FPGA IMPLEMENTATION OF A BINARY32 FLOATING POINT CUBE ROOT

Carlos Minchola Guardia
School of Engineering
Universidad Autónoma de Madrid
Madrid, Spain
e-mail: carlos.minchola@uam.es

Eduardo Boemo
School of Engineering
Universidad Autónoma de Madrid
Madrid, Spain
e-mail: eduardo.boemo@uam.es

***Abstract*—This paper presents the implementation of a sequential hardware core to compute a single floating point cube root compliant with the current IEEE 754-2008 standard. The design is based on Newton-Raphson recurrence, reciprocal and cube root units are implemented. Optimal performance requires two iterations for reciprocal and one for cube root units obtaining an accurate approximation of +/- 3 least significant bits. Our proposal is able to be performed up to 149 Mhz over Virtex5. The hardware cost occupies 230 *Slices* and 12 *Dsp48s* taking a latency of 19 clock cycles.**

Keywords— FPGA; cube root; reciprocal; IEEE 754-2008; binary floating point

I. INTRODUCTION

Nowadays, many scientific applications demand to compute a large numbers of arithmetic operations involving implementations of complex algorithms [9]. Every so often operations such as logarithm, cube root, reciprocal and trigonometric functions take part of several scientific formulas being commonly utilized in many applications as digital signal processing, multimedia, commercial and financial, mobile robot navigation and so on [16],[8],[10]. However, Floating Point Units are commonly implemented as software which is dramatically slow [3]. A way of improving the speed is to develop customized hardware versions of floating point operations, therefore this work proposes the implementation of a single binary floating point (BFP) cube root.

Currently, there are far fewer proposals with regard to cube root under FPGA technology. This grants to our proposal to capture the attention in future publications as far as this operation is concerned.

Recently, some publications have come up focused on the mentioned function under technology ASICs, as the proposal in [7] that presents a Radix-2-digit-by-digit architecture that outperforms the *area x delay* parameter in comparison with the proposal in [6] for high radices. Regarding FPGA technology, many mathematical methods have appeared for extracting cube roots as [11] that presents a decimal digit-recurrence scheme reducing the computational complexity, and the proposal in [5] that gives a survey of a digit-by-digit arithmetic method applicable to this function and suitable for FPGA implementations.

As seen, FPGA cube root implementations have not received a resounding attention regardless of the presence of various mathematical methods for extracting cube root. We hope this proposal can stimulate future FPGA implementations of the analyzed operation.

The outline of the paper is as follows. In the next Section, we describe the background information on binary floating-point. In section III, the general structure of our IEEE 754-2008 binary32 floating point cube root is explained. Section IV describes the generation of initial approximated solutions. In Section V, an outline of our general design is detailed. Section VI presents the architecture of our proposal based on Newton-Raphson iterations. Implementation and performance of our design is explained in Section VII. Finally, Section VIII briefly sums up some conclusions.

II. BINARY FLOATING POINT IN IEEE 754-2008

The IEEE 754-2008 [1] standard specifies formats for both BFP and decimal floating-point numbers. The primary difference between the two formats, besides the radix, is the normalization of the significands (also coefficient or mantissa). BFP significands are normalized with the radix point to the right of the most significant bit (MSB).

The IEEE 754-2008 standard specifies BFP formats of 32, and 64 bits. Our design tackles 32-bit BFP that is coded in the following three fields (Figure 1):

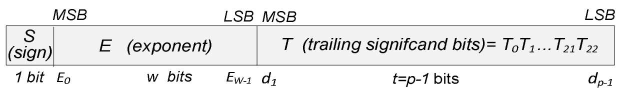

Fig. 1. Binary interchange floating-point format

a) 1-bit sign S.

b) *w*-bits biased exponent $E = q+bias$.($bias = 127$)

c) ($t = p - 1$)-bit trailing field digit string $T = d_1d_2d_3\ldots d_{p-1}$ the leading bit of d_0 is implicitly encoded in the biased exponent. In keeping with binary32, the values of *w* and *t* are 8 and 24 respectively. The BFP number obtained from its codification is expressed as:

$$B = -1^S * M * 2^q, q = E - 127 \quad (1)$$

where s is the sign bit, q is a 8-bit exponent belonging the range $[-127,128]$ and M is the non-negative integer significand formed as $M = 1.T_0T_1T_2 \dots T_{22}$ and normalized to the range $[1, 2\rangle$.

