Tradeoff of FPGA Design of a Floating-point Library for Arithmetic Operators

Daniel M. Muñoz[1], Diego F. Sanchez[1], Carlos H. Llanos[1], and Mauricio Ayala-Rincón[2]

[1]Department of Mechanical Engineering, University of Brasilia, Brasilia, D.F., Brazil
[2]Departments of Mathematics and Computer Sciences, University of Brasilia, Brasilia, D.F., Brazil
e-mail: damuz@unb.br

ABSTRACT

Many scientific and engineering applications require to perform a large number of arithmetic operations that must be computed in an efficient manner using a high precision and a large dynamic range. Commonly, these applications are implemented on personal computers taking advantage of the floating-point arithmetic to perform the computations and high operational frequencies. However, most common software architectures execute the instructions in a sequential way due to the *von Neumann* model and, consequently, several delays are introduced in the data transfer between the program memory and the Arithmetic Logic Unit (ALU). There are several mobile applications which require to operate with a high performance in terms of accuracy of the computations and execution time as well as with low power consumption. Modern Field Programmable Gate Arrays (FPGAs) are a suitable solution for high performance embedded applications given the flexibility of their architectures and their parallel capabilities, which allows the implementation of complex algorithms and performance improvements. This paper describes a parameterizable floating-point library for arithmetic operators based on FPGAs. A general architecture was implemented for addition/subtraction and multiplication and two different architectures based on the *Goldschmidt's* and the *Newton-Raphson* algorithms were implemented for division and square root. Additionally, a tradeoff analysis of the hardware implementation was performed, which enables the designer to choose, for general purpose applications, the suitable bit-width representation and error associated, as well as the area cost, elapsed time and power consumption for each arithmetic operator. Synthesis results have demonstrated the effectiveness of the implemented cores on commercial FPGAs and showed that the most critical parameter is the dedicated Digital Signal Processing (DSP) slices consumption. Simulation results were addressed to compute the mean square error (MSE) and maximum absolute error demonstrating the correctness of the implemented floating-point library and achieving and experimental error analysis. The *Newton-Raphson* algorithm achieves similar MSE results as the *Goldschmidt's* algorithm, operating with similar frequencies; however, the first one saves more logic area and dedicated DSP blocks.

Index Terms: Floating-point arithmetic, FPGAs.

1. INTRODUCTION

Most of the engineering and scientific applications involve the implementation of complex algorithms that are based on arithmetic operators [1]. The fixed-point arithmetic allows the computations to be performed with a high precision according to the bit-width representation. However, many applications require to work not only with a high precision, but also with a suitable format in order to represent large and small real numbers [2], [3]. Therefore, the floating-point arithmetic is a feasible solution for high performance computer systems, providing an appropriate dynamic range for representing real numbers and capabilities to retain its resolution and accuracy [4].

Floating-point based algorithms are commonly implemented as software and executed in microprocessors. Typically this solution requires to pay a performance penalty given that the conventional approaches require to perform the data transfer between the ALU and the program and the instruction memories [5]. This problem, well known as *von Neumann* bottleneck, has been partially overcome by using multicores microprocessors reducing the execution time.

Recently, Graphic Processor Units (GPUs) are being widely used to implement complex algorithms taking advantage of parallel floating-point arithmetic operators, increasing the throughput and achieving an expressive speed-up [6], [7]. Although GPUs work with a high frequency and improve the performance, it is a microprocessor based solution and as a consequence suffers of the *von Neumann* bottleneck when all the source data are accessed from global memory or when simultaneous accesses from different threads to memory have to be addressesd. Additionally, GPUs commonly operate at high frequencies increasing the power consumption of the overall system, being a drawback for embedded system applications.

Modern FPGAs are a suitable solution that provides hundred of thousand of logic elements and dedicated DSP blocks as well as several desired properties such as intrinsic parallelism, flexibility, low cost and customizable approaches. All this allows for a better performance and accelerated execution of the involved algorithms on mobile applications.

FPGA implementation of complex algorithms can improve the performance by exploring the parallel capabilities and using the required hardware resources. However, FPGAs only provide integer or fixed-point arithmetic, which implies that the computations are confined to a limited numeric range. Therefore, a floating-point FPGA implementation of arithmetic operators is an important issue for high performance applications.

There are several previous works covering FPGA implementations of floating-point arithmetic operations and different algorithms for computing the division and the square root [8], [9], [10]. Although, there exist different contributions for the FPGA implementations of floating-point addition and multiplication, support for division and square root have remained uneven. Several implementations for floating-point division and square root in FPGA are presented in [10], [11], [12], [13]. Wang [10] and Kwon *et al.* [11] presented a Taylor series expansion based approach. Shuang-yan *et al.* [12] applied a *Newton-Raphson* method and [13] a high radix SRT division algorithm and a binary restoring square root algorithm. These works, however, did not give enough attention to the tradeoff between the cost in area against the bit-width representation, as well as, to their respective error analysis and power consumption. Reference [14] presents an FPGA implementation of a floating-point library for arithmetic operations, which uses the *Goldschmidt's* algorithm for implementing division and square root. In that work the authors have included the accuracy as design criterion based on an experimental study of the error. However, the power consumption estimation of the implemented circuits is not considered.

This paper describes the FPGA implementation of an arithmetic floating-point library and presents a tradeoff analysis between the bit-width representation against the area cost, power consumption and the error associated. Two different architectures, based on the *Goldschmidt's* and *Newton-Raphson* iterative algorithms, have been implemented for division and square root allowing for a comparison between these approaches. This analysis allows the designer to choose the better implementation and the bit-width representation for general purpose applications. The implemented cores are based on the IEEE-754 format [15] and were described in VHDL (Very high speed integrated circuits Hardware Description Language). Synthesis results were performed for a commercial Virtex5 FPGA device and have demonstrated high throughput, high performance and low resources consumption. Simulation results were performed for different bit-width representations using the ModelSim® XE simulation tool [16] after the place and route process and the results were compared with a Matlab® R2008a implementation in order to compute the Mean Square Error (MSE) and the maximum absolute error.

This paper is organized as follows. Section 2 presents the related works covering FPGA implementations of floating-point arithmetic operators. Section 3 describes the general algorithm for addition/subtraction, multiplication and the *Goldschmidt's* and *Newton-Raphson* algorithms for division and square root operators. The implementations of the proposed architectures are described in Section 4 and before concluding, Section 5 presents the analysis of the synthesis and simulation results.

