Journal of VLSI Signal Processing 33, 125–145, 2003

Implementation of the Exponential Function in a Floating-Point Unit*

ÁLVARO VÁZQUEZ AND ELISARDO ANTELO
Department of Electronic and Computer Engineering, University of Santiago de Compostela, 15706 Santiago de Compostela, Spain

Received November 3, 2000; Revised September 7, 2001

Abstract. In this work we present an implementation of the exponential function in double precision, in a unit that supports IEEE floating-point arithmetic. As existing proposals, the implementation is based on the use of a floating-point multiplier and additional hardware. We decompose the computation into three subexponentials. The first and third subexponentials are computed in a conventional way (table look-up and polynomial approximation). The second subexponential is computed based on a transformation of the slow radix-2 digit-recurrence algorithm into a fast computation by using the multiplier and additional hardware. We present a design process that permits the selection of the most convenient trade-off between hardware complexity and latency. We discuss the algorithm, the implementation, and perform a rough comparison with three proposed designs. Our estimations indicate that the implementation proposed in this work presents better trade-off between hardware complexity and latency than the compared designs.

Keywords: exponential function, transcendental functions, floating-point unit, computer arithmetic

1. Introduction

General-purpose microprocessors and digital signal processors continuously improve their floating point capabilities, due to the increasing importance of new workloads (for instance digital signal processing and 3D graphics) that require floating-point processing. The majority of FPUs (Floating Point Units) in high-performance microprocessors and digital signal processors are IEEE-754/854 compliant, and support single, double and extended precision (or even quad precision). These units implement the basic floating-point operations in hardware: addition/subtraction, multiplication (or multiply-accumulate), division and square-root (and more recently the reciprocal square-root for 3D graphics). However there is an increasing interest in having hardware support for transcendental functions such as sine, cosine, tangent, arctangent, logarithm and exponential. Current microprocessors implement transcendentals by table-driven algorithms [1–5]. These algorithms make use of the combination of look-up tables and a fast floating-point pipelined multiplier. For these implementations there is a trade-off between the number of multiplications and the size of the tables.

In this work we present an algorithm for the hardware implementation of the exponential function in double precision. The exponential function is of particular interest in applications such as scientific computations, digital signal processing, neural networks or physical models for high-quality computer graphics and animation. Moreover, it is also useful for the evaluation of cosh and sinh, which are of interest for digital signal processing. The algorithm we propose is suitable for implementation in a FPU with a IEEE compliant floating-point pipelined multiplier (or floating-point multiplier-accumulator). Therefore, the proposed implementation may be useful for general purpose microprocessors, digital signal processors,

*This work was partially supported by Xunta de Galicia under contract PGIDT99PXI20601A.

and system-on-a-chip implementations for specific applications.

A common requirement for transcendentals is to be computed within 1 ulp of precision. Recently, there has been interest in obtaining exactly rounded results (1/2 ulp) with hardware algorithms[1] [2, 6]. In this work we concentrate on the latency hardware tradeoff of the implementation, so that we design the algorithm with the standard requirement of 1 ulp of precision.

The structure of the paper is as follows. In Section 2 we discuss general issues concerning the floating point computation of the exponential, and the existing algorithms for the computation of the significand. In Section 3 we present our algorithm. Section 4 shows the implementation using a floating-point multiplier. In Section 5 we compare it with existing algorithms, and finally we present the conclusions.

2. Double Precision Floating-Point Exponential Computation

We assume an input argument X represented in IEEE double precision format in the form $X = (-1)^s \cdot M \cdot 2^e$, where $M \in [1, 2)$ and $e \in [-56, e_m]$. The lower limit on e prevents the underflow exception, and the upper limit should verify $e_m \leq 11$ to prevent overflow, since the result of the computation must be a representable IEEE double precision number.

A possible sequence of steps for the floating-point exponential evaluation is as follows:

1. Denormalize the input argument so that $X = z + x$, where z is the integer part and x the fractional part. To simplify the computation of the exponential of the fractional part, it is useful to have X represented in two's complement. This implies that the fractional part after the conversion (x) takes values in the positive interval $[0, 1)$. In this work we adopt this scheme.[2]

   Therefore, z is an integer with e_m input bits in two's complement, and x is a fractional number within the interval $[0, 1)$. This allows the computation of the exponential as the product of two subexponentials (exponential of the integer part, and exponential of the fractional part of X) since

$$\exp(X) = \exp(z + x) = \exp(z) \times \exp(x)$$

2. Compute:

   - The exponential of the integer part of X ($\exp(z)$). This result is obtained in the form $\exp(z) = y \times 2^{e_y}$, where y takes values within the interval $[1, 2)$, and e_y is the partial exponent of the result. In [7] a method based on table look-up is proposed to compute this part.[3]
   - Compute $y \times \exp(x)$, which determines the significand part of the result. In this work we concentrate on this evaluation. A final step of adjustment of the exponent e_y may be required to normalize the result.

An important design parameter is the datapath width available for the computation, since this determines the truncation errors. In this work, we assume that a datapath of 63 fractional bits is available.[4]

We now review several existing methods to compute $y \times \exp(x)$. Existing algorithms (basically table-driven algorithms, based on the combination of tables and polynomial approximations and digit-recurrence algorithms[5]) and the algorithm we propose, are based on the following property of the exponential function

$$y \times \exp(x) \approx y \times \exp\left(\sum_{j=1}^{k} a[j]\right) = y \times \prod_{j=1}^{k} \exp(a[j]) \tag{1}$$

That is, the argument may be decomposed in k terms,[6] so that the computation of the exponential is reduced to: (1) computation of the exponential of each part (subexponentials) and (2) multiplication of the subexponentials to obtain the final result.

For instance, when $a[j] = \ln(1 + d_j 2^{-j})$, and $k = n$, where n is the required fractional-bit precision of the result (this corresponds roughly to a precision of 2^{-n} in the exponential calculation) we obtain the well-known radix-2 *digit-recurrence algorithm*[7] to compute $y \times \exp(x)$ [2, 8–11]. In this case the d_j values are selected in each iteration from a digit set $\{\alpha, \ldots, \beta\}$ (usually $\{0, 1\}$, $\{-1, 1\}$ or $\{-1, 0, 1\}$) so that

$$x \approx \sum_{j=1}^{n} \ln(1 + d_j\, 2^{-j}) \tag{2}$$

and therefore

$$\begin{aligned} y \times \exp(x) &\approx y \times \prod_{j=1}^{n} \exp(\ln(1 + d_j\, 2^{-j})) \\ &= y \times \prod_{j=1}^{n} (1 + d_j\, 2^{-j}) \end{aligned} \tag{3}$$

This algorithm is conventionally implemented by means of two recurrences: a recurrence v to compute (3), and a recurrence c to obtain the digits d_j to assure (2).

These recurrences are obtained by defining

$$v[j] = y \times \prod_{i=1}^{j}(1 + d_i\ 2^{-i})$$

$$\text{and} \quad c[j] = \sum_{i=1}^{j} \ln(1 + d_i\ 2^{-i})$$

resulting in

$$v[j+1] = v[j] + d_j\ 2^{-j}\ v[j]$$
$$c[j+1] = c[j] - \ln(1 + d_j\ 2^{-j})$$

with $v[1] = y$, $c[1] = x$, $1 \leq j \leq n$, and $d_j = \text{SEL}(c[j])$, where SEL( ) is the selection function, so that $c[j]$ tends to zero (and therefore $\sum_{j=1}^{n} \ln(1 + d_j\ 2^{-j}) \rightarrow x$).

The basic hardware requirements are: two adders, a shifter and a small table look-up to store the logarithms. Roughly, for n fractional bit-precision, n iterations are required. Conventional implementations use carry-propagate adders. To reduce the latency, there are implementations with carry-free additions [8], and unfolding of several iterations with improved schemes for a fast selection [10].

This idea has been extended to a higher radix than two (see [12] for a radix-16 implementation and [13] and [14] for a very-high radix implementation), which reduces the number of iterations. Although the iteration and the selection of the digits d_j are more complex, these implementations obtain a significant speed-up.

Table-driven algorithms are another approach, which in a simple implementation (see [15] for instance[8]) take $k = 2$, and therefore

$$y \times \exp(x) = y \times \exp(a[1]) \times \exp(a[2])$$

In this case $a[1]$ corresponds to a reduced number of leading bits of x, and $a[2]$ corresponds to the remaining bits. The value of $\exp(a[1])$ is obtained from a look-up table (the size of this table depends on the number of bits of $a[1]$) and is multiplied by y to obtain the partial result $R = y \times \exp(a[1])$. The value of $\exp(a[2])$ is approximated by a polynomial $P = 1 + p_o + p_1 r + p_2 r^2 + \cdots + p_t r^t$ (for instance a MacLaurin series expansion). The final result is obtained by the product $R \times P$, by means of the Horner's rule, using multiply-accumulate operations. Note that there is a trade-off between the size of the initial table and the degree of the polynomial, which determines the number of multiply-accumulate operations.

Variations of this basic approach are proposed in [7, 16, 17]. These algorithms use rectangular multipliers apart from a full multiplier, and reduce the latency significantly by introducing hardware parallelism.

For the algorithm proposed in [7], $k = 3$, so that

$$y \times \exp(x) = y \times \exp(a[1]) \times \exp(a[2]) \times \exp(a[3])$$

The values of $a[1]$ and $a[2]$ are chosen so that the number of bits of their exponentials result in a reduced fraction of the word size, allowing rectangular multiplications. Additional tables are necessary to obtain the exponentials with a reduced number of bits. The exponential $\exp(a[3])$ is evaluated by means of a polynomial (a MacLaurin series expansion in the case of [7]). This polynomial is obviously of lower degree than in the basic table-driven algorithm. In the algorithm proposed in [16], $k = 1$ so that $y \times \exp(x) = y \exp(a[1])$. The function $\exp(a[1])$ is approximated by a polynomial (specifically matched interpolation polynomials) in a point determined by the 14 leading bits of $a[1]$. Therefore the coefficients of the polynomial are determined by these 14 leading bits. They use a third order polynomial, using a second order interpolation to compute some of the terms (with less significance). The multipliers are simplified according to the significance of the terms computed.

For the algorithm proposed in [17], $k = 4$, so that

$$y \times \exp(x) = y \times \exp(a[1]) \times \exp(a[2] + a[3] + a[4])$$

The argument $a[1]$ corresponds to the 14 leading bits of x, so that $\exp(a[1])$ is obtained from a look-up table. The remaining bits of x are divided in groups of 14 bits with a decreasing weight. Then $\exp(a[2] + a[3] + a[4])$ is computed with a third degree MacLaurin series expansion. The polynomial is expanded in terms of $a[2]$, $a[3]$ and $a[4]$ so that the terms with weight lower than the required precision are avoided in the computation. The resultant expression corresponds to small multiplications and additions between the terms $a[2]$, $a[3]$ and $a[4]$.