The exponent q is obtained as a function of biased non-negative integer exponent E. Stored values of q as -127 and 128 are interpreted as denormalized numbers producing *zero* and *+/- infinity* respectively.

As an example of BFP number, a float F is set to **1** 01111100 **10110000000000000000000** where bold types represent sign and fraction of mantissa respectively. In this example:

a) *S = **1***

b) *E = 01111100,* then *q = 124 - 127 = -3*

c) *M = **1**. (**10110000000000000000000**) = 1.6875*

Thus, the BFP value is $-1^1x1.6875 \, x \, 2^{-3} = -0.2109375$

III. Cube Root Based On Newton-Raphson Method

Newton-Raphson (NR) iteration is a convergence method based on approximating the root of a non-linear function [2]. This procedure consists of finding values of x such that $f(x) = 0$. A way to estimate roots to $f(x) = 0$ is to make a reasonable guess to the neighborhood of a solution. A value denoted as $x_0 \approx x$ is considered as an approximated initial solution. It is observed in Figure 2 that the equation of the tangent line to $y = f(x)$ at $x_0 = x$ presents a x-intercept by x_1. A linear representation at x_0 is expressed as follow:

$$y = f(x_1) = f(x_0) + f'(x_0) * (x_1 - x_0) \qquad (2)$$

The solution of this for x_1 produces the next relation:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (3)$$

If the process is repeated, using x_1 as new guess, a new accurate approximation is provided:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \qquad (4)$$

A general expression for this function iteration is determined by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (5)$$

Hence, the function $f(x_{n+1})$ is closer to zero when n gradually increases.

Following, the calculated iteration function is utilized to find the corresponding approximate solution of a cube root. If we figure out $\sqrt[3]{c}$ using the NR algorithm then $f(x) = x^3 - c$ should be analyzed. Right off, the derivative of $f(x)$ is computed as $f'(x) = 3x^2$ then both of these expression are substituted in the previous four equations. This produces the seed formula for iterative cube root:

$$x_{n+1} = \frac{1}{3} * \left(2x_n + \frac{c}{{x_n}^2}\right) \qquad (6)$$

The previous function requires a division $c/{x_n}^2$ at each iteration. As a workaround the division is computed as $c * (1/{x_n}^2)$ therefore an approximation of a reciprocal function $1/x_n$ is proposed. The seed expression for iterative [4] reciprocal is expressed as:

$$y_{n+1} = y_n * (2 - x_n y_n) \qquad (7)$$

Where y_{n+1} represents the approximated solution of $1/x_n$. The above explanation refers to two approximated processes: cube root and reciprocal. When the process begins, the first task consists in finding the initial values for both of them from a memory ROM. As practical illustration, Figure 2 depicts the approximation of the solution of $x = \sqrt[3]{7}$, the equation is rewritten as $f(x) = x^3 - 7 = 0$ for extracting zero solutions. The rest of the process bases on executing the previous expressions from (2) to (7) resulting a x_1 that represents an approximated zero to $f(x)$. On account of the need for accuracy, the system could execute further iterations.

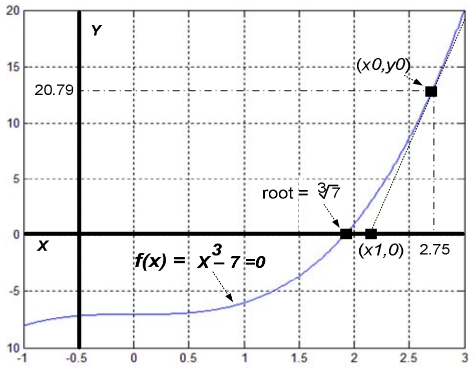

Fig. 2. Newton's method to approximate $\sqrt[3]{7}$

The diagram of the proposal is set forth in Figure 3. Before estimating $\sqrt[3]{X}$, the BFP number X is normalized such that $0.5 < X \leq 1$. Straightaway, the process gets going which will be detailed in the succeeding Sections.

