High-Performance Hardware Operators for Polynomial Evaluation

Arnaud Tisserand

High-performance hardware operators for polynomial evaluation

Arnaud Tisserand

Laboratory of Computer Science, Robotics and Microelectronics (LIRMM),
National Center for Scientific Research (CNRS),
University of Montpellier 2,
161 rue Ada,
Montpellier cedex 5, 34392, France
E-mail: arnaud.tisserand@lirmm.fr

Abstract: This paper presents some recent works on hardware evaluation of functions. A method for the automatic generation of high-performance arithmetic operators based on polynomial approximations is described. It deals with the bit-level representation of the polynomial coefficients, the intermediate computations width, the approximation and the rounding errors. The generated operators are small, fast and numerically validated at design time. Some examples have been implemented on Field Programmable Gate Arrays (FPGAs).

Keywords: arithmetic operator; hardware operator; function approximation; polynomial evaluation; polynomial coefficient; approximation error; rounding error.

Reference to this paper should be made as follows: Tisserand, A. (2007) 'High-performance hardware operators for polynomial evaluation', *Int. J. High Performance Systems Architecture*, Vol. 1, No. 1, pp.14–23.

Biographical notes: Arnaud Tisserand is a Senior Researcher at CNRS (National Center for Scientific Research) in LIRMM (Laboratory of Computer Science, Robotics and Microelectronics), Montpellier, France. He received a PhD (1997) in Computer Science in ENS Lyon, France. He was (1997–1999) a Research Expert at CSEM (Swiss Center for Electronics and Microtechnology) in Neuchâtel, Switzerland. He was (1999–2005) a Researcher at INRIA (National Institute for Research in Computer Science and Control) and LIP (Laboratory of Computer Science) in Lyon, France. His research interests include computer arithmetic, computer architecture, VLSI and FPGA design, low-power design and applications in scientific computing, digital signal processing, multimedia, digital control and cryptography.

1 Introduction

This paper presents some recent works on hardware evaluation of functions. A method for the automatic generation of high-performance arithmetic operators based on polynomial approximations is described. Some parts of this paper have been presented by Michard et al. (2006) and Tisserand (2006).

The design of high-performance arithmetic operators is an important issue in Application Specific Integrated Circuits (ASICs), Systems on Chip (SoCs) and Field Programmable Gate Arrays (FPGAs) implementations. Basic operations such as addition/subtraction or multiplication have always been implemented as high-performance operators in digital circuits (see Ercegovac and Lang, 2003). Some recent applications require fast evaluation of more complex operations such as division, reciprocal, square-root, inverse square-root, trigonometric functions, logarithm or exponential. Evaluation of compositions of such functions is also required.

Function approximation is often performed using polynomial evaluation in software as well as in hardware implementations. For instance, elementary functions (sine, cosine, exponential, logarithm, arc-tangent, etc.) are often evaluated using polynomials (see Muller, 2006). In some digital-signal processing applications, such as frequency demodulation, low degree polynomials are often used for evaluating reciprocals. Other algebraic functions, such as square root or square root reciprocal can be efficiently approximated using polynomials.

In this paper, a method that produces polynomial approximations 'well-suited' for high-performance hardware implementations is described. It faces with two problems: the generation of the polynomial coefficients that ensure low approximation errors and the sizing of intermediate computations that provide low round-off errors. The complete method has been implemented as an automatic code generator. The generated operators are small and fast. They are also numerically validated at compile time. The method determines the maximum total error (approximation and round-off) possible for each generated operator and set of parameter.

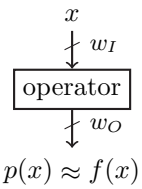
This paper is organised as follows. Notations and background on function approximation are presented in Section 2. Previous works are summarised in Section 3. The generation method is described in Section 4. Section 5

illustrates the method on several examples implemented on FPGAs. Section 6 concludes this paper.

2 Notations and background

The target function is f with input and output in fixed-point format (2's complement notation). The function f is approximated using the degree-d polynomial p. The argument x is in the domain $[a, b]$ and the result $p(x)$ is in the range $[a', b']$. Extension to other forms of input/output intervals such as $[a, b]$ is straightforward and is not considered here. Here, we consider function evaluation without range reduction (see Muller, 2006). The generation method below can be used in more complex schemes including range reduction. The argument x is a w_I-bit number and the output $p(x)$ is a w_O-bit number. The input argument x is considered as exact. The position of the binary point (i.e. the number of integer bits) is fixed by the format of the smallest representation that include both a and b. The number of fractional bits will be computed by the generation method to fit the target accuracy requirements. The polynomial coefficients are denoted p_0, p_1, p_2, ..., p_d, then $p(x) = \sum_{i=0}^{d} p_i x^i$. The polynomial coefficients are represented using 2's complement or borrow-save notation (see below). Basic notations are summarised in Figure 1.

Figure 1 Operator notations



Several evaluation schemes may be used to compute the value $p(x)$ in practice. In this work we only consider the *direct* and *Horner* evaluation schemes:

$$p(x) = \begin{cases} p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d & \text{direct} \\ p_0 + x\,(p_1 + x\,(p_2 + x(\cdots + x p_d)\cdots)) & \text{Horner} \end{cases}$$

The Horner scheme is implemented using basic blocks called Fused Multiply and Adds (FMAs). A FMA computes $uv + w$ with more or less the same cost (delay and circuit area) than a single multiplication uv. The evaluation schemes differ on several aspects: computation cost, internal parallelism and accuracy. The direct scheme leads to a cost of d additions and $d + \lceil \log_2 d \rceil$ multiplications while the Horner scheme only requires d additions and d multiplications. The Horner scheme is a sequential structure while the direct scheme allows some internal parallelism (at the monomial level). From the accuracy point of view, the Horner scheme is known to be slightly more accurate than the direct scheme (see Higham (2002) for the accuracy analysis in scientific computing applications). Our results also show that the Horner is slightly more accurate than the direct one (see Section 5).

