

A correctly rounded implementationof the exponential function on the Intel Itanium architecture

Christoph Quirin Lauter

▶ **To cite this version:**

Christoph Quirin Lauter. A correctly rounded implementationof the exponential function on the Intel Itanium architecture. [Research Report] RR-5024, LIP RR 2003-54, INRIA, LIP. 2003. inria-00071560

HAL Id: inria-00071560
https://inria.hal.science/inria-00071560

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A correctly rounded implementation of the exponential function on the Intel Itanium architecture

Christoph Quirin Lauter ,

Technische Universität München / École Normale Supérieure de Lyon

N° 5024

Decembre 2003

THÈME 2



ISSN 0249-6399 ISRN INRIA/RR--5024--FR+ENG



A correctly rounded implementation of the exponential function on the Intel Itanium architecture

Christoph Quirin Lauter ,
Technische Universität München / École Normale Supérieure de Lyon

Thème 2 — Génie logiciel
et calcul symbolique
Projet Arénaire

Rapport de recherche n° 5024 — Decembre 2003 — 61 pages

Abstract: This article presents an efficient implementation of a correctly rounded exponential function in double precision on the Intel Itanium processor family. This work combines advanced processor features (like the double-extended precision fused multiply-and-add units of the Itanium processors) with recent research results giving the worst-case precision needed for correctly rounding the exponential function. We give and prove an algorithm which returns a correctly rounded result (in any of the four IEEE-754 rounding modes) within 172 machine cycles on the Intel Itanium 2 processor. This is about four times slower than the less accurate function present in the standard Intel mathematical library. The evaluation is performed in one phase only and is therefore fast even in the worst case, contrary to other implementations which use a multilevel strategy [18, 6]: We show that the worst-case required precision of 157 bits can always be stored in the sum of two double-extended floating-point numbers. Another algorithm is given with a 92 cycles execution time, but its proof has to be formally completed.

Key-words: Correct rounding, IEEE-754, exponential, Itanium

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Une implémentation de l'exponentielle avec arrondi correct sur architecture Intel Itanium

Résumé : Cet article présente une implémentation efficace de la fonction exponentielle en double précision avec arrondi correct sur la famille de processeurs Intel Itanium. Ce travail combine les caractéristiques avancées de ces processeurs (en particulier la présence d'unités de multiplication-accumulation en double précision étendue) avec les résultats de travaux récents sur les pires cas d'arrondi correct pour l'exponentielle. Nous décrivons et prouvons un algorithme qui calcule le résultat arrondi correctement (dans l'un quelconque des quatre modes d'arrondis spécifiés par la norme IEEE-754) en 172 cycles sur l'Itanium 2 d'Intel, soit environ quatre fois plus lentement que la fonction exponentielle standard, moins précise. L'évaluation est réalisée en une seule étape, et est donc rapide également au pire cas, au contraire d'autres implémentations qui utilisent une stratégie multi-niveaux : nous montrons que les résultats intérmédiaires (qui ont besoin d'une précision jusqu'à 157 bits) peuvent toujours être représentés comme la somme de deux nombres flottants en double précision étendue. Nous présentons également un algorithme amélioré qui réalise le calcul en 92 cycles, mais sa preuve reste à formaliser complètement.

Mots-clés : Arrondi correct, IEEE-754, exponentielle, Itanium

A correctly rounded implementation of the exponential function on the Intel Itanium architecture

Christoph Quirin Lauter

1 Introduction

When the **IEEE 754 standard** was finally adopted in 1985 [5] it represented a breakthough towards the standardisation of floating point arithmetics. Before, the behaviour of a programm depended on the platform it was executed on.[6] The standard defines two main floating point formats called single and double precision and defines the exact semantics of the four standard operations which are addition, substraction, multiplication and division. All this operations can be considered to produce correctly rounded results, i.e. if the operation were carried out in infinite precision and if this intermediate result were rounded using one of the four rounding modes defined by the standard[1] the obtained result would be same as that one produced by the underlying finite precision hardware. In addition to the four general operations, the square root function and the remainder operation have been normalized as to be implemented with a correct rounding [5, 6].

Unfortunately, this standard has some major defaults. The first one is a historical one. When the standard has been set up, processors integrated floating point units aiming other goals than nowadays' processors which include new features. In fact, modern processors support much wider formats than the double precision format and some supplementary 'basic' ternary operations with only one rounding; e.g. the Intel Itanium Processor has a 82 bit wide floating point register format in addition to the 64 bit IEEE double precision format and it implements the so called fused-multiply-and-add instruction `fma` which calculates $a \cdot b + c$ as if it were calculated in infinite intermediate precision and than rounded to the operational precision, i.e. 82 bit.[11] A programmer using the IEEE standard as a reference would actually expect in this case an intermediate rounding of the multiplicative term of the expression $a \cdot b$ before the addition to the term c. Despite this, as we will see, the described behaviour of this instruction is very useful for some techniques allowing us to 'emulate' a very large floating point format by means of two working precision numbers.

The second problem encountered when one is speaking about the IEEE standard is that it does not specify the behaviour of the standard transcendental functions so as the exponential

[1] The four IEEE rounding modes are: round to nearest (RTN), round towards plus infinity (RTPI), round towards minus infinity (RTMI) and round towards zero (RTZ), cf. [5].

function. As we will see, if there were a standard also for transcendental functions, or if they were included in the IEEE standard, some advantages could be observed.[5, 2] As Muller mentions in [2] some of the transcendental functions are even implemented very so bad in common runtime libraries that they produce completely wrong results on some arguments. Since the common application programmer using a library is not supposed to think about the numerical stability of the transcendental functions implemented there this could potentially lead to buggy behaviour of some applications. In addition, the lack of an exact definition of the results to be returned by standard libraries prohibits portability of applications over serveral platfoms and makes dressing up correctness proofs for these applications completely impossible. In the domains where guarantees on the correct working of a programm must be given, e.g. medical applications or avionics, this can be a major issue. In contrast, it should be mentionned that some mathematical properties are incompatible with correct rounding so it can also be a problem.[6]

But one question remains to be answered: why does the IEEE 754 standard remain silent on transcendental functions? As [2, 7, 6] point out in their respective works, the main issue in the calculation of elementary functions with a correct rounding is a problem which was already known before the invention of computers and which is therefore called the **table maker's dilemma**: we can guarantee the correct rounding of an approximation to a transcendental function only if we can detect in the finite precision intermediate result we use a digit pattern that guarantees that the rounding will always be the same regardless of the following digits of the infinite development of the transcendental function. For example, if we calculate a 64 bit approximation to some function on some argument that we want to round in round-to-nearest mode to a 53 bit double precision number and bits 54 to 64 are of the form 01111111111 we cannot guarantee that rounding a more precise, i.e. a fortiori infinite precise result would not lead to a different final result: what would be if we had made an approximation error so that the actual sequence would be 10000000000? The final result would have to be rounded upwards at the 53rd digit and not downwards as one may suspect at the first sight. To conclude[2], the intermediate precision needed is much higher than the final result precision one wants to obtain. In addition, since is it can not be analytically calculated, the actual precision needed was not known at the time the standard was defined. This were in fact the main reasons for not including elementary functions in the IEEE 754 standard. Despite, some new results on research on this issue and the evaluation of processors since the creation of this standard seem to provide a sufficient base for a correct rounding implementation of elementary functions which can bear comparison with less accurate ones. This article aims also this target.

Before considering more closely our techniques to achieve this goal, let us brievely mention the different steps in the techniques recommanded for a correct implementation of the exponential function.

As one can read in [2, 18], Ziv uses a multilevel evaluation of the exponential function calculating more and more significative digits of an approximation until the rounding can be done without any error. This technique has two main defaults: as far as we know, the

[2] For a more complete description of the problem the reader could refer to [2].

algorithm has been proven to be correct only in a general way, the implementation itself has not, and further, which is the major issue with it, there are no bounds on the timings of the algorithm as it uses a loop running until the rounding can be done. This can be a problem in applications where the timing is critical or where ressources are restricted.

In order to improve this multilevel technique, Lefèvre [7] has done research on the problem and provides us some bounds on the maximal intermediate precision needed for a final double precision result. These results are exploited by Defour [6] to lead to a two level evaluation algorithm which is the first implementation of a correctly rounded exponential with bounded timings. Unfortunately, the algorithm presents still some defaults which could prevent its application on a everyday-everywhere-basis. Its maximum timings are still more than 20 times higher than its average timings which are also not very attractive in comparison to less accurate industrial implementations which facilitate their use in applications with real-time behaviour. In this context, 'less accurate' means that these functions return an approximation within typical e.g. 0.502ulp for the exponential function but that they do not guarantee the correct rounding of the result. [11]

By means of using some supplementary hardware features which are not exploited in Defour's algorithm for a reason of portability, we can now present a one phase correctly rounding algorithm for the exponential function which is completely self-contained, relatively small in size and, having a latency of 172 cycles on Intel Itanium 2, presents quite good performance on nowadays' processors.

Before continuing presenting the algorithm and the techniques used to guarantee that it calculates a correctly rounded approximation to the exponential function, let us give a short overview of the next sections. We will first give a general description of the algorithm and will then proof an accuracy bound for the main domain of the input values. This proof is relativly complex but contains only a few new ideas compared to common accuracy proofs. If the technical points of the main proof of the algorithm are not in the reader's interest, he or she can skip it at first reading this article. Let us anyway point out that the proof of the accuracy of the range reduction (lemma 2.8, page 12) differs significantly from the point of views of other authors (cf. [6]). In a second step (beginning on page 36), we will prove the correctness of the rounding in the cases where the table maker's dilemma can be observed with a critical precision of up to 157 bits. This part of the article is particularly important since the accuracy proof can only be done by means of a somewhat new technique developed. The final correctness theorem is theorem 2.24, page 39, it uses theorems 2.22, page 36, and 2.23, page 36. To conclude, we will give an overview of the testing and timing results in comparision to the algorithms presented by others.

2 The algorithm

The algorithm for the correctly rounded `exp` function has been written in assembler Itanium but other platforms could also be used if they support some features of modern floating point units such as `fma` , extended precision and perhaps, for performance reasons, also several different floating point status (control) registers.

Generally speaking the algorithm is a table driven one which is common use for an implentation of the exponential function [1, 3, 4]. A 6 degree polynomial is used for the approximation of the reduced argument, almost all calculations are carried out in an internal representation of intermediate results as a non-evaluated sum of two double extended floating point registers.

Similarly to less accurate implementations, one has to consider very precisely all floating point calculations while trying to optimize for performance; in contrast the length of a formal proof of the accuracy is more than twice the one needed for simpler implementations. For the final rounding, the algorithm relies on the correctness of the underlying floating point hardware only and does not check if the rounding can actually be done without commiting an error as others like [6] do.

2.1 General overview of the algorithm

The general idea of the algorithm can be seen in the following equation — the argument x is decomposed by range reduction into several parts which are used for indexing (by i_1 and i_2) two tables of precalculated values and for calculating an approximation to the exponential function on a small domain by evaluation of a polynomial:

$$e^x = 2^M \cdot 2^{\frac{i_1}{2^7}} \cdot 2^{\frac{i_2}{2^{14}}} \cdot (1 + p\,(r)) \cdot (1 + x^l) + \Delta$$

In this equation, the values M, i_1, i_2, r and x^l constitute the result of the range reduction done on x as follows[3]:

$$\begin{aligned} k &= \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil \\ x &= k \cdot \frac{\ln 2}{2^{14}} + r + x^l + \tau \\ k &= 2^{14} \cdot M + 2^7 \cdot i_1 + i_2 \end{aligned}$$

where $i_1, i_2 \in \mathbb{N} \cap \left[0; 2^7 - 1\right]$ and $M \in \mathbb{N}$.

The variables Δ and τ stand for the errors induced by the floating point arithmetic and the mathematical error of a polynomial approximation.

This general outline of the algorithm is an adapted version of other algorithms such as the algorithm for single precision presented in [1]. The choice of a range reduction using a scale constant of $\frac{\ln 2}{2^{14}}$ is the result of a relatively complex study on the following four points that infere on this issue: the size of the two tables which can be optimized when an even scale constant exponent, 14, is used, the degree of the polynomial which depends on the size of the maximum magnitude of the final reduced argument r, the degree of the lower

[3] Throughout this article we will make a light abuse of the gaussian floor value notation: we define

$$k = \lfloor t \rceil \Leftrightarrow k \in \mathbb{N} \text{ so that } \neg\exists k' \in \mathbb{N}. |k' - t| < |k - t|$$

part reduced argument x^l approximation and some storage issues encountered with the scale constant $\frac{\ln 2}{2^{14}}$.

The polynomial used for approximating the exponential function on the reduced argument r is a modified 6 degree minimax polynomial obtained by Remez approximation as described in [2]. The algorithm calculates it in the following form:

$$p(r) = r + \frac{1}{2}r^2 + r^3 \cdot \left(\left(a_4^h + a_4^l\right) + \left(a_5 + \left(a_6 + a_7 \cdot r\right) \cdot r\right) \cdot r\right)$$

As one will expect the values for a_4^h, a_4^l, a_5, a_6 and a_7 are near to respectively $\frac{1}{6}$, $\frac{1}{24}$, $\frac{1}{120}$ and $\frac{1}{720}$; the exact values, which differ slitly from the values calculated by the Remez algorithm since they must be represented in working precision, will be given below. The reason for not using Horner's polynomial evaluation form to the very end is the complete sequenciality of this algorithm. In order to win some time on a fully pipelined processor such as the Intel Itanium , we split up the evaluation in the way given. Finer approaches exist [4, 20] but they are merely useful if a higher precision presentation of the intermediate results is used as we do. The instruction sequence carrying out the calculation of this polynomial gives as a result two double extended precision numbers p^h and p^l whose sum represents a approximation to the value of the polynomial. This instruction sequence is the most time consuming phase of the algorithm and the most difficult in terms of the proof of its accuracy.

The two values $t_1^h + t_1^l = 2^{\frac{i_1}{2^7}} + \delta$ and $t_2^h + t_2^l = 2^{\frac{i_2}{2^{14}}} + \delta$ are read in two tables of 128 entries each, where they are stored as two double extended precision numbers. They are multiplied using another instruction sequence to obtain two double extended numbers $t^h + t^l = \left(t_1^h + t_1^h\right) \cdot \left(t_2^h + t_2^l\right) + \Delta$ which will be used in the final reconstruction step that calculates the product of $t_2^h + t_2^l$, $1 + \left(p^h + p^l\right)$ and $1 + x^l$ to lead to a value which must simply be multiplied by 2^M and rounded to get an approximation to the exponential which is (in the most cases) good enough to be correctly rounded to double precision.

The size of the table is $2 \cdot 128 \cdot 2 \cdot 16 = 8192$ bytes which is a relatively great value [6, 1] for an algorithm for the exponential function. This is in other words the prize to pay for correctly rounded results. On the other hand, the table could fit 12 times in the L2 cache[4] of the Itanium processor which can be accessed in only 6 cycles.[11] Furthermore, its size is less than the sizes of the tables used by other algorithms proposed by Defour [6] and Ziv.[18]

The instruction sequences given in the different parts of the following accuracy proof are joined by some few other instructions needed for indexing the table, loading the constants needed etc. We assume that the reader will be able to figure out their use in the complete algorithm listing (cf. 7) and to convince himself of the correctness of this instructions. In addition, the final algorithm to used on a Itanium system has been scheduled in order to get a maximum performance. As a result of scheduling, the different part such as they are presented here are somewhat 'interlaced' which does not improve the readability of the code unfortunately.

