metamorphosis of partial full adder is shown in figure 3 using which a 24 bit carry look ahead adder has been constructed and performed the addition operation.

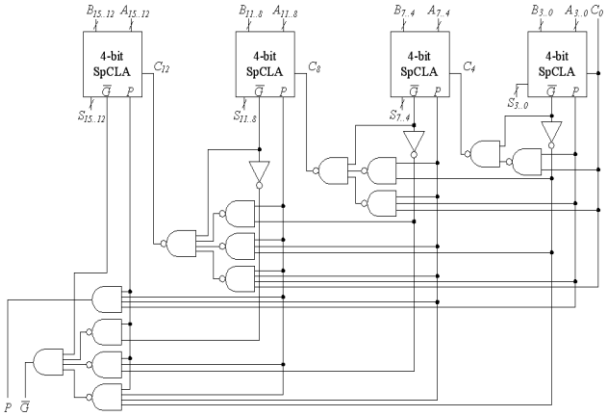

Fig 2: 16 bit modified carry look ahead adder [4]

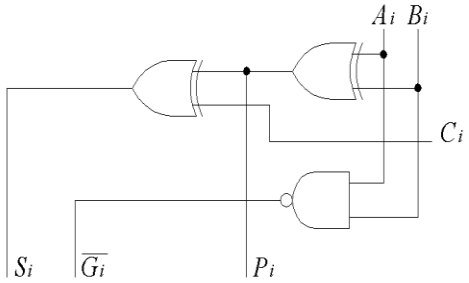

Fig 3: Metamorphosis of partial full adder [4]

3.2 Subtraction Unit

Subtraction operation is is implemented by taking 2's complement of second operand. Similar to addition operation, subtraction consists of three major operations, pre normalization, addition of mantissas, post normalization and exceptional handling[4]. Addition of mantissas is carried out using the 24 bit modified carry look ahead adder .

3.3 Multiplication

Constructing an efficient multiplication module is a iterative process and 2n-digit product is obtained from the product of two n-digit operands. In IEEE 754 floating-point multiplication, the two mantissas are multiplied, and the two exponents are added. Here first the exponents are added from which the exponent bias (127) is removed. Then mantissas have been multiplied using feasible algorithm and the output sign bit is determined by exoring the two input sign bits. The obtained result has been normalized and checked for exceptions.

To multiply the mantissas Bit Pair Recoding (or Modified Booth Encoding) algorithm has been used, because of which the number of partial products get reduces by about a factor of two, with no requirement of pre-addition to produce the partial products. It recodes the bits by considering three bits at a time. Bit Pair Recoding algorithm increases the efficiency of multiplication by pairing. To further increase the efficiency of the algorithm and decrease the time complexity, Karatsuba algorithm can be paired with the bit pair recoding algorithm.

One of the fastest multiplication algorithm is Karatsuba algorithm which reduces the multiplication of two n-digit numbers to 3nlog32 ~ 3n1.585 single-digit multiplications and therefore faster than the classical algorithm, which requires n2 single-digit products [11]. It allows to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or

y, with some additions and digit shifts instead of four multiplications[11]. The steps are carried out as follows

Let x and y be represented as n-digit numbers with base B and m<n.

$$x = x1Bm + x0$$

$$y = y1Bm + y0$$

where x0 and y0 are less than Bm [11]. The product is then xy = (x1Bm + x0)(y1Bm + y0)= c1B2m + b1Bm + a1

Where c1 = x1y1

$$b1 = x1y0+ x0y1$$

$$a1 = x0y0.$$

$$b1 = p1- z2 - z0$$

$$p1 = (x1 + x0)(y1 + y0)$$

Here c1, a1, p1 has been calculated using bit pair recoding algorithm. Radix-4 modified booth encoding has been used which allows for the reduction of partial product array by half [n/2]. The bit pair recoding table is shown in table 3. In the implemented algorithm for each group of three bits (y2iþ1, y2i, y2i_1) of multiplier, one partial product row is generated according to the encoding in table 3. Radix-4 modified booth encoding signals and their respective partial products has been generated using the figures 4 and 5. For each partial product row, figure 4 generates the one, two, and neg signals. These values are then given to the logic in figure 5 with the bits of the multiplicand, to produce the whole partial product array. To prevent the sign extension the obtained partial products are extended as shown in figure 6 and the the product has been calculated using carry save select adder.