The numerical quality of the approximation to f using the polynomial p deals with two components: the *approximation error* and the *round-off error*. The approximation error measures the distance between the mathematical function f and the function used for the approximation, here the polynomial p. The approximation error is also called the *method error*. In order to measure the theoretical approximation error ϵ_{app} due to the use of the polynomial p to approximate the function f on $[a, b]$, we use the distance:

$$\epsilon_{\text{app}} = \|f - p\|_\infty = \max_{a \le x \le b} |f(x) - p(x)| \tag{1}$$

Here, the value $p(x)$ is the mathematical value computed using an infinite precision. The approximation error ϵ_{app} is the smallest theoretical error that can be obtained using the polynomial p for approximating f. Due to the finite precision of the coefficients and intermediate computations during the practical evaluation of p, we will have to deal with larger errors. In this work, the value of ϵ_{app} is numerically estimated using the Maple infnorm command. We assume that the approximated functions are 'smooth' enough to ensure that infnorm result is correct (see Muller, 2006). Furthermore, we will use overestimations of ϵ_{app} to ensure that the generated operators fit the target accuracy.

The polynomial approximations used in the following are based on the *minimax* polynomial approximation as a starting point. The degree-d minimax polynomial approximation to f on $[a, b]$ is the polynomial p^* that satisfies:

$$\|f - p^*\|_\infty = \min_{p \in \mathcal{P}_d} \|f - p\|_\infty \tag{2}$$

where $\mathcal{P}_d$ is the set of polynomials with real coefficients and degree at most d. Minimax approximations can be computed thanks to an algorithm due to Remes (1934). More details about minimax approximations may be found by Muller (2006). In this work, the minimax polynomials are numerically computed using the Maple minimax command.

The *round-off error* or *rounding error* due to the discrete nature of the intermediate and final values adds up to the approximation error. It is the difference between the calculated approximation of a number and its exact mathematical value. This error is small for one single operation, that is a fraction of the weight of the Least Significant Bit (LSB). But during a sequence of operations, these small errors may accumulate themselves and significantly degrade the accuracy of the final result. In order to limit the rounding error, we introduce g additional *guard bits* for the intermediate computations (i.e. they are done on words of $w_O + g$ bits).

In the following, errors are expressed directly or as equivalent accuracy. The accuracy is the number of correct or significant bits. The relation between the error ϵ and the accuracy μ is $\mu = -\log_2 |\epsilon|$. For instance, the error $\epsilon = 0.0000107$ is equivalent to an accuracy of $\mu = 16.5$ correct or significant bits.

The notation $()_2$ denotes the binary representation of a value, for example, $3.125 = (11.001)_2$. The polynomial coefficients will be represented in the *borrow-save* format (see Ercegovac and Lang, 2003), that is radix-2 redundant representation with the digit set $\{-1, 0, 1\}$. Borrow-save representation is denoted $()_{\text{bs}}$. Bits with a negative weight are denoted by $\overline{1}$. In the borrow-save fixed-point format with k integer bits and l fractional bits, values are represented by $\sum_{i=-l}^{k} v_i 2^i$ with $v_i \in \{\overline{1}, 0, 1\}$. This representation avoids

long strings of ones, for example, $63.0 = (100000\overline{1}.0)_{bs}$ (see Ercegovac and Lang (2003) for examples of such representations in multipliers or other arithmetic operators). As we deal with hardware implementations, multiplications by powers of 2 reduce to shifts (only routing).

3 Previous works

A wide variety of algorithms for the approximation of functions has been proposed. Among them table based methods (Section 3.1), polynomial or rational approximations (Section 3.2), digit-recurrence algorithms (Section 3.3) and combinations of these solutions are the most commonly used. This section presents some of the main previous works on arithmetic operators dedicated to function approximation in hardware. More detailed presentations may be found by Ercegovac and Lang (2003) for computer arithmetic and by Muller (2006) and Markstein (2000) for elementary functions evaluation algorithms. In Section 3.4, this section ends with some methods and tools proposed to bound the total error (approximation and round-off) in arithmetic operators.

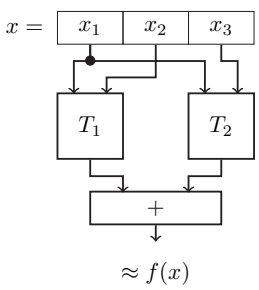
3.1 Table based methods

One of the first methods proposed to approximate functions was to tabulate the values of the target function for each possible input. Obviously, this solution is very limited due to the exponential size of the table: 2^{w_I} words of w_O bits. In practice the maximum number of address bits of the table w_I is in the range 8–12 depending on the technology.

Table-based methods are frequently used in FPGAs because these devices are very well suited for the implementation of tables as they are composed on small programmable memories. For instance, Xilinx FPGAs are based on basic cells composed of two 4 input bits Lookup Table (LUTs) and some programmable logical resources. Those LUTs can be used to implement arbitrary logic functions or very small memories.

Higher accuracies can be reached by combining tables and arithmetic operations. Architectures based on tables and additions/subtractions are very common for function approximation (see de Dinechin and Tisserand (2005) for a small survey on table-and-addition methods). The bipartite method uses two tables and only one final addition as illustrated on Figure 2.

Figure 2 Bipartite method architecture

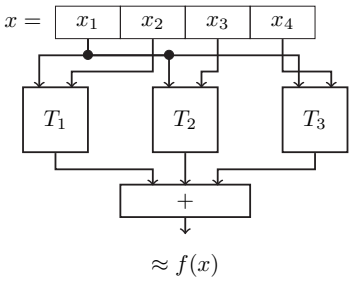


The bipartite method uses a decomposition of x into 3 subwords x_1, x_2 and x_3 of length w_1, w_2 and w_3, respectively such that $w_I = w_1 + w_2 + w_3$. The domain of x is decomposed into 2^{w_1} intervals. On each interval the function is approximated by 2^{w_2} affine segments with a constant slope. Table T_1 stores the initial value for each segment and table T_2 stores the offset to the initial value for each point of the segments (there are 2^{w_3} points per segment).

The bipartite table method leads to tables with a total of only $2^{2w_I/3}$ address bits compared to the 2^{w_I} address bits of the single table solution. The authors from Muller (1999) and Stine and Schulte (1999) have expressed the bipartite method in terms of Taylor approximation, which allows a formal error analysis. Some optimisations are possible at the architecture level. For instance in Schulte and Stine (1999), the authors have remarked that it is possible to exploit the symmetry of the segments on each small interval to halve the size of table T_2 using some additional XOR gates.

