Double Precision Floating Point Square Root Computation

Najib Ghatte#1, Shilpa Patil#2, Deepak Bhoir#3

Fr. Conceicao Rodrigues College of Engineering
Fr. Agnel Ashram,
Bandstand, Bandra (W), Mumbai: 400 050, India

***Abstract*— Square Root operation has found its prominence in many digital signal processing but it is very elusive to implement on FPGA due to its complicated computations. Many iterative algorithms which include restoring and non-restoring algorithms, SRT were proposed. Most of them implement with slow or large components which are less suitable for real-time applications than the addition or multiply components.**
This paper deals with the novel algorithm of square root computation of double precision floating point division. Verilog Code is written and implemented on Virtex-5 FPGA series.

Keywords— **Double precision, Binary square root, Vedic, Virtex, FPGA, Dvanda, IEEE-754.**

I. INTRODUCTION

The term floating point implicates that there is no fixed number of digits before and after the decimal point; i.e. the decimal point can float. Floating-point representations are slower and less accurate than fixed-point representations, but can handle a larger range of numbers.[1] Because mathematics with floating-point numbers requires a great deal of computing power, many microprocessors come with a chip, called a floating point unit (FPU), specialized for performing floating-point arithmetic. FPUs are also called math coprocessors and numeric co-processors. Floating-point representation has a complex encoding scheme with three basic components: mantissa, exponent and sign. Usage of binary numeration and powers of 2 resulted in floating point numbers being represented as single precision (32-bit) and double precision (64-bit) floating-point numbers. Both single and double precision numbers as illustrated in Fig. 1 are defined by the IEEE 754 standard.

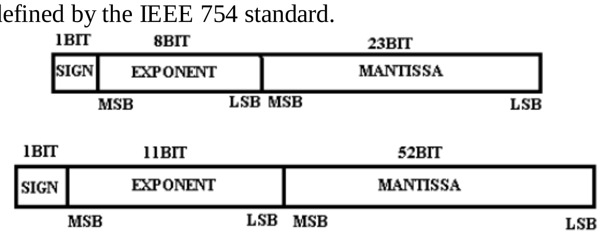

Fig. 1 IEEE-754 Floating-point Representation Standards

For a single precision format, 8-bits are reserved for exponent thereby having a bias value of +127 and 23 bits are reserved for mantissa. When sign bit is 1, it indicates negative number and when it is 0, it argues as a positive number.

The similar explanation is extended for double precision format where exponents are biased to +1023.

Division and square root are important operators in many digital signal processing (DSP) applications including matrix inversion, vector normalization, and Cholesky decomposition. The floating-point divide and square root operators support many different floating-point formats including IEEE standard formats. Both modules demonstrate a good trade-off between area, latency and throughput. They are also fully pipelined to aid the designer in implementing fast, complex, and pipelined designs. [2] [3]

It is the most important goal of a designer to enhance the performance of the ALU thereby reducing its design complexity to have better figure of merit. Due to the latency gap between addition/multiplication and division/square root, the latter operations increasingly become performance bottlenecks. As the performance gap widens between addition/subtraction/multiplication and division/square root, many signal and image processing algorithms involving floating-point operations have been rewritten to avoid the use of division and square root. Moreover, it is difficult to implement square root on hardware. [4] Therefore, poor implementations of floating-point division and square root results in severe performance degradation.

Vedic math is known to more optimised and efficient than algorithms based on conventional logic. [5] The sutras defined can be used in digital design to improve the performance of ALUs based on conventional logic. *Dvanda Yoga Sutra* deals with the division [6]. These sutras find its limitations when number of bits are increased as this paper deals with IEEE-754 floating-point representation.

II. VARIOUS SQUARE ROOT ALGORITHMS

Researchers have proposed many algorithms and procedural architectures to carry out square root in order to reduce the computational time and thus enhancing the performance.

A. Radix-2 SRT Algorithm

Named for its creators (Sweeney, Robertson, and Tocher), SRT for radix-2 is an iterative method to compute square root of a number. Each iteration deals with left-shift and addition of a digit. The algorithm is rather complex especially for more precision. Also, it may generate a wrong resulting value.

