Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree*

J. A. Piñeiro[§], J. D. Bruguera[§], J. M. Muller[¶]
§ Department of Electrical and Computer Engineering
Univ. Santiago de Compostela, Spain.
¶ CNRS, Project CNRS/ENSL/INRIA/ARENAIRE
LIP, Ecole Normale Superieure de Lyon, France.
e-mail: alex,bruguera@dec.usc.es, jmmuller@ens-lyon.fr

Abstract

A method for the calculation of faithfully rounded single-precision floating-point powering (X^p) is proposed in this paper. This method employs table look-up and a second-degree minimax approximation, which allows the employment of reduced size tables to store the coefficients from the polynomial approximation. A specialized squaring unit and a fused accumulation tree carry out with the computation of the quadratic polynomial. Both unfolded and pipelined architectures are presented, and the results of a pre-layout synthesis performed using CMOS 0.35 μm technology are shown, achieving a 50% area reduction from linear approximation methods, and with improved speed over other second-degree approximation based algorithms. The pipelined architecture has a latency of three cycles and a throughput of one result per cycle.

1. Introduction

Powering (X^p) is a very interesting function for applications such as computer 3D graphics and digital image and signal processing. It can also be a very efficient way to compute other important functions. The reciprocal (X^{-1}), the square root ($X^{1/2}$), the inverse square root ($X^{-1/2}$), the reciprocal square (X^{-2}), the cube (X^3), the reciprocal cube (X^{-3}), and others, are just specific cases of the powering function, with the fixed parameter p taking different values. These functions are important for several applications [5][16].

For very low precision calculations, it is possible to employ direct table look-up, but its high memory requirements make it an inefficient method for single–precision floating–point format. Polynomial and rational approximations [8] could be another way to implement powering, and also iterative algorithms, such as linear convergence methods (digit-recurrence algorithms [4]), and quadratic convergence methods (multiplicative-based [5] Newton–Raphson and Goldschmidt algorithms). However, one of the most efficient methods for computing powering function, in a single–precision floating–point format, are table–driven algorithms [8][19], which are halfway between direct table look-up and polynomial and rational approximations. The use of the polynomial approximation allows the table size to be to significantly reduced, and the table look-up allows the reduction of the degree of the polynomial employed to compute the function approximation.

Among the previous more important table-based methods it is possible to find two main groups: first–order approximation algorithms and second–order interpolation methods. The first–order algorithms can be bipartite approximations [3][10], which store a borrow–save representation of the function values in two tables, using an adder to perform a later assimilation, or piecewise linear approximations [18], which employ a multiplier, reducing thus the size of the required look-up tables. The second–order interpolation methods [6][2] propose architectures with a latency of 2 cycles, with the main advantage of the smaller tables employed, at the expense of more combinational logic than other methods. No one of these algorithms employs a minimax approximation, which allows a more important reduction on the table size.

Table 1 summarizes the hardware requirements for all these methods for single–precision floating–point computation. A more detailed description can be found in [9], and general considerations regarding exactly rounded function approximation with look–up tables and k–degree polynomials can be found in [12].

The method we propose here consists of using a second-

*This work was supported by the Secretaria Xeral de Investigacion e Desenvolvemento de Galicia (Spain), under contract PGIDT99PXI20602B

Method	Table size (bits)	Multiplier (bits)	Adder (bits)	Others
Direct	$2^{3m} \times 3m$	–	–	–
Bipartite tables	$(2^{2m} \times 3m) + (2^{2m} \times m)$	–	–	RB Booth
SBTM	$(2^{2m} \times 3m) + (2^{2m-1} \times m)$	–	3m	–
Piecewise linear approx.	$2^{3m/2} \times (3m/2 + 3m)$	$3m/2 \times 3m/2$	$3m$	–
Linear operand modif. [18]	$2^{3m/2} \times 3m$	$3m \times 3m$	–	$[RBBooth]$
2^{nd}-degree interp. [6]	$2^m \times (3m + 2m + m + m + 6)$	2 of $(3m \times 3m)$	2 of $3m$	–
2^{nd}-degree interp. [2]	$2^m \times (3m + 2m + 3)$	2 of $(3m \times 3m)$	5 of $3m$	–