The bipartite table method has been extended to a number of tables larger than 2, it is called the multipartite method. It uses several tables looked-up in parallel and final addition of the all tables' contributions as illustrated on Figure 3. Multipartite table methods have been the subject of much recent attention (de Dinechin and Tisserand, 2005; Muller, 1999; Schulte and Stine, 1999; Stine and Schulte, 1999). They allow computing commonly used functions with low accuracy (up to 24 bits) with significantly lower hardware cost than that of a straightforward table implementation, while being faster than digit-recurrence algorithms or basic polynomial approximations.

Figure 3 Multipartite method architecture for three tables



Tables are also use to provide initial seeds to iterative methods such as the Newton-Raphson algorithm for division and square root which is commonly used in floating-point units of current processors (see Ercegovac et al., 2005). Other solutions propose to perform some precomputations, several parallel table lookups and a final large addition as in Wong and Goto (1995).

3.2 Polynomial or rational approximations

The main drawback of table based methods is the fact that the tables are dedicated to only one specific function. Sharing tables among several functions is not possible in practice. Some 'compression' is possible at the logical level, but the gain is very limited. Using polynomial or rational approximations, the basic operators (mainly adders and multipliers) required for the evaluation can be shared or reused. Indeed, one just needs to change the coefficients to perform an approximation to another function or to the same function but with a different domain. Based on this property, function approximation in software is often performed using polynomial or rational approximations (see Muller (2006) for a complete presentation).

Rational approximations are not frequently used in hardware due to the high latency of the final division. Some methods have been proposed to limit the cost of the final division. As an example in Ercegovac et al. (1995), rational approximations are evaluated using a digit-recurrence algorithm based on the E-method. In this work, we only deal with polynomial approximations.

Polynomial approximations are used in hardware since a long time (e.g. Burleson, 1990; Duprat and Muller, 1988). In hardware implementation, the size of the multipliers is a major concern. Several solutions have been investigated to limit their sise. In Pineiro et al. (2001) a method based on a degree-2 polynomial and a specialised squaring unit is proposed. This solution leads to 50% area savings compared to standard methods.

Several solutions based on modifications of polynomial approximations have been proposed in the literature. Weighted sum methods are examples of such modifications (see Hassler and Takagi, 1995; Johansson et al., 2006). The coefficients of the polynomial are distributed at the bit level. Thus, the polynomial is rewritten as the sum of a huge set of weighted products of the bits of x. Some of these terms are neglected. This method leads to interesting practical results for some functions.

Polynomial approximations are also combined with tables. A lot of methods based on this combination have been proposed. A method based on tables and a single multiplication is proposed by Detrey and de Dinechin (2004). In Ercegovac et al. (2000) a method based on argument reduction and series expansions is used for the evaluation of reciprocals, square roots, reciprocal square roots and some elementary functions using small multipliers and tables. A recent method based on polynomials and tables is proposed by Detrey and de Dinechin (2005).

We will see below that the determination of polynomials with coefficients exactly representable in the target format is one of the main problems in the implementation of polynomial approximations. A general but quite slow method dedicated to this problem is proposed by Brisebarre et al. (2006). This method has been used by Brisebarre et al. (2004) for the automatic generation of best polynomial approximations dedicated to hardware implementation. The generated polynomial approximations lead to high-speed and small hardware operators because of the use of sparse coefficients (i.e. they include fixed strings of zeros in the binary representation of the coefficients).

3.3 Digit-recurrence algorithms

Digit-recurrence algorithms, also called *shift-and-add* algorithms, produce one digit of the result every iteration starting from the Most Significant Digit (MSD). For instance when computing the division x/y, the paper and pencil method gives one decimal digit per iteration. A complete presentation of this kind of algorithm may be found in Ercegovac and Lang (2003) for division and square root operations. In Flynn and Oberman (2001) high-performance implementations of these algorithms are presented in the context of floating-point units.

Those algorithms seem to be simple at a very high level, but their practical implementation is not straightforward as it is demonstrated by the numerous works on this topic over the years. Some small mistakes can occur in the numerous equations and bounds computed during their optimisation. For instance the Pentium division unit has some small errors in a small selection table (see Edelman, 1997). The two most well known digit-recurrence algorithms are: the SRT algorithm for division, square root and other algebraic functions and the Coordinate Rotations on a Digital Computer (CORDIC) algorithm for elementary functions.

The SRT algorithm was invented independently by Sweeney, Robertson and Tocher at around the same time (see Robertson, 1958; Tocher, 1958). This algorithm leads to a small number of iterations using a high radix. It uses a redundant number system for the result digits produced at each iteration. This allows correcting some small errors from previous iterations due to the reduced internal precision. The combination of a redundant number system and reduced precision leads to very fast iterations. Many variations of these algorithms exist and it is very difficult to decide which one is best suited with respect to some constraints. Divgen is a fixed-point divider unit generator designed to explore the design space (see Michard et al., 2005). Given a set of parameters and options, it automatically generates an optimised VHDL description of the corresponding divider operator for some specific implementation target constraints.

Several digit-recurrence hardware operators have been developed in the past for more complex algebraic functions. For instance a reciprocal square root operator is presented by Takagi (2001). The case of an operator dedicated to the computation of the Euclidean norm of a 3D vector is presented by Takagi and Kuwahara (2000).

The CORDIC algorithm was introduced by Volder (1959) and extended by Walther (1971). It only uses additions and shift to approximate some elementary functions. Then it only produces one new digit of the result at each iteration. This algorithm is very well suited for low-area hardware implementations due the very simple resources required during the computation. For instance, it was used in some HP pocket calculators. A complete description of this algorithm and its numerous variations can be found by Muller (2006).

3.4 Errors bounds

We have seen in Section 2 that two kinds of error occur during the evaluation of a polynomial in a digital integrated circuit (processor, ASIC, SoC, FPGA, etc.). The approximation error due to the mathematical quality of the approximation and the round-off error due the finite precision computations.

One can deal with the approximation error by using more accurate approximations. This mainly corresponds to use higher degree minimax polynomials. In practice bounding the approximation error is not really difficult using tools such as the numapprox package in Maple.

But bounding the round-off error is a very complex task. This problem leads to a very active research field during the last years. In digital-signal processing some methods have been proposed to model round-off errors using noise (see Ménard and Sentieys, 2002). In scientific computing several tools have been developed for floating-point arithmetic. As an example, FLUCTUAT, presented by

Goubault et al. (2006) is a static code analyser which allows detecting some accuracy degradation in floating-point codes. The CADNA software, presented by Chesneaux et al. (2006), implements a stochastic arithmetic in order to analyse the average accuracy of floating-point programs (a fixed-point version is under development).