2. RELATED WORKS

On the one hand, early works on floating-point arithmetic operations have been targeted to primitive FPGAs only for adders and multipliers [17], [18], [19]. However, given the reduced number of logic elements in early FPGAs, the implementation of other arithmetic operators such as division, square root and even transcendental functions was impractical for hardware designers. These early results point-out that implementing IEEE single precision addition and multiplication operations was feasible but impractical on FPGAs.

On the other hand, modern FPGAs have hundred of thousand of logic elements and several dedicated DSP blocks, which allows for embedding more complex computations and new algorithms [20]. Recently, FPGA floating-point arithmetic operations have been implemented for both 32-bit single precision and 64-bit double precision [9], [21], [22]. However, many of the engineering applications need different precision requirements; thus, parameterizable floating-point libraries for arithmetic operations in FPGAs are relevant for scientific applications. For instance, finding a suitable weight precision and a feasible number of iterations of algorithms for computing division and square root is an important factor in order to avoid over dimensioning hardware implementations.

Hardware architectures for computing division and square root have been proposed in [13] and [23]. For instance, Wang and Nelson [13] implemented in hardware the SRT method presenting results for iterative and pipelined hardware implementations. Montuschi and Mezzalama [24] presented an analysis of different techniques such as direct algorithms, non-restoring algorithms, SRT square rooting algorithms and approaches based on *Newton-Raphson* and CORDIC. References [8] and [9] describe parameterizable libraries for floating-point arithmetic operations, implementing Radix-4 and Radix-2 SRT algorithms for computing division and square root, respectively. Synthesis results for double precision are presented in [8] and the aspects related to the area consumption for different bit-width precision are presented in [9]. References [10] and [11] presented a Taylor series expansion for computing division and square root. Wang [10] described a variable-precision floating-point implementation showing area consumption results for different bit-width representation, exploring the capability of their applications on image and signal processing. Kwon *et al.* [11] performed a comparison among the Taylor series expansion against the Intel Pentium 4 processor (that uses the SRT algorithm) and the AMD K7 core (that uses *Goldschmidt's* technique).

In the hardware design of floating-point units it is important to point-out two main aspects: (a) the selection of a suitable bit-width in a such way that dynamic range is large enough to guarantee that saturation will not occur for a general-purpose application and (b) the tradeoff between the level of precision of the operators against their implementation cost in logic area. To include the accuracy as design criterion of digital circuits it is important to analyze behavioral aspects in terms of the tradeoff between the cost in logic area against the error associated as well as the computing time and the power consumption.

Although different algorithms have been proposed for computing addition/subtraction, multiplication, division and square root, covering floating-point arithmetic operations, which have been implemented in FPGAs, these implementations have not received enough attention with regard to the error analysis. In this work, the authors also present an experimental error analysis of the implemented floating-point architectures, using the Matlab® R2008a, which works in a double precision format, as statistical estimator. Different bit-width representations have been implemented for each one of the arithmetic operators and both the *Goldschmidt's* and *Newton-Raphson* algorithms (for division and square root) have been simulated in order to compute the MSE and maximum absolute error of the proposed circuits.

3. BACKGROUND

In this section the IEEE-754 standard format for representing floating-point numbers and the general algorithms for implementing the floating-point addition/subtraction, multiplication, division and square root are presented.

A. The IEEE standard

The IEEE-754 standard [15] is a floating-point number representation in a bit string format, characterized by three components: a sign *S*, a biased exponent *E* with E_w bit-width, and a mantissa *M* with M_w bit-width, as shown in Figure 1. A constant (bias) is added to the exponent in order to make the exponent's range non negative. Additionally, the mantissa represents the magnitude of the number.

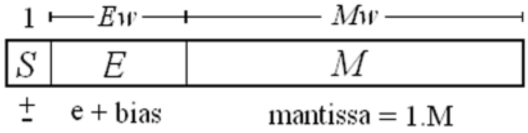

Figure 1. The IEE-754 standard

This standard allows the user to work not only with the 32-bit single precision and 64-bit double precision, but also with a flexible and suitable precision according to the application requirements. This is suitable for supporting variable precision floating-point operations [10]. A higher precision means fewer quantization errors in the final implementations, as long as a lower precision leads to straightforward designs, higher speed and reductions in area requirements and power consumption. Standard embedded microprocessors, such as *NIOS* and *Microblaze*, work only with single precision (generally in hardware) and double precision (generally in software), limiting in this way the flexibility of the implementations [25]. This problem can be solved by using parameterizable floating-point operations directly in hardware.

B. The floating-point addition/subtraction

The steps for performing the floating-point Addition/Subtraction are shown below.

1) Separate the sign, exponent and mantissa of the inputs, and check whether the inputs are zero, infinity or an invalid representation in IEEE 754 standard. Add the hidden bit to the mantissa.

2) Compare the two inputs: a logical shift right operation must be performed over the smaller of the two numbers. The number of bits of the mantissa shifted right, dependent on the exponent's difference, and this difference is a preliminary exponent calculation result. Finally, add/sub the current mantissas.
3) Shift left the achieved mantissa until its most significant bit (MSB) is 1. For each shift decreases the current exponent by 1. Finally, concatenate the sign, exponent and mantissa of the final results.

C. The floating-point multiplication

The steps for performing the floating-point multiplication are shown below.

1) Separate the sign, exponent and mantissa of the inputs, and check whether the inputs are zero, infinity or an invalid representation in IEEE-754 standard. Add the hidden bit to the mantissas.
2) Multiply mantissas, add exponents, and determine the product sign.
3) Whether the MSB is 1 in the mantissas multiplication result, hence, no normalization is needed. The current mantissa is shifted left until a 1 is achieved. For each shift operation the current exponent is decreased by 1. Finally, concatenate the sign, exponent and mantissa of the final results.

D. The floating-point division

A generalized algorithm to calculate the division is described in [14], whose steps are included below.

1) Let X and Υ be real numbers represented in IEEE-754 standard, where X represents the dividend and Υ the divisor.
2) Separate the sign, exponent and mantissa of X and Υ, adding the 1 hidden bit to the mantissa and detecting zero and invalid inputs.
3) Calculate the mantissa result using the *Goldschmidt's* or *Newton-Raphson* algorithms for division described below. In parallel to this, evaluate the result exponent, namely exponent(X) – exponent(Υ) + *Bias*, and evaluate the sign of the result.