[4]The processor cannot load floating point registers from L1 cache. [11]

2.2 Proof of the accuracy of the algorithm

In order to prove in the end of this section, that our algorithm correctly calculates the exponential function `exp` in double precision for the main part of the input domain, we will first give some general results and assumptions, and than give a complete error estimate of each of the parts of the algorithm. Finally, an over all estimate will be done.

Throughout the following, two different, concurrent notations will be used. The first one comes close to Intel Itanium assembler code and is used as a reference in the proof. The second one, makes abstraction of assembly language by being more close to a mathematical notation. The main operations will be defined below but the following conventions will be used:

- Rounding a higher or infinite precision value $\hat{x}$ to a floating point format offering n bits of mantissa will be noted: $x \leftarrow \langle \hat{x} \rangle_n$. If the context permits it, the indication of the resulting precision will be left out.
- The arithmetical operators corresponding to the mathematical operations $+$, $-$ and $\cdot$ will be noted $\oplus$, $\ominus$ and $\otimes$.

Definition 2.1 *Let a and b be two double extended precision numbers. So the instruction noted*

`fadd.s1 c = a, b` $\qquad c \leftarrow a \oplus b$

will calculate a double extended precision number c so that

$$c = (a + b) \cdot (1 + \epsilon)$$

holds with $|\epsilon| \leq 2^{-64}$ as long as the result does not under- or overflow.

Definition 2.2 *Let a, b and c be three double extended precision numbers.*
So the instruction

`fma.s1 d = a, b, c` $\qquad d \leftarrow \langle a \cdot b + c \rangle$

will calculate a double extended number d so that

$$d = (a \cdot b + c) \cdot (1 + \epsilon)$$

holds with $|\epsilon| \leq 2^{-64}$ and that there is no intermediate rounding between the calculation of the product and the sum.

The same property can be observed for the `fms` instruction which calculates $a \cdot b - c$.

Lemma 2.3 *Let a and b be two double extended precision numbers.*

So the instruction sequence

```
fma.s1 c = a, b, f0
fms.s1 d = a, b, c
```

$c \leftarrow \langle a \cdot b + 0 \rangle$
$d \leftarrow \langle a \cdot b - c \rangle$

calculates two double extended numbers c and d so that the equations

$$c + d = a \cdot b$$

and

$$|d| \leq |c| \cdot 2^{-64}$$

hold and c is the double extended nearest to $a \cdot b$.

Proof We know that the first instruction of the sequence calculates c so that

$$c = a \cdot b \cdot (1 + \epsilon)$$

with $|\epsilon| \leq 2^{-64}$. Thus c is trivially the double extended number nearest to $a \cdot b$.

We know also that there is no immediate rounding in the calculation between the calculation of $a \cdot b$ and the substraction with c in the second instruction. It can be clearly seen that two double extended precision numbers multiplied in infinite precision, produce a result that has at most 128 significative bits. As the first 64 bits of the infinite precision product of a and b all cancellate with c, there are only 64 bits left to be "rounded" to the 64 bits of the mantissa of d. Hence the result follows. ∎

This instruction sequence will be called below the ***Fast2Mult*** - sequence and noted also as $c + d \leftarrow$ ***Fast2Mult*** (a, b).

Lemma 2.4 *Let a and b be two double extended numbers so that $|a| > |b|$. So the instruction sequence*

```
fadd.s1 c = a, b
fsub.s1 t = c, a
fsub.s1 d = b, t
```

$c \leftarrow a \oplus b$
$t \leftarrow c \ominus a$
$d \leftarrow b \ominus t$

calculates two double extended precision numbers c and d so that the equations

$$c + d = a + b$$

and

$$|d| \leq |c| \cdot 2^{-64}$$

hold and that c is the double extended precision number nearest to $a + b$.

Proof This property has been shown by Knuth, Møller and Dekker. [4, 12, 13, 21] ∎

This instruction sequence will be called below the ***Fast2Sum*** - sequence and noted also as $c + d \leftarrow$ ***Fast2Sum*** (a, b).

Corollary 2.5 *Let a and b be two double extended numbers. So the instruction sequence*

`fadd.s1 c = a, b`	$c \leftarrow a \oplus b$
`famin.s1 t_1 = a, b`	$t_1 \leftarrow x$ *so that* $x \in \{a, b\} \wedge$ $\neg\exists x' \in \{a, b\}. \lvert x'\rvert < \lvert x\rvert$
`famax.s1 t_2 = b, a`	$t_2 \leftarrow x$ *so that* $x \in \{b, a\} \wedge$ $\neg\exists x' \in \{b, a\}. \lvert x'\rvert > \lvert x\rvert$
`fsub.s1 t_3 = t_2, c`	$t_3 \leftarrow t_2 \ominus c$
`fadd.s1 d = t_3, t_1`	$d \leftarrow t_3 \oplus t_1$

calculates two double extended precision numbers c and d so that the equations

$$c + d = a + b$$

and

$$|d| < |c| \cdot 2^{-64}$$

hold and that c is the double extended precision number nearest to $a + b$.

Proof As one can read in [4], this corollary is a trivial result of the previous Lemma 2.4 and a code sequence determining the minimum and maximum of the absolute value of two double extended numbers by means of the `famin` and `famax` instruction. [4] precises that the order of the arguments of the `famin` must be opposite to that one in which they are given to the `famax` instruction, this subtility has been respected. ∎

Remark: The preceeding instruction sequence is relatively long. Though having the same latency than the uncondition version of the sequence, it should be avoided as it makes a higher use of the processor's ressources thus perventing a easy scheduling of the englobing code. It can be eluded by analysing at compile time the magnitudes of the arguments a and b. A positive side of the sequence is that the high part of the result, c, is available after 4-5 cycles, thus the latency of the second result can usually be "hidden" in the latency of other calculations.
We will call this algorithm the ***conditional Fast2Sum*** sequence and note it also $c + d \leftarrow$ ***CondFast2Sum*** (a, b).

Theorem 2.6 *Let*

$$\begin{aligned} a_4^h &= \frac{12297829382473034411}{73786976294838206464} \\ a_4^l &= \frac{-3098113767111307095}{6805647338418769269267492148635364229 12} \\ a_5 &= \frac{12297829382473034411}{295147905179352825856} \end{aligned}$$

$$\begin{aligned} a_6 &= \frac{4919131753080926207}{590295810358705651712} \\ a_7 &= \frac{13117684675687318145}{9444732965739290427392} \end{aligned}$$

and let $x \in \left[-\frac{\ln 2}{2^{15}}; \frac{\ln 2}{2^{15}}\right]$.
The polynomial

$$p(x) = x + \frac{1}{2}x^2 + x^3 \cdot \left(\left(a_4^h + a_4^l\right) + \left(a_5 + \left(a_6 + a_7 \cdot x\right) \cdot x\right) \cdot x\right)$$

approximates $e^x - 1$ *so that the inegality*

$$|e^x - 1 - p(x)| < 2^{-123}$$

holds and the values a_4^h, a_4^l, a_5, a_6, a_7 *are all exactly representable in double extended precision.*
In addition, for $x \in \left[-2^{-30}; 2^{-30}\right]$ *the inegality holds with*

$$|e^x - 1 - p(x)| < 2^{-161}$$

Proof This property can simply be proven by locating numerically the local maximum values of $|e^x - 1 - p(x)|$ in the given intervals with a tool such as Maple.
The fact that the polynomial coefficients are all representable in double extended precision can trivially be verified by converting them to this format and calculating the rounding error, which must be zero.

Lemma 2.7 *Let*

$$l^h = \frac{3196577161300663915}{75557863725914323419136}$$

and

$$l^l = \frac{-15596301547560248643}{22300745198530623141535718272648361505980416}$$

So the following inequation holds

$$\left|\frac{\ln 2}{2^{14}} - \left(l^h + l^l\right)\right| < 2^{-150}$$

l^h *and* l^l *are exactly representable in double extended precision and* l^h *is the double extended number nearest to* $\frac{\ln 2}{2^{14}}$.

Proof Trivial, one has only to calculate the given difference and to test if the values are really representable in double extended precision. ∎

Lemma 2.8 *Let x be a double precision floating point number in the range $x \in [-745.13; 709.78]$*[5] *normalized to register format,*
let `inv_ln2` *be the double extended precision number nearest to $\frac{1}{\ln 2}$,*
let l^h be the double extended precision number nearest to $\frac{\ln 2}{2^{14}}$,
let l^l be the double extended number so that $l^h + l^l = \frac{\ln 2}{2^{14}} + \delta$ with $|\delta| < 2^{-150}$ and $|l^l| < |l^h| \cdot 2^{-64}$,
let `fRSH_2_49` *be the double extended number representing $1.5 \cdot 2^{112}$,*
let `fRSHF` *be the double extended number for $1.5 \cdot 2^{63}$ and*
let `f2_M_49` *be the double extended number 2^{-49}.*
Then, the following instruction sequence calculates two double extended numbers r and x^l so that

$$x = \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil \cdot \frac{\ln 2}{2^{14}} + r + x^l + \Delta$$

holds with

$$|\Delta| < 2^{-118}$$

or with

$$e^{r+x^l} = e^{\hat{r}} \cdot (1 + \xi)$$

where $|\xi| < 2^{-118}$ and $\hat{r} = x - \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil \cdot \frac{\ln 2}{2^{14}}$.
In both cases we have $|r| < \frac{ln2}{2^{15}}$ and $|x^l| < |r| \cdot 2^{-64}$.
In addition, for $x \in \left[-2^{-30}; 2^{-30}\right]$, the instruction sequence calculates r and x^l so that the following equalities hold exactly:

$$r = x$$

$$x^l = 0$$

and that $k = 0$.
Instruction sequence:

`fma.s1 q_1 = x, inv_ln2, fRSHF_2_49`	$q_1 \leftarrow \langle x \cdot inv_ln2 + fRSHF_2_49 \rangle$
`fms.s1 k = q_1, f2_M_49, fRSHF`	$k \leftarrow \langle q_1 \cdot f2_M_49 - fRSHF \rangle$
`fnma.s1 r_l = k, fl_l, f0`	$r^l \leftarrow \langle -k \cdot l^l + 0 \rangle$
`fnma.s1 r_h = k, fl_h, x`	$r^h \leftarrow \langle -k \cdot l^h + x \rangle$
`fadd.s1 r = r_h, r_l`	
`famin.s1 q_5_0 = r_h, r_l`	
`famax.s1 q_5_1 = r_l, r_h`	
`fsub.s1 q_5_2 = q_5_1, r`	
`fadd.s1 x_l = q_5_2, q_5_0`	$r + x^l \leftarrow$ `CondFast2Sum`$\left(r^h, r^l\right)$

[5] i.e. the range in which e^x is representable in double precision or is not simply zero (resp. $0 + 1$ulp for some rounding modes).

Proof
The last 5 instructions of the given sequence are the ***conditional Fast2Sum*** sequence which is correct on all arguments. This sequence ensures that $|x^l| \leq |r| \cdot 2^{-64}$ and thus $|x^l| < 2^{-79}$ if $|r| < 2^{-15}$ which can easily be verified. So it suffices to prove that

$$x = \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor \cdot \frac{\ln 2}{2^{14}} + r^h + r^l + \Delta$$

with $|\Delta| < 2^{-118}$ and that $|r^h| < |r^l|$.
Let us first show that the first 2 steps of the instruction sequence correctly calculate $k = \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor$. Let us note the rounded result $\langle z \rangle$ of the mathematical value z. So we have

$$q_1 = \left\langle x \cdot \frac{1}{\ln 2} + 1.5 \cdot 2^{112} \right\rangle$$

and we know that $\mathtt{1ulp}(q_1) = 2^{112-63} = 2^{49}$ thus the lower part of the double extended number (when we not consider the 2 leading ones of the $1.5 \cdot 2^{112}$ part) contains $\left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor$ scaled by 2^{49}. In fact, in the next instruction, the value q_1 is rescaled by 2^{-49} and the trailing ones are removed by substraction of $1.5 \cdot 2^{63}$. So k contains the integer value nearest to $x \cdot \frac{2^{14}}{\ln 2}$.
Let us show now, that $r^h + r^l = \left(x - k \cdot \frac{\ln 2}{2^{14}}\right) + \Delta$ with $|\Delta| < 2^{-118}$. One can simply verify that due to the restricted range of the value x, we can assume that $k \in \left[-2^{25}; 2^{25}\right]$. Further, because $\left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor \cdot \frac{\ln 2}{2^{14}}$ is a very good approximation to x (cf. [2]) and because the intermediate calculations during a `fma` instruction are carried out in infinite precision, we can assume[6] that the result of the third instruction is exactly $r^h = -k \cdot l^h + x$.
Additionally, the use of the `fma` instruction dispenses us from storing l^h with some trailing zeros as [1, 14, 15, 6] do and we can further assume that the significant bits debording from a 64 bit representation which are a result form the intermediate multiplication of l^h by k cannot lead to a complete cancellation and are not lost to the final result. This behaviour leads to a significative simplification of the range reduction sequence in comparision to e.g. [6]. Therefore, as [2] notices, too, it is not necessary to use methods such as the Payne and Hanek algorithm for the range reduction of the exponential function. For instance, even if Kahan's algorithm calculating the maximal number of cancelled bits [2] gives us the number of 72 cancelled bits, this cancellation is not a problem. Looking at the reduction formula, which we consider here as a mathematically exact proposition,

$$r = x - \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor \cdot \frac{\ln 2}{2^{14}}$$

one sees that the cancellation on the arithmetical computation of an approximation to r is the greater the term $\left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor \cdot \frac{\ln 2}{2^{14}}$ is nearer to x and thus the cancellation is important only

[6] For a pedantic proof we could use a similiar argumentation as in [8] which is based on a lemma proven by Sterbenz in [16].

if the real reducted argument r is very small. In other words we have cancellation only iff the functional argument x is very near to a table entry value. Since we have $e^r = \sum_{i=0}^{\infty} \frac{1}{i!} r^i = 1 + r + \frac{1}{2}r^2 + \ldots$, the impact of the reduced argument r, $|r| < 1$, on the final result for e^x will be very small (the leading 1 is much greater in magnitude than r or even r^2) and even more, the impact of the error made due to cancellation on the final result of the exponential function after the multiplication by the table values will be so small that we don't have to care too much about it. What matters is the absolute error of the arithmetically calculated reduced argument $r^h + r^l$ in comparison to the exact real value $x - \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rfloor \cdot \frac{\ln 2}{2^{14}}$, which we are going to give the bound of 2^{-118} for.
Let us consider this issue in a more formal way by dealing not with the absolute but with the relative error impact of the cancellation on the final result. If we note $\hat{r}$ the mathematically exact value of the reduced argument, we observe

$$r^h + r^l = \hat{r} \cdot (1 + \epsilon)$$

where ϵ is the relative error of the reduced argument. Using this, we can approximate formally the impact of this error on the relative error ξ of the final result:

$$e^{r^h + r^l} = e^{\hat{r} \cdot (1+\epsilon)} = e^{\hat{r}} \cdot e^{\hat{r}\epsilon} = e^{\hat{r}} \cdot \left(1 + \hat{r}\epsilon + \frac{1}{2}\hat{r}^2\epsilon^2 + \ldots\right)$$