IV. ROM Initial Aproximation Generation

This Section explains the generating of initial approximations for each cube root and reciprocal units. These values are stored in ROM.

The approximation of six initial solutions are sketched out in Table I. The first column represents the ROMs index. In the next column, X represents any normalized number BFP. The subsequent ones show the result of each mentioned process with their binary representations respectively.

Some restriction should be taken in account to generate the ROM values as: a) Initial approximations are coded into 24-bit precision. b) Is known the any input X is fitted to the range $\langle 0.5, 1]$, this prompts that the result of the reciprocal and cube root operations lie in the range $[1, 2\rangle$ and $\langle 0.79, 1]$ respectively. The last ranges assure the presence of 1-bit leading one for reciprocal (normalized as *1.xx..xx*) and 2-bit leading ones for cube root (normalized as *0.11xx..xx*). By this reason, memories of 22- and 23-bit ROM of size of 32 words are considered for each operation. The indexes of the ROM are formed by the 5 bits subsequent to the bits leading ones. c) It is worth pointing out that the bits leading ones and ROM values generate the 24-bit initials guess.

TABLE I. ROM INITIAL VALUES GENERATION

Ind. ROM	**$X \in \langle 0.5, 1]$ [X]**	**$[\frac{1}{X}]$**	**$\frac{1}{X_{bin}} = 1.x...x$ ROM[Ind.] $[xx..xx]_{23\ bits}$**	**$\sqrt[3]{X}$**	**$\sqrt[3]{X}_{bin} = .11x...x$ ROM[Ind.] $[xx..xx]_{22\ bits}$**
0	0.500	2.000	1111111111111 1111111111	0.793	00101100101111 11110110
4	0.562	1.777	1100011100011 1000111001	0.825	01001101010010 11000111
9	0.640	1.561	1000111110011 1000001101	0.862	01110010101111 10011000
16	0.750	1.333	0101010101010 1010101011	0.908	10100010010111 01101001
26	0.906	1.103	0001101001111 0111001100	0.967	11011110111100 01110010
31	0.984	1.015	0000010000010 0000100001	0.994	11111010101000 11100000

A brief example, let number of 9-bit $k = 448$ be, before completing the full operation is necessary that k to be normalized to $448/2^9$ such that $k \in \langle 0.5, 1]$. Normalization provides a result of *0.875* whose binary representation is 0.111. The input 0.111 is extended to a 9-bit operand defined as 111000000 which feeds the proposed cube root unit. So far as the initial value selection is concerned, the five bits following to the 2-bit leading ones are taken representing the ROMs index. In keeping with the example the index is 10000 (*16 decimal*). Observing Table 1, an index = 16 prompts an .11 & 10100010010111101101001 as initial cube root value, immediately the five bits following to the 2-bit leading ones are captured to find the initial reciprocal value. This new index is 11010 (*26 decimal*) and according to Table 1 the value of the selected word is 1. & 00011010011110111001100. This word represents the reciprocal initial solution. In turn, the alignment indicated by the binary point should be taken in consideration during calculations.

V. SINGLE FLOATING POINT CUBE ROOT DESIGN

A general overview of proposed system is presented in Figure 3. Arrows are used to show the direction of data flow and the shadow blocks indicate the main stages of the design.

Initially, an IEEE-754 standard operand (*X*) is decoded into three components: 1- sign (*S*), 8- exponent (*Exp*) and 23-bit fraction mantissa. A 1-bit leading one and the fraction mantissa are concatenated to achieve the true value of the mantissa (*Man*). Cube root implies in dividing the exponent over 3, as a workaround the exponent is expressed as multiple of 3 and a remainder. Man is turned into $Man' = Man * 2^{23}$ and the initial exponent ($Iexp$) is updated to $Exp - bias - 23$. The sign of $Iexp$ will be utilized later.