Table 1. Comparison of different table–driven methods to obtain $n = 3m$-bit accuracy

degree minimax approximation to faithfully compute X^p as $C_2X_2^2 + C_1X_2 + C_0$, where X_2 is the lower part of X, for a single–precision floating–point format. The three coefficients, C_0, C_1 and C_2, are obtained by using the computer algebra system Maple [20], minimizing their wordlengths for the required precision. These coefficients are stored in look-up tables, and selected by X_1, the m-bit upper part of X, and then the quadratic function is evaluated. When this most m significant bits of the operand are employed to address the tables, an accuracy over n=$3m$ bits is obtained, which allows a reduction on the table size to about $2^m \times \{(m-1) + (2m-1) + 3m\}$ (depending on p; about $12Kbits$ when m=8). The evaluation of the quadratic function is done using a fused accumulation tree with less size and about the same delay as a n-bit by n-bit standard multiplier. Besides, a small specialized squaring unit, which benefits from some mathematical properties to significantly reduce its size and delay over a standard squaring unit, and a *carry–save* (CS) to *signed–digit* (SD) radix 4 recoding unit, carry out with the computation of X_2^2. There is no restriction in our method to the value of parameter p, which can be either integer or non–integer, positive or negative, and the output results are guaranteed to be faithfully rounded to the nearest. The speed of first order approximations is achieved, employing tables slightly smaller than the tables of second degree interpolation methods.

2. Faithful Powering Computation

In this section we propose a method for the faithfully rounded computation of powering function (X^p), in a single–precision floating–point format, by means of a second-degree minimax approximation with table look-up, a specialized squaring unit and a fused accumulation tree. Faithful rounding to the nearest means that the returned value is guaranteed to be one of the two floating-point numbers that surround the exact value, which in most cases is the exactly rounded result.

2.1. Method

With IEEE single–precision floating–point format, a floating point number M is represented using a sign bit s_m, an 8-bit biased exponent e_m, and a 24-bit significand X. If M is a normalized number, it represents the following value:

$$M = (-1)^{s_m} \times (1 + f_x) \times 2^{e_m - 127},$$

where X=$1 + f_x$, $1 \leq X < 2$, and f_x is the fractional part of the normalized number (the 23 stored bits).

Our method deals only with the powering of the significand, X^p, since the sign and exponent treatment is straightforward, and can be done in parallel:

$$\begin{aligned} M^p &= [(-1)^{s_m} \times (X) \times 2^{e_m - 127}]^p \\ &= (-1)^{s_m p} \times (X)^p \times 2^{(e_m - 127)\cdot p} \end{aligned}$$

For p=$\{1/2, -1/2\}$, we guarantee faithfully rounded results for input operands in the interval $1 \leq X < 4$, and for p=$\{1/3, -1/3\}$, input operands in the interval $1 \leq X < 8$ are allowed. This suffices to guarantee faithful results in the full single–precision range.

Second–order minimax approximation

Our method for computing X^p is based on a minimax approximation (which is known to be the optimal polynomial approximation) of the function. We take advantange of its high accuracy to significantly reduce the wordlenghts of the coefficients to employ. The n-bit binary input significand, X, is split into an upper part X_1 and a lower part X_2, as in the piecewise linear approximation:

$$\begin{aligned} X_1 &= [1.x_1x_2 \ldots x_m] \\ X_2 &= [.x_{m+1} \ldots x_n] \times 2^{-m} \end{aligned}$$

An approximation to X^p in the range $X_1 \leq X < X_1 + 2^{-m}$ can be performed evaluating the expression