In [18] a table driven algorithm is also considered but using rational approximations. The drawback of this approach is that it needs a fast divider to be competitive.

In Section 5 we discuss more details of these algorithms and their implementation, and compare them with our proposal. Specifically, we compare our proposal with the algorithms proposed in [7, 10, 15, 17]. These designs are representative of the existing approaches to compute the exponential in double precision. We concluded that the designs with better trade-off between latency and hardware complexity are [15, 17] and our design. The algorithm proposed in [15] represents a low hardware complexity solution, with 1/5th of the additional hardware (apart from an existing floating point multiplier) compared with our design, but requires 1.8 times more latency. The design of [17] represents a low latency solution with 0.9 times the latency of our scheme, but with 2.5 more additional hardware.

3. An Algorithm for Exponential Calculation

In this section we propose an algorithm to compute the exponential function. As with existing algorithms, our approach consists in dividing the argument of the exponential into three parts, computing each of the subexponentials in the most convenient way, and cross-multiplying the results. In Fig. 1 we show the division of the argument in three parts (A, B and C), and their corresponding binary weights[9] (determined by p and q). Therefore, a direct implementation results in

$$y \times \exp(x) = y \times \exp(A) \times \exp(B) \times \exp(C)$$

However, to improve the implementation, we introduce an operand D, so that the exponential is decomposed as

$$y \times \exp(x) = y \times \exp(A) \times \exp(D) \times \exp(K)$$

where $K = C + (B - D)$, and D represents an approximation of B to facilitate a fast implementation. Since D is an approximation of B, we included the term $(B - D)$ in the computation of the third subexponential ($B - D$ should be of the same order as C, that is of the order of 2^{-q}). The introduction of D gives more flexibility to the design process and permits us to determine the most desirable tradeoff between latency and hardware complexity.

Our design is a compromise among the following objectives:

- Implement the exponential using a floating-point pipelined multiplier, and additional hardware.
- Minimize the number of cycles (latency) and maximize the throughput of the exponential function.
- Maintain the performance of the multiplication instruction.

In the following subsections we show the method used to compute each of the subexponentials. At the end of the section we present a design process that permits the selection of the parameters of the design (for instance, the values of p and q) to achieve the desired trade-off. We first discuss the computation of $y \times \exp(A)$ and $\exp(K)$ since we use conventional methods in these parts. After that, we discuss the implementation of $\exp(D)$ that requires a more detailed explanation.

3.1. *Computation of $y \times exp(A)$*

We use a table look-up directly to compute $\exp(A)$. Conventional methods to approximate the exponential (for instance a polynomial) are not suitable for this part, because of the rough granularity of A (p is a low value compared with the wordlength of the operands). This is the conventional approach used in existing algorithms based on tables and polynomials.

Since A corresponds to the bit stream $[x_1, \ldots, x_p]$, the table should have p input bits. The number of output bits corresponds to the wordlength used in the datapath. To complete the operation, the output of the table is multiplied by y, requiring a full multiplication.

3.2. *Computation of exp(K)*

Since the argument of this exponential should have a considerable number of leading zeros (the argument should be of the order of 2^{-q}), we use a direct

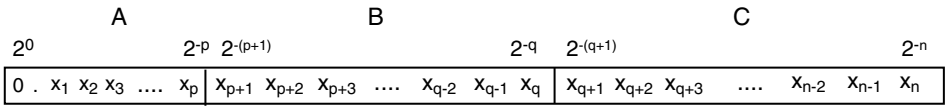


Figure 1. Decomposition of the argument in A, B and C.

polynomial approximation of the exponential around zero. This is also the usual approach followed by existing algorithms based on tables and polynomials.

Therefore, the exponential may be approximated by

$$\exp(K) \approx P(K,t) = p_0 + p_1 K + p_2 K^2 + p_3 K^3 + \cdots + p_t K^t$$

where the p_i's are the polynomial coefficients, and their values depend on the actual type of polynomial approximation used. Note that the polynomial is of degree t.

We define Δ as

$$\Delta = |\exp(K) - P(K,t)| \tag{4}$$

which represents the error of the polynomial approximation. Δ is determined by the maximum value of K (which is determined by the value of D, since $K = C + (B - D)$), and the type and degree of the polynomial used. Moreover, the maximum value of Δ is constrained by the desired accuracy of the result.

It is useful to obtain a bound on Δ. This bound will be used later in the design process to select the parameters of the algorithm. To obtain this bound it is necessary to choose the type of polynomial approximation. For this work we use a MacLaurin series polynomial. Therefore

$$P(K,t) = \sum_{j=0}^{t} \frac{1}{j!} K^j$$

Moreover, since

$$\exp(K) = \sum_{j=0}^{\infty} \frac{1}{j!} K^j$$

From expression (4) we obtain

$$\Delta = \sum_{j=t+1}^{\infty} \frac{1}{j!} K^j$$

A bound of this value is given by [19]

$$\Delta \le \frac{1}{(t+1)!} K^{t+1} \left(1 + \frac{K}{2}\right) \tag{5}$$

3.3. Computation of exp(D)

For the computation of this subexponential, first we have to determine the expression of D. We have the following constrains

- D should be selected to obtain a small value of K (which implies that D should be a good approximation of B, with a precision of the order of 2^{-q}), since we want to obtain the third subexponential by a polynomial approximation (note that $K = C + (B - D)$, and that C and B are part of the input operand).
- The implementation of $\exp(D)$ should be fast and cost-effective.

To achieve these goals, we formulate the computation of this subexponential in terms of the radix-2 exponential algorithm reviewed in the introduction. However, since in the conventional implementation this scheme is slow, as we show later, we modify the implementation to have a fast computation. Therefore, we use the following expression for D

$$D = \ln\left(1 + d_{p+1}\left(2^{-(p+1)} + a\, 2^{-2(p+1)-1}\right)\right) + \sum_{j=p+2}^{q} \ln(1 + d_j\, 2^{-j}) \tag{6}$$

where the values of a and d_j are in the digit-set $\{0, 1\}$. The parameter a permits (or not) a double shift in the first logarithm, and its usefulness will be justified later.

The value of D is determined by the sequence of d_j values (note that a is an algorithm parameter to be determined in the design process). Therefore the value of K (argument for the third subexponential) is given by

$$K = C + \left(B - \ln\left(1 + d_{p+1}\left(2^{-(p+1)} + a\, 2^{-2(p+1)-1}\right)\right) + \sum_{j=p+2}^{q} \ln(1 + d_j\, 2^{-j})\right) \tag{7}$$

Taking into account the expression of D, the computation of the exponential $\exp(D)$ is transformed into the following product

$$\exp(D) = \exp\left(\ln\left(1 + d_{p+1}\left(2^{-(p+1)} + a\, 2^{-2(p+1)-1}\right)\right) + \sum_{j=p+2}^{q} \ln(1 + d_j\, 2^{-j})\right) = \left(1 + d_{p+1}\left(2^{-(p+1)} + a\, 2^{-2(p+1)-1}\right)\right) \times \prod_{j=p+2}^{q} (1 + d_j\, 2^{-j}) \tag{8}$$

which, as mentioned before, corresponds to the basic radix-2 digit-recurrence algorithm, that with a straightforward implementation does not result in a fast solution. We now show a modified implementation of this algorithm to satisfy the design goals.

The issues to consider are:

- Fast determination of the d_j values.
- Fast computation of

$$\exp(D) = \left(1 + d_{p+1}\left(2^{-(p+1)} + a\,2^{-2(p+1)-1}\right)\right) \times \prod_{j=p+2}^{q} (1 + d_j\,2^{-j}) \tag{9}$$

- Computation of K (see (7)), which is the argument for the third exponential, with a reduced hardware complexity.

3.3.1. Determination of the d_j Values. To obtain a fast determination of the d_j values[10] we use a prediction algorithm, a technique first used for CORDIC algorithms [20]. This method allows us to obtain all the sequence of d_j's in parallel, permitting a parallel and fast computation of (9).

Now we obtain a bound on K using the prediction scheme for the selection of the d_j values, in terms of the parameters a, p and q. This bound is useful to relate all of the design parameters of the algorithm.

Since K is the argument of the third subexponential, which is computed by a polynomial approximation, it is desirable to obtain a small value of K. Taking into account that $B = \sum_{j=p+1}^{q} x_j$ and $C = \sum_{j=q+1}^{63} x_j$, K is given by the following expression

$$\begin{aligned} K = {} & \left(x_{p+1}\,2^{-(p+1)}\right. \\ & \left. - \ln\left(1 + d_p\left(2^{-(p+1)} + a\,2^{-2(p+1)-1}\right)\right)\right) \\ & + \sum_{j=p+2}^{q} (x_j\,2^{-j} - \ln(1 + d_j\,2^{-j})) + \sum_{j=q+1}^{63} x_j\,2^{-j} \end{aligned} \tag{10}$$

Prediction is based on selecting $d_j = x_j$ for $j = p+1$ to q. This is based on the fact that when $d_j = x_j$

$$x_j 2^{-j} - \ln(1 + d_j\,2^{-j}) = d_j^2 2^{-2j-1} - \cdots \tag{11}$$

which is small, and decreases as j increases, and therefore it may be a potential selection scheme suitable for obtaining a small value of K. Moreover, for the first logarithm, when $d_{p+1} = x_{p+1}$ we have

$$\begin{aligned} & x_{p+1}\,2^{-(p+1)} - \ln\left(1 + d_{p+1}\left(2^{-(p+1)} + a\,2^{-2(p+1)-1}\right)\right) \\ & \quad = d_{p+1}(d_{p+1}\,a\,2^{-3p-4} + d_{p+1}\,a^2\,2^{-4p-7} \\ & \qquad + 2^{-2p-3}(d_{p+1} - a)) - \cdots \\ & \quad = d_{p+1}(a\,2^{-3p-4} + a^2\,2^{-4p-7} \\ & \qquad + 2^{-2p-3}(1 - a)) - \cdots \end{aligned} \tag{12}$$

Therefore, when $a = 0$ this difference is of the order of 2^{-2p-3}, and is reduced to about 2^{-3p-4} when $a = 1$. Thus, the introduction of a double shift in the first logarithm leads to smaller differences between the logarithm and the power of two, allowing to obtain even a smaller value of K. As we explain later, we do not consider double shifts (or multiple shifts) in the remaining logarithms, since the reduction in the difference is not significant, and the increase in hardware complexity for the evaluation of $\exp(D)$ is large.