The $abs(Iexp)$ is utilized as index of a memory ROM of size of 151 words. The aforesaid indexes select 8-bit words which provide information of any multiple of 3 (n) and a remainder (r) of its respective index. Each word can be represented as:

$$index = 3n + r \tag{10}$$

$$ROM[index] = n_{binary} \,\&\, r_{binary} \tag{11}$$

A partial exponent, $Pexp$, is generated equal to $n.\ Man'$, represented as $Man * 2^{23}$, is passed on to cube root unit, as seen in previous sections. The analyzed function is focused in computing normalized numbers in the range $\langle 0.5, 1]$ therefore Man' is processed as $(Man'/2^{24}) * 2^{24}$. Its architecture is in-depth analyzed in the next Section.

In parallel special cases are detected and represented by: infinity, non-a-number ($sNaN, qNaN$), overflow and underflow.

The cube root unit begins when a *start* signal is asserted and the task is to calculate $\sqrt[3]{(Man'/2^{24}) * 2^{24}}$. For the sake of simplicity the multiplicand 2^{24} does not take part of the calculation but will affect the final exponent summing 8. Continuing with the explanation, as soon the process is *done* the output $rQ = \sqrt[3]{(Man'/2^{24})}$ is normalized according to the reminder r and the sign of $Iexp$.

Reminder r ($r < 3$) represents a 24-bit constant of the form $\sqrt[3]{2^r}$ whose result could be zero when $= 0$, $\sqrt[3]{2}$, when $r = 1$, and $\sqrt[3]{4}$ when $r = 2$. The constants are stored in ROM.

As explained earlier, the set (r, *sign of Iexp*) selects the new exponent $Pexp'$ and a binary *multiplication-by-constant* operation, therefore a rough approximation of the cube root operation is obtained.

The selection is carried out as follow:

$$rQ' = \begin{cases} rQ * zeros, & r = 0 \\ rQ * cube2, & r = 1 \land sign(Iexp) = 0 \\ rQ * recube2, & r = 1 \land sign(Iexp) = 1 \\ rQ * cube4, & r = 2 \land sign(Iexp) = 0 \\ rQ * recube4, & r = 2 \land sign(Iexp) = 1 \end{cases} \tag{11}$$

$$Pexp' = \begin{cases} Pexp, & sign(Iexp) = 0 \\ \overline{Pexp} + 1, & r = 0 \land sign(Iexp) = 1 \\ \overline{Pexp}, & sign(Iexp) = 1 \end{cases} \tag{12}$$

In (11) cube2, cube4, recube2, recube4 corresponding to 32-bit constants $\sqrt[3]{2}$, $\sqrt[3]{4}$, $1/\sqrt[3]{2}$, $1/\sqrt[3]{4}$ respectively. The expression $sign(Iexp)$ means sign of $Iexp$. In (12) $\overline{Pexp} = not\ Pexp$. In order to meet the normalization, the generated rQ' should belong to the range $\langle 0.5, 1]$ therefore a subsequent process verifies this requirement. A new product Q is constructed using a chunk of 32 bits of 56-bit rQ' as follows:

$$Q = \begin{cases} 32\ bits\ MSB\ of\ rQ', & rQ' \in \langle 0.5, 1] \\ 32\ bits\ MSB\ of\ (rQ' = rQ' \ll 1), & rQ' \notin \langle 0.5, 1] \end{cases} \tag{13}$$

At the same time, the exponent $Pexp'$ changes to:

$$Pexp = \begin{cases} Pexp' + 1, & r = 1 \lor r = 2 \land rQ' \in \langle 0.5, 1] \\ Pexp', & rQ' \notin \langle 0.5, 1] \end{cases} \tag{14}$$

The least significant bit (LSB), guard (*G*), round (*R*) and sticky (*STK*) bit are generated capturing the *LSB* and subsequent bits of Q. The 3-bit MSB of the captured data represents the LSB, *G*, and *R* respectively. The *STK* is obtained by means of or-chain operations of the remaining bits of the captured data.