$$X^p \approx C_0 + C_1X_2 + C_2X_2^2 \qquad (1)$$

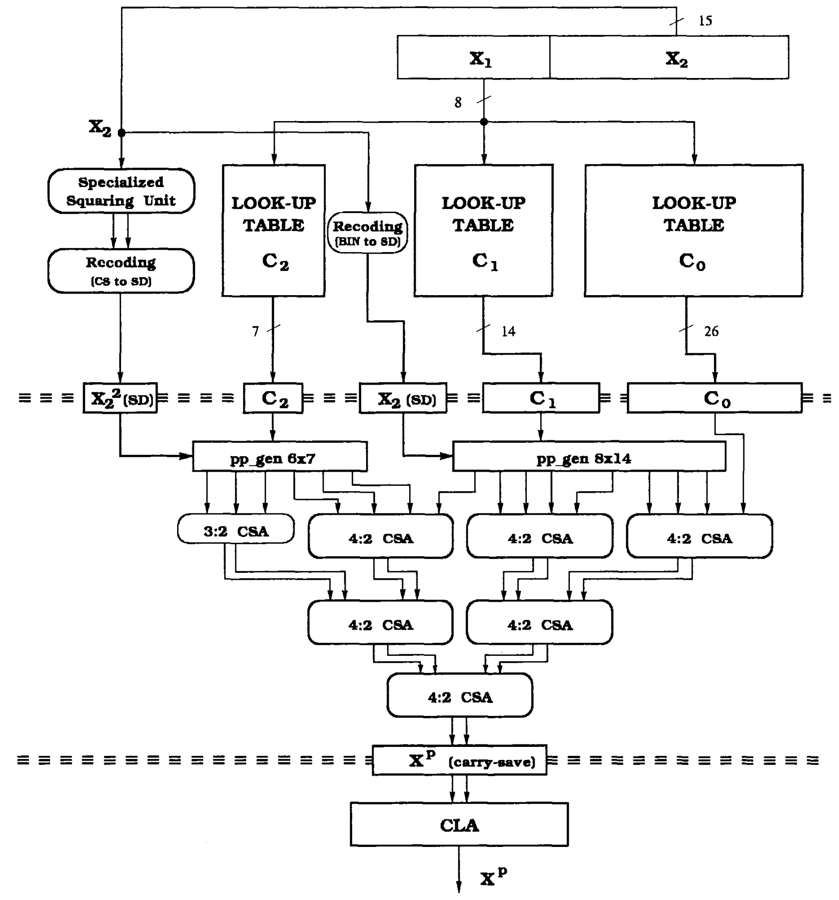

Figure 1. Pipelined unit scheme ($X^{-1/2}$ computation; $m = 8$)

where the coefficients C_0, C_1 and C_2 are obtained employing the computer algebra system **Maple** [20], which performs minimax approximations using Remes algorithm.

The values of these coefficients depend only on the value of X_1, the m most significant bits of X, and on the parameter p. Therefore, C_0, C_1 and C_2 can be stored in look-up tables of 2^m input values.

A diagram block of our method is shown in Figure 1, for the specific case of $p=-1/2$, the inverse square root computation, with a pipelined architecture.

Example

We wish to implement function $1/\sqrt{x}$ between 1 and 2, and the coefficients of a degree-2 approximation will be stored in a table with 8 address bits. At address i in the table, we will find the coefficients of the approximation for the interval $\{1 + i/256, 1 + (i+1)/256\}$. Let us see how to compute coefficients for $i = 37$. The Maple input line

```
> minimax(1/sqrt(1+37/256+x),x=0..1/256,
            [2,0],1,'err');
```

asks for the minimax approximation in that domain (variable x represents X_2, the low-order bits of the input value). We get the approximation

```
> 0.93472998018 + (- 0.40834453917 +
      + 0.26644775593 x ) x
```

The error of this approximation is 3.60739×10^{-9}. If we round the order-1 coefficient to 14 bits and the order-2 coefficient to 6 bits, we find the new approximation error with the input commands

```
> C0 := .9347299801784;
> C1 := round(2^15*(-.40834453917))/2^15;
> C2 := round(2^7*.26644775593)/2^7;
> infnorm(1/sqrt(1+37/256+x)-C0-C1*x-
            C2*x^2,x=0..1/256);
```

The new error is 5.58×10^{-8}, which is much larger than the initial error. Now, we can compute the best approximation polynomial among the polynomials with the 14-bit number $C1$ as order-1 coefficient as follows. We need to approximate

$$\frac{1}{\sqrt{1+37/256+x}} - C1 \times x$$

```
computecoeffts := proc(cc, p, q)
local errmax, i, pol1, j, weight, C1, pol2, C2, p0, C0, err;
errmax:=0;
for i from 0 to 255 do
    pol1 := minimax(1/sqrt(1 + 1/256*i + x), x = 0 .
        .1/256, [2, 0], 1, 'err');
    j := 0;
    weight := 1;
    while abs(coeff(pol1, x)) < weight do
        weight := .5*weight; j := j + 1
    od;
    C1 := 2^(-j-p+1)*round(coeff(pol1, x)*2^(j+p-1));
    pol2 := minimax(
        1/sqrt(1 + 1/256*i + sqrt(X)) - C1*sqrt(X),
        X = (0)^2 .. (1/256)^2, [1, 0], 1, 'err');
    j := 0;
    weight := 1;
    while abs(coeff(pol2, X)) < weight do
        weight := .5*weight; j := j + 1
    od;
    C2 := 2^(-j-q+1)*round(coeff(pol2, X)*2^(j+q-1));
    p0 := minimax(1/sqrt(1+1/256*i+x)-C1*x-C2*x^2,
        x = 0 .. 1/256, [0, 0], 1, 'err');
    C0 := round(2^(cc+1)*( tcoeff(p0)) )/2^(cc+1);
    err := infnorm(1/sqrt(1+1/256*i+x)-C0-C1*x-
        C2*x^2, x=0..1/256);
    if errmax < err then errmax:=err fi;
od;
errmax
end
```