From (10)–(12), we obtain a bound on K

$$\begin{aligned} K \le {} & d_{p+1}(a\,2^{-3p-4} + d_{p+1}\,a^2\,2^{-4p-7} \\ & + 2^{-2p-3}(d_{p+1} - a)) - \cdots \\ & + \sum_{j=p+2}^{q} \left[d_j^2\,2^{-2j-1} - \cdots\right] + \sum_{j=q+1}^{63} (x_j\,2^{-j}) \end{aligned} \tag{13}$$

Since

$$\begin{aligned} & \sum_{j=p+2}^{q} d_j^2\,2^{-2j-1} - \cdots < \sum_{j=p+2}^{q} 2^{-2j-1} \\ & \quad = \frac{1}{24}(2^{-2p} - 2^{-2q}) < \frac{1}{24} 2^{-2p}, \end{aligned} \tag{14}$$

$$\sum_{j=q+1}^{63} x_j\,2^{-j} < \sum_{j=q+1}^{63} 2^{-j} = 2^{-q} - 2^{-63} < 2^{-q} \tag{15}$$

and

$$\begin{aligned} & d_{p+1}(a\,2^{-3p-4} + d_{p+1}\,a^2\,2^{-4p-7} + 2^{-2p-3}(d_{p+1} - a)) \\ & \quad - \cdots < a\,2^{-3p-4} + a^2\,2^{-4p-7} + (1 - a)2^{-2p-3} \end{aligned} \tag{16}$$

a condition that verifies (13) is

$$K \le 2^{-q} + \frac{(4 - 3a)}{24} 2^{-2p} + a\left(\frac{1}{16} 2^{-3p} + \frac{a}{128} 2^{-4p}\right) \tag{17}$$

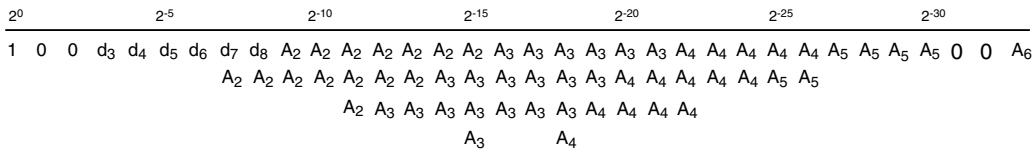


Figure 2. Computation of $\exp(D)$ as a sum of terms (example for $p = 2$, $q = 8$ and $a = 0$; A_i means AND operation of i d_j values).

Therefore, we have obtained a bound on K from the design parameters p, q and a.

3.3.2. Fast Computation of exp(D). We now consider the computation of (9). The straightforward evaluation of this expression implies $p-q-1$ shift and add serial operations. This is a slow scheme for our implementation. However, this expression may be transformed by cross-multiplying all the terms, transforming (9) in a sum of terms. Since these terms depend on the d_j values, which are obtained in parallel, the sum of terms can be evaluated in a fast tree structure of counters.

In Fig. 2 we show an example, illustrating the computation of $\exp(D)$ as a sum of terms for the case $p = 2$, $q = 8$ and $a = 0$. The expression as a product of terms is of the form

$$\begin{aligned}\exp(D) = {} & (1 + d_3\,2^{-3}) \times (1 + d_4\,2^{-4}) \times (1 + d_5\,2^{-5}) \\ & \times (1 + d_6\,2^{-6}) \times (1 + d_7\,2^{-7}) \\ & \times (1 + d_8\,2^{-8}) \end{aligned} \tag{18}$$

where in a conventional implementation, five sequential add and shift operations would be needed. In Fig. 2 we show the equivalent expression as a sum of terms. The result of the transformation is an array of terms that can be evaluated in a fast tree-like structure. The terms are products of d_j values. Since d_j takes values in the set $\{0, 1\}$, the products are computed as simple AND operations. Note that we only consider double shifts in the first logarithm, since the introduction of additional shifts significantly increases the number of terms to be added.

The complexity (in time and hardware) of this computation depends on the values of the design parameters p, q and a. Therefore, the evaluation of $\exp(D)$ is an important consideration in the selection of the values of these design parameters.

We may introduce a trade-off between hardware complexity and computation time by decomposing $\exp(D)$ in several subproducts. For instance, considering two subproducts results in

$$\exp(D) = P_1 \times P_2 \tag{19}$$

with

$$\begin{aligned} P_1 = {} & \left(1 + d_{p+1}\left(2^{-(p+1)} + a\,2^{-2(p+1)-1}\right)\right) \\ & \times \prod_{j \in I_1} (1 + d_j\,2^{-j}) \end{aligned}$$

and

$$P_2 = \prod_{i \in I_2} (1 + d_i\,2^{-i})$$

where I_1 and I_2 are a set of indexes so that $I_1 \cup I_1 = \{p+1, \ldots, q\}$. For this case we introduce an additional multiplication, but the evaluation of P_1 and P_2 might result in less hardware complexity than the direct evaluation of $\exp(D)$. Note that the indexes in I_1 and I_2 may be selected to optimize the hardware complexity.

3.3.3. Computation of K. The evaluation of K (see expression (7)) may be performed using a tree structure since all the d_j values are determined in parallel. Therefore the computation of this part is fast.

The main concern regarding this part is the hardware complexity due to the addition of the constants $\ln(1 + d_j\,2^{-j})$. However, using a customized array (there are positions with a bit set always to zero, independent of the d_j value, or strings of the form $111\ldots1$ that may be recoded to $1000\ldots\bar{1}$) the hardware complexity may be significantly reduced. In Section 4.2 we show the simplifications to reduce the hardware complexity of this part.

3.4. Algorithm Design

In this section we perform the design process of the algorithm, by determining the relation of the design parameters a, p and q, with the required accuracy of the algorithm, the latency and the hardware complexity.

3.4.1. Design Parameters and Accuracy of the Algorithm. We want results for double precision IEEE-754 format with one ulp of accuracy (that is, the maximum error after truncation should be bounded by 2^{-52}).

The components of the error are as follows:

- *Approximation error of the third subexponential* (δ). The error in the approximation (by a polynomial) of the third subexponential, Δ, is multiplied by the value of the first two subexponentials. However, as the multiplications are performed, we assume that left shifts are performed after every multiplication to have the result within $[1, 2)$. This way, this result can be used again directly in the multiplier. Moreover, when left shifts are performed, it is necessary to increase the exponent of the final result. Therefore, the factor that multiplies the error of the third subexponential is less than 2.

  We then conclude that

$$\delta \leq 2 \times \Delta$$

  Moreover, from (5) we obtain the bound

$$\delta \leq 2 \times \left(\frac{1}{(t+1)!} K^{t+1} \left(1 + \frac{K}{2} \right) \right)$$

- *Truncation errors due to a finite datapath width* (α). The value of these truncation errors depends on the number of multiplications performed, and the datapath width (m) used. At this point the value of α cannot be determined. Therefore, we first perform the design of the algorithm for $m = \infty$. Later we determine the datapath width required to maintain the same accuracy.[11]
- *Rounding error* (β). Since the result of the exponential has more bits than required by the format, it is necessary to round the result. Since the weight of the least significant bit of the format is 2^{-52}, this process introduces an error bounded by $\beta \leq 2^{-53}$.

Therefore, the bound for the total error is

$$\delta + \alpha + \beta \leq 2 \times \left(\frac{1}{(t+1)!} K^{t+1} \left(1 + \frac{K}{2} \right) \right) + \alpha + 2^{-53}$$

Since we require that

$$\delta + \alpha + \beta < 2^{-52}$$

and taking into account the bound on K given in (17), we obtain the condition:

$$\begin{aligned} 2 \frac{1}{(t+1)!} \Bigg(& \left[2^{-q} + \frac{(4-3a)}{24} 2^{-2p} + a \left(\frac{1}{16} 2^{-3p} + \frac{1}{128} 2^{-4p} \right) \right]^{t+1} \\ & + \frac{1}{2} \left[2^{-q} + \frac{(4-3a)}{24} 2^{-2p} + a \left(\frac{1}{16} 2^{-3p} + \frac{1}{128} 2^{-4p} \right) \right]^{t+2} \Bigg) \\ & + \alpha + 2^{-53} < 2^{-52} \end{aligned} \tag{20}$$

As mentioned before, we cannot determine a bound on α at this point of the design process. Therefore, we assume a value of $\alpha = 0$ (that is $m = \infty$) to obtain an expression that relates all the design parameters, except the datapath width (m). Later, with actual values of a, p and q, we determine a maximum bound for α (using the previous expression) and then we determine the datapath width.

Therefore, we obtain the final expression that relates the design parameters (except m)

$$\begin{aligned} \frac{1}{(t+1)!} \Bigg(& \left[2^{-q} + \frac{(4-3a)}{24} 2^{-2p} + a \left(\frac{1}{16} 2^{-3p} + \frac{1}{128} 2^{-4p} \right) \right]^{t+1} \\ & + \frac{1}{2} \left[2^{-q} + \frac{(4-3a)}{24} 2^{-2p} + a \left(\frac{1}{16} 2^{-3p} + \frac{1}{128} 2^{-4p} \right) \right]^{t+2} \Bigg) < 2^{-54} \end{aligned} \tag{21}$$

3.4.2. Design Space and Trade-Offs. From Eq. (21) we obtain values of the design parameters p, q and t, that determine different design solutions in terms of hardware complexity and latency. For each value of a (0 and 1), and t (we considered values of t from 1 to 4) we obtained the minimum values for p and q (which corresponds to the best solution given a value of a and t). For each parameter set we determined the latency and the additional hardware complexity (hardware complexity required apart from the existing multiplier). For the estimation of the hardware complexity we used a first order model, computing the equivalent number of simple gates for each design.

Table 1. Comparison for different design parameters.

a	t	p	q	$\lvert p-q\rvert$	Table size	$\exp(D) = P_1 \times P_2$	Complexity[a]	Number of $\times$[b]	Latency (cycles)
	1	14	27	13	14×64	No	141	3 (1)	8
						Yes	140	4 (2)	9
	2	9	18	9	9×64	No	6.5	4 (2)	9
0						Yes	5.5	5 (2)	9[c]
	3	7	13	6	7×64	No	1.6	6 (4)	11[c]
						Yes	1.4	7 (2)	11[c]
	4	5	10	5	5×64	No	1.1	7 (5)	13[c]
						Yes	1.1	8 (5)	13[c]
	1	12	27	15	12×64	No	40	3 (1)	8
						Yes	36	4 (2)	9
	2	7	18	11	7×64	No	9	4 (2)	8[c]
1	2	7	18	11	7×64	Yes	2.3	5 (2)	9[c]
	3	5	13	8	5×64	No	2.9	6 (4)	11[c]
						Yes	1.2	7 (4)	11[c]
	4	3	10	7	3×64	No	1.9	7 (5)	13[c]
						Yes	1	8 (5)	13[c]

[a]Ratio between the additional complexity of each design and the least complex design.
[b]Total Multiplications (dependent multiplications).
[c]Table look-up or evaluation of products are fast enough to be evaluated in the first cycle.