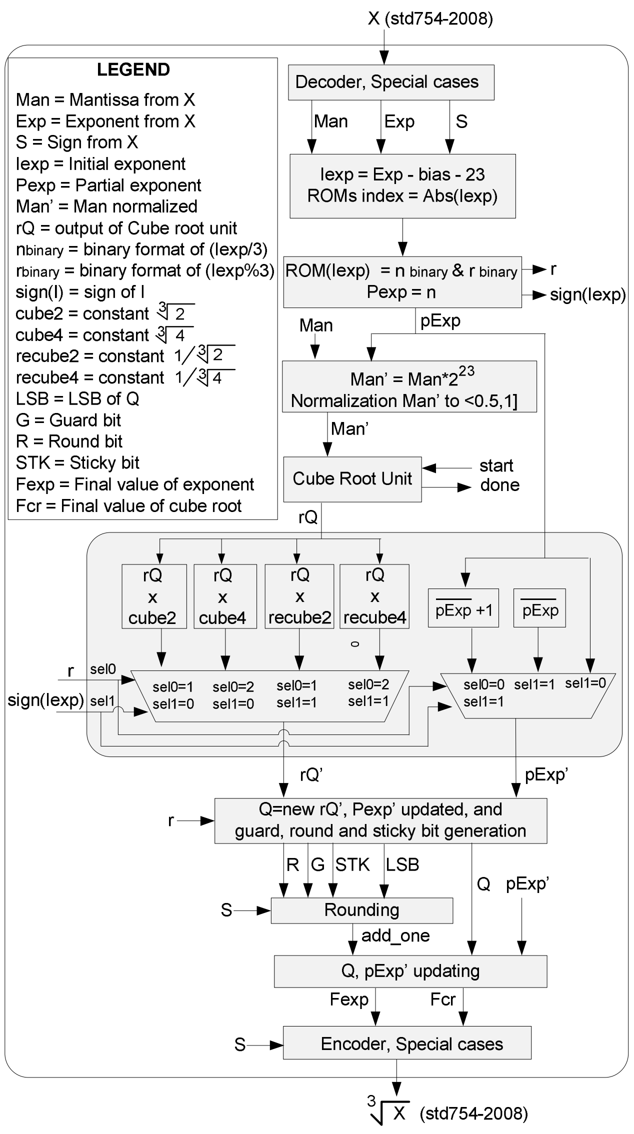

Fig. 3. High-level Binary Floating-Point Cube Root Diagram

To continuation the rounding unit is prompted. This process is feed by the previous bits and issues the signal *add_one* that is summed to Q. This sum provides a 32-bit result that is truncated to 24-bit MSB of Q producing the final cube root (Fcr) normalized into $\langle 0.5, 1]$. It is worth remarking that *round_ties_to_even* is the rounding strategy utilized.

In the rounding stage, special cases as overflow and underflow are tested again. In compliance with IEEE 754-2008 standard the mantissa is defined into $\langle 1, 2]$. Therefore a left-shift operation by 1-bit on *Fcr* is executed if its respective MSB is zero. The final exponent (*Fexp*) is calculated as follow:

$$Fexp = \begin{cases} Pexp' + 9 + bias, & if\ MSB\ of\ Fcr = 1 \\ Pexp' + 8 + bias, & if\ MSB\ of\ Fcr = 0 \end{cases} \tag{15}$$

Finally, the final mantissa is made up of the first 23 bits subsequent to MSB of $Fexp$, the final exponent (*Fexp*), sign (*S*), and special cases (*SC*) are processed by the Encoder stage to generate the IEEE-754-2008 format final output. During the final process the overflow and underflow should be analyzed and signaled.

Regarding cycles cost, the cube root unit is fully sequential and is made up for reciprocal and cube root cells which need 3 and 4 clock cycles respectively. The remaining process presents combinational blocks which are: decoder, exponent recalculating, index generation, mantissa and exponent normalization, cube root output updating, signals required by rounding, rounding, re-evaluation of final mantissa and exponent, and encoder as last stage. All of them are executed in 9 clock cycles.

The total cycles of the system can be calculated using the expression: *total_cycles = 9 + 2 x n_iter_rec + n_iter_cr,* where *n_iter_rec* and *n_iter_cr* corresponding the number of iterations for reciprocal and cube root respectively. The design was simulated and verified executing large amounts of data. A reliable approximation was detected when *n_iter_rec = 2* and *n_iter_cr = 1*. Our proposal needs 19 cycles clock. In regard to size of memory, the design requires 346x8 bits ROM.