Table 3. Bit-Pair Recoding [11]

BIT PATTERN				OPERATION
0	0	0	NO OPERATION	
0	0	1	1xa	prod=prod+a;
0	1	0	2xa-a	prod=prod+a;
0	1	1	2xa	prod=prod+2a;
1	0	0	-2xa	prod=prod-2a;
1	0	1	-2xa+a	prod=prod-a;
1	1	0	-1xa	prod=prod-a;
1	1	1	NO OPERATION	

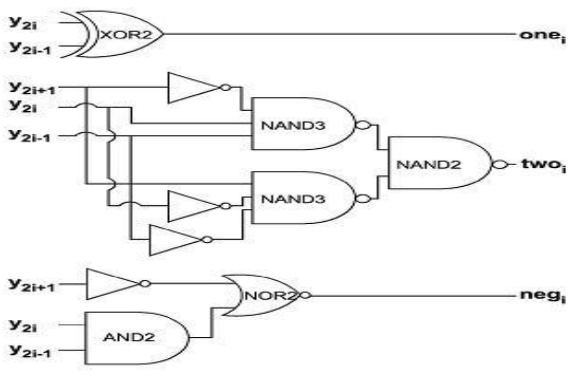

Fig 4: MBE Signal Generation [10]

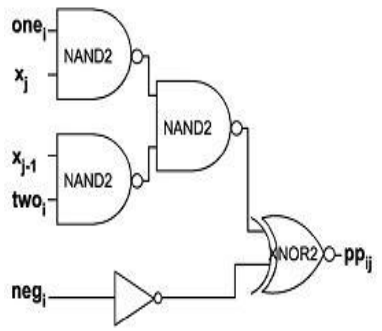

Fig 5: partial product generation [10]

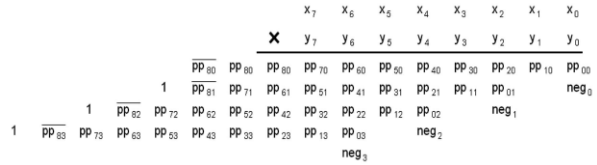

Fig 6: Sign Prevention Extension of Partial Products [10]

3.4 Division Algorithm

Division is the one of the complex and time-consuming operation of the four basic arithmetic operations. Division operation has two components as its result i.e. quotient and a remainder when two inputs, a dividend and a divisor are given. Here the exponent of result has been calculated by using the equation, e0 = eA – eB + bias (127) -zA + zB followed by division of fractional bits [5] [6]. Sign of result has been calculated from exoring sign of two operands. Then the obtained quotient has been normalized [5] [6].

Division of the fractional bits has been performed by using non restoring division algorithm which is modified to improve the delay. The non-restoring division algorithm is the fastest among the digit recurrence division methods [5] [6]. Generally restoring division require two additions for each iteration if the temporary partial remainder is less than zero and this results in making the worst case delay longer[5] [6]. To decrease the delay during division, the non-restoring division algorithm was introduced which is shown in figure 7. Non-restoring division has a different quotient set i.e it has one and negative one, while restoring division has zero and one as the quotient set[5] [6] Using the different quotient set, reduces the delay of non-restoring division compared to restoring division. It means, it only performs one addition per iteration which improves its arithmetic performance[6].

The delay of the multiplexer for selecting the quotient digit and determining the way to calculate the partial remainder can be reduced through rearranging the order of the computations. In the implemented design the adder for calculating the partial remainder and the multiplexer has been performed at the same time, so that the multiplexer delay can be ignored since the adder delay is generally longer than the multiplexer delay. Second, one adder and one inverter are removed by using a new quotient digit converter. So, the delay from one adder and one inverter connected in series will be eliminated.

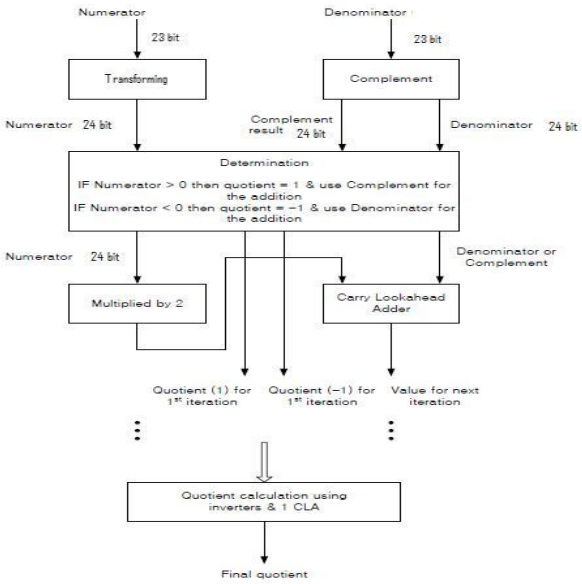

Fig 7: Non Restoring Division algorithm

4. RESULTS

4.1 Addition Unit

The single precision addition operation has been implementation in modelsim for the inputs, input1=25.0 and input2=4.5 which is input2=4.5 shown in figure 8 for which result has been obtained as 29.5