1) The Goldschmidt's algorithm for division

Assume two n-bit inputs N and D, which satisfy $1 \leq N, D < 2$. The idea of the *Goldschmidt's* algorithm is to calculate $Q = N/D$, starting from an initial seed equal to $1/D$, and then to approximate the quotient through successive multiplications of the seed times N [26]. This work takes as reference the algorithm presented in [1]:

1) Move the fixed point for N and D to locations such that $N \geq 1$ and $D < 2$.
2) Start with an initial approximation to $1/D$ by using a look-up table and call it L_0.
3) Calculate the first approximation to $q_0 = L_0 \times N$ and the error term $e_0 = L_0 \times D$.
4) Refine the approximations using the following iterative equations:

$$L_{i+1} = - e_i \quad (1a)$$
$$e_{i+1} = e_i \times L_{i+1} \quad (1b)$$
$$q_{i+1} = q_i \times L_{i+1} \quad (1c)$$

After each iteration of the algorithm, e_i approximates to 1 (the denominator D is multiplied by $1/D$) and q_i approximates the true quotient Q. Notice that equations (1b) and (1c) can be computed in a parallel approach.

2) The Newton-Raphson algorithm for division

The *Newton-Raphson* algorithm has two n-bits inputs N and D, that satisfy $1 \leq N, D < 2$, starting from an initial approximation to $y_0=1/D$. Equations 1(a) and 1(b) must be executed in a iterative way [27].

$$p = D \times y_i \quad (2a)$$
$$y_{i+1} = y_i \times (2-p) \quad (2b)$$

After the i^{th} iteration, multiplying $N \times y_{i+1}$ is yielding an approximation to N/D. The *Newton-Raphson* iteration differs from *Goldschmidt's* algorithm by referring to the initial divisor during each iteration [27].

E. The floating-point square root

A generalized algorithm to calculate the FP square root is described in [14].

1) Let X be a real number represented in IEEE-754 standard, whose square root is required.
2) Separate the sign, exponent and mantissa of X adding the 1 hidden bit of the mantissa, and detecting negative, zero and invalid inputs. Whether the exponent of X is even then multiply the mantissa of X by 2.
3) Calculate mantissa results using any of the algorithms for calculating the square root presented below. Parallel to this, evaluate the result of the exponents that is equal to the exponent of $(X + Bias)/2$.
4) Finally, concatenate the sign, exponent and mantissa of the result and remove the hidden bit of the resulting mantissa.

This algorithm deals mainly with the mantissa calculation because of the resulting exponent is obtained in a straightforward way. Therefore, this allows the designer to carry out the mantissa calculation using both addition and multiplication operations in fixed-point only, obtaining therefore a less resource consumption implementation.

1) The Goldschmidt's algorithm for square root

For a given variable *b*, this algorithm calculates *sqrt(b),* starting from an initial seed equal to 1/*sqrt(b)*, and the result is improved through an iterative process, whose number or iterations is defined by the user. This work takes as reference the algorithm introduced in [27], who proposed the flowing steps.

Set $b_0 = b$ and let y_0 be a suitable good approximation to 1/*sqrt*(b_0), such that $1/2 \leq b_0 \mathrm{x} y_0^2 \leq 3/2$. Set $g_0 = b_0 \mathrm{x} y_0$, and $h_0 = y_0/2$. Then, for $i > 0$, in the i^{th} iteration compute:

$$r_{i-1} = 0.5 - g_{i-1} \mathrm{x} h_{i-1} \quad (3a)$$

$$g_i = g_{i-1} + g_{i-1} \mathrm{x} r_{i-1} \quad (3b)$$

$$h_i = h_{i-1} + h_{i-1} \mathrm{x} r_{i-1} \quad (3c)$$

At each iteration, a closer value to *sqrt(b)* is computed in *g* and the variable *h* approximates to 1/(2xsqrt(*b*)). This method eliminates the divisions by two for each iteration and only needs three multipliers and three addition/subtraction modules. Notice, that a parallel calculation of the equations (3b) and (3c) is possible.

2) The Newton-Raphson algorithm for square root

For a given *b*, this algorithm calculates *sqrt*(*b*), starting from an initial seed equal to y_0=1/*sqrt*(*b*) and refining it with the following iteration.

$$y_{i+1} = 0.5 \mathrm{x} y_i \mathrm{x} (3 - b \mathrm{x} y_i^2) \quad (4)$$

After the i^{th} iteration, the variable y_{i+1} accumulates a more accurate value of 1/*sqrt*(*b*). Finally, multiplying y_{i+1}x*b* is obtained a value of *sqrt*(*b*) [28].

4. HARDWARE IMPLEMENTATIONS

In this section the hardware implementations of the floating-point arithmetic operators are described. All the components are based on the IEEE-754 standard, supporting exceptions and are parameterizable by bit-width for all the arithmetic operators and also are parameterizable by the number of iterations for both the *Goldschmidt's* and the *Newton-Raphson* algorithms (in case of division and square root).

A. FPGA implementation for division

Figures 2a and 2b show the developed *Goldschmidt's* and *Newton-Raphson* architectures respectively, for a division operator. As described in Section 3.D this algorithm operates over mantissa. Therefore the range of the inputs values is [1,2) and the seeds (1/*D*) are stored in a look-up table for different bit-width representations. In this approach, the number of the seeds to be stored is computed by dividing the range by the size of the look-up table (number of words). Successive iterations for refinement of the initial approximation are then executed, using a Finite State Machine (FSM) controller.