B. Use of Look-up Table

Algorithms such as Newton-Raphson's method as depicted in Eq. 1, is used to carry out computation, which is pretty faster, but due to presence of single precision division makes it tedious and performance degrading. [7]

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \quad (1)$$

To get rid of complex computation of division, use of look-up table is incorporated. Taylor series/Maclaurin series expansion for square root is considered as formulated in Eq. 2.[7]

$$\sqrt{x} = \sqrt{x_0} + \frac{x - x_0}{2x_0} = \frac{\sqrt{x_0}}{2} + \frac{1}{2\sqrt{x_0}}x \quad (2)$$

Then, a range of possible values of fraction f (0 to ~1) is divided into n sub-ranges by using log_2^n bits of f as an index into a table which contains the first two co-efficients of the Taylor expansion of the square root of the mantissa (1.0 to ~2) over that the sub-range.

C. Vedic Approach : Dvanda Yoga Sutra

Vedic Sutra which is used as an alternative for simplified mathematical computation. Fig. 2 shows the computational steps to evaluate square root of 17689 which comes out as 133.

1. Group terms in a pair of 2 from right to left. Since it has 5 digits, 1st group will have 1 digit (1).
   Write down as shown besides.
2. Find the perfect square of leftmost number (1) which is 1.
   To left of 1st vertical line write 1 and add same to it. So add = 2.
   Below 1, write same square root (1) and carry forward the difference between above number and perfect square (i.e. 1 - 1 = 0)
3. Now divide underline 07 by obtained 2 = {Quotient = 3 and Remainder = 1}.
   Write Quotient (3) below and carry forward Remainder (1).
4. Calculate *Dvanda* of numbers present after 2nd vertical line D (3) = 9.
   Subtract above 9 from underlined 16 (=7).
   Divide 7 by 2 = {Quotient = 3 and Remainder = 1}.
   Write Quotient {3} below and carry forward Remainder (1).
5. Calculate *Dvanda* of numbers present after 2nd vertical line D (33) = 18.
   Subtract above 18 from underlined 18 = (0).
   Divide 0 by 2 = {Quotient = 0 and Remainder = 0}.

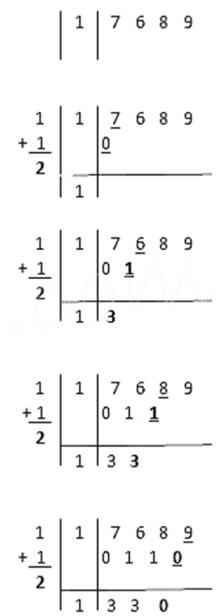

Fig. 2: Dvanda Yogi Sutra for Square Root: Stepwise Computation

Due to increased complexity with increase in the bits (52-bit mantissa), this sutra found some limitations with its implementation.

D. Paper and Pencil Method

It is one of the most orthodox and traditional method to compute square root [8].Fig. 3 illustrates the square root of 16 (10000_2) of which answer comes out as 4 (100_2).The stepwise algorithm for binary numbers is extended as follows:

1. Group the bits from right towards left. Since, it has 5 bits, the first group is of single bit (1).
2. Find the perfect square of leftmost number (1) which is 1.
   Write 1 in the quotient as well down below and add it to get 10.
3. Find the number suffixing 10 to get the product of it less than or equal to the partial difference and next group of bits.
4. Repeat the procedure till all the group of bits are done.

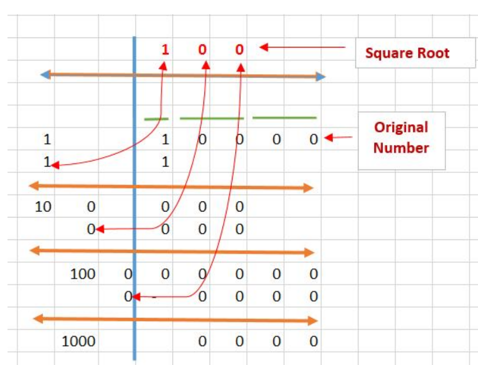

Fig. 3: Paper and Pencil Method for Square Root: Stepwise Computation

III. PROPOSED SQUARE ROOT ALGORITHM

This paper deals with the efficient algorithm which incorporates the positive attributes of the square root computational method. The square root algorithm find its limitations with the wide increase in the number of bits as this paper deals with the floating point representation where mantissa is of 52-bit wide.