Noting

$$\xi = \hat{r}\epsilon + \frac{1}{2}\hat{r}^2\epsilon^2 + \ldots = \hat{r}\epsilon\left(1 + \frac{1}{2}\hat{r}\epsilon + \frac{1}{6}\hat{r}^2\epsilon^2 \ldots\right)$$

we have

$$e^{r^h + r^l} = e^{\hat{r}} \cdot (1 + \xi)$$

and we are able to give the following estimate:

$$|\xi| = |\hat{r}| \cdot |\epsilon| \cdot \left|1 + \frac{1}{2}\hat{r}\epsilon + \frac{1}{6}\hat{r}^2\epsilon^2 \ldots\right| < |\hat{r}| \cdot |\epsilon| \cdot 2$$

If we have j bits that are lost by cancellation then we still have $(128 - j)$ correct bits of our potentially 128 bit precise result, so in this case

$$|\epsilon| < 2^{-128+j}$$

So if we want to have 118 correct bits in the final result, we must not loose more than j by cancellation for all $\hat{r}$ in the following range

$$|\hat{r}| > 2^{9-j}$$

because if this condition is respected we obtain the desired result

$$|\xi| < |\hat{r}| \cdot |\epsilon| \cdot 2 < 2^{9-j} \cdot 2^{-128+j} \cdot 2 = 2^{-118}$$

Let assume now that

$$|\hat{r}| = \left| x - \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil \cdot \frac{\ln 2}{2^{14}} \right| > 2^{9-j}$$

and that we loose more than j bits by cancellation. Since $r^h + r^l = \langle \hat{r} \rangle$ and $|\hat{r}| < 2^{-15}$ we know that the most significant bit in r^h has a maximal weight of 2^{-15}. So if j bits are cancelled, the final result $r^h + r^l$ is in magnitude

$$|r^h + r^l| < 2^{-15-j}$$

but as we have assumed that $|\hat{r}| > 2^{9-j}$, we have a contradiction. We can therefore assume that the error due to cancellation in the range reduction phase does not affect the bits higher that bit 118.
So we have to consider only the error due to the rounding of l^h and l^l, which we will do now:

$$\begin{aligned} r^h + r^l &= -k \cdot l^h + x - k \cdot l^l + k \cdot l^l \cdot \epsilon \\ &= x - k \cdot \left(l^h + l^l\right) + k \cdot l^l \cdot \epsilon \\ &= x - k \cdot \left(\frac{\ln 2}{2^{14}} + \delta\right) + k \cdot l^l \cdot \epsilon \\ &= x - k \cdot \frac{\ln 2}{2^{14}} + k \cdot \delta + k \cdot l^l \cdot \epsilon \\ &= x - \left\lfloor x \cdot \frac{2^{14}}{\ln 2} \right\rceil \cdot \frac{\ln 2}{2^{14}} + k \cdot \delta + k \cdot l^l \cdot \epsilon \end{aligned}$$

where $|\epsilon| < 2^{-64}$ and $|\delta| < 2^{-128}$.
Further we know that $|l^l| < |l^h| \cdot 2^{-64} \approx \frac{\ln 2}{2^{14}} \cdot 2^{-64} < 2^{-80}$. Thus

$$\begin{aligned} |\Delta| &< |k \cdot \delta + k \cdot l^l \cdot \epsilon| \\ &< |k \cdot \delta| + |k \cdot l^l \cdot \epsilon| \\ &< 2^{25} \cdot 2^{-150} + 2^{25} \cdot 2^{-80} \cdot 2^{-64} < 2^{-118} \end{aligned}$$

For a complete proof of the lemma, we still have to show that for $x \in \left[-2^{-30}; 2^{-30}\right]$ the value $r = x$ and that $x^l = 0$. This can be shown relatively trivially: As $|x| < 2^{-30}$ we have that

$$\left| x \cdot \frac{2^{14}}{\ln 2} \right| < \frac{1}{2}$$

so k will be

$$k = 0$$

Thus the result leading to r^l and r^h will give

$$r^l = 0$$

and

$$r^h = x$$

As we have shown that the additional ***conditional Fast2Sum*** sequence is errorless, the property follows. ∎

Remark: Looking on the preceeding instruction sequence one could suggest that it would suffice to take r^h and r^l as the higher and the lower part of the reduced argument. This would save us the use of the ***conditional Fast2Sum*** sequence whose use should be avoided as discussed above. Unfortunately this approach cannot be done because of the following reason: we know that r^l might be as great as 2^{-56} in magnitude by[7]

$$|r^l| = |l^l| \cdot |k| > 2^{-81} \cdot 2^{25} = 2^{-56}$$

With such a great lower part reduced argument, the quadratic term of the Taylor development of e^{x^l} would have to be evaluated to obtain the desired precision. Using x^l as a lower part reduced argument with $|x^l| < 2^{-79}$ allows us to approximate e^{x^l} only by $1 + x^l$ as we actually do.
Additionally, we cannot avoid to use the conditional version of the ***Fast2Sum*** sequence. As a result of cancellation, r^h can be less in magnitude than r^l. Though this case seems to be extremely rare, even almost impossible to our opinion, there is no mathematical reason that it cannot eventually happen. Further, the use of the ***conditional Fast2Sum*** algorithm is not very costly. As stated already above, we can hide the latency of the supplementary tests completely in the latency of the polynomial evaluations.

Theorem 2.9 *Let*

$$\begin{aligned} a_4^h &\approx \frac{1}{6} \\ a_4^l &\approx a_4^h \cdot 2^{-64} \\ a_5 &\approx \frac{1}{24} \\ a_6 &\approx \frac{1}{120} \\ a_7 &\approx \frac{1}{720} \\ h &= \frac{1}{2} \end{aligned}$$

let x be a double extended precision number where $x \in \left[-\frac{\ln 2}{2^{15}}; \frac{\ln 2}{2^{15}}\right]$ and let $p(x)$ be the polynomial

$$p(x) = x + h \cdot x^2 + x^3 \cdot \left(\left(a_4^h + a_4^l\right) + \left(a_5 + \left(a_6 + a_7 \cdot x\right) \cdot x\right) \cdot x\right)$$

[7] To be completely mathematically exact, we would have to quantify the proposition by an existential quantifier - in contrast to all other propositions in this paper which are implictly universally quantified:

$$\exists x. |r^l| = |l^l| \cdot |k| > 2^{-81} \cdot 2^{25} = 2^{-56}$$

So the following instruction sequence calculates two double extended numbers p^h and p^l so, that

$$p^h + p^l = p(x) + \Delta$$

holds with $|\Delta| < 2^{-124}$, that p^h is the double extended number nearest to $p(x)$ and that $|p^l| < |p^h| \cdot 2^{-64}$.
In addition, for $x \in \left[-2^{-30}; 2^{-30}\right]$ the arithmetical error Δ of the sequence is less than

$$|\Delta| < 2^{-141}$$

Instruction sequence:

Step 1*:*

`fma.s1 z_1 = a_7, x, a_6`	$z_1 \leftarrow \langle a_7 \cdot x + a_6 \rangle$
`fma.s1 z_2 = z_1, x, a_5`	$z_2 \leftarrow \langle z_1 \cdot x + a_5 \rangle$
`fma.s1 z_3_h = z_2, x, f0`	
`fms.s1 z_3_l = z_2, x, z_3_h`	$z_3^h + z_3^l \leftarrow$ ***Fast2Mult***(z_2, x)

Step 2*:*

`fadd.s1 z_4 = a_4_h, z_3_h`	
`fsub.s1 z_5 = z_4, a_4_h`	
`fsub.s1 z_6 = z_3_h, z_5`	$z_4 + z_6 \leftarrow$ ***Fast2Sum***$\left(a_4^h, z_3^h\right)$
`fadd.s1 z_7 = z_3_l, z_6`	$z_7 \leftarrow z_3^l \oplus z_6$
`fadd.s1 z_8 = a_4_l, z_7`	$z_8 \leftarrow a_4^l \oplus z_7$
`fadd.s1 z_9_h = z_4, z_8`	
`fsub.s1 z_10 = z_9_h, z_4`	
`fsub.s1 z_9_l = z_8, z_10`	$z_9^h + z_9^l \leftarrow$ ***Fast2Sum***(z_4, z_8)

Step 3*:*

`fma.s1 z_11_h = x, x, f0`	
`fms.s1 z_11_l = x, x, z_11_h`	$z_{11}^h + z_{11}^l \leftarrow$ ***Fast2Mult***(x, x)

Step 4*:*

`fma.s1 z_12 = z_11_h, x, f0`	
`fms.s1 z_13 = z_11_h, x, z_12`	$z_{12} + z_{13} \leftarrow$ ***Fast2Mult***$\left(z_{11}^h, x\right)$
`fma.s1 z_14 = z_11_l, x, z_13`	$z_{14} \leftarrow \left\langle z_{11}^l \cdot x + z_{13} \right\rangle$
`fadd.s1 z_15_h = z_12, z_14`	
`fsub.s1 z_16 = z_15_h, z_12`	
`fsub.s1 z_15_l = z_14, z_16`	$z_{15}^h + z_{15}^l \leftarrow$ ***Fast2Sum***(z_{12}, z_{14})

Step 5*:*

`fma.s1 z_17 = z_9_h, z_15_h, f0`	

`fms.s1 z_18 = z_9_h, z_15_h, z_17`	$z_{17} + z_{18} \leftarrow \mathit{Fast2Mult}\left(z_9^h, z_{15}^h\right)$
`fma.s1 z_19 = z_9_l, z_15_h, z_18`	$z_{19} \leftarrow \left\langle z_9^l \cdot z_{15}^h + z_{18} \right\rangle$
`fma.s1 z_20 = z_9_h, z_15_l, z_19`	$z_{20} \leftarrow \left\langle z_9^h \cdot z_{15}^l + z_{19} \right\rangle$
`fadd.s1 z_21_h = z_17, z_20`	
`fsub.s1 z_22 = z_21_h, z_17`	
`fsub.s1 z_21_l = z_20, z_22`	$z_{21}^h + z_{21}^l \leftarrow \mathit{Fast2Sum}(z_{17}, z_{20})$

Step 6*:*

`fma.s1 z_23 = z_11_h, h, f0`	
`fms.s1 z_24 = z_11_h, h, z_23`	$z_{23} + z_{24} \leftarrow \mathit{Fast2Mult}\left(z_{11}^h, h\right)$
`fma.s1 z_25 = z_11_l, h, z_24`	$z_{25} \leftarrow \left\langle z_{11}^l \cdot h + z_{24} \right\rangle$
`fadd.s1 z_26_h = z_23, z_25`	
`fsub.s1 z_27 = z_26_h, z_23`	
`fsub.s1 z_26_l = z_25, z_27`	$z_{26}^h + z_{26}^l \leftarrow \mathit{Fast2Sum}(z_{23}, z_{25})$

Step 7*:*

`fadd.s1 z_28 = z_26_h, z_21_h`	
`fsub.s1 z_29 = z_28, z_26_h`	
`fsub.s1 z_30 = z_21_h, z_29`	$z_{28} + z_{30} \leftarrow \mathit{Fast2Sum}\left(z_{26}^h, z_{21}^h\right)$
`fadd.s1 z_31 = z_21_l, z_30`	$z_{31} \leftarrow z_{21}^l \oplus z_{30}$
`fadd.s1 z_32 = z_26_l, z_31`	$z_{32} \leftarrow z_{26}^l \oplus z_{31}$
`fadd.s1 z_33_h = z_28, z_32`	
`fsub.s1 z_43 = z_33_h, z_28`	
`fsub.s1 z_33_l = z_32, z_34`	$z_{33}^h + z_{33}^l \leftarrow \mathit{Fast2Sum}(z_{28}, z_{32})$

Step 8*:*

`fadd.s1 z_35 = z_33_h, x`	
`fsub.s1 z_36 = z_35, x`	
`fsub.s1 z_37 = z_33_h, z_36`	$z_{35} + z_{37} \leftarrow \mathit{Fast2Sum}\left(z_{33}^h, x\right)$
`fadd.s1 z_38 = z_33_l, z_37`	$z_{38} \leftarrow z_{33}^l \oplus z_{37}$
`fadd.s1 p_h = z_35, z_38`	
`fsub.s1 z_39 = p_h, z_35`	
`fsub.s1 p_l = z_38, z_39`	$p^h + p^l \leftarrow \mathit{Fast2Sum}(z_{35}, z_{38})$

In order to be able to give a final proof of this theorem, we will use several lemmata for the different steps. The proof of the theorem will then simply consist in a summing up of the weighted sum of the different errors accumulated in the different steps.

Lemma 2.10 *The instruction sequence called "step 1" in the algorithm above calculates z_3^h and z_3^l so that*

$$z_3^h + z_3^l = (a_5 + (a_6 + a_7 x)\, x)\, x + \Delta_1$$

holds with $|\Delta_1| < 2^{-82}$ and that z_3^h is the double extended number nearest to the exact result.

Proof The last 2 instructions of this step form the `Fast2Mult` algorithm which has been proven above to be errorless. So we have:

$$\begin{aligned} z_3^h + z_3^l &= x \cdot ((a_7 x + a_6)(1+\epsilon_1) + a_5) \cdot (1+\epsilon_2) \\ &= \dots \\ &= (a_5 + (a_6 + a_7 x)\, x)\, x + \Delta_1 \end{aligned}$$

where

$$\Delta_1 = a_7 x^3 \epsilon_1 + a_6 x^2 \epsilon_1 + a_5 x \epsilon_2 + a_7 x^3 \epsilon_2 + a_6 x^2 \epsilon_2 + a_7 x^3 \epsilon_1 \epsilon_2 + a_6 x^2 \epsilon_1 \epsilon_2$$

with $|\epsilon_1| < 2^{-64}$ and $|\epsilon_2| < 2^{-64}$. Since

$$|\Delta_1| = |a_7 x^3 \epsilon_1| + |a_6 x^2 \epsilon_1| + |a_5 x \epsilon_2| + |a_7 x^3 \epsilon_2| + |a_6 x^2 \epsilon_2| + |a_7 x^3 \epsilon_1 \epsilon_2| + |a_6 x^2 \epsilon_1 \epsilon_2|$$

and

$$\begin{gathered} |a_7 x^3 \epsilon_1| < \frac{1}{720} \cdot 2^{-46} \cdot 2^{-64} < 2^{-109} \\ |a_6 x^2 \epsilon_1| < 2^{-101} \\ |a_5 x \epsilon_2| < 2^{-83} \\ |a_7 x^3 \epsilon_2| < 2^{-109} \\ |a_6 x^2 \epsilon_2| < 2^{-101} \\ |a_7 x^3 \epsilon_1 \epsilon_2 < 2^{-160} \\ |a_6 x^2 \epsilon_1 \epsilon_2| < 2^{-150} \end{gathered}$$

we have

$$|\Delta_1| < 2^{-109} + 2^{-101} + 2^{-83} + 2^{-109} + 2^{-101} + 2^{-160} + 2^{-150} < 2^{-82}$$

∎

Lemma 2.11 *The sequence called "step 2" in the algorithm above calculates the values z_9^h and z_9^l so that*

$$z_9^h + z_9^l = \left(\left(a_4^h + a_4^l\right) + \left(z_3^h + z_3^l\right)\right) + \Delta_2$$

holds with $|\Delta_2| < 2^{-130}$ and that z_9^h is the double extended number nearest to the exact result.