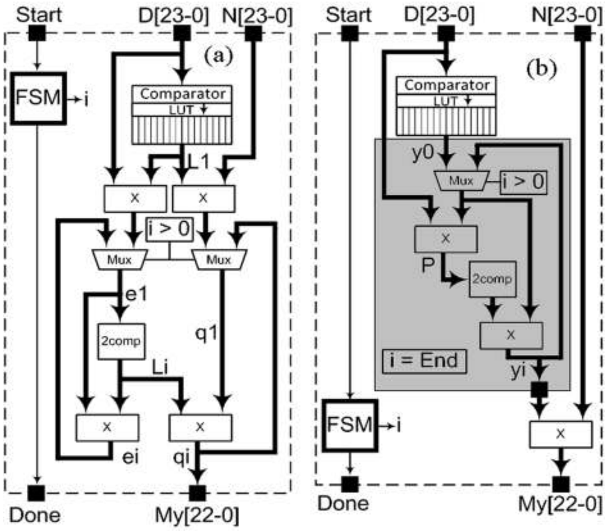

Figure 2. Iterative architecture for division (a) *Goldschmidt's* algorithm (b) *Newton-Raphson* algorithm

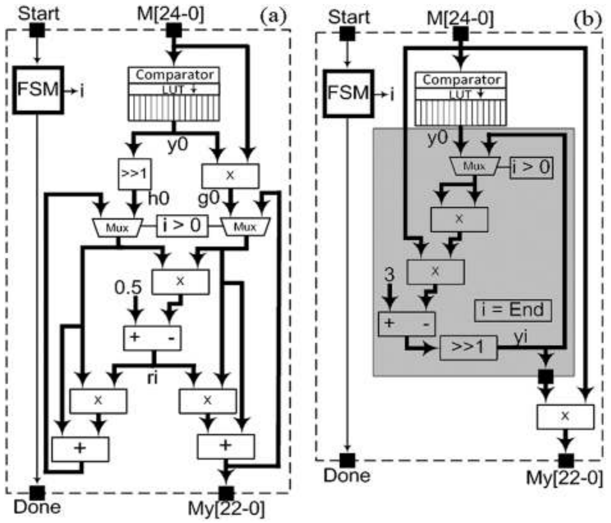

Figure 3. Iterative architecture for square root (a) *Goldschmidt's* algorithm (b) *Newton-Raphson* algorithm

Initially, the *Goldschmidt's* architecture chooses as initial approximation $L_1=1/D$ (stored in the LUT of seeds) and uses the first two parallel fixed-point multipliers to compute the values of the quotient approximation $q_1=L_1 \mathrm{x} N$ and the error $e_1=L_1 \mathrm{x} D$. At the first iteration, the two parallel multiplexers select between the values of q_1 and e_1 and a two-complement operation is performed in order to compute the value $L_2 = -e_1$. Afterward, two parallel fixed-point multipliers are used to compute the new quotient approximation $q_2 = q_1 \times L_2$ and the new error $e_2 = e_1 \times L_2$. Finally, two parallel multiplexers allow the algorithm to feedback the current approximation and then the new quotient approximation $q_i = q_{i-1} \times L_i$ and the new error $e_i = e_{i-1} \times L_i$ are computed. The FSM evaluates possible exceptions, controls the number of the iteration and synchronizes the hardware components. After a suitable number of iterations, previously set-up by the user, the quotient result (qi) represents the mantissa value result.

The *Newton-Raphson* architecture starts with the initial approximation $y_1 = 1/D$ (stored in the LUT of seeds). The multiplexer selects the first approximation and then, the fixed-point multiplier is used to compute the value of $p = D \times y_{0.}$ Afterward, the two-complement operation is used to compute the value of $(2-p)$. Finally, a last fixed-point multiplier computes the value of the new approximation $y_2 = y_1 \times (2-p)$. This approximation value is used to compute a new p value $p = D \times y_{1,}$ which is used to compute the next approximation $y_{i+1} = y_i \times (2-p)$. The FSM evaluates possible exceptions, synchronizes the hardware components and controls the number of iterations. After a correct number of iterations, previously set-up, the same fixed-point multiplexer is used to evaluate the final approximation $N \times y_{i+1}$, which represents the mantissa value result.

It can be observed that *Goldschmidt's* architecture yields at each iteration a new approximation of the final result (namely, N/D), whereas in the *Newton-Raphson* architecture each iteration refines the value of $y = 1/D$ and after n iterations the approximation to N/D is performed multiplying $y \times N$.

B. FPGA implementation of square root

Figures 3a and 3b respectively show the architecture for computing the square root using *Goldschmidt's* and *Newton-Raphson* algorithms. According to the general algorithm for computing square root, the range of the input values is [1,4) (where $2M \in [1,4)$) and the seed ($1/sqrt(M)$) is stored in the look-up table, being M the mantissa. Look-up tables of 8, 16, 32, 64 and 128 addressable words are used for storing different initial seeds that were computed splitting the previous defined input range depending on the size of the look-up table (namely, the number of addressable word). Successive refinements over the partial results are executed in an iterative process for improving the result. The overall process is controlled by a FSM, which allows for sharing multipliers and adders in order to reduce the resources consumption.

As explained in Section 3, the *Goldschmidt's* architecture operates over the mantissa and computes the value of $sqrt(M)$. In this case, it selects a an initial approximation $y_0 = 1/sqrt(M)$ (stored in a LUT of seeds), the first $sqrt(M)$ approximation is performed by computing $g_0 = y_0 \times M$ and the value $h_0 = y_0/2$ is obtained by using a right shift register. Afterward, the iterative process starts and two parallel multiplexers select between the first stage approximation and the next stages approximations. The values $g_0 \times h_0$ and $r_1 = 0.5 - g_0 \times h_0$ are computed in a sequential way. Finally, two parallel fixed-point multipliers and two parallel fixed-point adders are used to compute $g_1 = g_0 + g_0 \times r_0$ and $h_1 = h_0 + h_0 \times r_0$ in a parallel approach, where g_1 is the new square root approximation. The FSM supports the exceptions, synchronizes the different hardware components and feedback the g_1 and h_1 values in order to start a new iteration. After a suitable number of iterations, previously defined by the user, the value of $g_i = g_{i-1} + g_{i-1} \times r_{i-1}$ represents the mantissa value result of the square root.

The *Newton-Raphson* architecture operates over the mantissa and computes the value of $sqrt(M)$ as follow. An initial approximation equal to $y_0 = 1/sqrt(M)$ (stored in a LUT of seeds) is upload and the algorithms starts to iterate. One multiplexer chooses between the approximation at the first iteration or the other approximations computed on the consecutive iterations. Afterward, one fixed-point multiplier is used to compute the value of y_0^2 and then this result is used to compute the value $M \times y_0^2$. The next two stages compute the value $y_1 = \mathrm{y}_0 (3 - M \times \mathrm{y}_0^2)$. Finally, a right shift register is used to divide by two the current result, obtaining a new square root approximation value that is used in the next iteration. After a suitable number of iterations, previously defined by the user, the last approximation value (y_i) represents the mantissa value result of the square root.

In the *Goldschmidt's* algorithm for computing square root, each iteration approximates a new value of $sqrt(b)$, as long as in the *Newton-Raphson* architecture, each iteration refines the value of $y = 1/sqrt(b)$ and after n iterations, the approximation to $sqrt(b)$ is obtained by multiplying $y \times b$.