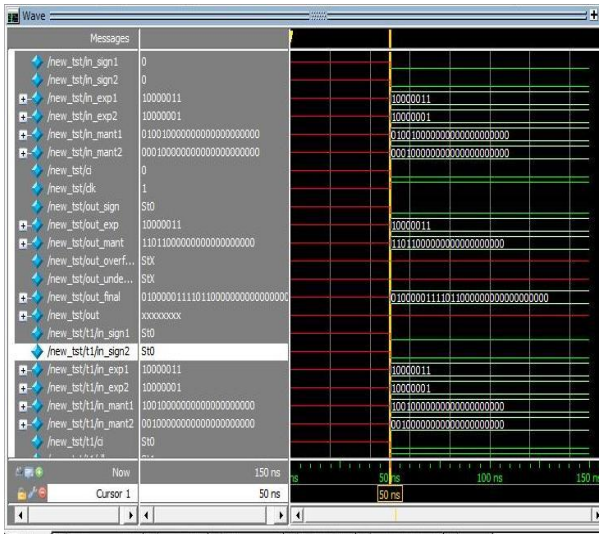

Fig 8: Implementation of 32 bit Addition operation

4.2 Subtraction Unit

The single precision addition operation has been implementation in modelsim for the inputs, input1=25.0 and input2=4.5 which is shown in figure 9 for which result has been obtained as 20.5.

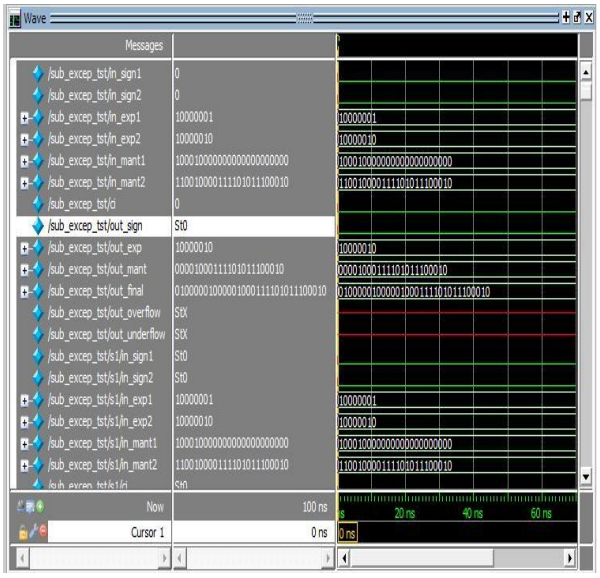

Fig 9: Implementation of 32 bit Subtraction operation

4.3 Multiplication Unit

The single precision multiplication operation has been implementation in modelsim is shown in figure 10. For inputs in_sign1=1'b0,in_sign2=1'b0;in_exp1=8'b10000011,in_exp2 =8'b10000010,in_mant1=23'b00100,in_mant2=23'b001100 and the output obtained is out_sign=1'b0;out_exp=8'd131; ,out_mant=23'b00101011.

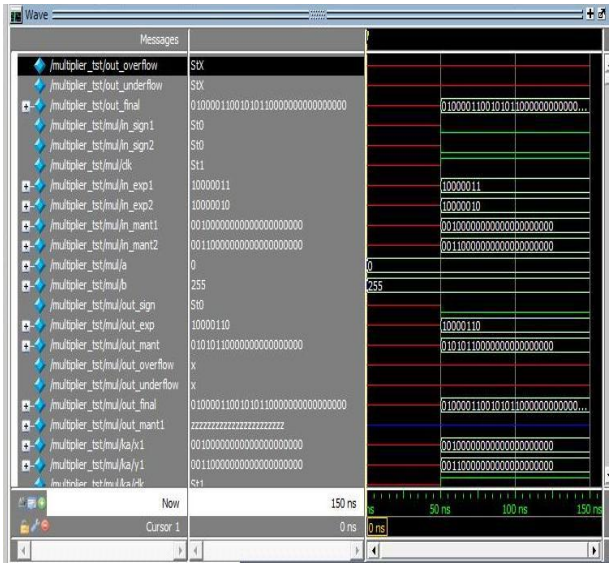

Fig 10: Implementation of 32 bit Multiplication operation

4.4 Division Operation

The single precision division operation has been implementation in modelsim for the inputs, input1=32'd100 and input2=32'd36 which is shown in figure 11 for which quotient has been obtained as 23'd2 and the remainder as 23'd28.