Proof The first three and the last three instructions of the step are the *Fast2Sum* algorithm. To ensure that the results it produces are correct we must prove that the conditions on the magnitude of its input values are respected.
We have:

$$|a_4^h| \approx \frac{1}{6}$$
$$|z_3^h| \approx |a_5 x + \delta|$$

so one can easily see that the input condition is respected.
Secondly, we have

$$|z_4| \approx |a_4^h + z_3^h|$$
$$|z_8| \leq |a_4^l + z_3^l + \left(a_4^h + z_3^h\right) \cdot 2^{-64}| < 3 \cdot |a_4^h + z_3^h| \cdot 2^{-64}$$

so the input condition is respected also for the second utilisation of the *Fast2Sum* algorithm.
Thus we can assume that we have

$$z_4 + z_6 = a_4^h + z_3^h$$

exactly and that it suffices to show that

$$z_4 + z_8 = \left(\left(a_4^h + a_4^l\right) + \left(z_3^h + z_3^l\right)\right) + \Delta_2$$

Hence

$$\begin{aligned} z_4 + z_8 &= z_4 + \left(a_4^l + \left(z_3^l + z_6\right)(1+\epsilon_1)\right)(1+\epsilon_2) \\ &= \ldots \\ &= z_4 + z_6 + a_4^l + z_3^l + \Delta_2 \\ &= \ldots \\ &= \left(\left(a_4^h + a_4^l\right) + \left(z_3^h + z_3^l\right)\right) + \Delta_2 \end{aligned}$$

where Δ_2 is

$$\Delta_2 = z_3^l\epsilon_1 + z_6\epsilon_1 + a_4^l\epsilon_2 + z_3^l\epsilon_2 + z_6\epsilon_2 + z_3^l\epsilon_1\epsilon_2 + z_6\epsilon_1\epsilon_2$$

We have

$$|z_3^l\epsilon_1| < 2^{-148}$$
$$|z_6\epsilon_1| < 2^{-132}$$
$$|a_4^l\epsilon_2| < 2^{-132}$$
$$|z_3^l\epsilon_2| < 2^{-148}$$
$$|z_6\epsilon_2| < 2^{-132}$$
$$|z_3^l\epsilon_1\epsilon_2| < 2^{-200}$$
$$|z_6\epsilon_1\epsilon_2| < 2^{-196}$$

Therefore

$$|\Delta_2| < 2^{-131} + 2^{-147} + 2^{-196} + 2^{-200} < 2^{-130}$$

∎

Lemma 2.12 *The instruction sequence called "step 3" in the algorithm given above calculates z_{11}^h and z_{11}^l so that*

$$z_{11}^h + z_{11}^l = x^2$$

exactly and that z_{11}^h is the double extended precision number nearest to x^2.

Proof Trivial, the instructions sequence is the `Fast2Mult` algorithm whose correctness has already been proven.

Lemma 2.13 *The instruction sequence labelled "step 4" in the given algorithm calculates z_{15}^h and z_{15}^l so that*

$$z_{15}^h + z_{15}^l = \left(z_{11}^h + z_{11}^l\right) \cdot x + \Delta_4$$

holds with $|\Delta_4| < 2^{-173}$ and that z_{15}^h is the double extended precision number nearest to the exact result.

Proof One notices that the first two lines of the step are the `Fast2Mult` algorithm and that the last three constitute the `Fast2Sum` algorithm. So it suffices to prove the input conditions and to estimate the error due to the other instruction.
Concerning the input condition, we have:

$$|z_{12}| \approx |z_{11}^h \cdot x|$$

$$|z_{14}| \leq |z_{11}^l \cdot x + z_{11}^h \cdot x \cdot 2^{-64}| \approx |z_{11}^h \cdot x| \cdot 2^{-63}$$

so the input condition is respected.
Thus, as we can assume a correct result for

$$z_{12} + z_{13} = z_{11}^h \cdot x$$

we observe the following

$$\begin{aligned} z_{12} + z_{14} &= z_{12} + \left(z_{11}^l \cdot x + z_{13}\right)(1 + \epsilon_1) \\ &= \ldots \\ &= \left(z_{11}^h + z_{11}^l\right) \cdot x + \Delta_4 \end{aligned}$$

where Δ_4 is

$$\Delta_4 = z_{11}^l x \epsilon_1 + z_{13}\epsilon_1$$

with $|\epsilon_1| < 2^{-64}$.
Since

$$|z_{11}^l x \epsilon_1| < 2^{-174}$$

$$|z_{13}\epsilon_1| < 2^{-174}$$

we have the result $|\Delta_4| < 2^{-173}$. ∎

Lemma 2.14 *The instruction sequence named "step 5" calculates two double extended numbers z_{21}^h and z_{21}^l so that*

$$z_{21}^h + z_{21}^l = \left(z_{15}^h + z_{15}^l\right) \cdot \left(z_9^h + z_9^l\right) + \Delta_5$$

holds with $|\Delta_5| < 2^{-173}$ and that z_{21}^h is the double extended number nearest to the exact result.

Proof The first two lines of the step are the ***Fast2Mult*** sequence, the last three are the ***Fast2Sum*** algorithm. So it suffices to show that the input condition of the ***Fast2Sum*** algorithm is respected and to estimate the error induced by the other 2 operations.
Concerning the input conditions we observe:

$$|z_{17}| \approx |z_9^h \cdot z_{15}^h|$$

$$|z_{20}| < |z_9^h \cdot z^l 15 + z_9^l \cdot z_{15}^l + z_9^h \cdot z_{15}^h \cdot 2^{-64}| \leq |z_9^h \cdot z_{15}^h| \cdot 2^{-63}$$

thus the condition on the magnitude of the input values is respected.
Since we have an exact result for

$$z_{17} + z_{18} = z_9^h \cdot z_{15}^h$$

we can deduce

$$\begin{aligned}
z_{17} + z_{20} &= z_{17} + \left(z_9^h \cdot z_{15}^l + \left(z_9^l \cdot z_{15}^h + z_{18}\right)(1+\epsilon_1)\right)(1+\epsilon_2) \\
&= \ldots \\
&= z_9^h \cdot z_{15}^h + z_9^h \cdot z_{15}^l + z_9^l \cdot z_{15}^h + \\
&\quad z_9^l \cdot z_{15}^l - z_9^l \cdot z_{15}^l + z_9^l \cdot z_{15}^h \cdot \epsilon_1 + \\
&\quad z_{18} \cdot \epsilon_1 + z_9^h \cdot z_{15}^l \cdot \epsilon_2 + z_9^l \cdot z_{15}^h \cdot \epsilon_2 + \\
&\quad z_{18} \cdot \epsilon_2 + z_9^l \cdot z_{15}^h \cdot \epsilon_1\epsilon_2 + z_{18} \cdot \epsilon_1\epsilon_2 \\
&= \left(z_{15}^h + z_{15}^l\right) \cdot \left(z_9^h + z_9^l\right) + \Delta_5
\end{aligned}$$

where Δ_5 is

$$\Delta_5 = -z_9^l z_{15}^l + z_9^l z_{15}^h \epsilon_1 + z_{18}\epsilon_1 + z_9^h z_{15}^l \epsilon_2 + z_9^l z_{15}^h \epsilon_2 + z_{18}\epsilon_2 + z_9^l z_{15}^h \epsilon_1\epsilon_2 + z_{18}\epsilon_1\epsilon_2$$

with $\epsilon_1 < 2^{-64}$ and $\epsilon_2 < 2^{-64}$.
With the following estimates

$$\begin{aligned}
|-z_9^l z_{15}^l| &< 2^{-176} \\
|z_9^l z_{15}^h \epsilon_1| &< 2^{-244} \\
|z_{18}\epsilon_1| &< 2^{-176} \\
|z_9^h z_{15}^l \epsilon_2| &< 2^{-176}
\end{aligned}$$

$$|z_9^l z_{15}^h \epsilon_2| < 2^{-176}$$
$$|z_{18}\epsilon_2| < 2^{-176}$$
$$|z_9^l z_{15}^h \epsilon_1 \epsilon_2| < 2^{-244}$$
$$|z_{18}\epsilon_1\epsilon_2| < 2^{-244}$$

we get

$$|\Delta_5| < 2^{-173}$$

∎

Lemma 2.15 *The instuction sequence of the algorithm called "step 6" calculates two double extended precision numbers z_{26}^h and z_{26}^l so that*

$$z_{26}^h + z_{26}^l = h \cdot \left(z_{11}^h + z_{11}^l\right) + \Delta_6$$

holds with $|\Delta_6| < 2^{-159}$ and that z_{26}^h is the double precision number nearest to the exact result.

Proof One remarks that the first two lines in this step are the ***Fast2Mult*** sequence and that the last three are the ***Fast2Sum*** algorithm. So it suffices to show that the input condition of the ***Fast2Sum*** algorithm is respected to be able to assume correct results for these two parts of the step.
Let us first show the input condition:
We have

$$|z_{23}| \approx \left|\frac{1}{2} \cdot z_{11}^h\right|$$
$$|z_{25}| \approx \left|\frac{1}{2} \cdot z_{11}^l + \frac{1}{2} \cdot z_{11}^h \cdot 2^{-64}\right| < |z_{11}^h \cdot 2^{-64}|$$

Thus the input condition is respected.
So we have

$$z_{23} + z_{24} = h \cdot z_{11}^h$$

exactly and therefore

$$\begin{aligned} z_{23} + z_{25} &= z_{23} + \left(z_{11}^l \cdot h + z_{24}\right)(1+\epsilon_1) \\ &= \ldots \\ &= h \cdot \left(z_{11}^h + z_{11}^l\right) + \Delta_6 \end{aligned}$$

where Δ_6 is $\Delta_6 = h \cdot z_{11}^l \cdot \epsilon_1 + z_{24} \cdot \epsilon_1$ with $\epsilon_1 < 2^{-64}$
One can easily check that

$$|h \cdot z_{11}^l \cdot \epsilon_1| < 2^{-160}$$
$$|z_{24} \cdot \epsilon_1| < 2^{-160}$$

so we have as a result

$$|\Delta_6| < 2^{-159}$$

∎

Lemma 2.16 *The instruction sequence called "step 7" in the algorithm given above calculates two double extended numbers z_{33}^h and z_{33}^l so that*

$$z_{33}^h + z_{33}^l = \left(\left(z_{26}^h + z_{26}^l\right) + \left(z_{21}^h + z_{21}^l\right)\right) + \Delta_7$$

holds with $|\Delta_7| < 2^{-158}$ and that z_{33}^h is the double extended precision number nearest to the exact result.

Proof As one can see, the first three and the last three instructions in the sequence of step 7 are two times the `Fast2Sum` sequence. We will first check their input condition:

$$|z_{26}^h| \approx |h \cdot x^2| \approx \frac{1}{2} \cdot x^2$$

$$|z_{21}^h| \approx |x^3 \cdot (a_4 + (a_5 + (a_6 + a_7 x)\, x)\, x)\,|$$

To prove that $|z_{26}^h| > |z_{21}^h|$, we locate numerically the minimum of

$$\frac{1}{2} \cdot x^2 - \left|x^3 \cdot (a_4 + (a_5 + (a_6 + a_7 x)\, x)\, x)\right|$$

in the given range for x using a tool such as Maple. The minimum is 0 for $x = 0$, thus the input condition is respected.
Concerning the second use of the `Fast2Sum` sequence, we observe as follows:

$$|z_{28}| \approx |z_{26}^h + z_{21}^h|$$

$$|z_{32}| \approx |z_{26}^l + z_{21}^l + \left(z_{26}^h + z_{21}^h\right) \cdot 2^{-64}| < |\left(z_{26}^h + z_{21}^h\right) \cdot 2^{-63}|$$

so the input condition is verfied also for this sequence.
Let us now estimate the error due to the other two instructions of step 7.
We have exactly $z_{28} + z_{30} = z_{26}^h + z_{21}^h$ and thus

$$\begin{aligned} z_{28} + z_{32} &= z_{28} + \left(z_{26}^l + \left(z_{21}^l + z_{30}\right)(1+\epsilon_1)\right)(1+\epsilon_2) \\ &= \ldots \\ &= \left(\left(z_{26}^h + z_{26}^l\right) + \left(z_{21}^h + z_{21}^l\right)\right) + \Delta_7 \end{aligned}$$

where Δ_7 is

$$\Delta_7 = z_{21}^l\epsilon_1 + z_{30}\epsilon_1 + z_{26}^l\epsilon_2 + z_{21}^l\epsilon_2 + z_{30}\epsilon_2 + + z_{21}^l\epsilon_1\epsilon_2 + z_{30}\epsilon_1\epsilon_2$$

with $|\epsilon_1| < 2^{-64}$ and $|\epsilon_2| < 2^{-64}$ So we can give some estimates for the different terms of the sum which we have commonly calculated using a numerical tool:

$$|z_{21}^l\epsilon_1| < 2^{-177}$$

$$|z_{30}\epsilon_1| < 2^{-160}$$

$$|z_{26}^l \epsilon_2| < 2^{-160}$$
$$|z_{21}^l \epsilon_2| < 2^{-177}$$
$$|z_{30} \epsilon_2| < 2^{-160}$$
$$|z_{21}^l \epsilon_1 \epsilon_2| < 2^{-241}$$
$$|z_{30} \epsilon_1 \epsilon_2| < 2^{-224}$$

Therefore we have

$$|\Delta_7| < 2^{-158}$$

∎

Lemma 2.17 *The sequence labelled "step 8" of the algorithm given above calculates two double extended numbers p^h and p^l so that*

$$p^h + p^l = \left(\left(z_{33}^h + z_{33}^l\right) + x\right) + \Delta_8$$

holds with $|\Delta_8| < 2^{-142}$ and that p^h is the double extended number nearest to the exact result.