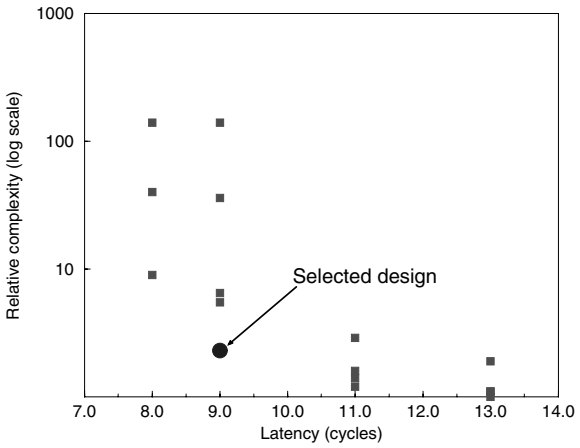


Figure 3. Design space.

In Table 1 we show the results obtained. The additional hardware complexity is given in relative terms. In Fig. 3 we represent the data from Table 1 (latency and relative hardware complexity). This graph gives us a clear idea of the tradeoff between hardware complexity and latency, and permits to design the algorithm according to the desired tradeoff.

To illustrate the implementation of the algorithm in this work, we select the design solution with best balanced tradeoff, which is given in Table 1 within a rectangle (and corresponds to the design marked with a circle in graph 3). Therefore for the implementation we use the design parameters: $a = 1$, $t = 2$, $p = 7$, $q = 18$, decomposing the computation of $\exp(D)$ into two products.

In Fig. 4 we show the sequence of operations that corresponds to the design parameters selected. The operations in the same row can be performed in parallel. From Fig. 4, the algorithm we implement requires the following steps (the R_i values correspond to partial results):

1. Evaluation of P_1 and P_2:

$$P_1 = (1 + d_8(2^{-8} + 2^{-17})) \prod_{j=10,11,12,16} (1 + d_j 2^{-j}) \tag{22}$$

and

$$P_2 = \prod_{j=9,13,15,17,18} (1 + d_j 2^{-j}) \tag{23}$$

This computation is performed in parallel to steps 3 and 5. The indexes of each product were selected

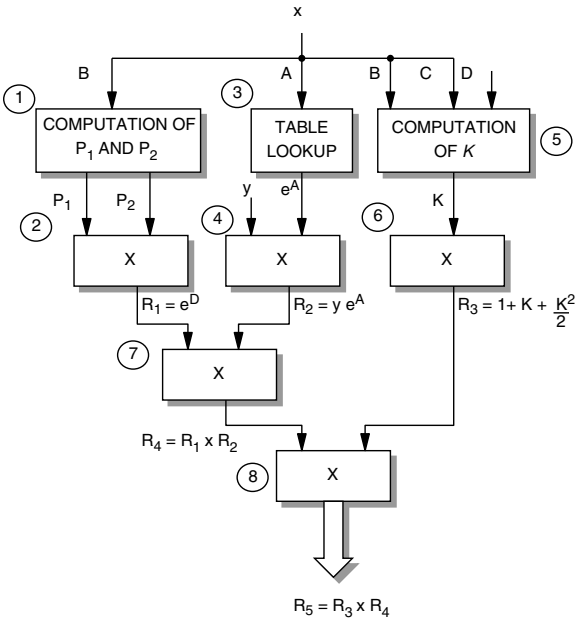


Figure 4. Flow diagram for the algorithm.

to reduce the hardware complexity of the evaluation of P_1 and P_2.

2. Multiplication 1

$$R_1 = P_1 \times P_2 \tag{24}$$

performed in parallel to steps 4 and 6.

3. Evaluation of $\exp(A) = \exp(x_1, \ldots x_7)$ using a table look-up with 7 input and m output bits.
4. Multiplication 2

$$R_2 = y \times \exp(A) \tag{25}$$

5. Evaluation of K

$$K = \sum_{j=19}^{63} x_j\, 2^{-j} + (x_8\, 2^{-8} - \ln(1 + d_8(2^{-8} + 2^{-17}))) + \sum_{j=9}^{18} (x_j\, 2^{-j} - \ln(1 + d_j\, 2^{-j})) \tag{26}$$

6. Evaluation of $\exp(K)$ by means of the polynomial approximation $1 + K + \frac{1}{2}K^2$, implementing the operation $1 + K \times (1 + \frac{K}{2})$. To obtain $(1 + \frac{K}{2})$, K is shifted right one bit and the most significant bit is set to one. After this, the following multiplication is performed

$$R_3 = K \times \left(1 + \frac{K}{2}\right) \tag{27}$$

To complete the polynomial approximation the most significant bit of R_3 is set to one.

7. Multiplication 4

$$R_4 = R_1 \times R_2 \tag{28}$$

This step depends on steps 2 and 4.

8. Multiplication 5

$$R_5 = R_3 \times R_4 \tag{29}$$

This step depends on steps 6 and 7.

Now we have to determine the required datapath width (m) for the design parameters selected. We proceed as follows:

- From the sequence of operations, determine an expression for α (truncation error due to a finite datapath width) in terms of m.
- From (20), using the selected values of a, t, p and q, and the expression of α in terms of m, we obtain a lower bound for m.

In [19] we obtain the following expression for α:

$$\alpha = 2^{-m+5} + 2^{-m-p+3} + 2^{-m-q+5} + 2^{-m-q+2} + 2^{-m-p-q+4}$$

Substituting this value in expression (20) and using the actual values of the design parameters, results in the condition

$$2^{-m}(2^5 + 2^{-4} + 2^{-13} + 2^{-16} + 2^{-21}) + \frac{9}{5}2^{-53} < 2^{-52} \tag{30}$$

From this expression we obtain $m \geq 59$. Therefore our algorithm can be implemented in a datapath with 63 fractional bits, which is a common choice in double precision floating point units. In the following sections we show the implementation of this algorithm in a floating-point unit with an existing multiplier.

4. Implementation Using a Floating-Point Multiplier

Generally, the efficient computation of transcendental functions requires enhancing a standard floating-point unit's multiplier. We present here an implementation of our algorithm using a standard multiplier, minimizing

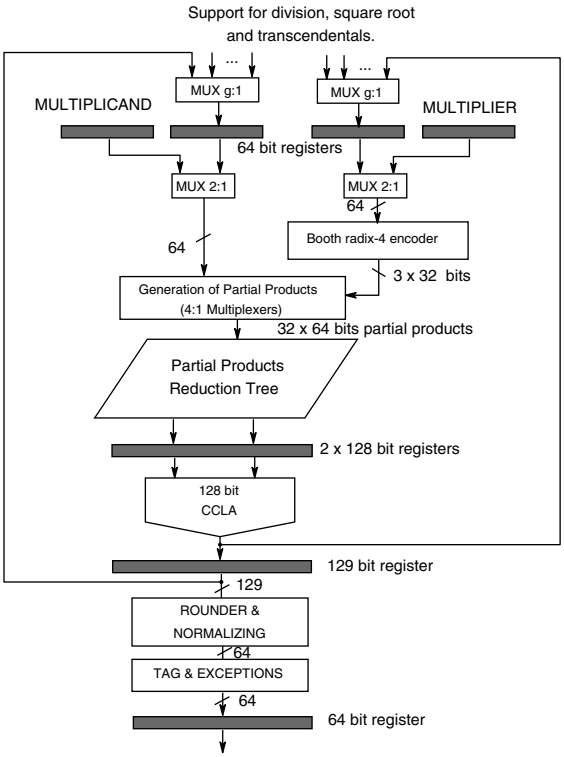


Figure 5. Architecture of the 64-bit × 64-bit multiplier.

the hardware added to the critical path of the multiplication operation to maintain the performance. Moreover, additional blocks are optimized to reduce the amount of hardware.

4.1. Floating-Point Multiplier

For the implementation we consider the 64-bit × 64-bit multiplier shown in Fig. 5. We assume a multiplier that includes the following functional stages to perform a complete IEEE floating-point multiplication[12]

- *Booth recoding and partial products generation*. In order to reduce the number of partial products the multiplier is recoded to radix-4 digits in the set $\{-2, -1, 0, 1, 2\}$.
- *Counter tree*. The partial products are reduced to a carry-save word in a tree structure of counters.
- *Carry propagate adder*. The redundant carry-save word is added in a fast carry propagate adder.
- *Rounding and normalization*. A final stage is needed for normalizing the significand and rounding the result according to the IEEE rounding modes.
- *Exception handling*. Overflow, underflow, invalid operation and inexact exceptions defined by the Standard must be determined and appropriate flags must be set.

We assume a pipelined implementation with three stages as indicated in Fig. 5. To allow the implementation of arithmetic algorithms (for instance quadratic convergence multiplicative division and square-root, and transcendentals) we assume a feedback (bypass) in the pipeline and multiplexers to select the suitable input to the multiplier.

4.2. Additional Hardware

First we need to obtain the expressions of $\exp(A)$, $\exp(D)$ and $\exp(K) = \exp(C + (B - D))$.

- For the evaluation of $\exp(A) = \exp(0.x_1x_2 \ldots x_7)$ we use a table look-up of 7 inputs and 64 outputs.
- The evaluation of

$$\begin{aligned}\exp(D) &= \exp(0.00 \ldots x_8x_9 \ldots x_{18}) \\ &= (1 + d_8 \cdot 2^{-8} + d_8 \cdot 2^{-17}) \\ &\quad \times \prod_{i=9}^{18} (1 + d_i \cdot 2^{-i})\end{aligned}$$

where $d_i = x_i$, and $i = 8, \ldots, 18$, is performed in a module that we call *generator of products*. The previous expression is decomposed into two parts

$$P_1 = (1 + d_8(2^{-8} + 2^{-17})) \prod_{i=10,11,12,16} (1 + d_i \cdot 2^{-i}) \tag{31}$$

and

$$P_2 = \prod_{i=9,13,14,15,17,18} (1 + d_i \cdot 2^{-i}) \tag{32}$$

and are transformed separately into a sum of terms. The selection of the product terms for P_1 and P_2 was optimized to reduce the hardware complexity. The implementation consist of two parts:

– Generate the AND of the corresponding d_j values.
– Reduction by means of a tree of counters to a binary value.