The basic algorithm for the proposed design as follows:

1) Sign Bit

Sign bit of the result is same as the sign bit of the original number.

2) Exponent Computation

Exponent of the result depends on the biased exponent of the number. If biased exponent is odd, then 1023 is added to it and final sum is right shifted (divide by 2 operation).

$$E_r = \frac{E_a + 1023}{2}$$

If biased exponent is even, then 1022 is added and final sum is right shifted (divide by 2 operation). In addition, shift flag is set to indicate that the mantissa should be shifted to the left by 1 bit before computing its square root

$$E_r = \frac{E_a + 1022}{2}$$

3) Mantissa (Square Root) Evaluation

The block of code which carry out square-root computation is based on iterative approach where it deals with two registers namely temp and ANS of 56-bit wide and M_r of 55 bit wide.

Consider an example of evaluating square root of 16 (10000_2).

Initialize TEMP as 0000…0000 and ANS as 0100…0000. The iteration process is carried out.

On first iteration, TEMP is loaded with mantissa and then is compared with ANS, shift operation is carried out depending on the comparison results.

If it is greater or equal to ANS, the contents of TEMP are subtracted from those of ANS, shifted to the left, and stored in TEMP. The contents of ANS are shifted to the left by one bit starting from the current pointer position. Then, a 1 is inserted in the current bit position in ANS. Note that in each iteration, a pointer points to the current bit that will be replaced in ANS.

If TEMP is less than ANS, its contents are shifted to the left and stored in TEMP. The contents of ANS are shifted to the right by one bit starting from the current pointer position. Then, a 0 is inserted in the current bit position bit in ANS.

After the last iteration, the contents of ANS, with the exception of the two least significant bits are the bits which considered as final result.

Fig. 4 shows the architectural block diagram of the square root computation.

IV. DESIGN IMPLEMENTATION

Verilog HDL code for Square Root Computation of IEEE-754 Double Precision Numbers is being developed and then is simulated using ModelSim SE Plus 6.5.

Verilog HDL code was break down into modules which deals with the exponent computation (11-bit) and mantissa evaluation (52-bit).Top module connects all of them as shown in Fig. 5.

Various sets of inputs are fed to the top modular block to get the results. The further part of the document deals with simulation and synthesis results.

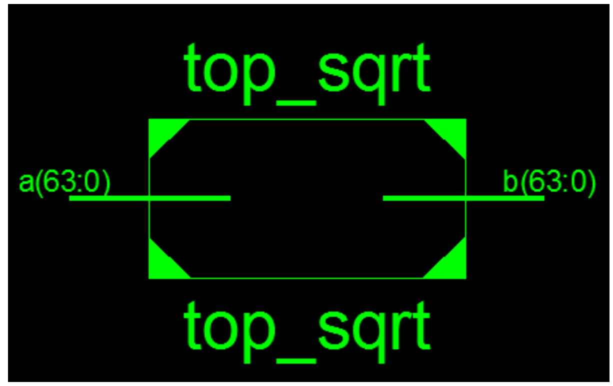

Fig. 5 Double Precision Square Root Computation: Block Diagram

A. ModelSim Simulation

Consider square root computation of a number, a = 16 (0 10000000011 0000000000000000000000000000000000000000000000000000) were fed to the algorithm to get the desired output as b = 4 (0 10000000001 0000000000000000000000000000000000000000000000000000) as shown in Fig. 6.

B. Xilinx ISE Synthesis

Verilog HDL Code for square root computation of IEEE-754 Double Precision (64-bit) numbers are then synthesized for device XC5VLX30 having package as FF324 of Virtex™-5 FPGA family. From the datasheet cited in [9], this device has following attributes manifests in Table I.