Proof Also in this sequence, we see that the first three and the last three instructions of the step consist in the ***Fast2Sum*** algorithm. We will thus show that the input conditions are respected:
We have to show first that $|x| > |z_{33}^h|$ is true.

$$|z_{33}^h| \approx \left|\frac{1}{2}x^2 + x^3 \cdot (a_4 + (a_5 + (a_6 + a_7 x)\, x)\, x)\right|$$

Since

$$x \in \left[-\frac{\ln 2}{2^{14}}; \frac{\ln 2}{2^{14}}\right] \subset]-1; 1[$$

this condition is verified.
Concerning the second use of the ***Fast2Sum*** sequence, we have to show that $|z_{35}| > |z_{38}|$. We have

$$|z_{35}| \approx |z_{33}^h + x|$$
$$|z_{38}| \approx |z_{33}^l + \left(z_{33}^h + x\right) \cdot 2^{-64}| < |z_{33}^h \cdot 2^{-63}| + |x \cdot 2^{-64}|$$

Thus the input condition is respected.
Let us now estimate the error induced by the other instruction.
We have $z_{35} + z_{37} = z_{33}^h + x$ exactly.
Therefore

$$\begin{aligned} z_{35} + z_{38} &= z_{35} + \left(z_{33}^l + z_{37}\right)(1 + \epsilon_1) \\ &= \dots \\ &= \left(\left(z_{33}^h + z_{33}^l\right) + x\right) + \Delta_8 \end{aligned}$$

where Δ_8 is $\Delta_8 < z_{33}^l \epsilon_1 + z_{37}\epsilon_1$ with $\epsilon_1 < 2^{-64}$.
We can check easily using a numerical tool, that

$$|z_{33}^l \epsilon_1| < 2^{-160}$$

$$|z_{37}\epsilon_1| < 2^{-143}$$

Thus $|\Delta_8| < 2^{-142}$. ∎

We know now the different errors Δ_i, $i = 1 \ldots 8$ due to the serveral steps of the algorithm that calculates the polynomial approximation of reduced argument of the `exp` function. Using this facts, we can now give a proof for the accuracy theorem given above.
In this proof, we will sum up the different error, weighted differently. To simplify it a little bit and to increase its readablity, we will use the following conventions, notations and assumptions:

- A sum of two double extended precision numbers that represent in some way the higher and the lower part of a higher precision number, will be noted $c^{h/l}$ instead of $\left(c^h + c^l\right)$.
- We will also make a light abuse of the operation signs $+$ and $\cdot$. They will be used either on double extended precision numbers either on sum of two double extended precision numbers noted $c^{h/l}$; e.g. $c_1^{h/l} \cdot c_2^{h/l}$ will represent $\left(c_1^h + c_1^l\right) \cdot \left(c_2^h + c_2^l\right)$.
- We will note δ_1, δ_2, δ_3 and δ_4 the errors made when we represent the constants $a^{h/l}$, a_5, a_6 and a_7 in double extended precision. We know that:

$$|\delta_1| < 2^{-128}$$

$$|\delta_2| < 2^{-64}$$

$$|\delta_3| < 2^{-64}$$

$$|\delta_4| < 2^{-64}$$

Proof Using the assumptions and lemmata shown above, we get following equation:

$$\begin{aligned}
p^{h/l} &= x + z_{33}^{h/l} + \Delta_8 \\
&= x + z_{26}^{h/l} + z_{21}^{h/l} + \Delta_7 + \Delta_8 \\
&= x + h \cdot z_{11}^{h/l} + z_{15}^{h/l} \cdot z_9^{h/l} + \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 \\
&= x + h \cdot \left(x^2 + \Delta_3\right) + \left(z_{11}^{h/l} \cdot x + \Delta_4\right) \cdot \left(\left(a_4^{h/l} + \delta_1\right) + z_3^{h/l} + \Delta_2\right) + \\
&\quad + \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 \\
&= \ldots \\
&= x + h \cdot x^2 + \left(x^3 + \Delta_3 x + \Delta_5\right) \left(a_4^{h/l} + \delta_1 + ((a_5 + \delta_2) + ((a_6 + \delta_3)\right. \\
&\quad \left. + \ (a_7 + \delta_4) \cdot x) \cdot x) \cdot x + \Delta_1 + \Delta_2\right) \\
&\quad + \Delta_5 + h \cdot \Delta_3 + \Delta_6 + \Delta_7 + \Delta_8
\end{aligned}$$

At this point, we will simplify a little bit the calculations using some approximations. First, since $|\Delta_3| = 0$ we get

$$\begin{aligned} p^{h/l} &= x + h \cdot x^2 + \\ &+ \left(x^3 + \Delta_4\right) \cdot \\ &\cdot \left(a_4^{h/l} + \delta_1 + (a_5 + \delta_2 + (a_6 + \delta_3 + (a_7 + \delta_4)\, x)\, x)\, x + \Delta_1 + \Delta_2\right) + \\ &+ \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 \end{aligned}$$

Secondly, we can give an approximation of the final error Δ_9 due to the errors δ_1, δ_2, δ_3 and δ_4. As $|x^3| < 2^{-46}$, $|x^4| < 2^{-62}$, $|x^5| < 2^{-77}$ and $|x^6| < 2^{-93}$ we will get in a rather pessimistic approximation

$$|\Delta_9| = |f\left(\delta_1, \delta_2, \ldots\right)| < 2^{-46} \cdot 2^{-128} + 2^{-62} \cdot 2^{-64} + 2^{-77} \cdot 2^{-64} + 2^{-93} \cdot 2^{-64} + \hat{\delta}$$

where $|\hat{\delta}| < 2^{-128}$.
Thus

$$|\Delta_9| < 2^{-125}$$

Using this observations, we obtain

$$\begin{aligned} p^{h/l} &= x + h \cdot x^2 + x^3 \cdot \left(a_4^{h/l} + (a_5 + (a_6 + a_7 x)\, x)\, x\right) + \\ &+ x^3\Delta_1 + x^3\Delta_2 + \Delta_4\Delta_1 + \Delta_4\Delta_2 + \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 + \Delta_9 \\ &= x + h \cdot x^2 + x^3 \cdot \left(a_4^{h/l} + (a_5 + (a_6 + a_7 x)\, x)\, x\right) + \Delta \end{aligned}$$

where Δ is

$$\Delta = x^3\Delta_1 + x^3\Delta_2 + \Delta_4\Delta_1 + \Delta_4\Delta_2 + \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 + \Delta_9$$

So, by summing up the different values, one gets:

$$|\Delta| < 2^{-124}$$

We must now prove the additional property pronounced in the theorem which says that for $x \in \left[-2^{-30}; 2^{-30}\right]$ we have

$$|\Delta| < 2^{-141}$$

For doing this we will only reconsider the error Δ_9 and the last estimate step which will give us a sufficient bound. We had that

$$\Delta = x^3\Delta_1 + x^3\Delta_2 + \Delta_4\Delta_1 + \Delta_4\Delta_2 + \Delta_5 + \Delta_6 + \Delta_7 + \Delta_8 + \Delta_9$$

and for the given particular domain of x we have
$|x^3| < 2^{-90}$, $|x^4| < 2^{-120}$, $|x^5| < 2^{-150}$ and $|x^6| < 2^{-180}$.
Thus

$$|\Delta_9| < 2^{-180}$$

and we get the following final summing up

$$\begin{aligned} |\Delta| &< 2^{-90} \cdot 2^{-82} + 2^{-90} \cdot 2^{-130} + 2^{-173} \cdot 2^{-82} + \\ &+ 2^{-173} + 2^{-159} + 2^{-158} + 2^{-142} + 2^{-180} \\ &< 2^{-141} \end{aligned}$$

∎

Theorem 2.18 *Let be $i_1 \in \{0, ..., 127\}$ and $i_2 \in \{0, ..., 127\}$. Let us assume that we dispose of two tables in which approximations to the exact values $2^{\frac{i_1}{2^7}}$ respectively $2^{\frac{i_1}{2^{14}}}$ are stored. Let us assume that each entry consists in 2 double extended numbers t_1^h and t_1^l (respectively t_2^h and t_2^l) that satisfy the following conditions:*

- *t_j^h is the double extended precision number nearest to $2^{\frac{i_j}{2^{(j \cdot 7)}}}$*
- *$|t_j^l| < |t_j^h| \cdot 2^{-64}$*
- *$t_j^h + t_j^l = 2^{\frac{i_j}{2^{(j \cdot 7)}}} + \delta_j$ with $|\delta_j| < 2^{-128}$*

So the following instruction sequence calculates two double extended precision numbers t^h and t^l so that

$$t^h + t^l = 2^{\frac{i_1}{2^7}} \cdot 2^{\frac{i_1}{2^{14}}} + \Delta$$

where $|\Delta| < 2^{-123}$ and that t^h is the double extended number nearest to the correct result. Instruction sequence:

`fma.s1 u_1 = t_1_h, t_2_h, f0`	
`fms.s1 u_2 = t_1_h, t_2_h, u_1`	$u_1 + u_2 \leftarrow$ ***Fast2Mult*** $\left(t_1^h, t_2^h\right)$
`fma.s1 u_3 = t_1_h, t_2_l, u_2`	$u_3 \leftarrow \left\langle t_1^h \cdot t_2^l + u_2 \right\rangle$
`fma.s1 u_4 = t_1_l, t_2_h, u_3`	$u_4 \leftarrow \left\langle t_1^l \cdot t_2^h + u_3 \right\rangle$
`fadd.s1 t_h = u_1, u_4`	
`fsub.s1 u_5 = t_h, u_1`	
`fsub.s1 t_l = u_4, u_5`	$t^h + t^l \leftarrow$ ***Fast2Sum***(u_1, u_4)

Proof We observe that the first two steps of the instruction sequence given consist in the ***Fast2Mult*** algorithm and that the last three constitute the ***Fast2Sum*** sequence.
Let us thus show first that the input condition of the ***Fast2Sum*** algorithm is respected:

$$|u_1| \approx |t_1^h \cdot t_2^h|$$

$$\begin{aligned} |u_4| &\approx |t_1^l \cdot t_2^h + t_1^h \cdot t_2^l + t_1^h \cdot t_2^h \cdot 2^{-64}| \\ &< |t_1^l \cdot t_2^h| + |t_1^h \cdot t_2^l| + |t_1^h \cdot t_2^h \cdot 2^{-64}| \\ &< \dots \\ &< |t_1^h \cdot t_2^h| \cdot 2^{-63} < |t_1^h \cdot t_2^h| \end{aligned}$$

So the input condition is respected.
Therefore, it suffices to estimate the error of $u_1 + u_4$.
We know that $u_1 + u_2 = t_1^h \cdot t_2^h$ correctly.
So we get:

$$\begin{aligned} u_1 + u_4 &= u_1 + \left(t_1^l \cdot t_2^h + \left(t_1^h \cdot t_2^l + u_2\right)(1+\epsilon_1)\right)(1+\epsilon_2) \\ &= \dots \\ &= \left(t_1^h + t_1^l\right) \cdot \left(t_2^h + t_2^l\right) + \Delta_1 \end{aligned}$$

where Δ_1 is

$$\Delta_1 = t_1^h t_2^l \epsilon_1 + u_2\epsilon_1 + t_1^l t_2^h \epsilon_2 + t_1^h t_2^l \epsilon_2 + u_2\epsilon_2 + t_1^h t_2^l \epsilon_1\epsilon_2 + u_2\epsilon_1\epsilon_2 - t_1^l t_2^l$$

with $|\epsilon_1| < 2^{-64}$ and $|\epsilon_2| < 2^{-64}$. As one can easily check, we have:

$$|t_1^h t_2^l \epsilon_1| < 2^{-127}$$
$$|u_2\epsilon_1| < 2^{-127}$$
$$|t_1^l t_2^h \epsilon_2| < 2^{-127}$$
$$|t_1^h t_2^l \epsilon_2| < 2^{-127}$$
$$|u_2\epsilon_2| < 2^{-127}$$
$$|t_1^h t_2^l \epsilon_1\epsilon_2| < 2^{-191}$$
$$|u_2\epsilon_1\epsilon_2| < 2^{-191}$$
$$|t_1^l t_2^l| < 2^{-127}$$

Thus $|\Delta_1| < 2^{-124}$.
To this error, we must add the error due to the rounding of the stored values which can be approximated as follows:

$$\begin{aligned} 2^{\frac{i_1}{2^7}} \cdot 2^{\frac{i_1}{2^{14}}} + \Delta_2 &= \left(2^{\frac{i_1}{2^7}} + \delta_1\right) \cdot \left(2^{\frac{i_1}{2^{14}}} + \delta_2\right) \\ &= 2^{\frac{i_1}{2^7}} \cdot 2^{\frac{i_1}{2^{14}}} + \\ &+ 2^{\frac{i_1}{2^7}} \cdot \delta_2 + 2^{\frac{i_1}{2^{14}}} \cdot \delta_1 + \delta_1\delta_2 \end{aligned}$$

We can estimate the different terms as given below:

$$|2^{\frac{i_1}{2^7}} \cdot \delta_2| < 2^{-127}$$

$$|2^{\frac{i_1}{2^{14}}} \cdot \delta_1| < 2^{-127}$$
$$|\delta_1 \delta_2| < 2^{-256}$$

Therefore the error due to the incorrect memorization of the table values can be bound by

$$|\Delta_2| < 2^{-125}$$

and we can deduce the final error Δ which is

$$|\Delta| < 2^{-123}$$

∎

Theorem 2.19 *Let p^h and p^l be two double extended precision numbers that satisfy the following conditions:*
$p^h + p^l \in \left[-2^{-15}; 2^{-15}\right]$ *and* $p^l \in \left[-2^{-79}; 2^{-79}\right]$.
Let t^h and t^l be two double extended numbers that satisfy:
$t^h + t^l \in [1; 2[$ *and* $t^l \in \left[-2^{-63}; 2^{-63}\right]$.
Let x^l be a double extended number so that $x^l \in \left[-2^{-79}; 2^{-79}\right]$.
So the instruction sequence given below calculates two double extended numbers s^h and s^l so that

$$s^h + s^l = \left(t^h + t^l\right) \cdot \left(1 + \left(p^h + p^l\right)\right) \cdot \left(1 + x^l\right) + \Delta$$

where $|\Delta| < 2^{-123}$ and that s^h is the double extended number nearest to the correct result.
Instruction sequence:[8]

(i)	`fma.s1 v_1_h = x_l, t_h, f0`	
(ii)	`fms.s1 v_1_l = x_l, t_h, v_1_h`	$v_1^h + v_1^l \leftarrow$ ***Fast2Mult*** $\left(x^l, t^h\right)$
(iii)	`fadd.s1 v_2 = t_l, v_1_l`	$v_2 \leftarrow t^l \oplus v_1^l$
(iv)	`fadd.s1 v_3_h = t_h, v_1_h`	
(v)	`fsub.s1 v_10 = v_3_h, t_h`	
(vi)	`fsub.s1 v_3_l = v_1_h, v_10`	$v_3^h + v_3^l \leftarrow$ ***Fast2Sum*** $\left(t^h, v_1^h\right)$
(vii)	`fadd.s1 v_15 = v_3_l, v_2`	$v_{15} \leftarrow v_3^l \oplus v_2$
(viii)	`fma.s1 v_4_h = p_h, t_h, f0`	
(ix)	`fms.s1 v_4_l = p_h, t_h, v_4_h`	$v_4^h + v_4^l \leftarrow$ ***Fast2Mult*** $\left(p^h, t^h\right)$
(x)	`fma.s1 v_5 = x_l, v_4_h, v_4_l`	$v_5 \leftarrow \left\langle x^l \cdot v_4^h + v_4^l \right\rangle$
(xi)	`fma.s1 v_6 = p_l, t_h, v_5`	$v_6 \leftarrow \left\langle p^l \cdot t^h + v_5 \right\rangle$
(xii)	`fma.s1 v_7 = t_l, p_h, v_6`	$v_7 \leftarrow \left\langle t^l \cdot p^h + v_6 \right\rangle$

[8] The roman numbers in the beginning of each line are not part of the algorithm, they will be useful during the proof.