The GAPPA software, presented by Melquiond (2006), allows to evaluate and to produce a proof of mathematical properties on numerical codes. The main useful characteristic of GAPPA used in this work is its capability to tightly bound round-off errors and prove these bounds are below some threshold. GAPPA can generate a file for the Coq proof assistant (see The Coq Development Team, 2004).

Figure 4 presents an example of GAPPA input for a polynomial approximation to e^x on the domain $[1/2, 1]$. All the computations are supposed to be performed on 10-bit fixed-point format with 1 integer bit (noted 1Q9). The polynomial is

$$p(x) = \frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$$

Figure 4 GAPPA example

```
1 p0 = 571/512; p1 = 275/512; p2 = 545/512;
2 x = fixed<-9,dn>(Mx);
3 x2 fixed<-9,dn>= x * x;
4 p fixed<-9,dn>= p2 * x2 + p1 * x + p0;
5 Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
6 {Mx in [0.5,1] /\ |Mp-Mf| in [0,0.001385]
7   -> |p-Mf| in ? }
```

The first line specifies the coefficients (exactly representable in the target 1Q9 format). By convention in this work variables starting by a capital M denote mathematical values (with an infinite precision), all other variables denotes actual values in the circuit (input/output or intermediate registers). Line 2 specifies that x is the 'circuit version' of the mathematical argument Mx. The notation fixed<-9,dn> is a rounding mode declaration for fixed-point format with a LSB weight of 2^{-9} and rounding direction set to down dn (this is the standard truncation in the fixed-point format). Line 3 specifies that the variable x2 is the circuit result of the square x^2. Line 4 describes how the polynomial is computed in the circuit while line 5 describes its mathematical computation (with an infinite precision because there is no fixed<·,·> rounding mode). Lines 6 and 7 specify the target property (between { and }). The left hand side of the sign -> is the assumption set. Here there are two assumptions: $x \in [1/2, 1]$ and the fact that the approximation error is less 0.001385 (the approximation error is the distance between Mp and Mf). The right hand side of the sign -> asks to GAPPA in which interval (in ?) is the total error (the distance between the evaluated polynomial p and the mathematical function Mf). The total error includes both the approximation error and the round-off error. The result from GAPPA (version 0.7.2) is:

```
Results for Mx in [0.5, 1] and |Mp - F|
   in [0, 0.001385]: |p - F|
in [0, 232010635959353905b-64
   {0.0125773, 2^(-6.31303)}]
```

GAPPA recalls the assumptions and indicates that there is a bound for $|p - f|$ and the final accuracy is at least 6.31 correct bits.

In Figure 4, the polynomial is evaluated using the direct scheme. Figure 5 presents the required modifications to specify an evaluation using the Horner scheme. The final accuracy is then 6.57 correct bits.

Figure 5 Modification of GAPPA example from Figure 4

```
3 y1 fixed<-9,dn>= p2 * x + p1;
4 p  fixed<-9,dn>= y1 * x + p0;
5 Mp = (p2 * Mx + p1) * Mx + p0;
```

For a given argument x it is possible to measure the effective total error $\epsilon = f(x) - \texttt{output}(p(x))$ which includes all kinds of error. Here, $\texttt{output}(p(x))$ is the result of the evaluation of $p(x)$ by the circuit using finite precision computations. Indeed, for all possible values of x, the effective result is evaluated and compared to the theoretical value.

4 Generation method

The generation method described below has been presented by Michard et al. (2006) and Tisserand (2006). It provides an answer to two practical questions about implementation of polynomial approximation. The first one is 'What values should be used for the implemented coefficients?' The second one is 'What is the minimal size for the intermediate computations?' The generation method answers to the first question using coefficients that ensure low approximation errors and that are representable in the target format. The answer to the second one is provided by the use of intermediate computations with minimal size and that provide low round-off errors.

As our method does not explore all the parameter space corresponding to these two questions, the result may not be optimal. At the theoretical point of view, the optimal result is not known! In practice, the generation method provides very good results as reported in Section 5.

The input parameters of the method are:

- f the function to be evaluated
- $[a, b]$ the domain of the argument x
- the argument format x (size w_I bits)
- μ the total maximal target absolute error (the accuracy constraint).

The results from the method are:

- d the degree of the polynomial
- p_0, p_1, p_2,…, p_d the coefficients values (representable in the target format)
- n the coefficient size
- n' the data-path size.[1]

The generation method consists in three main steps and an optional step:

Step 1: Determination of the initial polynomial

Step 2: Coefficients optimisation

Step 3: Data-path optimisation

Step 4: Post-optimisations (optional).

Steps 1–4 are respectively described in Sections 4.1–4.4. There may be no result produced by a given step or the result is not considered 'good enough'. Then it is necessary to loop back to a previous step. This may lead to loops that are discussed in Section 4.5. Finally, a summary of the method is presented in Section 4.6.

4.1 Step 1: Determination of the initial polynomial

The first step defines a good starting point for the method. We use a minimax polynomial as a starting point (see Section 2) provided by the MAPLE minimax function. We look for the minimax polynomial p^* with the smallest degree d and accurate enough to approximate f on $[a, b]$ with an error less than μ, that is, $\epsilon^*_{\text{app}} < \mu$ with $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$. We start with $d = 1$ and d is incremented until p^* leads to an approximation error ϵ^*_{app} such that $\epsilon^*_{\text{app}} < \mu$.

Below is an example with $f = \log_2(x)$ and x in $[1, 2]$:

```
1 > minimax(log[2](x),x=1..2,[1,0],1,'err'); -log[2](err);
2   -.9570001094+1.000000000*x
3   4.537124583
4 > minimax(log[2](x),x=1..2,[2,0],1,'err'); -log[2](err);
5   -1.674903474+(2.024681754-.3448476634*x)*x
6   7.659796889
7 > minimax(log[2](x),x=1..2,[3,0],1,'err'); -log[2](err);
8   -2.15362071+(3.04788416+(-1.05187503+.158248704*x)*x)*x
9   10.61615211
```

Lines 1, 4 and 7 are the user commands (the greater than sign is the MAPLE prompt). Results are shown in italic. Line 1 means that we are looking for a degree-1 polynomial and the approximation error ϵ^*_{app} will be assigned to the variable err. Lines 2 and 3 are the results: the polynomial p^* and the number of correct bits corresponding to ϵ^*_{app}.