TABLE I
XILINX VIRTEX™-5 XC5VLX30 ATTRIBUTES

Device	CLB Array (One CLB = Four Slices × 2			Total Slices	Max. User I/O
	Rows	Column	Total		
xc5vlx30	80	30	4800	19,200	220

Table II shows the Device Utilisation Summary of the Verilog HDL code, so written, it is been observed that number of device parameters used are very less. Hence, an optimum Device Utilisation is obtained.

From the timing report obtained, it is found that the maximum combinational path delay is 129.684 ns. Maximum combinational path delay is only for paths that start at an input to the design and go to an output of the design without being clocked along the way.

TABLE II
FLOATING POINT SQUARE ROOT COMPUTATION (DOUBLE PRECISION):
DEVICE UTILISATION SUMMARY

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice LUTs	4,789	19,200	24%
Number used as logic	4,789	19,200	24%
Number using O6 output only	3,683		
Number using O5 output only	2		
Number using O5 and O6	1,104		
Number of occupied Slices	1,576	4,800	32%
Number of LUT Flip Flop pairs used	4,789		
Number with an unused Flip Flop	4,789	4,789	100%
Number with an unused LUT	0	4,789	0%
Number of fully used LUT-FF pair	0	4,789	0%
Number of slice register sites lost to control set restrictions	0	19,200	0%
Number of bonded IOBs	128	220	58%
Average Fanout of Non-Clock Nets	4.76		

V. PERFORMANCE ANALYSIS: VEDIC VERSUS CONVENTIONAL

It can be easily deduced from the Table III which shows the performance analysis of the proposed square root computation with that of conventional algorithm based square root computational block that proposed algorithm is more optimistic.

TABLE III
PERFORMANCE ANALYSIS:
PROPOSED WORK (VEDIC) VERSUS CONVENTIONAL

Parameters	This Work	[2]	[10]
Floating point Precision	52	52	52
Target Device	Virtex 5 XC5VLX30 -3 FF324	Virtex-II XC2V6000	Virtex-II XC2V6000
Number of Slices	1576	1572	405
Number of 4-input LUTs	4789	---	---
Number of IOBs	128	---	---
Estimated Time Delay (nanosecond)	129.684	165	532

The proposed algorithm is not able to optimise with the area utilisation with that of other algorithms. But it can be found out, that there is a significant improvement in the time required for computation which is nearly 126 nanoseconds compared to other algorithms, which suffer time delay of nearly 165 nanoseconds. Thus, there is nearly 23 % enhancement in the speed at which square root computation can be carried out.

VI. CONCLUSION

The importance and usefulness of floating point format nowadays does not allow any discussion. Any computer or electronic device, which operates with real numbers, implements this type of representation and operation.

Square Root is one of the most important arithmetic operation and difficult to implement in terms of hardware. Various architectures were proposed which include recursive iterations to increase the computational performance of the ALU.

The proposed design uses algorithm which makes the system more efficient as it enhances the speed at which device can operate. It is found that this work works nearly 23% faster than the prior algorithmic design where synthesis carried on Virtex – 5 platform.

Thus, proposed design is more efficient than traditional ALUs and serve as better optimisation technique as per today's need and wants.

REFERENCES

[1] Alex N. D. Zamfirescu, "Floating Point Type for Synthesis", CA USA, 2000.

[2] Xiaojun Wang, "Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms" Ph. D thesis, Department of Electrical and Computer Engineering, Northeastern University, December 2007.

[3] Tole Sutikno, Aiman Zakwan Jidin, Auzani Jidin,Nik Rumzi Nik Idris, "Simplified VHDL Coding of Modified Non-Restoring Square Root Calculator" in *International Journal of Reconfigurable and Embedded Systems*,Vol. 1, pp. 37-42, Mar. 2012.

[4] K. Piromsopa, C. Aporntewan, P. Chongsatitvatana, "An FPGA Implementation of a Fixed-Point Square Root Operation" Thailand, Feb. 2002.

[5] Deepa and Sanal's (2005) Vedic Mathematics [Online]. Available: http://www.sanalnair.org/articles/vedmath/intro.htm

[6] Rahul Bhangale (2012) Dvanda Yoga [Online]. Available: http://mathlearners.com/vedic-mathematics/squares/dvanda-yoga/

[7] Guillaume Bedard, Frederic Leblanc, Yohan Plourde, Pierre Marchand, "Fast Square Root Calculation" *in Mactech | The Journal of Apple Technology*, Vol. 14, pp. 1-5, 1998.