5. RESULTS

This section summarizes the synthesis and simulation results of the proposed circuits. All the arithmetic cores are based on the IEEE-754 standard and were validated using four different bit-width representations (including the simple and double precision formats) that were implemented, synthesized and simulated in order to obtain the area cost, elapsed time, power consumption and accuracy of the proposed circuits.

A. Synthesis results

The floating-point cores have been described in VHDL hardware description language using the Xilinx ISE 10.1 development tool [29]. Table I presents the synthesis results of the arithmetic operators using a Xilinx Virtex5 FPGA family (device xc5vlx110T).

Table I. Synthesis results.

Bit-width (Exp,Man)	Floating-point core	FF 69120	LUTs 69120	DSP48Es 64	Freq. MHz
24 (6,17)	Add/sub	53	561	8	201.11
	Multiplier	27	59	5	587.98
	Division GS	197	372	10	153.14
	Division NR	173	279	11	120.59
	Square root GS	206	344	14	141.73
	Square root NR	128	239	13	144.42
32 (8,23) single precision	Add/sub	69	962	8	184.58
	Multiplier	35	74	5	576.45
	Division GS	257	496	10	153.14
	Division NR	227	375	11	120.59
	Square root GS	270	448	14	141.73
	Square root NR	168	313	13	143.94
43 (11,31)	Add/sub	91	1420	8	184.19
	Multiplier	46	95	7	569.17
	Division GS	338	648	14	102.82
	Division NR	300	482	17	87.04
	Square root GS	356	646	20	97.50
	Square root NR	222	407	22	98.53
64 (11,52) double precision	Add/sub	134	3150	6	185.67
	Multiplier	67	250	15	568.10
	Division GS	527	1256	29	78.65
	Division NR	468	1114	40	70.10
	Square root GS	568	1907	50	60.16
	Square root NR	352	1527	62	60.50

The area cost in registers (FF), Look-up Table (LUTs) and embedded DPS blocks as well as the performance (given in Mega-Hertz) are presented for different bit-width representations. A *Goldschmidt's* implementation is represented by *GS* and a *Newton-Raphson* implementation is represented by *NR*. These results were achieved by using 16-addressable words LUT in the cases of division and square root for both the *Goldschmidt's* and *Newton-Raphson* architectures.

As expected, large bit-width representations have a higher area cost and low performance than the small bit-width representations. The synthesis tool reports a high performance for the multiplier core (around 568 MHz). However it is important to take into account that the large clock frequency supported by the FPGA device is 500MHz. The *GS* algorithm achieves best operational frequency than the *NR* algorithm for division operator. It can be explained due to the two parallel approaches for implementing equations (1b) and (1c). The *GS* and *NR* algorithms present similar performance for square root operator.

Figures 4, 5 and 6 show that the area cost has an exponential behavior with the bit-width representation. All the floating-point cores are feasible implemented in terms of registers (Flip-flops), LUTs, and DSP48E blocks.

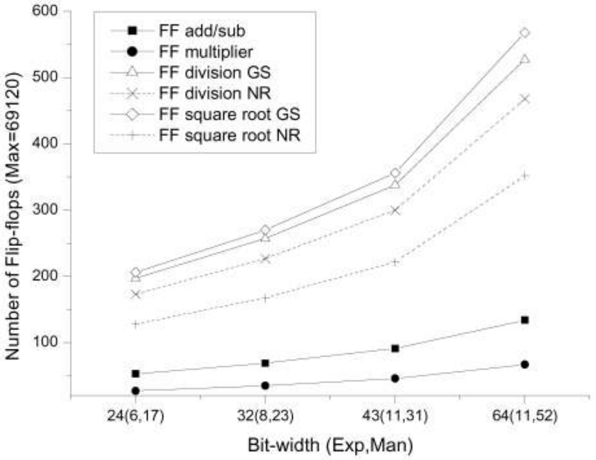

Figure 4. Flip-flops consumption

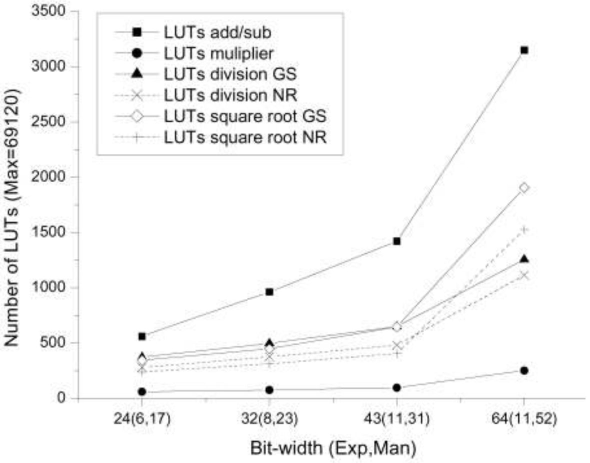

Figure 5. LUTs Consumption

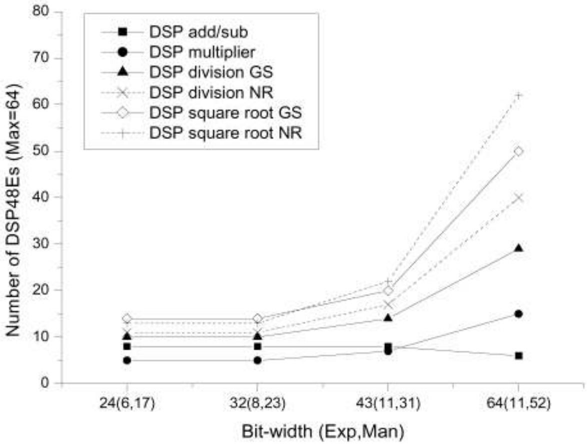

Figure 6. DSP48Es consumption

It can be observed that the addition/subtraction core is more expensive than the other operators. A comparison between the experimental cost in area associated to the *Goldschmidt's* and *Newton-Raphson* implementations for division and square root have demonstrated that the first algorithm requires more logic area than the second one.

However, the *NR* algorithm requires to perform more fixed-point multiplications and as a consequence more dedicated DSP blocks are used. It can be observed that in the case of a double precision format (64-bit), the square root core consumes a significant number of DSP blocks, specifically for *Newton-Raphson* implementation, limiting the hardware resources for applications where a large number of arithmetic operations are required.