The first step provides:

- d the minimal degree of the approximation polynomial
- $p^*(x) = \sum_{i=0}^{d} p_i^* x^i$ the theoretical polynomial (real coefficients)
- ϵ^*_{app} the *minimal* theoretical error between the output of the circuit and f (using infinite accuracy for the coefficients and the computations).

The polynomial result p^* will be modified in the next steps. The next steps will degrade the accuracy (i.e. lead to errors larger than ϵ^*_{app}). Then some 'margin' between ϵ^*_{app} and μ is necessary as seen in the following sections.

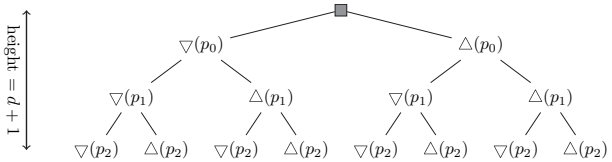
In Section 5.2, we will see that some variable changes may be useful in order to get values with similar magnitude order. This will avoid some shifts in the fixed-point computations. A standard modification occurs when evaluating $f(x)$ over $[a, b]$ with $a \neq 0$, then it may interesting to consider $f(x+a)$ over $[0, b-a]$. Some functions cannot be approximated using low degree polynomials, see (Muller, 2006, chap. 3) for the elementary functions for instance.

4.2 Step 2: Coefficients optimisation

In this step, we look for the size n and the values p_i of the coefficients in the target format such that $\epsilon_{\text{app}} < \mu$ where $\epsilon_{\text{app}} = ||f - p||_\infty$ and $p(x) = \sum_{i=0}^{d} p_i x^i$. Notice that n may be smaller than the width of the target format. As seen in Section 3.2 the choice of the coefficients is a complex problem.

The proposed solution is based on an exploration over the rounded coefficients. Each coefficient p_i^* may be rounded up $p_i = \triangle(p_i^*)$ or down $p_i = \triangledown(p_i^*)$. There are 2 choices per coefficient, then 2^{d+1} different polynomials to test as illustrated in Figure 6. For each possible polynomial p the value $\epsilon_{\text{app}} = ||f - p||_\infty$ is evaluated using the infnorm MAPLE function. In our applications d is small ($d \leq 6$) then the total exploration time is small.

Figure 6 Tested rounding modes for p^* of degree $d = 2$



We designed a MAPLE program that tests the 2^{d+1} polynomials corresponding to the different rounding modes of the $d + 1$ coefficients p_i^*. The program starts with $n = \lceil -\log_2 |\mu| \rceil$, tests all the rounding modes of the $d + 1$ coefficients for the current n size. If there are solutions (at least one), that is, polynomials such that $\epsilon_{\text{app}} < \mu$, it returns the list of those polynomials for the next step (those where ϵ_{app} is minimal). If there is no solution then n is incremented and all rounding modes tested.

In this step, we assume that all coefficients are represented using the same size n. Some post-optimisations allow using different sizes (see Section 4.4).

4.3 Step 3: Data-path optimisation

In this step, we look for the data-path size n'. Only two polynomial evaluation schemes are supported in the method: the direct and the Horner schemes (see Section 2). The idea is very simple. We start with $n' = n$ and we check that the data-path of size n' fulfils the accuracy constraint μ using GAPPA. If the total error bound computed by GAPPA is less than μ then this step is finished. If not, the size n' is incremented and the new data-path should be tested using GAPPA.

The difference between n' and n is called the number of *guard bits*. Using a coefficient size n smaller than n' allows a reduction in the memory size required to store the coefficients without degradation on the final accuracy.

4.4 Step 4: Post-optimisations (optional)

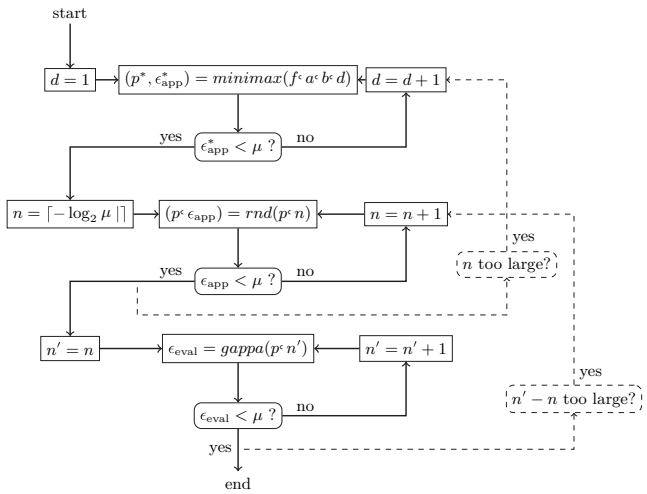
One kind of frequent post-optimisation is to simplify the hardware when some coefficients are very close to power of 2. As an example, if one coefficient from step 2 is the value $0.5002441406 = (0.100000000001)_2$ and the target accuracy μ is about 12 fractional bits, this coefficient may be rounded to 0.5. In that case, the multiplication by 0.5002441406 is replaced by a simple right shift

(see Section 5.2 for a concrete example of this kind of post-optimisation).

4.5 Loops

From a given step, it may be necessary to move back to a previous step in some cases. For instance, there may be no result from a given step or its result may not be considered 'good enough'. These possible loops correspond to the dashed lines in the Figure 7.

Figure 7 Summary of the generation method



As an example, the second step may not produce representable coefficients that ensure $\epsilon_{\text{app}} < \mu$. This case is due to insufficient margin between ϵ^*_{app} and μ in the first step. In that case, one should move back to step 1 and try with a higher degree polynomial (i.e. $d \leftarrow d + 1$).

Another standard loop occurs at step 3 when n' is considered too huge in front of n. In that case, it may be more efficient to move back to step 2 and try a larger n. By doing this, ϵ_{app} will be smaller which leads to more margin for round-off errors.

4.6 Summary of the method

The complete method is summarised in Figure 7. The method has been implemented in a collection of MAPLE and C++ programs.

5 Examples

The examples below have been implemented on Xilinx FPGAs XCV200-5 using ISE8.1i tools. Synthesis and place/route have been optimised for an area target with high effort. The reported results include all the resources required for the implementation (logic cells and registers).