(xiii)	`fadd.s1 v_8_h = v_3_h, v_4_h`	
(xiv)	`fsub.s1 v_11 = v_8_h, v_3_h`	
(xv)	`fsub.s1 v_8_l = v_4_h, v_11`	$v_8^h + v_8^l \leftarrow \mathit{Fast2Sum}\left(v_3^h, v_4^h\right)$
(xvi)	`fadd.s1 v_9 = v_8_l, v_7`	$v_9 \leftarrow v_8^l \oplus v_7$
(xvii)	`fadd.s1 v_13 = v_9, v_15`	$v_{13} \leftarrow v_9 \oplus v_{15}$
(xviii)	`fadd.s1 s_h = v_8_h, v_13`	
(xix)	`fsub.s1 v_14 = s_h, v_8_h`	
(xx)	`fsub.s1 s_l = v_13, v_14`	$s^h + s^l \leftarrow \mathit{Fast2Sum}\left(v_8^h, v_{13}\right)$

Remark: The error Δ given in the last theorem expresses only the error due the calculation of the result value with the given input values. The error due to the errors of the input values is not considered here which it will be in another section.

Proof
As usual, this instruction sequences uses the *Fast2Sum* and the *Fast2Mult* algorithm. The latter can be seen in lines *(i)* and *(ii)* and *(viii)* and *(ix)*. As proven above it is exact for all input values.
The lines *(iv)* - *(vii)*, *(xiii)* - *(xv)* and *(xviii)* - *(xx)* constitute the *Fast2Sum* sequence. Let us show first that their input conditions are respected.

- Lines *(iv)* - *(vi)*

  $$|v_1^h| \approx |x^l \cdot t^h| < |t^h| \cdot 2^{-79} < |t^h|$$

  so the input condition is respected.

- Lines *(xiii)* - *(xv)*

  $$|v_3^h| > |t^h|$$
  $$|v_4^h| \approx |p^h \cdot t^h| < |t^h| \cdot 2^{-15} < |t^h| < |v_3^h|$$

  thus the input condition is respected also here.

- Lines *(xviii)* - *(xx)*

  $$|v_8^h| > |v_3^h| > |t^h|$$

$$\begin{aligned} |v_{13}| &\approx |v_9 + v_{15}| \\ &\approx |v_8^l + v_7 + v_{15}| \\ &< |v_8^h| \cdot 2^{-64} + |v_7| + |v_{15}| \\ &\approx |v_3^h + v_4^h| \cdot 2^{-64} + |t^l \cdot p^h + v_6| + |v_3^l + v_2| \end{aligned}$$

$$\begin{aligned} &< |v_3^h| \cdot 2^{-64} + |v_4^h| \cdot 2^{-64} + |t^h \cdot p^h| \cdot 2^{-64} + |v_6| + |v_3^h| \cdot 2^{-64} + |v_2| \\ &\approx \ldots \\ &< \ldots \\ &< |t^h| \cdot 2^{-62} < |t^h| < |v_8^h| \end{aligned}$$

For this reason the input condition is respected.

So we can assume the following mathematically exact results:

- $v_1^h + v_1^l = x^l \cdot t^h$
- $v_3^h + v_3^l = t^h + v_1^h$
- $v_4^h + v_4^l = p^h \cdot t^h$
- $v_8^h + v_8^l = v_3^h + v_4^h$
- $s^h + s^l = v_8^h + v_{13}$

Therefore we can do the following estimates:

$$\begin{aligned} s^h + s^l &= v_8^h + v_{13} \\ &= v_8^h + (v_9 + v_{15}) \cdot (1 + \epsilon_1) \\ &= v_8^h + \left(\left(v_8^l + v_7\right) \cdot (1 + \epsilon_2) + \left(v_3^l + v_2\right) \cdot (1 + \epsilon_3)\right)(1 + \epsilon_1) \\ &= \ldots \\ &= v_8^h + v_8^l + v_7 + v_2 + v_3^l + \Delta_1 \end{aligned}$$

with

$$\begin{aligned} \Delta_1 &= v_8^l\epsilon + v_7\epsilon_2 + v_3^l\epsilon_3 + v_2\epsilon_3 + v_8^l\epsilon_1 + v_7\epsilon_1 + v_8^l\epsilon_1\epsilon_2 + v_7\epsilon_1\epsilon_2 + v_3^l\epsilon_1 + \\ &+ v_2\epsilon_1 + v_3^l\epsilon_1\epsilon_3 + v_2\epsilon_1\epsilon_3 \end{aligned}$$

Thus

$$\begin{aligned} s^h + s^l &= v_3^h + v_4^h + v_7 + v_2 + v_3^l + \Delta_1 \\ &= t^h + v_1^h + v_4^h + v_7 + v_2 + \Delta_1 \\ &= t^h + v_1^h + v_4^h + v_7 + \left(t^l + v_1^l\right)(1 + \epsilon_4) + \Delta_1 \\ &= \ldots \\ &= t^h + t^l + x^l + t^h + v_4^h + v_7 + \Delta_2 \end{aligned}$$

with

$$\Delta_2 = t^l\epsilon_4 + v_1^l\epsilon_4 + \Delta_1$$

Therefore

$$\begin{aligned} s^h + s^l &= t^h + t^l + x^l t^h + v_4^h + \left(t^l p^h + v_6\right) \cdot (1+\epsilon_5) + \Delta_2 \\ &= \dots \\ &= t^h + t^l + x^l t^h + v_4^h + t^l p^h + t^h p^l + v_5 + \Delta_3 \end{aligned}$$

with

$$\Delta_3 = p^l t^h \epsilon_6 + v_5\epsilon_6 + t^l p^h \epsilon_5 + p^l t^h \epsilon_5 + v_5\epsilon_5 + p^l t^h \epsilon_5\epsilon_6 + v_5\epsilon_5\epsilon_6 + \Delta_2$$

Thus

$$\begin{aligned} s^h + s^l &= t^h + t^l + x^l t^h + v_4^h + t^l p^h + t^h p^l + \left(x^l v_4^h + v_4^l\right) \cdot (1+\epsilon_7) + \Delta_3 \\ &= \dots \\ &= t^h + t^l + x^l t^h + t^h p^h + t^l p^h + t^h p^l + x^l v_4^h + \Delta_4 \end{aligned}$$

with

$$\Delta_4 = x^l v_4^h \epsilon_7 + v_4^l \epsilon_7 + \Delta_3$$

So we have

$$\begin{aligned} s^h + s^l &= t^h + t^l + x^l t^h + t^h p^h + t^l p^h + t^h p^l + x^l p^h t^h \,(1+\epsilon_8) + \Delta_4 \\ &= \dots \\ &= t^h + t^l + x^l t^h + t^h p^h + t^l p^h + t^h p^l + x^l p^h t^h + \Delta_5 \end{aligned}$$

with

$$\Delta_5 = x^l p^h t^h \epsilon_8 + \Delta_4$$

So finally,

$$\begin{aligned} s^h + s^l &= t^h + t^l + x^l t^h + t^h p^h + t^l p^h + t^h p^l + x^l p^h t^h + x^l p^l t^h + x^l t^l + t^l p^l + \\ &+ t^l x^l p^h + t^l x^l p^l - x^l p^l t^h - x^l t^l - t^l p^l - t^l x^l p^h - t^l x^l p^l + \Delta_5 \\ &= \left(t^h + t^l\right)\left(1 + \left(p^h + p^l\right)\right) \cdot \left(1 + x^l\right) + \Delta \end{aligned}$$

with

$$\Delta = -x^l p^l t^h - x^l t^l - t^l p^l - t^l x^l p^h - t^l x^l p^l + \Delta_5$$

In all this estimates, the ϵ_j are $|\epsilon_j| < 2^{-64}$. And we can deduce the final error estimate:

$$\begin{aligned} |\Delta| &< |x^l p^l t^h| + |x^l t^l| + |t^l p^l| + |t^l x^l p^h| + \\ &+ |t^l x^l p^l| + |x^l p^h t^h \epsilon_8| + |x^l v_4^h \epsilon_7| + |v_4^l \epsilon_7| + \\ &+ |p^l t^h \epsilon_6| + |v_5\epsilon_6| + |t^l p^h \epsilon_5| + |p^l t^h \epsilon_5| + \\ &+ |v_5\epsilon_5| + |p^l t^h \epsilon_5\epsilon_6| + |v_5\epsilon_5\epsilon_6| + |t^l\epsilon_4| + \\ &+ |v_1^l\epsilon_4| + |v_8^l\epsilon_2| + |v_7\epsilon_2| + |v_3^l\epsilon_3| + \\ &+ |v_2\epsilon_3| + |v_8^l\epsilon_1| + |v_7\epsilon_1| + |v_8^l\epsilon_1\epsilon_2| + \\ &+ |v_7\epsilon_1\epsilon_2| + |v_3^l\epsilon_1| + |v_2\epsilon_1| + |v_3^l\epsilon_1\epsilon_3| + \\ &+ |v_2\epsilon_1\epsilon_3| \end{aligned}$$

By approximating all these different values as it has been shown in a similar way in the previous sections, one gets the following final result:

$$\begin{aligned}
|\Delta| &< 2^{-157} + 2^{-142} + 2^{-142} + 2^{-157} + 2^{-221} + 2^{-157} + 2^{-157} + \\
&+ 2^{-142} + 2^{-142} + 2^{-141} + 2^{-142} + +2^{-142} + 2^{-206} + 2^{-127} + 2^{-206} + \\
&+ 2^{-126} + 2^{-140} + 2^{-126} + 2^{-126} + 2^{-126} + 2^{-140} + 2^{-190} + \\
&+ 2^{-204} + 2^{-126} + 2^{-126} + 2^{-190} + 2^{-190} \\
&= 6 \cdot 2^{-126} + 2^{-127} + \ldots < 2^{-123}
\end{aligned}$$

∎

Lemma 2.20 *Let s^h and s^l be two double extended precision numbers so that $s^h + s^l \in [0..2[$ and that $s^l \in \left[-2^{-64}; 2^{-64}\right]$.*
Let `f2M` be a double extended precision number that represents a value 2^M, $M \in \mathbb{N}$.
So the following instruction sequence calculates $2^M \cdot \left(s^h + s^l\right)$ correctly and rounds this result to a double precision number `res` without any intermediate error as long as the final result does not underflow. The rounding is effectuated according to the user rounding mode set.
Instruction sequence:

`fma.s1 s_l_2M = s_l, f2M, f0`	$s^l_{2^M} \leftarrow s^l \otimes f2M$
`fma.d.s0 res = s_h, f2M, s_l_2M`	$res \leftarrow \left\langle s^h \cdot f2M + s^l_{2^M} \right\rangle_{53}$

Remark: The IEEE standard [5] applied on the calculation of a transcendental function such as `exp` require the correct setting of the "inexact" flag when underflow occurs and taking of a trap for exception handling. As the exponential function is monotonic increasing the underflow of the final result can simply be excluded from the main path of the algorithm by a preceeding tests on the argument of the function. So one has to think only of the underflow of intermediate results such as $s^l_{2^M}$ in the following proof.

Proof If neither the final nor the intermediate value $s^l_{2^M}$ underflow, the multiplication with a power of 2 is exact. So one can trivially see that all the properties are verified.
We have assumed that the final result does not underflow so we have only to consider the case in which the intermediate result $s^l_{2^M}$ underflows while the final result does not. In this case, we observe:

$$|res| \approx |s^h \cdot 2^M + s^l_{2^M}| > 2^{-1022}$$

because the final result does not underflow and

$$|s^l_{2^M}| \approx |s^l \cdot 2^M| < |s^h \cdot 2^M| \cdot 2^{-64}$$

So we can do the following approximation

$$|s^h \cdot 2^M| 2^{-1021}$$

and as we know that the value $s^l_{2^M}$ which is stored in register format with a 17 bit exponent underflows, we have

$$|s^l_{2^M}| < 2^{2^{17-1}-2} = 2^{65534}$$

So the value $s^l_{2^M}$ is so small in comparision to $s^h \cdot 2^M$ that, even when we loose all significant bits in $s^l_{2^M}$ this will not affect the final result.
If the final result underflows the sequence is still correct, since the intermediate calculations during the last `fma.d.s0` instruction are done in infinite precision before rounding and the temporary value $s^l_{2^M}$ result, stored in double extended register format, does not necessarily underflow when the final result does and if it does a similar argument to the preceeding one can be used. In contrast to the main path, a trap to an exception handler must be taken. ∎

Theorem 2.21 *Let an algorithm consist in all the steps given above (cf. 2.8, 2.9, 2.18, 2.19 and 2.20), which are the range reduction step, the table entry multiplication step, the polynomial approximation, the reconstruction and the final rounding step, and in some supplementary instructions which can be found in the complete listing (cf. 7) and whose correctness can be trivially proven.*
So this algorithm calculates an approximation $\hat{y}$ to the exponential function e^x so that at least significant 116 bit of $\hat{y}$ are correct and rounds this approximation to a double precision number according to the rounding mode set.

Proof As we just have proven that the final multiplication by 2^M can be done without any error and as we know all the errors of the different steps, it suffices to sum a the weighted sum of all these errors. We can postulate

$$s^{h/l} = \frac{e^x}{2^M} + \Delta$$

where Δ is

$$|\Delta| < |\Delta_{\mathbf{reduc}}| + |\Delta_{\mathbf{approx}}| + |\Delta_{\mathbf{arith}}|$$

where $\Delta_{\mathbf{reduc}}$ is the error due to range reduction, $\Delta_{\mathbf{approx}}$ is the error induced by approximating the exponential function by a polynomial and $\Delta_{\mathbf{arith}}$ is the error produced by the floating point arithmetic. Let us remind the complete term showing how the value $s^{h/l}$ is evaluated:

$$s^{h/l} = t^{h/l} \cdot \left(1 + p^{h/l}(r)\right) \cdot \left(1 + x^l\right) + \Delta.$$

As $t^{h/l} \in [0..2[$ and by using theorem 2.8 we obtain

$$|\Delta_{\mathbf{reduc}}| < 2 \cdot 2^{-118} = 2^{-117}$$

and similarly using theorem 2.6 we get

$$|\Delta_{\mathbf{approx}}| < 2 \cdot 2^{-123} = 2^{-122}$$

Further we can write:

$$|\Delta_{\mathbf{arith}}| < |\Delta_{\mathbf{reconst}}| + |\Delta_{\mathbf{table}}| + 2 \cdot |\Delta_{\mathbf{poly}}|$$

Using the theorems 2.19, 2.18 and 2.9 one gets

$$|\Delta_{\mathbf{poly}}| < 2^{-123} + 2^{-123} + 2 \cdot 2^{-123} < 2^{-121}$$

So the final overall error is

$$|\Delta| < 2^{-117} + 2^{-122} + 2^{-121} < 2^{-116}$$

and the property follows trivially using lemma 2.20. ∎

Theorem 2.22 (Lefèvre) *[23, 10]*
Let y be the exponential function value of a double precision floating point value x. Let y^* be an approximation to y so that the distance between the mantissas of y and y^* can be majored by ϵ. So

- *for $|x| \geq 2^{-30}$, if $\epsilon < 2^{-53-59} = 2^{-112}$, for all rounding modes, rounding y^* to double precision is equivalent to rounding y.*
- *for $|x| < 2^{-30}$, if $\epsilon < 2^{-53-104} = 2^{-157}$, rounding y^* is equivalent to rounding y.*

Proof see [10]

Corollary 2.23 *The algorithm given calculates a double precision number y which is a correctly rounded approximation to e^x for all double precision input values which satisfy the predicate $|x| < 2^{-53} \vee |x| > 2^{30}$.*

Proof We have proven above that the algorithm given calculates a 112 bit correct approximation of the exponential function for all inputs and that it rounds this result to double precision, the property to be shown follows trivially using Lefèvre's theorem. ∎

2.3 Towards a correctly rounded algorithm for all arguments

With the last corollary (cf. 2.23) we have proven that the algorithm we present in this article calculates a correctly rounded approximation to the exponential function unless the function's argument is not contained in a very small range of inputs, i.e. it is correct for all $x \notin \left[-2^{-30}; -2^{-53}\right] \cup \left[2^{-53}; 2^{-30}\right]$. In this section, we will show that the algorithm calculates a 157 bits correct result for input values $x \in \left[-2^{-30}; 2^{-53}\right] \cup \left[2^{-53}; 2^{-30}\right]$. For this we will use the supplementary results of some lemmata above that we have not used yet.