Figure 2. Maple program implementing our method (inverse square root computation)

by a polynomial of the form $C_0' + C_2' x^2$. By defining $X = x^2$ this is equivalent to approximating

$$\frac{1}{\sqrt{1 + 37/256 + \sqrt{X}}} - C1 \times \sqrt{X}$$

by the degree-1 polynomial $C_0' + C_2' X$. So we type

```
> minimax(1/sqrt(1+37/256+sqrt(X))-
    C1*sqrt(X), X = (0)^2..(1/256)^2,
    [1,0],1,'err');
```

After rounding to 6 bits the newly obtained coefficient C_2' (which give a number C_2''), we get a slightly better approximation error, namely 5.01×10^{-8}. We now have to compute a new order-zero coefficient to get the best polynomial among the polynomials whose order-1 coefficient is $C1$ and whose order-2 coefficient is C_2''. This is done as follows:

```
> minimax(1/sqrt(1+37/256+x)-C1*x-
    C2second*x^2,x=0..1/256,[0,0],1,'err');
```

The new coefficient is $C_0' = 0.934730008279251$. The error of this final approximation is 2.7740073×10^{-8}, which is less than 2^{-25}.

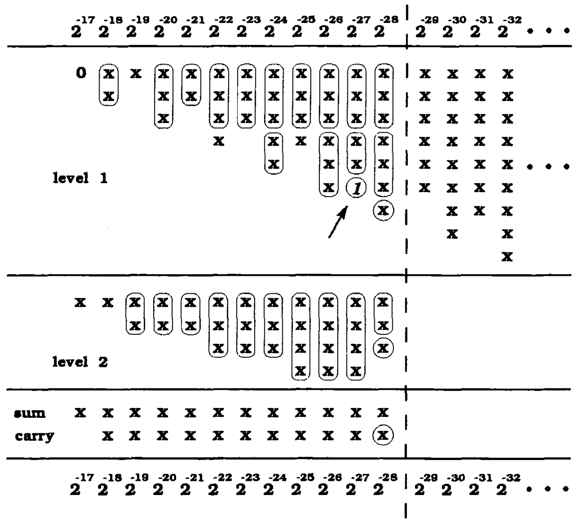

Figure 3. Specialized squaring unit operation (when $m = 8$)

The method presented here can be easily generalized. It allows to get small order 1 and 2 coefficients (or higher order coefficients for polynomials of degree greater than 2), which result in smaller and faster multipliers, and also smaller tables. Figure 2 shows a Maple program which implements our method.

Specialized squaring unit

The calculation of X_2^2 is necessary for the evaluation of the quadratic function. Rather than having a multiplier to compute the square, some strategies can be used to significantly reduce the area and delay of the unit which performs this calculation [7]: re-arranging the partial product matrix and considering the leading zeros of X_2.

- Since $x_i x_j = x_j x_i$, the partial product matrix is symmetric with respect to the anti-diagonal, which allows the reduction on the number of partial products to be accumulated by using the identities $x_i x_j + x_j x_i = 2x_i x_j$ and $x_i x_i = x_i$. Thus, the original matrix can be replaced by an equivalent matrix consisting of the partial products on the anti-diagonal, plus the partial products above it *shifted* one position to the left.

- To further reduce the size and delay of the equivalent matrix, it is also important to notice that $X_2 = [.x_{m+1} \cdots x_n] \times 2^{-m}$, and therefore X_2 has m leading zeros. This means that X_2^2 will have at least $2m$ leading zeros. If we truncate the partial products accumulation tree at position 2^{-2m-12}, which means that the wordlength of X_2^2 will be 12 bits, and add a 1 on

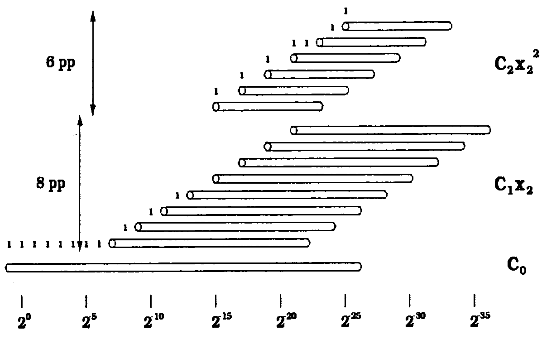

Figure 4. Partial products to be accumulated (when $m = 8$)

an empty position with weight 2^{-2m-11}, the maximum error on its computation has been proved to be bounded by

$$\sum_{j>2m+12} part\ prod_j < 3 \cdot 2^{-2m-11}$$

The only dependence on p is through the value of m.