5.1 Radix-2 exponential over $[0, 1]$

Here $f(x) = 2^x$ and $x \in [0, 1]$. The target accuracy is 12 bits, that is, $\mu = 2^{-12}$. We report the result from the first step for the theoretical minimax polynomial for degrees between 1 and 5. The corresponding accuracy is reported below in number of correct bits.

d	1	2	3	4	5
ϵ^*_{app}	4.53	8.65	13.18	18.04	23.15

Degree-1 and -2 theoretical minimax polynomials are not accurate enough with respect to the 12-bit target accuracy. Without the presented generation method a degree-4 polynomial would be required assuming worst case round-off error. Indeed for degree-3 one have $13.18 - d = 10.18 < 12$ while for degree 4 one have $18.04 - d = 14.04 > 12$. The values below represent all the rounding modes for the degree-4 solution and their accuracy. One can notice the large variation of the approximation errors for the rounding modes of the theoretical coefficients from 11.41 to 17.12 bits of accuracy.

$(\nabla, \nabla, \nabla, \nabla, \nabla)$	12.00	$(\nabla, \nabla, \nabla, \nabla, \triangle)$	13.00
$(\nabla, \nabla, \nabla, \triangle, \nabla)$	13.00	$(\nabla, \nabla, \nabla, \triangle, \triangle)$	14.03
$(\nabla, \nabla, \triangle, \nabla, \nabla)$	13.00	$(\nabla, \nabla, \triangle, \nabla, \triangle)$	14.55
$(\nabla, \nabla, \triangle, \triangle, \nabla)$	14.99	$(\nabla, \nabla, \triangle, \triangle, \triangle)$	13.00
$(\nabla, \triangle, \nabla, \nabla, \nabla)$	13.00	$(\nabla, \triangle, \nabla, \nabla, \triangle)$	*16.13*
$(\nabla, \triangle, \nabla, \triangle, \nabla)$	*17.12*	$(\nabla, \triangle, \nabla, \triangle, \triangle)$	13.00
$(\nabla, \triangle, \triangle, \nabla, \nabla)$	15.71	$(\nabla, \triangle, \triangle, \nabla, \triangle)$	13.00
$(\nabla, \triangle, \triangle, \triangle, \nabla)$	13.00	$(\nabla, \triangle, \triangle, \triangle, \triangle)$	12.00
$(\triangle, \nabla, \nabla, \nabla, \nabla)$	13.00	$(\triangle, \nabla, \nabla, \nabla, \triangle)$	13.00
$(\triangle, \nabla, \nabla, \triangle, \nabla)$	13.00	$(\triangle, \nabla, \nabla, \triangle, \triangle)$	13.00
$(\triangle, \nabla, \triangle, \nabla, \nabla)$	13.00	$(\triangle, \nabla, \triangle, \nabla, \triangle)$	13.00
$(\triangle, \nabla, \triangle, \triangle, \nabla)$	12.99	$(\triangle, \nabla, \triangle, \triangle, \triangle)$	12.00
$(\triangle, \triangle, \nabla, \nabla, \nabla)$	12.99	$(\triangle, \triangle, \nabla, \nabla, \triangle)$	12.98
$(\triangle, \triangle, \nabla, \triangle, \nabla)$	12.91	$(\triangle, \triangle, \nabla, \triangle, \triangle)$	12.00
$(\triangle, \triangle, \triangle, \nabla, \nabla)$	12.79	$(\triangle, \triangle, \triangle, \nabla, \triangle)$	12.00
$(\triangle, \triangle, \triangle, \triangle, \nabla)$	12.00	$(\triangle, \triangle, \triangle, \triangle, \triangle)$	11.41

Using the generation method we test the degree-3 solution. In that case, the theoretical minimax polynomial is:

$$p^*(x) = 0.9998929656 + 0.6964573949x + 0.2243383647x^2 + 0.0792042402x^3$$

Then $\epsilon_{\text{app}} = ||f - p^*||_\infty = 0.0001070344$, this corresponds to 13.18 bits of accuracy (assuming infinite precision for the coefficients and the computations). Due to the function, we consider a fixed-point format with one integer bit and $n - 1$ fractional bits. We report below the results from step 2 for several values of $n - 1$:

$n-1$	12	13	14	15	16
ϵ_{app}	12.38	12.45	13.00	13.00	13.02
# step 2 results	0	0	2	2	7

For the solution $n - 1 = 14$ bits, all the rounding modes possible in step 2 are reported below:

$(\nabla, \nabla, \nabla, \nabla)$	11.41	$(\nabla, \nabla, \nabla, \triangle)$	12.00
$(\nabla, \nabla, \triangle, \nabla)$	12.00	$(\nabla, \nabla, \triangle, \triangle)$	12.84
$(\nabla, \triangle, \nabla, \nabla)$	12.00	$(\nabla, \triangle, \nabla, \triangle)$	*13.00*
$(\nabla, \triangle, \triangle, \nabla)$	*13.00*	$(\nabla, \triangle, \triangle, \triangle)$	12.36
$(\triangle, \nabla, \nabla, \nabla)$	12.00	$(\triangle, \nabla, \nabla, \triangle)$	12.25
$(\triangle, \nabla, \triangle, \nabla)$	12.23	$(\triangle, \nabla, \triangle, \triangle)$	12.23
$(\triangle, \triangle, \nabla, \nabla)$	12.13	$(\triangle, \triangle, \nabla, \triangle)$	12.12
$(\triangle, \triangle, \triangle, \nabla)$	12.05	$(\triangle, \triangle, \triangle, \triangle)$	11.64

There are two possible solutions:

$$\frac{8191}{8192} + \frac{2853}{4096}x + \frac{1837}{8192}x^2 + \frac{649}{8192}x^3$$

and

$$\frac{8191}{8192} + \frac{2853}{4096}x + \frac{919}{4096}x^2 + \frac{649}{8192}x^3$$

Those polynomials lead to an approximation error equal to 0.0001220703 (13.00 bits of accuracy).

In step 3, we look for the data-path size of the operator. We report below the final accuracy evaluated using GAPPA for the first possible polynomial and for several sizes n':

n'	*14*	*15*	*16*	*17*	*18*	*19*	*20*
ϵ_{eval} Horner	11.32	11.93	12.36	12.65	12.81	12.90	12.95
ϵ_{eval} direct	11.24	11.86	12.32	12.62	12.79	12.89	12.94

The values obtained for the other polynomial ($p_2 = 919/4096$) are similar. Two solutions have been implemented. The first one corresponds to the standard solution (degree-4 polynomial, $n = n' = 18$). The second one is the result from the generation method ($d = 3$, $n - 1 = 14$, Horner evaluation scheme and $n' = 16$). The implementation results are reported in Table 1. The generation method leads to 17% smaller circuit and the degree-3 approximation saves 38% of the computation time.