This problem can be overcome by implementing the floating-point multipliers using the available logic area. However, in this case a low performance penalty must be considered due to the fact that the implemented multipliers over the Configurable Logic Blocks will have lower performance than the dedicated ones (which are implemented in the DSP blocks).

The XC5VLX110T is not the largest device from the Virtex5-LX FPGA family. However, according to the synthesis results, there are around 97% of the available FF and LUTs to implement other hardware components or pipeline architectures. Although large bit-width representations allow high precision computations, the large area cost associated to the floating-point solution is a critical problem if compared with fixed-point approaches [30], [31], [32], [33].

As shown in Section 3, the developed architectures for division and square root are based on iterative algorithms. Therefore, in order to analyze the tradeoff between area cost and error associated in the FPGA design of floating-point arithmetic cores is important to address simulation results for different bit-width and number of iterations.

B. Simulation results

The implemented operators were simulated using the ModelSim 6.3g simulator tool [16] after the *placement and routing* (PAR) process. A simulation environment to validate the behavior of the floating-point units was developed in Matlab® R2008a. One hundred random floating-point test vectors for each bit-width representation were used for each experiment and afterward the same were addressed as inputs of each arithmetic operator. The binary floating-point results were analyzed in the simulator environment in order to calculate the MSE of the implemented cores, for which the Matlab® results (which operates in a double precision format) were used as an statistical estimator.

Table II and III present the MSE and maximum absolute error achieved for each arithmetic core using different bit-width representations. Addition/ subtraction, multiplication and division cores were evaluated using 100 random input values between -1000 and 1000 and the square root core was tested using 100 random input values between 0 and 1000. The division and square root operators were calculated using 5 iterations and a 16-addressable words look-up table. As expected the best error results were achieved in the case of a double precision format and large errors are presented when using the 24 bit-width representation, specifically for the multiplier core, in which errors are introduced due to truncation problems.

Table II. MSE results. 16 LUT size and 5 Iterations

Floating-point Core	Bit-width (Exp,Man)			
	24 (6,17)	**32 (8,23)**	**43 (11,31)**	**64 (11,52)**
Add/sub	2.27E-07	2.76E-11	3.24E-15	1.26E-17
Multiplier	9.53E-04	1.53E-07	1.49E-11	5.87E-16
Division GS	9.27E-11	3.71E-12	3.16E-23	1.99E-25
Division NR	5.10E-11	3.82E-13	8.70E-24	1.99E-25
Square root GS	2.91E-08	8.62E-12	3.31E-20	1.94E-24
Square root NR	3.04E-08	8.48E-12	8.86E-21	1.94E-24

Table III. Maximum absolute error. 16 LUT size and 5 Iterations

Floating-point Core	Bit-width (Exp,Man)			
	24 (6,17)	**32 (8,23)**	**43 (11,31)**	**64 (11,52)**
Add/sub	1.02E-02	1.70E-04	1.15E-06	9.30E-07
Multiplier	9.09E-02	1.43E-03	1.19E-05	6.83E-06
Division GS	7.23E-04	6.32E-06	5.34E-07	1.90E-08
Division NR	5.15E-04	2.02E-06	2.95E-07	1.90E-08
Square root GS	7.60E-04	1.39E-5	6.32E-08	1.89E-08
Square root NR	4.91E-04	7.38E-06	3.00E-08	1.89E-08

The simple extended precision (43-bit implementation) achieves satisfactory results taking advantage of a small area cost which allows for increasing the throughput and performance, being suitable for a wide range of embedded applications.

Figure 7 summarizes the best MSE values achieved for different bit-width representations. It can be observed that the MSE for the multiplier and the add/sub cores is bigger than for the MSE of the division and the square root operators. This behavior can be explained because the successive refinements performed by *Goldschmidt's* and *Newton-Raphson* algorithms for computing the division and the square root accelerate convergence (and thus precision) of the computations based on these methods.

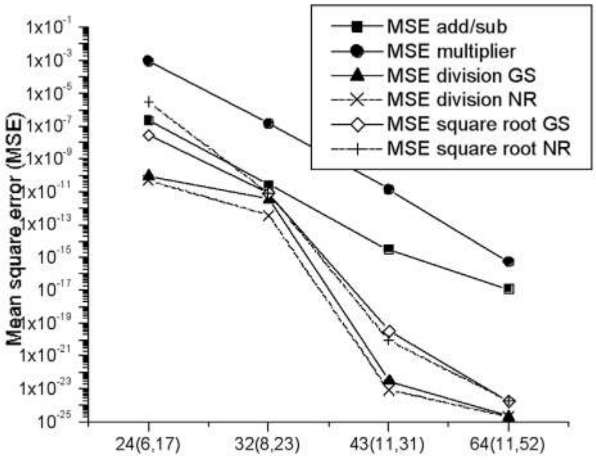

Figure 7. MSE order of magnitude

The MSE and area cost for division and square root operators can be slightly adjusted by modifying the size of the LUT to store the initial seeds. Table IV shows the MSE achieved for different size of look-up table, using a simple precision implementation (32 bits). The division core achieves the lower MSE using a LUT of 8 and 16-addressable words for *Goldschmidt's* and *Newton-Raphson* implementations respectively. The MSE for square root presents similar behavior for both *Goldschmidt's* and *Newton-Raphson* architectures, achieving the lower order of magnitude by using a LUT of 16-addressable words. It is observed that the LUT size does not affect significantly the MSE value; however, it can increase the area cost large enough to affect the performance of the arithmetic operator itself [14].

Table IV. MSE for variable LUT sizes and 3 iterations

Core	LUT size	MSE Goldschmidt's	MSE Newton-Raphson
Division	4	3.63E-11	3.41E-11
	8	3.67E-13	1.19E-11
	16	3.71E-12	3.82E-13
	32	3.87E-12	3.81E-13
	64	1.57E-12	1.57E-12
Square root	4	2.14E-11	1.81E-11
	8	8.62E-12	8.48E-12
	16	7.45E-12	6.71E-12
	32	7.24E-12	6.25E-12
	64	1.24E-11	5.93E-12

Table V. MSE and elapsed time in clock cycles (cc) for variable itera-tions (NI) and a 32-bits implementation.