The matrix of partial products to be accumulated in the squaring unit is shown in Figure 3. The partial products on the right of the dashed vertical line are not accumulated, so only two levels of adders (the first one, with 3:2 CSA; the second one, with 4:2 CSA), plus an initial *and* stage ($x_{ij} = x_i \cdot x_j$), are required to generate the carry–save (CS) representation of X_2^2.

As X_2^2 is going to be employed as a multiplier, it is possible to avoid the final assimilation by using a CS to SD radix 4 recoding to generate its SD representation. X_2^2 in SD radix 4 representation is employed for the partial–products generation in the accumulation tree.

Fused accumulation tree

Apart from the squaring evaluation, the other main problem to be overcome is the computation of the expression (1), once the values of C_0, C_1 and C_2 have been obtained through table look-up, and X_2^2 has been calculated in parallel using the specialized squaring unit.

We propose to employ an unified tree to accumulate the partial products both of C_1X_2 and of $C_2X_2^2$, plus the coefficient C_0 [17]. Considering the number of leading zeros of X_2 (m leading zeros) and of X_2^2 ($2m$ leading zeros), and employing SD radix 4 representation to reduce the number of partial products to be accumulated, the resulting tree has significantly less size and almost the same delay as a standard $(3m \times 3m)$-bit multiplier.

Let's consider, for instance, the inverse square root function ($X^{-1/2}$, i.e. p=$-1/2$, Figure 1). For this function, it will be seen that m=8, and therefore X_2 has 15 significant bits. X_2^2 has 16 leading zeros, and a wordlength of 12 bits. Employing SD radix 4 recoding, these wordlengths lead to a generation of 8 partial products to compute C_1X_2 and 6 partial products to compute $C_2X_2^2$. The number of products to be accumulated is therefore 15 (8 + 6 + 1; the last one is the coefficient C_0), and so 3 levels of 4:2 CSA adders, plus a final assimilation, will be needed.

The delay of the critical path for our fused accumulation tree is

$$t_{fused\_accum} = t_{pp\_gen} + 3 \cdot t_{4:2\ CSA} + t_{CLA},$$

while the delay of a standard (24 x 24)-bit multiplier is about

$$t_{mult} = t_{pp\_gen} + t_{3:2\ CSA} + 2 \cdot t_{4:2\ CSA} + t_{CLA}$$

Therefore, the fused accumulation tree we propose is only 0.5 t_{fa} slower than a standard multiplier, with significantly less area, because of the reduced wordlengths of our C_2 and C_1 coefficients.

Figure 4 shows the arrangement of the matrix of partial products to be accumulated, and gives an idea of the effect of reducing the wordlengths of these two coefficients on the total area of the accumulation tree (C_2 affects to 6 partial products, while C_1 is involved in the generation of 8). The circles at the beginnig of each word mean a complement of the sign bit.

2.2. Error computation

The total error of the output result can be expressed as the accumulation of the error of the result before rounding, ϵ_{interm}, and the rounding error, ϵ_{round}:

$$\epsilon_{total} = \epsilon_{interm} + \epsilon_{round} < 2^{-r}$$

where r depends on the function to compute. As $\epsilon_{round} \leq 2^{-r-1}$ (we perform rounding to the nearest), there is a bound on the error for the intermediate result of $\epsilon_{interm} < 2^{-r-1}$ to guarantee faithfully rounded final results. We perform the rounding to the nearest by adding a 1 in position 2^{-r-1}: C_0'=$C_0 + 2^{-r-1}$, and then truncating the intermediate result at position 2^{-r}.

The error of the intermediate result comes from two sources, the error of the second order minimax approximation, ϵ_{approx}, and the error due to the finite arithmetic employed to compute the quadratic approximating function, ϵ_{comput}:

$$\epsilon_{interm} = \epsilon_{approx} + \epsilon_{comput},$$

The term ϵ_{approx} can be obtained as a result of the program shown in Figure 2, and already takes into account the finite wordlength of the coefficients C_0', C_1 and C_2.