Table 1 Implementation results 2^x over [0, 1]

Solution	*Area [slices]*	*Period [ns]*	*#Cycles*	*Delay [ns]*
Degree-3, $n' = 16$	193	21.9	3	65.7
Degree-4, $n' = 18$	233	26.9	4	107.6

5.2 Square root

For the second example $f(x) = \sqrt{x}$, $x \in [1, 2]$ and a target accuracy of 8 correct bits is fixed ($\mu = 2^{-8}$). The first step leads to $d = 2$ and $\epsilon^*_{\text{app}} = 0.0007638369$ which corresponds to 10.35 correct bits (for $d = 1$ the accuracy is only 6.81 correct bits).

The minimax polynomial is $p^* = 0.4456804579 + 0.6262821240x - 0.7119874509x^2$. The fixed-point evaluation of this polynomial requires some scaling in the fixed-point format. With x in $[1, 2]$, 2-integer bits are required for x^2 while the other operations only require 1-integer bit. In order to avoid this scaling problem, we now consider $f(x) = \sqrt{1 + x}$ with $x \in [0, 1]$ (this is the exactly the same function). Then the minimax polynomial is: $1.0007638368 + 0.4838846338x - 0.7119874509x^2$. The theoretical approximation error also leads to 10.35 correct bits (the variable change $x = 1 + x$ does not modify the minimax polynomial quality).

The coefficients p_0 and p_1 are close to power of 2 and we will try to use the post-optimisations proposed in step 4 (see Section 4.4). The first optimisation consists in using $p_0 = 1$. The approximation polynomial $1 + 0.4838846338x - 0.7119874509x^2$ leads to a theoretical accuracy of 9.35 correct bits. The coefficient p_1 is close to 0.5. The approximation $1 + 0.5x - 0.7119874509x^2$ only leads to 6.09 correct bits. So p_1 cannot be replaced by 0.5. But we can try to recode p_1 with only a few non-zero digits (i.e. 1 or $-1 = \bar{1}$). The coefficient p_1 is close to $(0.10000\bar{1})_2$. The approximation polynomial $1 + (0.10000\bar{1})_2x - 0.7119874509x^2$ leads to an accuracy of 9.45 correct bits and the product p_1x is replaced by the subtraction $1/2x - 1/2^6x$. We try to recode p_2 using a few non-zero bits and we get $p_2 = (0.0001001)_2$. The product p_2x^2 is replaced by the addition $1/2^4x^2 + 1/2^7x^2$.

The approximation polynomial $1 + (0.10000\bar{1})_2x + (0.0001001)_2x^2$ leads to an accuracy of 9.49 bits and there is only one multiplication for the computation x^2. The GAPPA input below evaluates the total error corresponding to the last polynomial using $n' = 10$ for the data-path size.

```
p0 = 1; p1 = 31/64; p2 = -9/128;
x = fixed<-10,dn>(Mx);
x2  fixed<-10,dn>= x * x;
p   fixed<-10,dn>= p2 * x2 + p1 * x + p0;
Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
  -> |p-Mf| in ? }
```

GAPPA returns a total of 8.03 correct bits. But this GAPPA program corresponds to the use of multipliers for the products p_1x and p_2x^2. We need to modify the GAPPA input to take into account the new evaluation scheme using only one multiplication. This is the GAPPA input below. Step 3 of the proposed method leads to $n' = 13$. In that case the total error is equivalent to 8.07 correct bits with only one multiplication x^2 as illustrated in Figure 8 (gray circles denotes right shifts).

```
p0  = 1;
p11 = 1/2;   p12 = -1/64;
p21 = -1/16; p22 = -1/128;
x = fixed<-8,dn>(Mx);
x2 fixed<-16,dn>= x * x;
p fixed<-13,dn>= p21*x2 + p22*x2 + p11*x + p12*x + p0;
Mx2 = Mx * Mx;
Mp  = p21 * Mx2 + p22 * Mx2 + p11 * Mx + p12 * Mx + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
  -> |p-Mf| in ? }
```

The operator presented in Figure 8 and the standard solution (degree-2, Horner scheme with $n = n' = 11$ bits) have been implemented on FPGAs. The results are reported in Table 2. A 40% area reduction and a 51% speedup are obtained.

Figure 8 Post-optimised operator for $\sqrt{1 + x}$ over [0, 1]

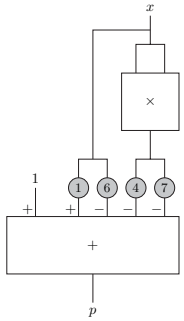


Table 2 Implementation results $\sqrt{1+x}$ over [0, 1]

Solutions	*Area [slices]*	*Period [ns]*	*#Cycles*	*Delay [ns]*
Degree-2 Horner	103	19.9	2	39.8
Degree-2 optimised	61	19.4	1	19.4

6 Conclusion

Some basic methods for the design and the optimisation of hardware operator dedicated to function evaluation have been presented. A complete method based on high-performance polynomial approximation is described. This method leads to small and fast hardware operators. The generated operators are numerically validated to design time. The method deals with both the approximation and the round-off errors which is a something new. Some results have been implemented, validated and compared to standard solutions. Some significant improvements are reported: up to 40% for circuit area reduction and up to 50% speed improvement.

References

Boullis, N. and Tisserand, A. (2005) 'Some optimizations of hardware multiplication by constant matrices', *IEEE Transactions on Computers*, Vol. 54, No. 10, pp.1271–1282.

Brisebarre, N., Muller, J-M. and Tisserand, A. (2004) 'Sparse-coefficient polynomial approximations for hardware implementations', *Proceedings of 38th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA: IEEE, pp.532–535.

Brisebarre, N., Muller, J-M. and Tisserand, A. (2006) 'Computing machine-efficient polynomial approximations', *ACM Transactions on Mathematical Software*, Vol. 32, No. 2, pp.236–256.

Burleson, W. (1990) 'Polynomial evaluation in VLSI using distributed arithmetic', *IEEE Transactions on Circuits and Systems*, Vol. 37, No. 10.