NI	Goldschmidt's				Newton-Raphson			
	division	**cc**	**sqrt**	**cc**	**division**	**cc**	**sqrt**	**cc**
3	3.7E-13	10	8.62E-12	25	3.82E-13	11	8.48E-12	16
5	7.7E-12	14	2.70E-11	34	3.72E-13	15	6.96E-12	24
7	3.4E-11	18	3.63E-11	43	3.72E-13	19	6.31E-12	32
9	8.1E-11	22	3.63E-11	52	3.72E-13	23	6.30E-12	40
11	1.5E-10	26	3.63E-11	61	3.72E-13	27	6.96E-12	48

Taking into account the results presented in Tables II, III and IV the best tradeoff between area cost and error associated can be achieved by using a single format or single extended format, a LUT of 8-addressable words in the case of the division using the *Goldschmidt's* algorithm and a LUT of 16-addressable words for the other cases. Therefore, these conditions can be used for addressing several simulations in order to analyze the tradeoff between the elapsed time against the error associated.

Table V shows the MSE behavior for different number of iterations, denoted by *NI,* using the best conditions described above. It can be observed that the lower order of magnitude of MSE is achieved for 3 iterations in all cases. The MSE grows slowly due to the round error propagated from the initial approximation. The error due to round-off after *i* iterations may be slightly over *i/2* ulp (Unit of Least Precision) [27]. Additionally, the elapsed time given in clock cycles and denoted by *cc*, is presented for the *Goldschmidt's* and *Newton-Raphson* algorithms computing the division and square root operators. As expected, the number of clock cycles increases linearly with the number of iterations. The square root operator requires more clock cycles than the division operator.

High-performance embedded systems require to operate not only with high frequency and high accuracy computations, but also with a low power consumption. Therefore, one important design criteria in the hardware implementation of digital circuits is the power consumption. Table VI presents the power consumption estimation for the developed architectures for different bit-width representations. Each one of the arithmetic operators were simulated (post-placement and routing process) using the same testbench to generate the respective Value Change Dump (VCD) files, which were used to provide stimulus data and toggle rates as input model of the Xpower Analyzer® tool (XPA) [34]. The XPA tool determines the total quiescent device power and dynamic power consumption. The quiescent power is represented by the addition of two main components, (1) the power consumed by the device when it is powered up without programming the user logic and (2) the power consumed by the user logic when the device is programmed and without any switching activity. On the other hand, the dynamic power consumption represents the power consumed when there is switching activity.

According to the power consumption estimation, the multiplier core requires more quiescent power than the addition/subtraction core, even requiring a less dynamic power. As expected, the double-precision format (64-bit) requires more power than the other bit-width representations. A comparison between the *Goldschmidt's* and *Newton-Raphson* architectures shows that both algorithms present similar quiescent power consumption. However, the first one requires more dynamic power for the division core, where as the *Newton-Raphson* approach requires more dynamic power consumption for the square root operator.

There is a tradeoff between four main variables, namely, area cost, error, elapsed time and power consumption, which can be improved by selecting a suitable value for three parameters, namely: bit-width precision, number of iterations and size of look-up table. The tradeoff analysis between the bit-width against cost in area and experimental error associated have pointed out that the *Newton-Raphson* architecture performs better than the *Goldschmidt's* one. The *Newton-Raphson* implementation achieves small error results than the *Goldschmidt's* implementation and requires less logic area and DSP blocks. Table VII shows a qualitative analysis among the three design parameters and the main variables for circuit design of arithmetic operators.

Table VI. Power consumption (Watts)

Floating-point Core	Bit-width (Exp,Man)				
	Power	24 (6,17)	32 (8,23)	43 (11,31)	64 (11,52)
Add/sub	Quiescent	0.959	0.959	0.959	0.961
	Dynamic	0.011	0.011	0.016	0.033
	Total	0.970	0.970	0.975	0.994
Multiplier	Quiescent	1.095	1.095	1.095	1.095
	Dynamic	0.006	0.006	0.009	0.013
	Total	1.101	1.101	1.104	1.108
Division Goldschmidt's	Quiescent	0.960	0.960	0.960	0.964
	Dynamic	0.018	0.023	0.021	0.062
	Total	0.978	0.983	0.981	1.025
Division Newton-Raphson	Quiescent	0.959	0.959	0.960	0.963
	Dynamic	0.013	0.016	0.023	0.059
	Total	0.972	0.975	0.983	1.023
Sqrt Goldschmidt's	Quiescent	0.960	0.960	0.960	0.964
	Dynamic	0.017	0.017	0.025	0.065
	Total	0.977	0.977	0.985	1.029
Sqrt Newton-Raphson	Quiescent	0.960	0.960	0.962	0.977
	Dynamic	0.016	0.020	0.050	0.218
	Total	0.976	0.980	1.012	1.195

Table VII. Qualitative tradeoff dependence

Parameter / variable	Area cost	Error	Elapsed time	Power
Bit-width pre-cision	High	High	Not depend	High
Number itera-tions	Not depend	High	High	Not depend
LUT size	Low	Low	Low	Low

6. CONCLUSIONS

This work describes an FPGA implementation of a floating-point library for arithmetic operators, including addition/subtraction, multiplication, division and square root. The floating-point cores for division and square root were developed using the *Goldschmidt's* and *Newton-Raphson* algorithms. The implemented cores are based on the IEEE-754 format and were described in VHDL.

A tradeoff analysis was performed allowing the designer to choose, for general purpose applications, the suitable bit-width representation and error associated, as well as the area cost, elapsed time and power consumption for each arithmetic operator. The addition/subtraction core requires more hardware resources than the multiplier core. However the last one consumes more embedded DSP blocks for large bit-width representations. The addition core requires more dynamic power consumption than the multiplier; however, due to the large DSP blocks required by the multiplier it is more expensive in terms of quiescent power consumption. Although the MSE presents similar behavior for both, *Goldschmidt's* and *Newton-Raphson* iterative architectures, the main advantage of the *Newton-Raphson* architecture is the lower hardware resources consumption (Flip-flops and LUTs) where as the *Goldschmidt's* architecture requires less dynamic power consumption. The tradeoff between bit-width representation and error associated demonstrates that the LUT size does not affect significantly the MSE value. However, it can increase the area cost large enough to affect the performance of the arithmetic operator itself.

Synthesis results have demonstrated that FPGAs are a feasible solution for implementing floating-point arithmetic operators. The logic area consumption is satisfactory in all the cases allowing the algorithms to be implemented using a pipeline approach. However, the large number of dedicated DSP blocks required by large bit-width representations is a critical parameter. Satisfactory performance results were achieved for all the arithmetic cores. The square root cores have the lower performance, around 60 MHz for a double-precision implementation and around 143 MHz for a single-precision implementation.