At first sight, one could argue that the algoritm simply cannot return a 157 bit correct result as all intermediate values are stored in two double extended precision registers with each 64 bits of mantissa. But one has to remember that this high precision is needed only

when the table maker's dilemma occurs. In the other cases less precision is sufficient. In fact if one considers more closely the contents of e.g. the two registers s^h and s^l which contain the result of the function before the final multiplication by 2^M and before rounding, one observes the following situation on an input argument on which the table maker's dilemma occurs e.g. in round to $+\infty$ mode[9]: the infinite precision mantissa of the result which must be approximated and stored in s^h and s^l is e.g. of the following form (cf. [2]):

$$2^k \cdot \overbrace{\underbrace{1.x_1x_2x_3x_4x_5\ldots x_{n-1}0}_{53 \text{ bits}}11111\ldots11111}^{m \text{ bits}} 0y_1y_2y_3y_4\ldots$$

where m is the table maker's dilemma's precision number and k the floating point format exponent.
Let us assume that the previous steps of the algorithm provide a sufficient precision, so this value is stored in s^h and s^l as follows:

$$s^h = 2^k \cdot 1.x_1x_2x_3\ldots x_{n-1}100000000000$$

$$s^l = -2^{k-m+11} \cdot 1.\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4\ldots$$

where $\bar{y}_i$ stands for the negated value of y_i which is $1 - y_i$. So as one can see, the long series of ones[10] which constitute the difficulty in the table maker's dilemma's special cases are stored in s^h and s^l only implicitly. This positive effect is actually the greater the more difficult the case is.

We will prove in the following that the given algorithm calculates s^h and s^l so that this 'phenomenon' can be observed and that actually at least 157 bits of the mantissa represented in this form with a implicit storing of a long suite of ones or zeros are significant. For this, we will show that for the input values in $\left[-2^{-30}; 2^{-53}\right] \cup \left[2^{-53}; 2^{-30}\right]$ on which the table maker's dilemma occurs some parts of the algorithm only deal with exact values without any intermediate rounding.

As we have already seen in lemma 2.8, in the given particular range, we can assume that $k = 0$, so $i_1 = 0$ and $i_2 = 0$ and that $r = x$ and finally $x^l = 0$.
We know that $2^{\frac{0}{2^7}} = 1$ and $2^{\frac{0}{2^{14}}} = 1$, so the respective table entries will be $t_1^h = 1$, $t_1^l = 0$, $t_2^h = 1$ and $t_2^l = 0$. As multiplications by 1 and additions with 0 can be done without any arithmetical error, one can easily see, considering the instruction sequence given in 2.18 that $t^h = 1$ and $t^l = 0$ exactly.
So we can observe in the final reconstruction sequence given in lemma 2.19 that we get the following exact results:

$$v_1^h = 0,\ v_1^l = 0$$

$$v_2 = 0$$

[9]The other rounding modes present similar situation or are 'easier' in terms of the maximum accuracy needed.
[10]or zeros in other cases

$$v_3^h = 1,\ v_3^l = 0$$

$$v_{15} = 0$$

$$v_4^h = p^h,\ v_4^l = 0$$

$$v_5 = 0$$

$$v_6 = p^l$$

$$v_7 = p^l$$

$$v_8^h + v_8^l = 1 + p^h$$

Let us now consider more precisely the contents of registers v_8^h and v_8^l. In particular we want to prove that

$$v_8^l = 0$$

As one can easily proof, for $2^{-53} < |x| < 2^{-30}$ we have $2^{-53} < |p^h + p^l| < 2^{-29}$ and we know that we are in a case where the table maker's dilemma occurs with more than 64 bits of critical accuracy. Assuming in a first step that $p^h + p^l$ provide enough accuracy, we observe the following situation[11]; the equation is meant to be read as a mathematical one:

$$1 + p^h + p^l = 1.\underbrace{0\ldots\ldots0}_{\geq 29 \text{ bits}}\overbrace{\underbrace{1x_1x_2\ldots x_j0_{53\text{rd}}}_{\leq 24 \text{ bits}}1\ldots1_{64\text{th}}\ \underbrace{1\ldots1}_{\geq 29 \text{ bits}}}^{\text{in } p^h}\overbrace{1\ldots10y_1y_2\ldots}^{\text{in } p^l}$$

So p^h and p^l will actually contain the following values:

$$p^h = 1.x_1x_2\ldots x_j10\ldots0$$

$$p^l = -1.\bar{y_1}\bar{y_2}\ldots$$

It is imporant to see that p^h will contain at least 29 trailing zeros.
So one can observe for the intermediate result $v_8^h + v_8^l$:

$$v_8^h + v_8^l = 1 + p^h = \overbrace{1.0\ldots01x_1x_2\ldots x_j10\ldots0_{64\text{th}}}^{\text{in } v_8^h}\overbrace{0\ldots0\ldots}^{\text{in } v_8^l}$$

Thus we know that $v_8^h = 1 + p^h$ is a exact result and that $v_8^l = 0$. Therefore as additions with 0 are errorfree, we have the exact results

$$v_9 = p^l$$

[11] in round-to-nearest mode; the other modes are similar

and

$$v_{13} = p^l$$

So $v_8^h + v_{13}$ contains without any rounding error $1 + p^h + p^l$.
As the ***Fast2Sum*** sequence leading to s^h and s^l is proven to be exact, we know that

$$s^h + s^l = 1 + p^h + p^l$$

exactly.
In order to prove that $s^h + s^l$ represents an approximation to the exponential function with at least 157 bits of accuracy if $x \in \left[-2^{-30}; 2^{-30}\right]$ and if the table maker's dilemma occurs, it suffices therefore to show that $p^h + p^l$ is accurate enough. We have proven by theorem 2.9 that the arithmetical error due to the evaluation of the polynomial in the given range is $|\Delta| < 2^{-141}$ and we have shown in 2.6 that the polynomial used approximates the exponential function with a maximal error of $|\Delta| < 2^{-161}$. So we can assume that at least 140 bits[12] of $p^h + p^l$ are correct. As we know that

$$|p^h + p^l| < 2^{-29}$$

the impact of this error on the final error in $s^h + s^l$ is scaled by at least 2^{-29}. So at least $140+29 = 169$ bits of the intermediate result before rounding are exact and we can affirmate

Corollary 2.24 *The algorithm given for the evaluation of the exponential function calculates a double precision number y which is a correctly rounded approximation to e^x for all double precision input values.*

Proof This property follows trivially from the results stated above and Lefèvre's theorem announced as theorem 2.22. ∎

3 Performance and accuracy testing

The implementation of the exponential function has been checked on two lists obtained using Lefèvre's algorithm [7] containing all worst case values with a critical intermediate precision over 100 bits. The first 2270 entry list contains argument-result pairs for the round-to-nearest rounding mode, the second one contains 4074 entries for the directed rounding modes. The algorithm returned a correct result for all entries of both lists in all corresponding rounding modes. This verification is a useful check that the theoretical proof given above is correct.

In addition the implementation of the exponential function has been tested using the Intel Math Libary Test Suite (IMLTS) which is a test system which is particulary adapted to this purpose. It uses a multiprecision library implementation for checking the correctness

[12] One remarks that as result of the use of the ***Fast2Sum*** sequence in the polynomial approximation phase, a similar effect as the one described here can be observed on $p^h + p^l$. For this reason, one can assume that a 140 bit precise result can be stored in two double extended numbers.

of the results of the standard mathmatical library functions on randomly produced or hand chosen arguments as a reference.

Performance testing is very important if one wants to provide a function which is not only correctly rounding but also highly optimized in speed. The Itanium processor instruction set supports the explicit use of parallelism in programs which has the consequence that the assembly code must be carefully scheduled for maximum performance. This processus resulted in a about 100 cycles gain between the first un-scheduled version of the code and final one as it can be found in section 7. The scheduling has been highly optimized for the Itanium 2 processor and will therefore provide slightly less excellent performance on Itanium 1 systems.

Concerning the accuracy checking, which includes the check if all flags needed to IEEE [5] compliance are correctly set, the function has been tested on 130000000 randomly chosen arguments in the definition domain of the function[13] in each of the 4 rounding modes two of which are actually identical due to the fact that the exponential function is positive on all arguments. Additionnally, the function has been tested on supplementary arguments in the range $x \in \left[-2^{-30}; -2^{-53}\right] \cup \left[2^{-53}; 2^{-30}\right]$ which is known to be particulary 'difficult' as one can see in the theorem on the rounding of the exponential function given in [6, 7]. In none of this ranges, the function showed up a rounding error greater than or equal to 0.5ulp respectively 1.0ulp for the directed modes. All flags were correctly set on all arguments including the under-/ overflow cases. This result is a positive experimental verification of the mathematical prove given above.

With respect to performance, the implementation has been compared on the one hand to the implementation given in [6] which has been proven to be correctly rounding. On the other hand, a comparison to the open-source, less accurate (≤ 0.502ulp), higher optimized library Intel provides [17] for Itanium based systems has been done. The first has been compiled using Intel 's `ecl` compiler for Itanium systems using some compiler options which are needed for the correctness of this library[14] but which affect the performance of the function. The latter has simply be assembled to a functional library. Unfortunately the portable correct rounding implementation provided by David Defour could only be tested in round to nearest mode since it implements the different IEEE standard rounding modes in different functions whereas the floating point status register for rounding control is not be taken in account which makes it completely incompatible to the timing system used.

The functions have been timed for all paths of the algorithm we presented. In addition, serveral points on which the second evaluation phase [6] is launched in Defour's implementation have been timed.

[13] i.e. in the range $x \in [-800; 800]$

[14] i.e. the `-QIPF_fma-` and the `-QIPF_fltacc-` options which prevent the compiler from optimizing expressions such as `a * b + c` by means of the `fma` instruction which does no intermediate rounding.

Range	our `crexp`	Defour's `exp`	Intel 's `exp`
$x = \pm 0$	9	38	10
$\|x\| < 2^{-53}$, $x \neq 0$	17	38	42
$-700 < x < 700$, $\|x\| > 2^{-53}$	172	255	42
$710 < x < 1000$ (overflow)	72	53	255
$-745 < x < 709$ (underflow)	4578	4807	3158
$-1000 < x < -750$ (zero)	1576	1934	1814
$x = +\infty$	12	47	11
$x = -\infty$	10	47	9
$x =$ `nan`	12	46	11
x positive denormal	373	3573	545
x negative denormal	383	3600	524
Hard case 1	172	4707	42
Hard case 2	172	4721	42
Hard case 3	172	3951	42
Hard case 4	172	4218	42

The hard cases mentionned in this list are 4 randomly chosen arguments on which Defour's implementation must launch the second phase to guarantee the correct rounding. A mean value timing on these cases has not been done, also other values may be different timings.

As the scheduling has explicitly be done for the latencies of the newer Intel Itanium 2 processor, the function will show less excellent performance on Itanium 1.

All these results show that the given implementation of a correct rounding exponential is competitive in comparison to implementations that provide a correctly rounded result and can be a good alternative to less accurate one.

4 Future work

A critical view on the given algorithm lets one easily discover that the main part of the latency of the algorithm is due to the polynomial evaluation. In fact, assuming the latency of an `fma` instruction on Intel Itanium 2 to be 4 cycles, analysis of the different steps of this phase gives a latency of at least 100 cycles for the evaluation of the polynomial until p^h is available. This makes one think that the parallelism the processor is capable of is not fully exploited.

Considerating even more precisely the polynomial and the magnitudes of the different values involved, one sees that in

$$p(x) = x + \frac{1}{2}x^2 + \left(a_4^h + a_4^l\right) x^3 + x^4 \cdot (a_5 + x \cdot (a_6 + a_7 x))$$

the higher degree terms $x^4 \cdot (a_5 + x \cdot (a_6 + a_7 x))$ will only affect the lower part p^l of the final result. This part of the polynomial can therefore be evaluated in parallel to the other parts

in native precision only. In addition, the intermediate value x^4, which is needed only with an accuracy of one double extended number, can be calculated as $x^2 \cdot x^2$ using the high part of the machine representation of x^2 in two double extended numbers. Further some intermediate normalizations, i.e. the use of a ***Fast2Sum*** sequence on potentially 'overlapping' numbers z^h and z^l, which are done in the algorithm we presented above, could be saved at the cost of a even more complex accuracy proof.

Let us give an instruction sequence for the polynomial evaluation that takes advantage of this deliberations. The sequence can be scheduled for Intel Itanium 2 in such a way that the higher part p^h of the result is available after only 42 machine cycles; the cycle in which the instructions can be executed are indicated in the beginning of each line. We will consider the accuracy of this sequence below.