Function	ϵ_{interm}	m	$\epsilon_{squaring}$	ϵ_{approx}	*Look–Up Table size*
$\sqrt{X}\,(X^{1/2})$	$< 2^{-24}$	7	$< 1.50 \cdot 2^{-26}$	$1.23 \cdot 2^{-25}$	$2^7 \times (26+14+6) = 5.75Kb$
$1/\sqrt{X}\,(X^{-1/2})$	$< 2^{-25}$	8	$< 1.14 \cdot 2^{-27}$	$1.36 \cdot 2^{-26}$	$2^8 \times (26+14+7) = 11.75Kb$
$1/X\,(X^{-1})$	$< 2^{-25}$	8	$< 1.50 \cdot 2^{-26}$	$0.93 \cdot 2^{-27}$	$2^8 \times (27+15+7) = 12.25Kb$
X^3	$< 2^{-24}$	9	$< 1.13 \cdot 2^{-25}$	$0.99 \cdot 2^{-26}$	$2^9 \times (25+17+7) = 24.5Kb$
$1/X^2\,(X^{-2})$	$< 2^{-26}$	9	$< 1.13 \cdot 2^{-28*}$	$1.23 \cdot 2^{-27}$	$2^9 \times (29+13+6) = 24Kb$
$1/X^3\,(X^{-3})$	$< 2^{-27}$	10	$< 1.13 \cdot 2^{-29*}$	$1.24 \cdot 2^{-28}$	$2^{10} \times (29+14+7) = 50Kb$

Table 2. Summary of design parameters for some functions

Thus, the error of the computation comes only from the error due to the finite precision computation performed by the specialized squaring unit ($\epsilon_{squaring}$),

$$\epsilon_{comput} = \epsilon_{squaring} < |C_2|_{max} \cdot 3 \cdot 2^{-2m-11}$$

Summarizing, to guarantee faithfully rounded results, the following bound is set:

$$\epsilon_{interm} = \epsilon_{approx} + |C_2|_{max} \cdot 3 \cdot 2^{-2m-11} < 2^{-r-1}$$

For each function, there is a minimum value of m which allows the achievement of faithful rounding. Once m is set, the minimax approximation for the specific function is computed using Maple, minimizing the wordlengths of the coefficients C_1 and C_2 within the bound set for the error in the intermediate result. For some functions it is necessary to employ an extended wordlength for X_2^2 (two more bits, which means that one more partial product for $C_2X_2^2$ is required) so that the error from the squaring is decreased to allow a smaller value of m. We will denote this as X_2^{2*}. The specialized squaring unit resulting from this extension is 0.5 t_{fa} slower than the one previously proposed. Table 2 shows a summary of the parameters to employ for implementing some interesting functions with our method.

2.3. Architecture

Our method to faithfully compute the powering function X^p in a single–precision floating–point format can be implemented by either an unfolded architecture or a pipelined architecture with a latency of three cycles and a throughput of one result per cycle. Figure 1 shows the pipelined unit scheme for the specific case of inverse square root computation (p=$-1/2$; m=8). It has the same structure as the unfolded unit, with strategically inserted registers. The placement and number of registers has as goals both to reduce their area cost and to allow the design of a unit with a high operation frequency.

In the first stage of this unit, the tables are addressed using the operand X_1 to look-up the values of coefficients C_0', C_1 and C_2. In parallel, the computation of X_2^2 in *carry-save* form is performed, and then recoded to *signed–digit* radix 4 to be stored in a register. A binary to SD radix 4 recoding of X_2 is also performed in parallel with the table look-ups. The total table size is $2^8 \times (26 + 14 + 7) = 11.75Kbits = 1.5KB$ (for a second–degree Taylor approximation, coefficients of wordlengths 29, 19 and 11 would be required to guarantee faithfully rounded results).

The second stage is the accumulation tree stage, beginning with the partial products generation (both of $C_2X_2^2$ and of C_1X_2), and then performing the accumulation of all these products plus the coefficient C_0'. The reduction on the coefficients wordlength allows an important reduction on the area of the accumulation tree.

The last stage of this pipelined architecture consists of a carry-lookahead adder (CLA) that performs the final assimilation from carry–save to binary representation of the result X^p, which is guaranteed to be faithfully rounded.

The critical path for the unfolded unit contains the specialized squaring unit, the *CS to SD* recoding unit and the fused accumulation tree, so

$$t_{unfolded} = t_{squaring} + t_{recoding} + t_{fused\_acc} + t_{reg}$$

Dividing the scheme into three stages, the total number of registers to be employed is very low, and the cycle time (and so the operation frequency) comes only from the accumulation stage, without including the final assimilation:

$$t_{pipelined} = t_{pp\_gen} + 3 \cdot t_{4:2\ CSA} + t_{reg}$$

Implementation results

The unfolded and pipelined architectures proposed have been synthesized using Synopsys Design Analyzer compiler tool, employing a CMOS 0.35 μm technology standard cell library from AMS [1], following a *VHDL*–based design flow [14][15]. A more complete and detailed analysis can be found in [9].