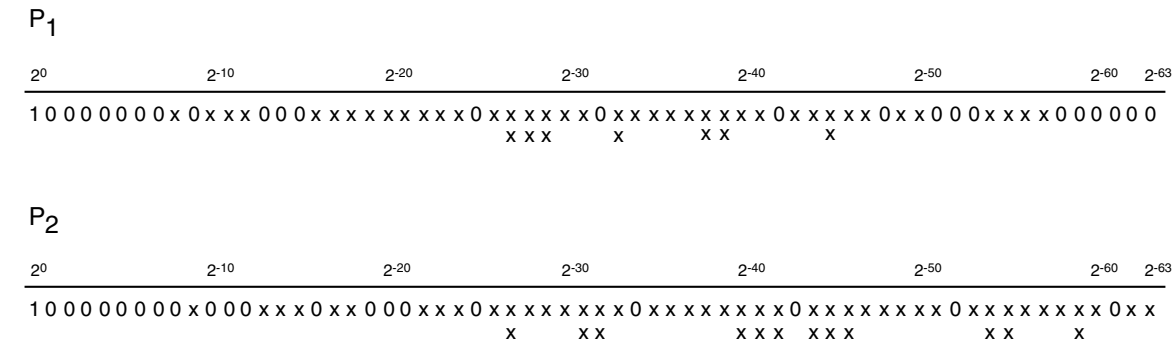


Figure 6. Representation of P_1 and P_2 as a sum of terms.

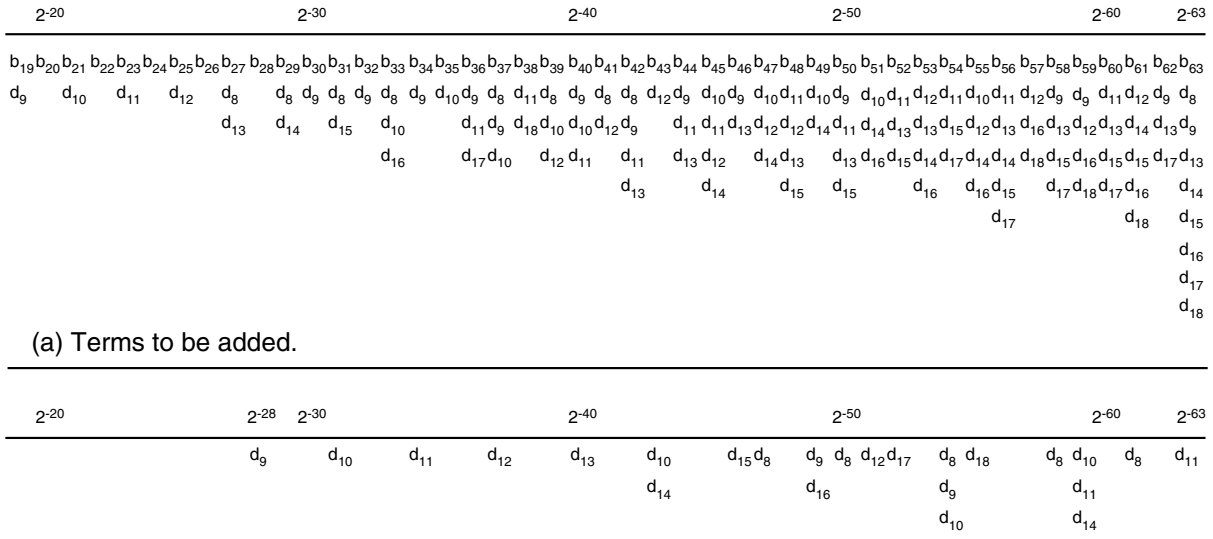


Figure 7. Addition of positive and negative values in the predictor.

Using the multiplier we obtain $\exp(D)$ introducing expression (31) as the multiplicand and expression (32) as multiplier.

Figure 6 shows the terms resulted from transforming P_1 and P_2 into a sum of terms. Each term is generated by an AND operation.

- The evaluation of K is performed in a module called *predictor*. It consists of an optimized tree of counters that compute

$$\sum_{i=9}^{18}(d_j 2^{-j} - \ln(1 + d_j 2^{-j})) + d_8 2^{-8} - \ln(1 + d_8(2^{-8} + d_8 2^{-17})) + \sum_{j=19}^{63} x_j 2^{-j} \quad (33)$$

An important simplification arises from the fact that there are bit positions with a zero independently of the d_j values. To reduce the complexity further, we transform patterns of the form $2^{-k} + 2^{-k-1} + \cdots + 2^{-l}$ into $2^{-k+1} - 2^{-l}$ where $l - k > 2$. Moreover, we add in a different tree of counters the weights with positive values, shown in Fig. 7(a) and the weights with negative values, shown in Fig. 7(b). Due to the simplicity of the pattern of negative terms we can make here many carry propagate additions of few bits obtaining a single word. Finally, another row of 3-to-2 counters subtract this binary word from the carry-save word of positive terms (after its reduction). A final carry propagate addition of 45 bits is performed with an area-optimized carry-propagate adder.

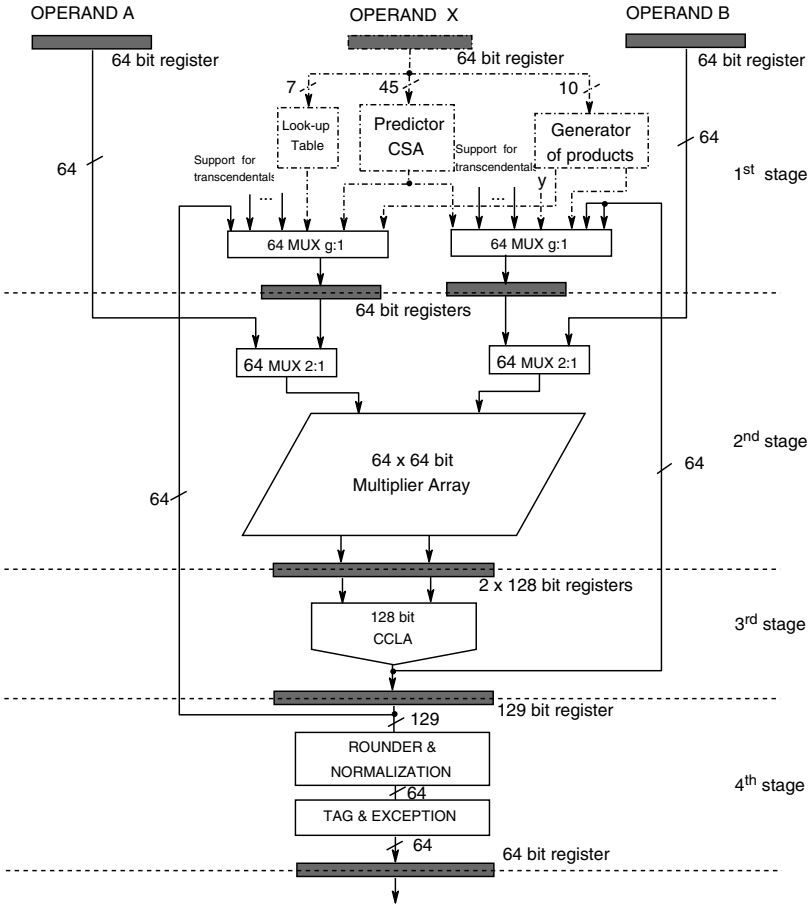


Figure 8. Architecture for IEEE double precision exponential evaluation.

Finally we use the computed value of K to approximate $\exp(K)$ by a polynomial using the multiplier.

Figure 8 shows the implementation of the enhanced floating-point multiplier to compute the exponential function. The additional hardware blocks are shown in dotted lines. In Table 2 we show the additional hardware needed for each part.

4.3. Sequence of Operations

In Fig. 9 we show the sequence of operations to compute the exponential. The exponential evaluation is completed with five multiplications in nine clock cycles. In the first cycle the products P_1 and P_2 are generated. As the generation of products has a short delay, the previous shifting and complementing required for a floating-point evaluation, can be performed in this cycle.[13] Moreover, the addition of a one needed to obtain the two's complement of the operand can be performed in the predictor, avoiding the delay of a carry-propagate adder. During the second cycle $\exp(A)$ is obtained from the table look-up, and the multiplier array computes the product $P = P_1 \times P_2$ in carry-save representation.

In the third cycle the predictor obtains the value of K. Before loading the multiplier register this value is shifted one position to the right, and the leading bit is

Table 2. Additional hardware.

Module	Hardware complexity
Table look-up 7×64	8192 bits
Generator products	75 and gates, 12 3:2 counters, 27 2:2 counters
Predictor	21 inverters, 26 2:2 counters, 116 3:2 counters 45 bit carry propagate adder
Registers	1×64-bit register

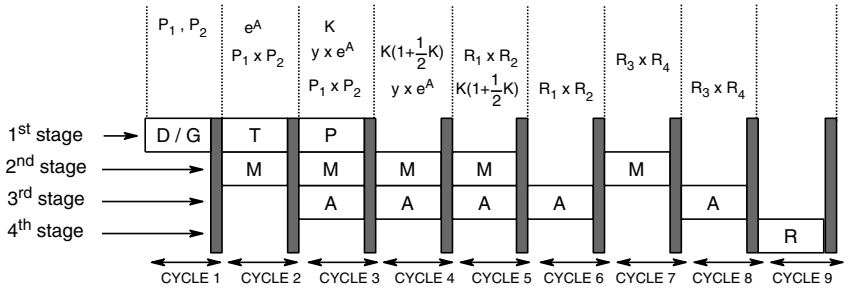


Diagram keys:
D : Denormalization.
G : Generation of products.
T : Table Look-up for e^A.
P : Prediction.
M: Booth encoding and partial products generation and reduction.
A: Carry Propagate Adder.
R: Rounding, normalizing and exception treatment.

Figure 9. Diagram of cycles for exponential evaluation.

changed from zero to one to obtain $1 + K/2$. Meanwhile, the multiplier array performs $R_1 = y \times \exp(A)$ and the fast carry-propagate adder obtains $R_1 = P_1 \times P_2$ as a binary word.

In the fourth cycle the value of R_1 is loaded into the register between the third and fourth stage. The multiplier array performs $R_3 = K \times (1 + K/2)$ and the adder reduces the carry-save value of $R_2 = y \times \exp(A)$ to a binary word. Note that two feeding back lines are needed, because a right shift of one bit may be necessary when R_2 belongs to the interval $[2, 4)$.