VI. Cube Root Unit Architecture

In Figure 4 exhibits the sequential architecture of our proposal. The rounded corners shadow blocks representing the mainly process of this unit. With regard to shadow embedded blocks are sub-stages based on binary operations. The thin dashed lines show the segmentation of each block. The worst data path is signaled by the thick dashed line.

This stage receives as input the normalized value $Man'/2^{24}$ that was explained in the previous Section. As soon as the input is read, a 24-bit word is drawn from ROM, using 5-bit address as it is seen in Figure 4. This word, CR_0, represents the initial solution for cube root approximation. Straightaway, 5-bit chunk is extracted from this word as was specified in the last Section producing the initial solution for reciprocal approximation denoted as REC_0. Get this execution done, all required inputs ($Man'/2^{24}$) and the two approximations (CR_0 and REC_0) are ready to be utilized.

As seen in earlier Sections, the proposed sequential circuit is targeted to design the iterative function represented in (6) and (7). Due to the presence of a division operation, seen in (6), we can profit from this equation as follows: the division is represented as a multiplication of the dividend and the reciprocal of the divisor.

Hence the fact that iterative approximation for cube root and reciprocal functions are implemented.

They both architectures are shown in Figure 4. The hardware can be dealt with describing two blocks. The beginning of the process means the first iteration as much for reciprocal, denoted by $n_0 = 0$, as for cube root, denoted by

$n_1 = 0$. There the variables y_0 and x_0 are initialized to REC_0 and CR_0 respectively.

The first block based on (7) generates the result of an initial reciprocal approximation. As is observed this block requires 3 clock cycles. In the first clock a squarer and shift register of reciprocal initial solution are computed, getting 48- and 26-bit operands as outputs respectively. The 48-bit output and the cube root initial solution are multiplied whose result is truncated to 48 bits. Finally, a subtraction of the 26-bit operand and the output from the multiplication is carried out. The first block is executed multiple times (n_0) according the number of programmed iterations for reciprocal approximation denoted by *rec_iter*. The output generated y_1 is truncated as is shown in Figure 4.