Table 3 shows the area and delay results for each individual component of the architecture, and also the total area and delay of the unfolded architecture. There are four main paths in this architecture, corresponding to the three look-up tables and the squaring unit, all of them connected to the

Component	Area (mm^2)	Delay (ns)
X_2 squaring	0.096	2.02
recoding CS to SD	0.092	1.35
recoding BIN to SD	0.017	0.33
registers	0.011	0.70
Table C_2	0.151	3.00
Table C_1	0.307	3.00
Table C_0'	0.401	3.50
Fused Accum. Tree	0.791	7.08
TOTAL Tables & extra	1.075	4.07
TOTAL	1.87	11.2

Table 3. Area and delay of circuit components (unfolded unit)

Pipeline stage	Area (mm^2)	Delay (ns)
First stage	1.108	4.2
Second stage	0.654	4.8
Third stage	0.171	3.7
TOTAL	1.93	4.8

Table 4. Area and delay of pipelined unit stages

fused accumulation tree. The critical path is the one containing the squaring unit and the CS to SD recoding, with a delay of about 11.2 ns. The total area is about 1.9 mm^2, with about the same contribution from the tables and from the fused accumulation tree.

Table 4 shows the area and delay for each stage of the pipelined architecture. A cycle time of about 5 ns is achieved, set by the delay of the second stage.

It is possible to design architectures for the computation of any two different functions (X^{p_1} or X^{p_2}) by only duplicating the table size and inserting multiplexers to select between the corresponding coefficients. The extra hardware inserted does not lead to an increase in the circuit delay, because the critical path is the one containing the specialized squaring and CS to SD recoding units. Only minor modifications on the accumulation tree are required.

3. Comparison

In this section we perform a comparison between our method and some previous table–driven algorithms.

The method presented in [12] computes exactly rounded function approximations (reciprocal, square root, logarithm and exponential), by using table look–up and a polynomial approximation: a k–degree Chebyshev interpolation. The effects of providing exactly rounded results ($\epsilon_{total} < 0.5ulp$) is to significantly increase the size of the tables employed (the tables are almost double-sized). This method could be employed to provide faithful results ($\epsilon_{total} < 1ulp$), but for k=2, the tables to employ are still slightly bigger than our tables, due to the fact that we have performed a minimax approximation, which is more accurate than a Chebyshev interpolation. Besides, the hardware we employ for the computation of the quadratic function (1), a specialized squaring unit and a fused accumulation tree, is optimized for obtaining a better speed/area tradeoff.

Table 5 shows a comparison of the main features of our unfolded and pipelined units with some of the more efficient previous table–driven methods which provide a final error less than 1 ulp: on one hand, the most efficient linear approximation algorithm, the linear approximation with operand modification [18], and on the other hand, two second-degree interpolation methods [6][2]. A more detailed comparison, and an exhaustive description of any of them, can be found in [9]. To allow a fair comparison, we have synthesized the architectures implementing these methods by means of the same synthesis CAD tool employed for our units. The standard cell library employed has also been the same one, a 0.35 μm technology from AMS. However, we have to remark that the results shown on the table are just *pre-layout* synthesis results.

These results show that our architectures require slightly less area than the second-degree interpolation methods ($\sim$ 2.2 mm^2), but the operation frequency achieved is significantly higher: the time needed to complete one operation is 11.2 ns, while for the second-degree interpolation is above 25 ns. This reduction in size comes from employing the minimax approximation, which allows the use of smaller tables, since the coefficients C_1 and C_2 have a reduced wordlength. Other important feature shown in the table is that the cycle time of our architectures is the same as the linear approximation with operand modification one ($\sim$ 11.2 ns), but the total area has been reduced from 4.3 mm^2 to about 1.9 mm^2. We also show the time needed to complete 10 operations, to emphasize the advantage of pipelining our unfolded design, which allows the achievement of a high operation frequency.

We have also performed an structural comparison with some specific units which employ a low precision approximation as a seed and then a modified Newton-Raphson iteration for the computation of the reciprocal [11] and the inverse square root [13] functions. The results of this comparison, which show the achievement of higher operation frequencies with less area, and a detailed description of these units, can also be found in [9].