In the fifth cycle the multiplier array performs $R_4 = R_1 \times R_2$ and the adder obtains the binary word for $R_3 = K \times (1 + K/2)$. Before loading this result in the multiplier register through the non-shifted feeding back line, a leading bit must be added to obtain the correct polynomial approximation. Therefore a two input AND gate is placed before the multiplexers to introduce the leading bit when necessary. The multiplier register with the value R_3 will not be used until the seventh cycle.

In the sixth cycle the adder is the only stage in operation, obtaining the binary word of $R_4 = R_1 \times R_2$. The last multiplication $R_3 \times R_4$ is evaluated from cycles seven through nine, where a final normalization and rounding is performed.

4.4. Latency and Throughput

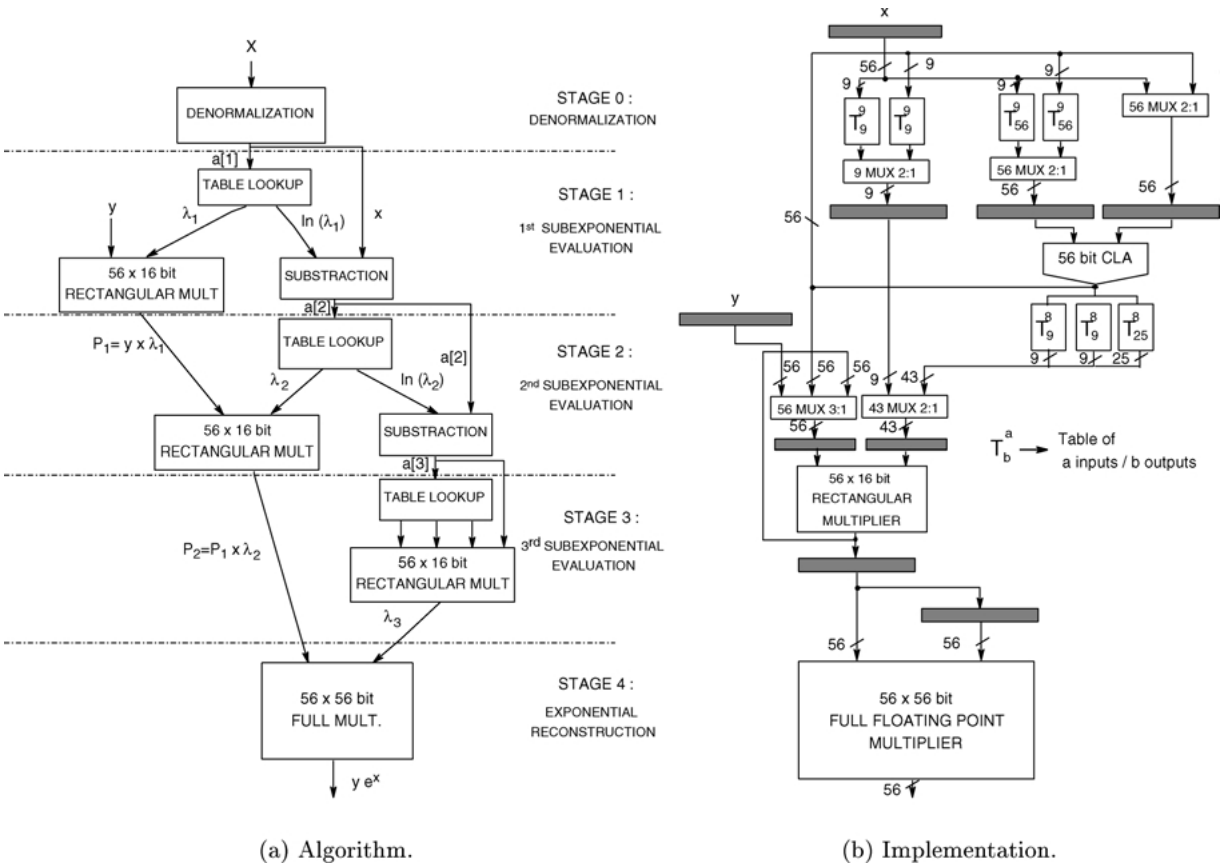
The latency of the exponential evaluation in the proposed architecture is of three multiplication times (nine cycles), while the latency for the multiply operation still remains in three cycles. The throughput of the exponential operation results in six cycles, since the first three cycles of each exponential evaluation can be overlapped with the three last cycles of the previous.

The throughput of the multiplication is of one cycle when only multiply operations are issued. However, this architecture allows the issue of multiplication and exponential operations, and therefore it is convenient to analyze the structural hazards for these operations. When intensive computation of both multiply and exponential operations is required, an exponential and two multiplications with a separation of five cycles can be issued every six cycles. When intensive multiplication is required, the introduction of a single exponential evaluation within the chain of multiplications requires the stalling of the issue of new multiplications for four cycles. When intensive evaluation of exponentials is required (this might be the case in specific applications), a single multiplication can be performed without changing the exponential instruction flow, if it is issued in the first or in the sixth cycle of any exponential.

5. Comparison

In this section we compare our design with four schemes proposed in the literature[14] [7, 10, 15, 17].

In order to evaluate the hardware complexity and latency, we assume a rough model based on the estimation of the total number of equivalent simple gates



(a) Algorithm. (b) Implementation.

Figure 10. Algorithm and implementation of [7].

and the equivalent number of simple gate levels along the critical path (t_g is the delay of a simple gate level). Although this is a very rough model (especially for deep submicron technologies) the ratios obtained might indicate the potential advantages and disadvantages of each scheme.

We assume 16 t_g for the delay of a carry propagate adder (full wordlength) and a hardware complexity of 14 N gates, where N is the length of the input operands to the adder. A 3:2 counter takes 4 t_g and has an equivalent complexity of 8 gates, while a 4:2 counter takes 6 t_g and has 17 gates. For h-to-1 multiplexers we estimate a complexity of 1.5 h gates and a delay of h t_g. For a N-bit register we assume a delay of 4 t_g and 4 N gates. To estimate the complexity and delay of the tables in gates we used synthesis tools (in some cases we extrapolated the synthesis results of one table to obtain the corresponding values for other tables without actually performing their synthesis).

As described in Section 2, the algorithm proposed in [7] decomposes the computation in three subexponentials. A graph of this algorithm is shown in Fig. 10(a). The first two subexponentials are evaluated so that the result has a small number of bits,[15] allowing the use of rectangular multiplications to obtain the final value. The third subexponential is approximated by a MacLaurin series expansion (quadratic approximation). To avoid the use of the full floating-point multiplier in this approximation, they use tables and take advantage of certain terms that do not contribute to the precision of the result. Therefore, for this MacLaurin series approximation, they use tables, additions and a rectangular multiplier. Finally, the exponential is obtained with two rectangular multipliers (multiplication of the first subexponential by y, and the result multiplied by the second subexponential) and a full multiplier to multiply by the result of the evaluation of the third subexponential.

In Fig. 10(b) we show an implementation of this algorithm. We considered that the rectangular multiplier-accumulator is also used as a multioperand addition unit for the MacLaurin series approximation.[16] In

Table 3. Additional hardware for the design of [7].

Module	Number of elements	Equivalent gates
2 Table look-up of $2^9 \times 56$	57344 bits	10000
2 Table look-up of $2^9 \times 9$	9216 bits	1600
2 Table look-up of $2^8 \times 9$	4608 bits	800
Table look-up of $2^8 \times 25$	6400 bits	1100
Rectangular multiplier	Radix-4 recoder	550
	Partial products generator	2250
	Reduction tree	3000
	72 bit CLA	1000
Counters	56 3:2 counters	475
Carry propagate adder	56 bit CLA	785
Multiplexers	164 mux 2:1	500
	56 mux 3:1	250
Registers	9-bit register	35
	7×56-bit regs	1570
	43-bit reg	175
Total		23500

Table 3 we show the additional hardware complexity for the implementation proposed in [7] using an existing FPU multiplier similar to the one shown in Fig. 5. An exponential evaluation can be performed in 9 cycles. We estimate that the initial denormalization of the significand is performed in the same cycle of the first look-up table of 9×56 bits. We assume that a table look-up and a carry-propagate addition of 56 bits can be performed in the cycle time of the FPU multiplier used in this work. For the 56×16 rectangular multiplier we estimate that a two pipelined stage implementation is required. In Fig. 11 we show a time diagram corresponding to this implementation.

The design proposed in [10] is based on a digit-recurrence algorithm using carry-save arithmetic. To determine the d_j values, small assimilations are used allowing a faster implementation, avoiding several wide carry propagate additions in each iteration. The implementation described performs eight unfolded iterations in one step (basically eight chained shift-and-add operations in carry-save using multiplexers, 4-to-2 and 3-to-2 csa). At the end of each step (eight iterations) a wide assimilation is performed. This algorithm does not make use of an existing floating-point multiplier, and all the hardware is dedicated. In Table 4 we show the hardware needed for this implementation. We show the hardware for the master part (that determines the d_j values) and for the slave part (shift-and-add operations). For double precision, this architecture requires the denormalization step, seven steps (8 bits of the result computed in each step) and the final normalization and rounding. We estimated that one step may be computed in four cycles (assuming the same cycle time used in our architecture). Therefore the latency of an exponential evaluation results in 1(denormalization) + 4(cycles/step) $\times$ 7(steps) + 1(normalization and rounding) = 30 cycles.

In [15] a scheme is proposed based on a table look-up and polynomial approximation for exponential evaluation.[17] In Table 5 we show the additional hardware needed to implement this algorithm in a three stage floating point multiplier. The table look-up of five inputs is used to store the exponential of the five leading bits of x. The table of three inputs is required to store the five coefficients of the polynomial. For a

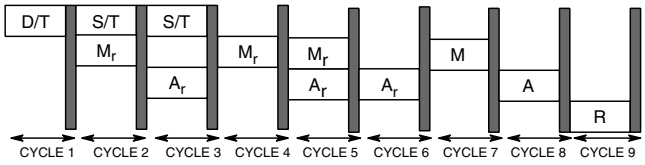


Diagram keys :
D : Denormalization .
T : Table lookup.
S : 56 bit carry-propagate subtraction.
M : Recoding, partial products generation and reduction (for the FULL MULTIPLIER).
M_r: Recoding, partial products generation and reduction (for the RECTANGULAR MULTIPLIER).
A : Carry-propagate addition (for the FULL MULTIPLIER).
A_r : Carry-propagate addition (for the RECTANGULAR MULTIPLIER).
R: Rounding, normalization and exception handling.