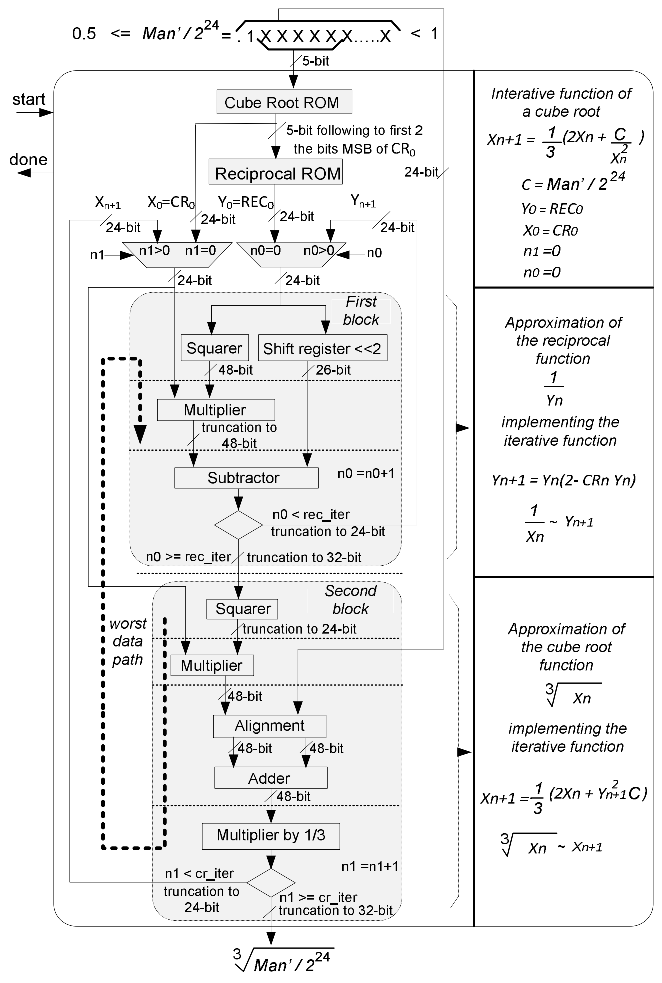

Fig. 4. Sequential Hardware of Cube Root Operation

The second block, related to (6), captures 32-bit y_1 ($y_1 \cong 1/x_0$) from the first one. The hardware requires 4 cycles. First, a squarer operation is executed prompting an output that is truncated to 24 bits. The next cycle, the last output and the initial cube root solution are multiplied whose result is a 48-bit product. In the third cycle, the normalized input, $Man'/2^{24}$, and the previous product are summed. The binary point is analyzed by an alignment process. Finally, a constant multiplication operation is executed in the last cycle.

A precision of 32 bits, denoted by x_1, is achieved as output. As was explained in the first block, numbers of executions (n_1) depend on the programmed iterations (*cr_iter)* for cube root approximation.

The hardware is based on one cycle 32x24- and 24x24 binary multiplier cells, which were fitted to *DSP48* Slice. The worst data path occurs having as source the squarer stage of second block and as destination the multiplier of the first one.

VII. FPGA Floating Point Cube Root Implementations

All circuits were described in VHDL. Some parts the proposal use low level component instantiation. For synthesis and implementation XST[15] and Xilinx ISE 14.2 tool[14] have been used respectively. The circuits were implemented in a Virtex-5 speed grade -3 using timing constraints[12].

Delay and area depend mainly on the number of iteration executed and the size of the multipliers utilized. As said before, at most 32x24 multipliers were considered. Insofar as more accuracy of our proposal is required, multiplications of larger binary numbers should be considered.

In Table II the results of the implementation are presented.

Unfortunately, publications based on cube root under FPGA techniques could not be found. By this reason, comparison with others works might be unfair.

TABLE II. Single Floating Point Cube Root Implementation

# Slices	# Flip Flop	# Lut	# Dsp48	Min Period (ns)	Max Freq (Mhz)	# Cycles	Delay (ns)	Bram 18kb	Mops
236	439	576	12	6.7	149	19	127.3	1	7.87

A. Verification

A behavioral model testbench, using ModelSim, was fed with large normalized BFP numbers of random vectors. As a result of these simulations, two iterations of reciprocal approximation and one of cube root provided a reasonable solution obtaining an error of +/- 10^{-6} in the mantissas. It was added to our testbench the *Real VHDL Package* utilizing the function *cbrtf* whose result was compared with the proposed hardware.

In order to carry out performance assessments of our circuit, a Win32 C++ application of millions of cube root operations (using the *cbrtf* function defined in the standard library *mat.h*) was evaluated whose final result was matched with our hardware proposal. These assessments are described below:

Execution times are shown as first test. The C++ script was evaluated on an Intel Core i-7 Processor @ 2.20 Ghz. The execution time required per cube root operation was around 0.11 ns. The time-consuming of our proposal was 127.3 ns @ 149 Mhz under FPGA.

As second test the speed-up of a Software / Hardware Codesign was evaluated. Therefore a Microblaze Soft-Core Embedded Processor with its Floating Point Unit respective was implemented [13]. The digital system based on PLB-bus was developed over Virtex5-vfx30t-3ff665@ 50Mhz. Then, a customized co-processor based on our design was implemented and connected the system central through Fast Simplex Link (FSL) bus. The C++ script was executed. The obtained result reveals a delay of 40 cycles for version hardware and 630 for version software producing an acceleration of 15.75. Despite that our proposal takes 19 cycles, Microblaze took further cycles executing FSL instructions.