Scheme	*latency*	*throughput*	*cycle t.(ns)*	*area* (mm^2)	*time/op.(ns)*	*time/10 op.(ns)*
Unfolded unit	1	1	**11.2**	**1.87**	11.2	112
Pipelined unit	3	1	**4.8**	**1.93**	14.2	**58**
Linear operand modif. [18]	1	1	**11.2**	4.28	11.2	112
Second–degree interp. [6]	2	2	13.4	2.15	26.8	268
Second–degree interp. [2]	2	1	12.6	2.80	25.2	139

Table 5. Architecture features comparison (faithfully rounded results)

4. Conclusion

A new method for the calculation of faithfully rounded single-precision floating-point format powering (X^p) has been presented. This method, which can be adapted to compute any elementary function, performs this calculation by using a second degree minimax approximation, which allows the achievement of minimum wordlengths for the polynomial coefficients. Three look-up tables addressed by the m-bit word X_1 are employed to store these coefficients, a specialized squaring unit with low area cost and delay performs the calculation of X_2^2, and the quadratic approximation is computed using a fused accumulation tree, with significantly less area and about the same delay as a standard (24×24)-bit multiplier.

Both an unfolded and a pipelined architecture have been proposed. Pre-layout synthesis results have been obtained employing Synopsys tools combined with a CMOS 0.35 μm standard cell library, showing a cycle time of 11.2 ns for our unfolded architecture, and about 4.8 ns for our pipelined architecture, whose latency is three cycles and whose throughput is one result per cycle.

Comparison results show that our method combines the main advantage of linear approximations, the speed, and the main advantage of the second-degree approximations, the reduced size of the circuit, also taking advantage of the high accuracy of the minimax approximation employed.

The common employment of an initial low–precision approximation (*seed*) and a later Newton–Raphson iteration is due to the huge size of the look-up tables employed in traditional table–driven methods, which make designers prefer this solution to save a great amount of area. The reduction in the table size of our method makes it suitable to directly compute X^p in single–precision format, with no later iteration, which usually leads to a reduced latency and/or an improved operation frequency.

References

[1] AMS. *0.35 μm CMOS Standard Cell Databook*, 2000.

[2] J. Cao and B. Wei. High-performance hardware for function generation. In *Proc. 13th Symp. Computer Arithmetic*, pages 184–188, 1997.

[3] D. DasSarma and D. W. Matula. Faithful bipartite rom reciprocal tables. In *Proc. 12th Symp. Computer Arithmetic*, pages 17–28, 1995.

[4] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit Recurrence Algorithms and Implementations.* Kluwer Academic Publishers, 1994.

[5] M. J. Flynn. On division by functional iteration. *IEEE Trans. On Computers*, 19:702–706, 1970.

[6] V. K. Jain, S. A. Wadecar, and L. Lin. A universal nonlinear component and its application to WSI. *IEEE Transactions On Components, Hybrids and Manufacturing Technology*, 16(7):656–664, 1993.

[7] T. Jayarshee and D. Basu. On binary multiplication using the quarter square algorithm. In *Spring Joint Computer Conference*, pages 957–960, 1974.

[8] J. M. Muller. *Elementary Functions. Algorithms and Implementation.* Birkhäuser, 1997.

[9] J. A. Piñeiro, J. D. Bruguera, and J. M. Muller. *Powering by Table look-up using a 2nd-degree minimax approximation*, 2000. Internal report, at http://www.ac.usc.es.

[10] M. J. Schulte and J. E. Stine. Symmetric bipartite tables for accurate function approximation. In *Proc. 13th Symp. Computer Arithmetic (ARITH13)*, pages 175–183, 1997.

[11] M. J. Schulte, J. E. Stine, and K. E. Wires. High-speed reciprocal approximations. In *Proc. 31st Asilomar Conference On Signals, Circuits and Systems*, pages 1178–1182, 1998.

[12] M. J. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In *Proc. IEEE 11th Int. Symp. Computer Arithmetic (ARITH11)*, pages 138–145, 1993.

[13] M. J. Schulte and K. E. Wires. High-speed inverse square roots. In *Proc. 14th Symp. Computer Arithmetic (ARITH14)*, pages 124–131, April 1999.

[14] S. Sjoholm and L. Lindh. *VHDL for designers*. Prentice Hall, cop., 1997.

[15] M. J. S. Smith. *Application-Specific Integrated Circuits.* Addison Wesley, 1997.

[16] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating point divide and square root implementations. *ACM Computer Surveys*, pages 518–564, 1996.

[17] E. E. Swartzlander. Merged arithmetic. *IEEE Transactions On Computers*, 29:946–950, 1980.

[18] N. Takagi. Powering by a table look–up and a multiplication with operand modification. *IEEE Trans. Computers*, 47(11):1216–1222, 1998.

[19] P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. *Argonne Nat. Lab. Rep.*, MCS-P194-1190, January 1991.

[20] Waterloo Maple Inc. *Maple V Programming Guide*, 1998.