Figure 11. Time diagram for the implementation of the algorithm proposed in [7].

Table 4. Hardware complexity of the design proposed in [10].

Module	Number of elements	Equivalent gates
	Master	
Tables of logarithms	4728 bits	900
Carry propagate adders	7 10-bit CLA	980
	9-bit CLA	125
	71-bit CLA	1000
	64-bit CLA	890
Carry-save adders	540 3:2 counters	4600
AND gates	512 AND-2	512
Multiplexers	80 mux 2:1	270
	56 mux 3:1	250
Registers	64-bit reg	255
	Slave	
Carry propagate adder	2 × 64-bit CLA	1780
Carry-save adder	64 3:2 counters	545
6 Cascaded carry-save adders	768 mux 8:1	9200
	768 mux 2:1	2300
	768 4:2 compressors	13000
Multiplexers	192 mux 2:1	580
	128 mux 8:1	1535
Registers	64-bit reg	255
Total		38500

Table 5. Additional hardware for the architecture that implements the algorithm proposed in [15].

Module	Number of elements	Equivalent gates
Table look-up of 5 × 56	1800 bits	400
Table look-up of 3 × 56	450 bits	100
Registers	One 56-bit reg	220
Total		700

double precision exponential evaluation, it is necessary an access to a table look-up of five input bits, and six multiplications free of rounding for evaluating a sixth degree polynomial. A final complete floating-point multiplication is needed to reconstruct the exponential evaluation from partial results. The multiplications in the polynomial evaluation are performed sequentially using the Horner's rule. The initial significand denormalization, and the access to the table look-up of 5×56 is performed in the first cycle. After that, six dependent multiply-accumulate operations are performed, requiring 12 cycles. The multiplication of y by the content of the table look-up of five input bits is performed in parallel to the multiply-accumulate operations, and therefore no additional latency is required for this operation. Finally the result is obtained by means of a multiplication with a final normalization and rounding step (three cycles). A complete exponential evaluation is performed in 1(denormalization) + 2 × 6(mults. free of rounding) + 3(final mult. with rounding) = 16 cycles.

The method proposed in [17] is based on the following computation

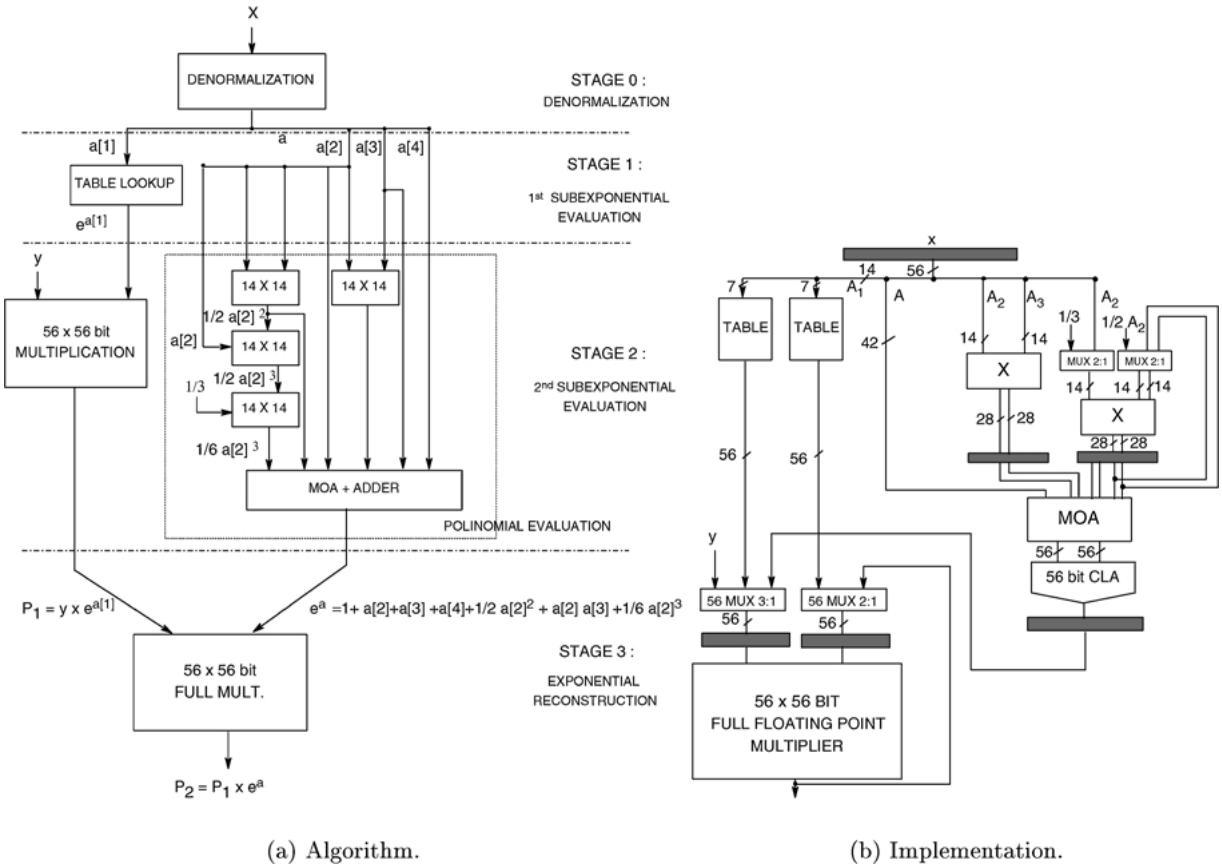
$$y \times \exp(x) = y \times \exp(a[1]) \times \left(1 + a[2] + a[3] + a[4] + \frac{1}{2}a[2]^2 + a[2] \times a[3] + \frac{1}{6}a[2]^3\right)$$

where each $a[i]$ has 14 bits. In Fig. 12(a) we show a graph that illustrates the algorithm. The computation of $\exp(a[1])$ is performed by a table look-up. This means that a table of 14×56 bit is necessary, which is a very large table. To have a moderate hardware cost we modify slightly the design of [17], by computing $\exp(a[1])$ as the product of $\exp(au[1]) \times \exp(al[1])$ where $au[1]$ are the 7 leading bits of $a[1]$ and $al[1]$ are 7 lower bits of $a[1]$. This way we introduce an additional full multiplication, but two tables of 7 inputs are used instead one table of 14 inputs. As we show below this modification does not introduce additional latency and saves a large hardware complexity.

In Fig. 12(b) we show a hardware implementation of the algorithm, and in Fig. 13 we show the time diagram. The two small multipliers of 14×14 bits perform the computation of $a[2]^2$ and $a[2] \times a[3]$ in parallel, and later perform sequentially[18] $a[2]^3 = a[2]^2 \times a[2]$ and $\frac{1}{6} \times a[2]^3$. These multipliers produce the result in carry-save so that we assume that their delay is within one cycle. The MOA module performs a multioperand addition. In parallel to the operation of the small multipliers, the full multiplier performs $y \times \exp(au[1]) \times \exp(al[1])$ without increasing the latency of the algorithm. Finally a full multiplication is performed to obtain the result. The computation of the exponential is performed in 8 cycles. In Table 6 we show the additional hardware required by this architecture.

In Table 7 we show the additional hardware required by our implementation. As was discussed in Section 4 the latency is of nine cycles.

Finally, in Table 8 we show the additional hardware complexity (apart from the existing floating-point multiplier) and latency comparison between the five implementations. For the implementation proposed in



(a) Algorithm. (b) Implementation.

Figure 12. Algorithm and implementation of [17].

[10] we assumed as additional hardware all the hardware needed, since the floating-point multiplier is not used in this algorithm. In Fig. 14 we represent the data of Table 8 to better appreciate the latency hardware trade-off for each of the designs. In this graph we also represent the estimated hardware complexity and latency of the floating point multiplier.

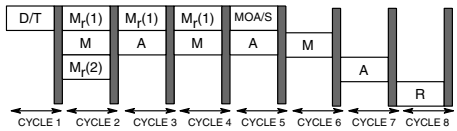


Diagram keys :
D : Denormalization .
T : Table lookup.
S : 56 bit carry-propagate addition.
MOA : Multiple operand addition.
M : Recoding, partial products generation and reduction (for the FULL MULTIPLIER).
$M_r(1)$: Recoding, partial products generation and reduction (for the 1st 14x14 bit multiplier).
$M_r(2)$: Recoding, partial products generation and reduction (for the 2nd 14x14 bit multiplier).
A : Carry-propagate addition (for the FULL MULTIPLIER).
R: Rounding, normalization and exception handling.

Figure 13. Time diagram for the implementation of the algorithm proposed in [17].

The comparison shows the potential advantages of our implementation. The architecture proposed in [7] presents the same latency as our design, but the additional hardware complexity is roughly six times higher. This reveals the fact that, in this case, the use of rectangular multipliers, when a pipelined floating point multiplier is already available, is not efficient,[19] since this significantly increases the additional hardware complexity without improving the latency.

As compared with the architecture that implements a similar algorithm to [15], there is a speedup of 1.8, although our design requires five times more additional hardware complexity. However for the current levels of integration, this increase in hardware is reasonable (the algorithm proposed in [15] represents a very-low hardware complexity solution). This result reveals that with a small amount of additional hardware the use of the pipelined multiplier can be more efficient (more

Table 6. Additional hardware for the algorithm proposed in [17].

Module	Number of elements	Equivalent gates
2 Table look-up 2 of $2^7 \times 56$	14336 bits	2600
14×14 bit multiplier	Carry-save to radix-4 recoder	150
	Partial products generator	800
	Reduction tree	750
14×14 bit multiplier	Binary to radix-4 recoder	60
	Partial products generator	800
	Reduction tree	750
MOA (multiple operand addition)	28 3:2 counters	250
	56 4:2 counters	1000
Carry propagate adder	56 bit CLA	790
Multiplexers	42 mux 2:1	125
Registers	2×56-bit reg	500
	6×28-bit reg	670
Total		9200

Table 7. Additional hardware for the proposed architecture.

Module	Hardware complexity	Equivalent gates
Table look-up 7×64	8192 bits	1500
Generator of products	75 and gates	300
	12 3:2 counters	
	27 2:2 counters	
Predictor	21 inverters	1600
	26 2:2 counters	
	116 3:2 counters	
	45 bit carry propagate adder	
Registers	1×64-bit register	250
Total		3700

Table 8. Comparison of additional hardware and latency.