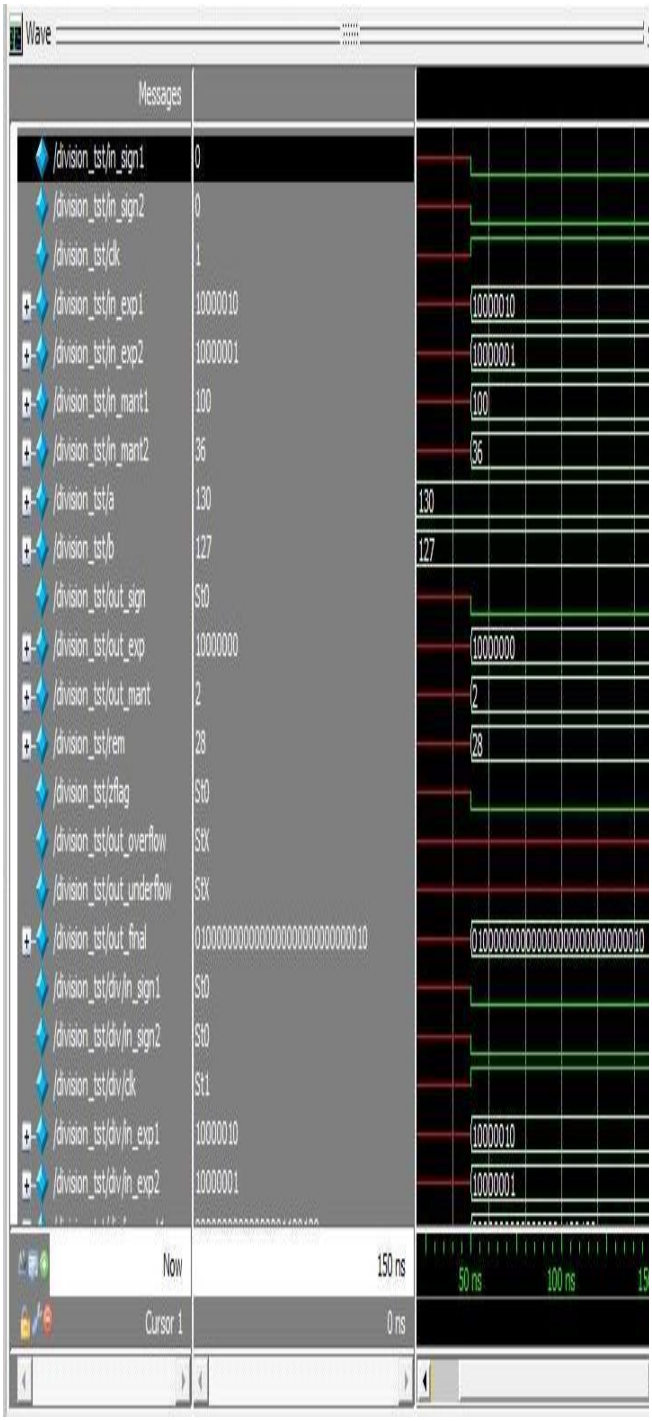

Fig 11: Implementation of 32 bit Multiplication operation

5. REFERENCES

[1] Rudolf Usselmann, "Open Floating Point Unit, The Free IP Cores Projects".

[2] Edvin Catovic, Revised by: Jan Andersson, "GRFPU – High Performance IEEE754 Floating Point Unit", Gaisler Research, Första Långatan 19, SE413 27 Göteborg, and Sweden.

[3] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, Vol 23, No 1, March 1991, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

[4] Yu-Ting Pai and Yu-Kumg Chen, "The Fastest Carry Lookahead Adder", Department of Electronic Engineering, Huafan University.

[5] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," IEEE Transactions on Computers, vol. 46, pp. 833–854, 1997.

[6] Milos D. Ercegovac and Tomas Lang, Division and Square Root: Digit-Recurrence Algorithms and Implementations, Boston: Kluwer Academic Publishers, 1994.

[7] ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[8] Behrooz Parhami, Computer Arithmetic - Algorithms and Hardware Designs, Oxford: Oxford University Press, 2000.

[9] Steven Smith, (2003), Digital Signal Processing-A Practical guide for Engineers and Scientists, 3rd Edition, Elsevier Science, USA.

[10] D. J. Bernstein. Multidigit Multiplication for Mathematicians. Advances in Applied Mathemat-ics, to appear

[11] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. Soviet Physics- Doklady, 7 (1963), 595-596.

[12] D. E. Knuth. The Art of Computer Programming. Volume 2: Seminumerical Algorithms.Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

Yamin Li and Wanming Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs", Proc of FCCM'97, IEEE Symposium on FPGAs for Custom Computing Machines, April 16 – 18, 1997, Napa, California, USA, pp.226-232.