B. Future Improvements

Several improvements of the analyzed operation under Newton- Raphson and FPGA techniques could be studied in a future advanced version. Firstly, the binary multipliers could be segmented, therefore an improvement of the frequency performance could be achieved. This would help us to propose a pipeline version of our design. Secondly, the design can be extended to compute doubles BFP, this would involve large hardware cost due to the fact that several multipliers would be required. As a workaround either efficient multipliers or intellectual property cores could be utilized. Thirdly, the division c/x_n^2 in (6) could be computed from a binary division instead of executing convergence methods for determining $1/x_n$. Finally, could be analyzed the implementation of a general solution $\sqrt[n]{x}$. This would entail to limit n to certain range of values. And as was tackled our proposal, the exponents would be expressed as multiples of the radix n.

VIII. Summary

This work presents the hardware of a single floating point cube root based on Newton-Raphson method and implemented on FPGA platform.

This architecture is capable of computing 7.8 MOPS.

Regarding *Dsp48s* slice consumption, the design occupies 12 slices. All the available *Dsp48s* in the utilized FPGA (Virtex-5) would only allow us to fit five cores of our proposal. Working with larger Virtex-5 such as the model xc5lx330t-2ff1738 would allow us to fit 16 units of our custom hardware.

A co-processor, connected to a Microblaze embedded processor, of the proposed circuit accelerates its version software up to 15.75 times.

Our core can be utilized in scientific calculation that involves cube root functions.

The resulting mantissa presents a decimal error of 10^{-6}.

Acknowledgment

This research has been supported by Fundación Vodafone España under contracts FUAM 085212 and 085206. Additional funds have been obtained from Euroform Polo Español.

References

[1] IEEE Standard for Floating-Point Arithmetic, 2008. IEEE Std 754-2008.

[2] Andrew Adler. Notes on Newton-Raphson method. *online available:http://www.math.ubc.ca/ anstee/math104/newtonmethod.pdf.*

[3] M. F. Cowlishaw. Decimal floating-point: algorism for computers. In *Proc. 16th IEEE Symp. Computer Arithmetic*, pages 104–111, 2003.

[4] Jean-Pierre Deschamps. *Synthesis of Arithmetic Circuits FPGA, ASIC and Embedded Systems*. New Jersey : John Wiley, 2006, 2006.

[5] M.D. Ercegovac. On Digit-by-Digit Methods for Computing Certain Functions . In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, pages 338 – 342, 2007.

[6] J.D. Bruguera J.-A Pineiro, M.D. Ercegovac. Algorithm and Architecture for Logarithm, Exponential and Powering Computation. In *Computers, IEEE Transactions on*, volume 53, pages 1085–1096, 2004.

[7] P. Montuschi J. Bruguera, F. Lamberti. A Radix-2 Digit-by-Digit Architecture for Cube Root. In *Computers, IEEE Transactions on*, volume 57, pages 562–566, 2008.

[8] Hyun-Chul Shin ; Jin-Aeon Lee ; Lee-Sup Kim. A minimized hardware architecture of fast Phong shader using Taylor series approximation in 3D graphics. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pages 286 – 291, 1998.

[9] K. E. Wires M. J. Schulte, J. E. Stine. High-speed reciprocal approximations. In *Signals, Systems & Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on*, volume 2, pages 1183–1187, 1997.

[10] P. Tang S. Story. New algorithms for improved transcendental functions on IA-64. In *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, pages 4–11, 1999.

[11] Brian J. Shelburne. Another Method for Extracting Cube Roots. *Dept. of Math and Computer Science-Wittenberg University.*

[12] Xilinx Inc. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics (DS202)*, v5.3 edition, May 5, 2010 2010.

[13] Xilinx Inc. *EDK Concepts, Tools, and Techniques*, 14.2 edition, April 2012.

[14] Xilinx Inc. *Xilinx Inc. Xilinx ISE Design Suite 14.2 Software Manuals*, v14.2 edition, June 2012.

[15] Xilinx Inc. *Xilinx Inc. XST User Guide 14.2*, v14.2 edition, June 2012.

[16] Chia-Sheng Chen-Hau-Zen Sze An-Peng Wang Ying-Shieh Kung, Kuan-Hsuan Tseng. FPGA-Implementation of Inverse Kinematics and Servo Controller for Robot Manipulator. In *Robotics and Biomimetics, 2006. ROBIO '06. IEEE International Conference on*, pages 1163–1168, 2006.