[8] Padala Nandeeswara Rao, "FPGA Implementation of Double Precision Floating Point Square Root with BIST Capability" M.Tech thesis, Department of Electronics and Communication Engineering, Thapar University, July 2009.

[9] *Virtex-5 Family Overview datasheet*, Xilinx 2009.

[10] Anuja Jayraj Thakkar, "Pipelining of Double Precision Floating Point Division and Square Root Operations on Field-Programmable Gate Arrays" M.Sc thesis, School of Electrical Engineering and Computer Science,University of Central Florida, 2006

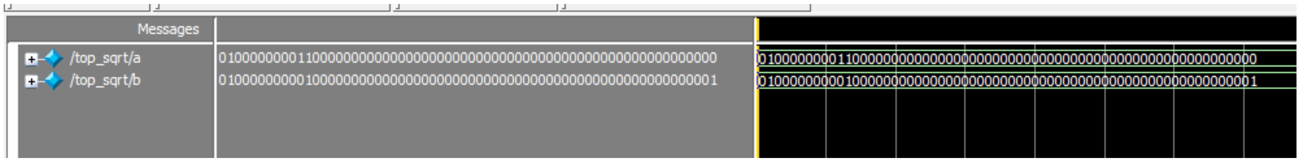

Fig. 6 Floating Point Square Root Computation (Double Precision): Timing Diagram

$\sqrt{16} = 04$

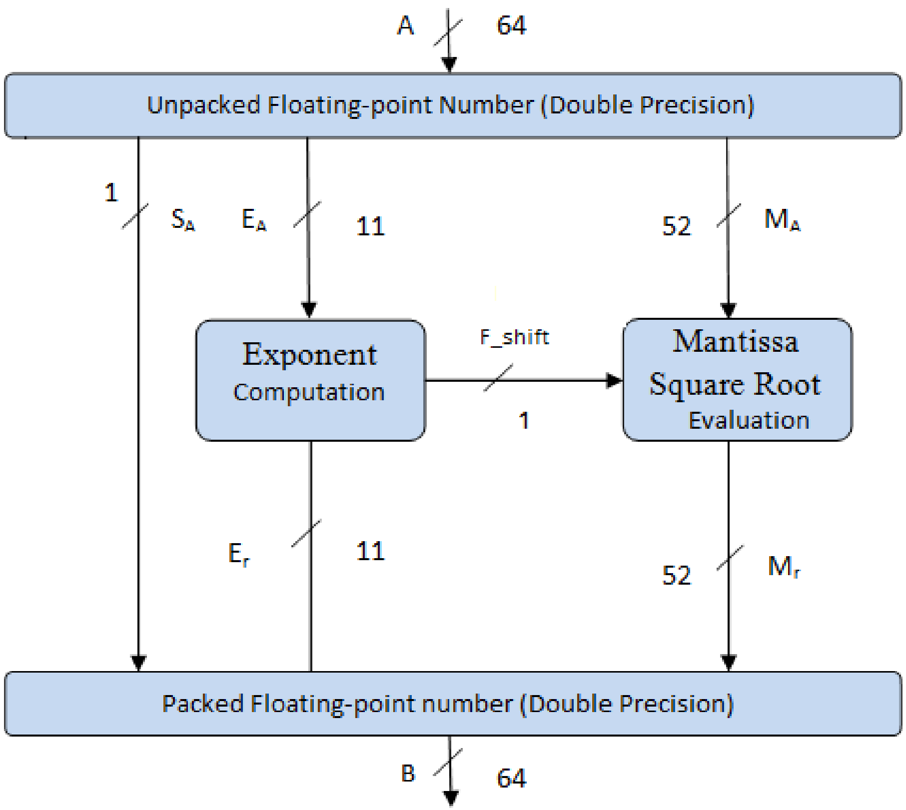

Fig. 4 Floating Point Square Root Computation (Double Precision): Architectural Block Diagram