Chesneaux, J-M., Didier, L-S., Jézéquel, F., Lamotte, J-L. and Rico, F. (2006) 'CADNA: control of accuracy and debugging for numerical applications', Available at: http://www-anp.lip6.fr/cadna/, LIP6–Univ. Pierre et Marie Curie.

Detrey, J. and de Dinechin, F. (2004) 'Second order function approximation using a single multiplication on FPGAs', *Proceedings of 14th International Conference on Field-Programmable Logic and Applications FPL*, Number 3203 in LNCS, Springer, pp.221–230.

Detrey, J. and de Dinechin, F. (2005) 'Table-based polynomials for fast hardware function evaluation', *Proceedings of 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, Samos, Greece: IEEE Computer Society, pp.328–333.

de Dinechin, F. and Tisserand, A. (2005) 'Multipartite table methods', *IEEE Transactions on Computers*, Vol. 54, No. 3, pp.319–330.

Duprat, J. and Muller, J-M. (1988) 'Hardwired polynomial evaluation', *Journal of Parallel and Distributed Computing*, Vol. 5, No. 3, pp.291–309.

Edelman, A. (1997) 'The mathematics of the Pentium division bug', *SIAM Reviews*, Vol. 39, No. 1, pp.54–67.

Ercegovac, M.D. and Lang, T. (2003) *Digital Arithmetic*, Morgan Kaufmann.

Ercegovac, M.D., Lang, T., Muller, J-M. and Tisserand, A. (2000) 'Reciprocation, square root, inverse square root, and some elementary functions using small multipliers', *IEEE Transactions on Computers*, Vol. 49, No. 7, pp.627–637.

Ercegovac, M.D., Muller, J-M. and Tisserand, A. (1995) 'FPGA implementation of polynomial evaluation algorithm', in J. Schewel (Ed). *Proceedings of Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, Vol. 2607, Philadelphia, PA: SPIE, pp.177–188.

Ercegovac, M.D., Muller, J-M. and Tisserand, A. (2005) 'Simple seed architectures for reciprocal and square root reciprocal', *Proceedings of 39th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA: IEEE, pp.1167–1171.

Flynn, M.J. and Oberman, S.F. (2001) *Advanced Computer Arithmetic Design*, Wiley-Interscience.

Goubault, E., Martel, M. and Putot, S. (2006) *FLUCTUAT: Static Analysis for Numerical Precision*, Available at: http://www-list.cea.fr/labos/fr/LSL/fluctuat/, CEA-LIST.

Hassler, H. and Takagi, N. (1995) 'Function evaluation by table look-up and addition', *Twelfth IEEE Symposium on Computer Arithmetic*, Bath, UK: IEEE Computer Society, pp.10–16.

Higham, N.J. (2002) *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM.

Johansson, K., Gustafsson, O. and Wanhammar, L. (2006) 'Approximation of elementary functions using a weighted sum of bit-products', *IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, pp.795–798.

Markstein, P. (2000) *IA-64 and Elementary Functions: Speed and Precision*, Prentice Hall: Hewlett-Packard Professional Books.

Melquiond, G. (2006) 'GAPPA: génération automatique de preuves de propriétés arithmétiques', Available at: http://lipforge.ens-lyon.fr/www/gappa/, ENS-Lyon: LIP.

Ménard, D. and Sentieys, O. (2002) 'Automatic evaluation of the accuracy of fixed-point algorithms', *Proceedings of Design, Automation and Test in Europe (DATE)*, pp.529–537.

Michard, R., Tisserand, A. and Veyrat-Charvillon, N. (2005) 'Divgen: a divider unit generator', in F.T. Luk (Ed). *Proceedings of Advanced Signal Processing Algorithms, Architectures and Implementations XV*, Vol. 5910, San Diego, CA: SPIE, p.59100M.

Michard, R., Tisserand, A. and Veyrat-Charvillon, N. (2006) 'Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales', *11ième SYMPosium en Architectures nouvelles de machines (SYMPA)*, Perpignan, France, pp.130–141.

Muller, J-M. (1999) 'A few results on table-based methods', *Reliable Computing*, Vol. 5, No. 3, pp.279–288.

Muller, J-M. (2006) *Elementary Functions: Algorithms and Implementation*, 2nd edition, Birkhäuser.

Pineiro, J.A., Bruguera, J.D. and Muller, J-M. (2001) 'Faithful powering computation using table look-up and a fused accumulation tree', *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, pp.40–47.

Remes, E. (1934) 'Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation', *CR Academic Science Paris*, Vol. 198, pp.2063–2065.

Robertson, J.E. (1958) 'A new class of division methods', *IRE Transactions Electronic Computers*, Vol. EC-7, pp.218–222.

Schulte, M. and Stine, J. (1999) 'Approximating elementary functions with symmetric bipartite tables.' *IEEE Transactions on Computers*, Vol. 48, No. 8, pp.842–847.

Stine, J. and Schulte, M. (1999) 'The symmetric table addition method for accurate function approximation', *Journal of VLSI Signal Processing*, Vol. 21, No. 2, pp.167–177.

Takagi, N. (2001) 'A hardware algorithm for computing reciprocal square root', *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, pp.94–100.

Takagi, N. and Kuwahara, S. (2000) 'A VLSI algorithm for computing the euclidean norm of a 3D vector', *IEEE Transactions on Computers*, Vol. 49, No. 10, pp.1074–1082.

The Coq Development Team (2004) 'The Coq proof assistant', Availbale at: http://coq.inria.fr/, INRIA.

Tisserand, A. (2006) 'Automatic generation of low-power circuits for the evaluation of polynomials', *Proceedings of 40th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA: IEEE.

Tocher, K.D. (1958) 'Techniques of multiplication and division for automatic binary compputers', *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 11, part 3, pp.368–384.

Volder, J. (1959) 'The CORDIC computing technique', *IRE Transactions on Computers*, Vol. EC-8, No. 3, pp.330–334.

Walther, J. (1971) 'A unified algorithm for elementary functions', *Joint Computer Conference Proceedings*, Reprinted in E.E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA,1990.

Wong, W. and Goto, E. (1995) 'Fast evaluation of the elementary functions in single precision', *IEEE Transactions on Computers*, Vol. 44, No. 3, pp.453–457.

Note

[1]In some cases, using different values for n and n' may reduce the size of the circuit.