As future works we intend to include rounding methods for improving the accuracy of the arithmetic operators. Also, an optimized compromise between the tradeoff variables can be performed in order to minimize the area cost, error, elapsed time and power consumption of the arithmetic operators.

REFERENCES

[1] S. Kilts, *Advanced FPGA Design, Architecture, Implementation and Optimization,* John Wiley & Sons, New Jersey, United States: 2007, pp. 117-139.

[2] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays,* Springer, Berlin, Germany: 2001, pp. 29-79.

[3] Y. Kung, K. Tseng, H. Sze and A. Wang, "FPGA implementation of inverse kinematics and servo controller for robot manipulator," in *Proceedings of the IEEE Conference on Robotics and Biomimetics,* 2006, pp. 1163-1168.

[4] M. Ibne, S. Islam and M. Sulaiman, "Pipeline floating point ALU design using VHDL," in *Proceedings of the IEEE Conference on Semiconductor Electronics,* 2002, pp. 204-208.

[5] S. Hauck and A. Dehon, *Reconfigurable Computing. The Theory and Practice of FPGA-based Computing,* Elsevier, Burlington, United States: 2008, pp. 62-63.

[6] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," in *Proceedings of the IEEE Conference on Evolutionary Computation,* 2009, pp. 1493-1500.

[7] L. Veronese and R. Krohling, "Swarms's Flight: Accelerating the particles using C-CUDA," in *Proceedings of the IEEE Conference on Evolutionary Computation,* 2009, pp. 3264-3270.

[8] G. Govindu, R. Scrofano and V. Prasanna, "A Library of parameterizable floating-point cores for FPGAs and their application to scientific computing," in *Proceedings of the IEEE Conference on Engineering of Reconfigurable Systems and Algorithms,* 2005, pp. 137-148.

[9] B. Lee, and N. Burgess, "Parameterizable Floating-point operations on FPGA," in *Proceedings of the Conference on Signals, Systems and Computers,* 2002, pp. 137-148.

[10] X. Wang, *Variable Precision Floating-point Divide and*

Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithm, Doctoral Thesis, Northeastern University, 2007.

[11] T. Kwon, J. Sondeen and J. Draper, "Floating-Point division and square root implementation using a Taylor-series expansion algorithm with reduced look-up tables," in *Proceedings of the IEEE Symposium on Circuits and Systems,* 2008, pp. 954-957.

[12] C. Shuang-yan, W. Dong-hui, Z. Tie-jun and H. Chao-huan, "Design and implementation of a 64/32-bit floating-point division, reciprocal, square root, and inverse square root unit," in *Proceedings of the IEEE Conference on Solid-Stated and Integrated Circuit Technology,* 2006, pp. 1976-1979.

[13] X. Wang and B. Nelson, "Trade-off of designing floating-point division and square root on Virtex FPGAs," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines,* 2003, pp. 195-203.

[14] D. Sánchez, D. Muñoz, C. Llanos, and M. Ayala-Rincón, "Parameterizable floating-point library for arithmetic operations in FPGAs," in *Proceedings of the ACM Symposium on Integrated Circuits and System Design,* 2009, pp. 253-258.

[15] IEEE standards Board, "IEEE standard for binary floating-point arithmetic," Technical Report *ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronic Engineers*, 1985.

[16] ModelSim, 2009, [Online]: <http://www.model.com>

[17] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," in *IEEE Transactions on VLSI Systems*, vol. 2, no. 3, 1994, pp. 365-367.

[18] L. Louca, T. Cook and W. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *Proceedings of the IEEE Symposium on FPGAs Custom Computing Machines*, 1996, pp. 107-116.

[19] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers and K. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 1998, pp. 206-215.

[20] M. Beauchamp, S. Hauck, K. Underwood and K. Hemmert, "Embedded floating-point units in FPGAs," in *Proceedings of the ACM Symposium on Field Programmable Gate Arrays*, 2006, pp. 12-20.

[21] J. Liang, R. Tessier and O. Mencer "Floating point unit generation and evaluation for FPGAs," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines,* 2003, pp. 185-194.

[22] K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-point performance," in *Proceedings of the ACM Symposium on Field Programmable Gate Arrays*, 2004, pp. 171-180.

[23] Y. Li and W. Chu, "Implementation of single precision floating point square root on FPGAs," in *Proceedings of the IEEE Symposium on Custom Computing Machines*, 1997, pp. 226-232.

[24] P. Montuschi and P. Mezzalama, "Survey of square rooting algorithms," in *IEEE Transactions on Computers and Digital Techniques*, vol. 137, no. 1, 1990, pp. 31-40.

[25] Processor Architecture, NIOS II Reference Handbook, 2008, [Online]: <http://www.altera.com>

[26] R. Goldschmidt, *Applications of division by convergence*, Master's Thesis, Massachusetts Institute of Technology, 1964.

[27] P. Markstein, "Software division and square root using Goldschmidt's algorithms," in *Proceedings of the Conference on Real Nambers ans Computers*, 2004, pp. 146-157.

[28] P. Soderquist and M. Leeser, "Division and square root choosing the right implementation," in *IEEE Micro,* vol. 17, no. 4, 1997, pp. 56-66.

[29] ISE 10.1 Development tool, Quick Start Tutorial, 2009, [Online]: <http://xilinx.com>

[30] G. Sutter, J. Deschamps, "High speed fixed point dividers for FPGAs," in *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*, 2009, pp. 448-452.

[31] J. Piromsopa, C. Aporntewan and P. Chongsatitvatana, "An FPGA implementation of a fixed-point square root operation," in *Proceedings of the IEEE Symposium on communication and Information Technologies*, 2001, pp. 587-589.

[32] J. Mailloux, S. Simard, R. Beguenane, "Implementation of division and square root using XSG for FPGA-based vector control drivers," in *IEEE Transactions on Electrical and Power Engineering*, vol. 5, no. 1, 2007, pp. 524-529.

[34] J. Deschamps, G. Bioul and G. Sutter, *Synthesis of Arithmetic Circuits*, John Wiley & Sons, New Jersey, United States: 2006, pp. 109-165.

[35] Xpower Tutorial: FPGA Design, 2009 [Online]: <ftp://ftp.xilinx.com/pub/documentation/tutorials/>