Architecture	Latency (cycles)	Latency ratio	Additional hardware	Hardware ratio
Ref. [10]	30	3.3	36000	10.4
Ref. [15]	16	1.8	700	0.2
Ref. [7]	9	1.0	22000	6.4
Ref. [17]	8	0.9	9200	2.5
Our design	9	1.0	3700	1.0

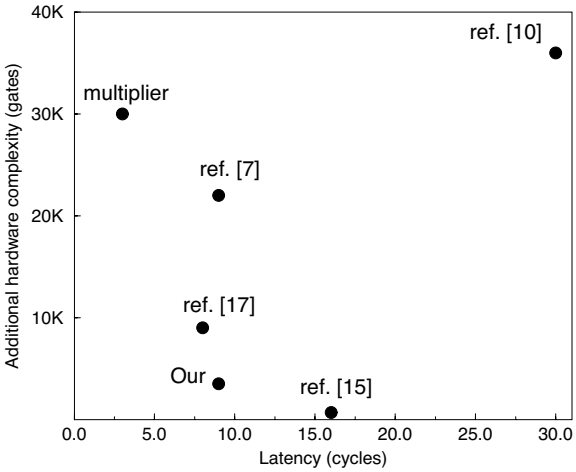


Figure 14. Latency vs. additional hardware complexity for the compared designs.

operations in parallel), reducing by a factor of 1.8 the latency.

Compared with [10], our design is roughly three times faster reducing the additional hardware by a factor of ten. Clearly a design with dedicated hardware does not result in an efficient alternative in this context.

Finally the design proposed in [17] has one cycle less of latency but at the cost of 2.5 times more additional hardware.

6. Conclusions

We presented an implementation of the exponential function in a floating-point unit. The proposed algorithm makes use of the existing floating-point multiplier and of additional hardware. The exponential calculation is decomposed into three subexponentials. The method used to compute the first and third subexponentials are table look-up and polynomial approximation respectively. For the second subexponential we improve the implementation of the slow radix-2 digit-recurrence algorithm by using the multiplier and additional hardware. The design process used to develop the algorithm allowed us to obtain a good trade-off between additional hardware complexity and latency. Therefore we designed the algorithm in a generic way until the algorithm parameters were determined, guided by a good trade-off goal. We presented a detailed implementation that assumes the availability of a floating-point multiplier and performed a rough comparison with existing proposals, that reveals potential advantages of our design (better area-latency trade-off). The

implementation can be easily extended to compute cosh and sinh, which are of interest for digital signal processing.

Notes

1. Software implementations also exist that obtain exactly rounded results [21].
2. To obtain the two's complement of X, it is necessary to complement each bit of X and add a one in the least significant place. To avoid a carry-propagate addition, we consider the addition of a one in the least significant place as a part of the computation of the exponential of the fractional part (see Section 4.2)
3. This may impose an upper limit on e_m to avoid a large table. If the whole range have to be covered in hardware, $e_m = 11$, which may require a large table. In this case, it is possible to decompose the table into two smaller tables (with half input bits) and multiply the result of the two tables. Alternatively, it is possible to perform the following transformation: $e^X = 2^{\frac{X}{\ln(2)}} = 2^{z+x} = 2^z e^{x \ln(2)}$. This solution does not need tables, but requires two additional multiplications.
4. This is a reasonable assumption for current microprocessor implementations.
5. We do not consider in this work algorithms based on pure table look-up methods (with addition) [22], since these algorithms are only practical up to floating-point single precision.
6. In a general case, x is approximated by the addition of the k terms, accurately enough to have the desired precision. Therefore, we use $\approx$ in expression (1).
7. The logarithm of the radix indicates roughly the number of bits of the result computed in each iteration.
8. The algorithm proposed in [15] actually computes $2^x - 1$. Here we adapt the algorithm to compute $y \exp(x)$.
9. Since we consider a datapath with 63 fractional bits, after the denormalization, x is truncated to 63 fractional bits.
10. Note that the conventional implementation determines one d_j value per iteration, which results in a slow scheme for our purposes.
11. We first assume that $m = \infty$, and solve the design problem, by determining the design parameters and the sequence of the operations. Later we compute the actual error in the calculation of the exponential for $m = \infty$, and the remainder from the required accuracy is assigned to α. Then, from α, we compute a finite value for m.
12. Instances of the implementation of this multiplier in microprocessors are [23] and [24] among others ([25] and [26] implement a similar algorithm but with a different pipeline organization).
13. The shifting is due to the denormalization step of the significand, and the complement is performed when the input argument is negative in order to obtain a two's complement representation.
14. We consider these designs representative algorithms, with similar design goals to the design proposed in this work. We do not compare with [4] since from their work it is not possible to obtain implementation details. In any case, the algorithms seem to be similar to those proposed in [15], but with emphasis in achieving 0.6 ulp instead of 1 ulp of precision, for extended precision and with tight limitations in the datapath width. We do not compare with [5] since in this case the algorithm is similar to the one proposed in [15] but using two floating-point multiplier-accumulation units operating in parallel to evaluate the polynomial.
15. This is implemented with two set of tables for each subexponential.
16. This actually determines the size of the rectangular mac.
17. The algorithm proposed in [15] is specifically for computing $2^x - 1$. We adapt the algorithm to compute $y \exp(x)$ to compare with our proposal.
18. These multiplications are of 14×14 since the lower bits of $a[2]^2$ and $a[2]^3$ have no significance in the computation.
19. However the use of a rectangular multiplier reduces the use of the full multiplier for exponential calculation, reducing the structural hazard with the multiplication instruction.

References

1. T. Lynch, A. Ahmed, and M. Schulte, "The K5 Transcendental Functions," in *Proc. 12th IEEE Symposium on Computer Arithmetic*, 1995, pp. 163–170.
2. J.M. Muller, *Elementary Functions: Algorithms and Implementations*, Birkhauser, 1997.
3. D.B. Papworth, "Tunning the Pentium Pro Microarchitecture," *IEEE Micro*, vol. 16, 1996, pp. 8–15.
4. S. Story and P.T.P. Tang, "New Algorithms for Improved Transcendental Functions on IA-64," in *Proc. 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 4–11.
5. J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang, "The Computation of Transcendental Functions on the IA-64 Architecture," *Intel Technology Journal*, Q4, 1999.
6. M. Schulte and E. Swartzlander, "Exact Rounding of Certain Elementary Functions," in *Proc. 11th IEEE Symposium on Computer Arithmetic*, 1993, pp. 138–145.
7. W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, 1994.
8. J.C. Bajard, S. Kla, and J.M. Muller, "BKM: A New Hardware Algorithm for Complex Elementary Functions," *IEEE Transactions on Computers*, vol. 4, 1994, pp. 955–96.
9. R.C. Chen, "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots," *IBM J. Res. Dev.*, vol. 16, 1972, pp. 380–388.
10. V. Kantabutra, "On Hardware for Computing Exponential and Trigonometric Functions," *IEEE Transactions on Computers*, vol. 45, no. 3, 1996, pp. 328–339.
11. W.H. Specker, "A Class of Algorithms for $\ln(x)$, $\exp(x)$, $\sin(x)$, $\cos(x)$, $\tan^{-1}(x)$ and $\cot^{-1}(x)$," *IEEE Trans. Electronic Computers*, vol. 14, 1965, pp. 85–86.
12. M.D. Ercegovac, "Radix-16 Evaluation of Certain Elementary Functions," *IEEE Trans. on Comput.*, vol. C-22, 1973, pp. 561–566.
13. P.W. Baker, "Parallel Multiplicative Algorithms for Some Elementary Functions," *IEEE Trans. on Comput.*, vol. 3, 1975, pp. 322–325.
14. E. Antelo, J.D. Bruguera, T. Lang, J. Villalba, and E.L. Zapata, "High-Radix CORDIC Rotation Based on Selection by Rounding," in *Lecture Notes in Computer Science* (EUROPAR-96: Parallel Processing, Worshop: Parallel Image/Video Processing and Computer Arithmetic), vol. 1124, 1996, pp. 155–164.

15. P.T. Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," in *Proc. 10th IEEE Symposium on Computer Arithmetic*, 1991, pp. 232–236.
16. V.K. Jain and L. Lin, "High-Speed Double Precision Computation of Nonlinear Functions," in *Proc. 12th IEEE Symposium on Computer Arithmetic*, 1995, pp. 107–114.
17. M.D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers," *IEEE Transactions on Computers*, vol. 49, no. 7, 2000, pp. 628–637.
18. I. Koren and O. Zinaty, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Transactions on Computers*, vol. 39, no. 8, 1990, pp. 1030–1037.
19. A. Vázquez and E. Antelo, "Implementation of the Exponential Function in a Floating-Point Unit," Internal Report. Dept. Electrónica y Computación, University of Santiago de Compostela, Spain, October 2000.
20. P.W. Baker, "Suggestion for a Fast Binary Sine/Cosine Generator," *IEEE Transactions on Computers*, vol. 25, 1976, pp. 1134–1136.
21. A. Ziv, "Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit," *ACM Transactions on Mathematical Software*, vol. 17, 1991, pp. 410–423.
22. M.J. Schulte and J.E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. on Computers*, vol. 48, no. 8, 1999, pp. 842–847.
23. G. Gerwig and M. Kroener, "Floating-Point Unit in Standard Cell Design with 116 Bit Wide Dataflow," in *Proc. 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 266–273.
24. R.K. Yu and G.B. Zyner, "167 MHz Radix-4 Floating-Point Multiplier," in *Proc. 12th IEEE Symposium on Computer Arithmetic*, 1995, pp. 149–154.
25. T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-bit Performance," *IEEE Micro*, vol. 19, 1999, pp. 73–85.
26. A. Naini, A. Dhablania, W. James, and D. Das Sarma, "1-GHz HAL SPARC64 Dual Floating-Point Unit with RAS Features," in *Proc. 15th IEEE Symposium on Computer Arithmetic*, 2001, pp. 173–183.

Álvaro Vázquez received the B.S. degree in physics in 1997 and the M.S. degree in 1999 both from the University of Santiago de Compostela, Spain. Currently, he is a Ph.D. candidate in the Department of Electronics and Computer Science at the University of Santiago de Compostela. His research interests include computer arithmetic, Intellectual Property design for SoC and 3D computer graphics.

Elisardo Antelo received the B.S. degree in 1991 and the Ph.D. in 1995, both in Physics, from the Universidade de Santiago de Compostela, Spain. In 1992, he joined the Departamento de Electrónica e Computación of the Universidade de Santiago de Compostela. From 1992 to March 1996 he was assistant professor and, since March 1996, he has been an associate professor in this department. His research interests are in computer arithmetic, computer architecture, 3D graphics and SoC design.
elisardo@dec.usc.es