(1)	`fma z_1 = x, x, f0`	$z_1 \leftarrow x \otimes x$
(1)	`fma z_2 = a_4_h, x, f0`	
(2)	`fma z_5 = a_7, x, a_6`	$z_5 \leftarrow \langle a_7 \cdot x + a_6 \rangle$
(2)	`fma z_10 = a_4_l, x, f0`	$z_{10} \leftarrow a_4^l \otimes x$
(5)	`fma z_3 = z_2, z_1, f0`	
(5)	`fma z_4 = z_1, z_1, f0`	$z_4 \leftarrow z_1 \otimes z_1$
(6)	`fma z_6 = z_5, x, a_5`	$z_6 \leftarrow \langle z_5 \cdot x + a_5 \rangle$
(6)	`fms z_9 = x, x, z_1`	$z_1 + z_9 \leftarrow$ ***Fast2Mult*** (x, x)
(7)	`fma z_11 = z_10, z_1, f0`	$z_{11} \leftarrow z_{10} \otimes z_1$
(7)	`fma z_13 = h, z_1, f0`	$z_{13} \leftarrow h \otimes z_1$
(8)	`fms z_21 = a_4_h, x, z_2`	$z_2 + z_{21} \leftarrow$ ***Fast2Mult*** $\left(a_4^h, x\right)$
(9)	`fms z_8 = z_2, z_1, z_3`	$z_3 + z_8 \leftarrow$ ***Fast2Mult*** (z_1, z_2)
(10)	`fma z_7 = z_4, z_6, f0`	$z_7 \leftarrow z_4 \otimes z_6$
(10)	`fma z_12 = z_2, z_9, f0`	$z_{12} \leftarrow z_2 \otimes z_9$
(11)	`fma z_14 = h, z_9, f0`	$z_{14} \leftarrow h \otimes z_9$
(11)	`fadd z_15 = x, z_13`	
(12)	`fma z_22 = z_21, z_1, f0`	$z_{22} \leftarrow z_{21} \otimes z_1$
(13)	`fadd z_23 = z_8, z_11`	$z_{23} \leftarrow z_8 \oplus z_{11}$
(14)	`fadd z_24 = z_7, z_12`	$z_{24} \leftarrow z_7 \oplus z_{12}$
(15)	`fadd z_18 = z_15, z_3`	
(16)	`fsub z_16 = z_15, x`	
(16)	`fadd z_25 = z_14, z_22`	$z_{25} \leftarrow z_{14} \oplus z_{22}$
(18)	`fadd z_26 = z_23, z_24`	$z_{26} \leftarrow z_{23} \oplus z_{24}$
(19)	`fsub z_19 = z_18, z_15`	
(20)	`fsub z_17 = z_13, z_16`	$z_{15} + z_{17} \leftarrow$ ***Fast2Sum*** (x, z_{13})
(22)	`fadd z_27 = z_25, z_26`	$z_{27} \leftarrow z_{25} \oplus z_{26}$
(23)	`fsub z_20 = z_3, z_19`	$z_{18} + z_{20} \leftarrow$ ***Fast2Sum*** (z_{15}, z_3)
(26)	`fadd z_28 = z_17, z_27`	$z_{28} \leftarrow z_{17} \oplus z_{27}$
(30)	`fadd z_29 = z_28, z_20`	$z_{29} \leftarrow z_{28} \oplus z_{20}$
(34)	`fadd p_h = z_18, z_29`	

(38) `fsub z_30 = p_h, z_18`
(42) `fsub p_l = z_29, z_30` $\qquad p^h + p^l \leftarrow \mathit{Fast2Sum}\,(z_{18}, z_{29})$

Concerning the accuracy of this sequence, let us show now that for $|x| < \frac{\ln 2}{2^{15}}$, $h = \frac{1}{2}$ exactly, $a_4^h + a_4^l \approx \frac{1}{6}$, $a_5 \approx \frac{1}{24}$, $a_6 \approx \frac{1}{120}$ and $a_7 \approx \frac{1}{720}$, the sequence calculates a result

$$p^h + p^l = p(x) + \Delta$$

with an error Δ with $|\Delta| < 2^{-126}$ and for $|x| < 2^{-30}$ with only $|\Delta| < 2^{-155}$

Proof Similarily to the sequences of the first version of algorithm whose accuracy has been proven above, this instruction sequence makes use of the *Fast2Sum* algorithm. Let us show first that the respective input conditions are respected.
The *Fast2Sum* is used first for calculating

$$z_{15} + z_{17} = \mathit{Fast2Sum}\,(x, z_{13})$$

We have $|z_{13}| \approx \frac{1}{2} \cdot x^2$ and $|x| < 1$ so the input condition is respected.
The second use of the *Fast2Sum* sequence is made for calculating

$$z_{18} + z_{20} = \mathit{Fast2Sum}\,(z_{15}, z_3)$$

We know that

$$|z_{15}| \approx \left| x + \frac{1}{2} \cdot x^2 \right|$$

and

$$|z_3| \approx \left| \frac{1}{6} \cdot x^3 \right|$$

So the input condition is respected also here.
The third time the sequence is used is for the calculation of

$$p^h + p^l = \mathit{Fast2Sum}\,(z_{18}, z_{29})$$

One can verify that for $2{-}53 < |x| < 2^{-15}$[15] we have

$$|z_{18}| \approx \left| x + \frac{1}{2} \cdot x^2 + \frac{1}{6} \cdot x^3 \right| > 2^{-52}$$

and

$$|z_{29}| < \left| 2^{-61} \cdot \left(\frac{1}{12} + \frac{61}{60} \cdot x + \frac{181}{180} \cdot x^2 + \frac{1}{6} \cdot x^3 \right) \right| < 2^{-64}$$

[15] The lower bound is a result of the fact that for $|x| < 2^{-53}$ we can simply return $\langle 1 + x \rangle_{53}$ as a correctly rounded double precision approximation to the exponential function; see [23].

So $|z_{18}| > |z_{29}|$ and thus, the input condition is respected.
Hence we get the exact results

$$z_1 + z_9 = x^2$$
$$z_{13} = \frac{1}{2} \cdot z_1$$
$$z_{14} = \frac{1}{2} \cdot z_9$$
$$z_2 + z_{21} = a_4^h \cdot x$$
$$z_3 + z_8 = z_2 \cdot z_1$$
$$z_{15} + z_{17} = x + z_{13}$$
$$z_{18} + z_{20} = z_{15} + z_3$$
$$p^h + p^l = z_{18} + z_{29}$$

So if one approximates the error for each of the instructions left in a similar way as shown above, one gets

$$p^h + p^l = x + h \cdot x^2 + x^3 \cdot \left(a_4^h + a_4^l\right) + a_5 \cdot x^4 + a_6 \cdot x^5 + a_7 \cdot x^6 + \Delta$$

with

$$\begin{aligned}
\Delta &= z_1 z_{21} \epsilon_{12} + z_1 z_{10} \epsilon_{13} + z_1 a_4^l x \epsilon_{14} + z_2 z_9 \epsilon_{15} + a_4^h x z_9 \epsilon_{16} - a_4^l x z_9 + \\
&+ a_7 x^6 \epsilon_{12} + a_6 x^5 \epsilon_{12} + z_5 x^5 \epsilon_{11} + a_5 x^4 \epsilon_{11} + 2 x^4 z_6 \epsilon_{10} + x^4 z_6 \epsilon_{10}^2 + \\
&+ z_1 z_1 z_6 \epsilon_9 + z_4 z_6 \epsilon_8 + z_7 \epsilon_7 + z_{12} \epsilon_7 + z_8 \epsilon_6 + z_{11} \epsilon_6 + z_{23} \epsilon_5 + z_{24} \epsilon_5 + \\
&+ z_{14} \epsilon_4 + z_{22} \epsilon_4 + z_{25} \epsilon_3 + z_{26} \epsilon_3 + z_{17} \epsilon_2 + z_{27} \epsilon_2 + z_{28} \epsilon_1 + z_{20} \epsilon_1
\end{aligned}$$

with $|\epsilon_i| \leq 2^{-64}$.
For $|x| < 2^{-15}$, $\left|x^2\right| < 2^{-30}$, $\left|x^3\right| < 2^{-45}$, $\left|x^4\right| < 2^{-60}$, $\left|x^5\right| < 2^{-75}$ and $\left|x^6\right| < 2^{-90}$ we get the results in the left column of the following table, for $|x| < 2^{-30}$, $\left|x^2\right| < 2^{-60}$, $\left|x^3\right| < 2^{-90}$, $\left|x^4\right| < 2^{-120}$, $\left|x^5\right| < 2^{-150}$ and $\left|x^6\right| < 2^{-180}$ the results given in the right column below:

$$|z_1 z_{21} \epsilon_{12}| < 2^{-175}$$
$$|z_1 z_{10} \epsilon_{13}| < 2^{-175}$$
$$\left|z_1 a_4^l x \epsilon_{14}\right| < 2^{-175}$$
$$|z_2 z_9 \epsilon_{15}| < 2^{-175}$$
$$\left|a_4^h x z_9 \epsilon_{16}\right| < 2^{-175}$$
$$\left|a_4^l x z_9\right| < 2^{-175}$$
$$\left|a_7 x^6 \epsilon_{12}\right| < 2^{-163}$$
$$\left|a_6 x^5 \epsilon_{12}\right| < 2^{-145}$$
$$\left|z_5 x^5 \epsilon_{11}\right| < 2^{-145}$$
$$\left|a_5 x^4 \epsilon_{11}\right| < 2^{-128}$$
$$\left|x^4 z_6 \epsilon_{10}\right| < 2^{-127}$$
$$\left|x^4 z_6 \epsilon_{10}^2\right| < 2^{-192}$$
$$|z_1 z_1 z_6 \epsilon_9| < 2^{-128}$$
$$|z_4 z_6 \epsilon_8| < 2^{-128}$$
$$|z_7 \epsilon_7| < 2^{-127}$$
$$|z_{12} \epsilon_7| < 2^{-175}$$
$$|z_8 \epsilon_6 + z_{11}| < 2^{-175}$$
$$|z_{11} \epsilon_6| < 2^{-175}$$
$$|z_{23} \epsilon_5| < 2^{-174}$$
$$|z_{24} \epsilon_5| < 2^{-126}$$
$$|z_{14} \epsilon_4| < 2^{-159}$$
$$|z_{22} \epsilon_4| < 2^{-175}$$
$$|z_{25} \epsilon_3| < 2^{-158}$$
$$|z_{26} \epsilon_3| < 2^{-125}$$
$$|z_{17} \epsilon_2| < 2^{-143}$$
$$|z_{27} \epsilon_2| < 2^{-124}$$
$$|z_{28} \epsilon_1| < 2^{-142}$$
$$|z_{20} \epsilon_1| < 2^{-141}$$

and therefore

$$|\Delta| < 2^{-122}$$

$$|z_1 z_{21} \epsilon_{12}| < 2^{-220}$$
$$|z_1 z_{10} \epsilon_{13}| < 2^{-220}$$
$$\left|z_1 a_4^l x \epsilon_{14}\right| < 2^{-220}$$
$$|z_2 z_9 \epsilon_{15}| < 2^{-220}$$
$$\left|a_4^h x z_9 \epsilon_{16}\right| < 2^{-220}$$
$$\left|a_4^l x z_9\right| < 2^{-220}$$
$$\left|a_7 x^6 \epsilon_{12}\right| < 2^{-253}$$
$$\left|a_6 x^5 \epsilon_{12}\right| < 2^{-210}$$
$$\left|z_5 x^5 \epsilon_{11}\right| < 2^{-210}$$
$$\left|a_5 x^4 \epsilon_{11}\right| < 2^{-188}$$
$$\left|x^4 z_6 \epsilon_{10}\right| < 2^{-187}$$
$$\left|x^4 z_6 \epsilon_{10}^2\right| < 2^{-252}$$
$$|z_1 z_1 z_6 \epsilon_9| < 2^{-188}$$
$$|z_4 z_6 \epsilon_8| < 2^{-188}$$
$$|z_7 \epsilon_7| < 2^{-187}$$
$$|z_{12} \epsilon_7| < 2^{-220}$$
$$|z_8 \epsilon_6 + z_{11}| < 2^{-220}$$
$$|z_{11} \epsilon_6| < 2^{-220}$$
$$|z_{23} \epsilon_5| < 2^{-219}$$
$$|z_{24} \epsilon_5| < 2^{-188}$$
$$|z_{14} \epsilon_4| < 2^{-189}$$
$$|z_{22} \epsilon_4| < 2^{-220}$$
$$|z_{25} \epsilon_3| < 2^{-188}$$
$$|z_{26} \epsilon_3| < 2^{-185}$$
$$|z_{17} \epsilon_2| < 2^{-158}$$
$$|z_{27} \epsilon_2| < 2^{-184}$$
$$|z_{28} \epsilon_1| < 2^{-157}$$
$$|z_{20} \epsilon_1| < 2^{-156}$$

and thus

$$|\Delta| < 2^{-155}$$

This are the properties to be proven. ∎

Further, on inspection of the reconstruction sequence, one notices also that the parallelism the processor is capable of is not fully exploited. In addition, as the particularities of this part of the algorithm for the cases where more than 128 bits are needed to decide the rounding are relatively well known, one can set the use of some supplementary normalization ***Fast2Sum*** sequences aside. This deliberations have led to a new reconstruction part by which the over all latency of the algorithm has been reduced to 92 cycles:

(1)	`fma u_1 = t_h, p_h, f0`	
(1)	`fma u_2 = x_l, t_h, t_l`	$u_2 \leftarrow \langle x^l \cdot t^h + t^l \rangle$
(2)	`fma u_5 = t_l, p_h, f0`	$u_5 \leftarrow t^l \otimes p^h$
(5)	`fms u_3 = t_h, p_h, u_1`	$u_1 + u_3 \leftarrow$ ***Fast2Mult*** (t^h, p^h)
(5)	`fma u_4 = t_h, p_l, u_2`	$u_4 \leftarrow \langle t^h \cdot p^l + u_2 \rangle$
(6)	`fadd s_h = t_h, u_1`	
(6)	`fma u_6 = u_1, x_l, u_5`	$u_6 \leftarrow \langle u_1 \cdot x^l + u_5 \rangle$
(9)	`fadd u_10 = u_3, u_4`	$u_{10} \leftarrow u_3 \oplus u_4$
(10)	`fsub u_8 = s_h, t_h`	
(13)	`fadd u_11 = u_10, u_6`	$u_{11} \leftarrow u_{10} \oplus u_6$
(14)	`fsub u_9 = u_1, u_8`	$s^h + u_9 \leftarrow$ ***Fast2Sum*** (t^h, u_1)
(18)	`fadd s_l = u_11, u_9`	$s^l \leftarrow u_{11} \oplus u_9$

One can easily verify that this sequence has the same behaviour as the reconstruction algorithm we showed above on the cases where the argument x of the exponential function is $|x| < 2^{-30}$. This property guarantees that the parts of the correction proof concerning the very hard cases of the table maker's dilemma are still verified.
Concerning the accuracy of this phase for other arguments in other ranges, we will show in the following that the sequence calculates

$$s^h + s^l = \left(t^h + t^l\right) \cdot \left(1 + \left(p^h + p^l\right)\right) \cdot \left(1 + x^l\right) + \Delta$$

with $|\Delta| < 2^{-122}$ if t^h, t^l, p^h, p^l and x^l are in the ranges given in theorem 2.19 on page 30.

Proof Using a similiar approach as for the sections above, one obtains that Δ can be expressed as

$$\begin{aligned} \Delta &= -t^l p^l - t^l x^l - t^h p^l x^l - t^l p^h x^l - t^l p^l x^l + t^l p^h \epsilon_8 + t^h p^h x^l \epsilon_7 + u_1 x^l \epsilon_6 + \\ &+ x^l t^h \epsilon_5 + t^l \epsilon_5 + t^h p^l \epsilon_4 + u_2 \epsilon_4 + u_3 \epsilon_3 + u_4 \epsilon_3 + u_{10} \epsilon_2 + u_6 \epsilon_2 + u_{11} \epsilon_1 + u_9 \epsilon_1 \end{aligned}$$

with $\epsilon_i \leq 2^{-64}$.
By locating the absolute maximum of each of the monoms of this expression, we get

$$\left|t^l p^l\right| < 2^{-141}$$

$$\left|t^l x^l\right| < 2^{-142}$$
$$\left|t^h p^l x^l\right| < 2^{-156}$$
$$\left|t^l p^h x^l\right| < 2^{-156}$$
$$\left|t^l p^l x^l\right| < 2^{-220}$$
$$\left|t^l p^h \epsilon_8\right| < 2^{-141}$$
$$\left|t^h p^h x^l \epsilon_7\right| < 2^{-156}$$
$$\left|u_1 x^l \epsilon_6\right| < 2^{-156}$$
$$\left|x^l t^h \epsilon_5\right| < 2^{-142}$$
$$\left|t^l \epsilon_5\right| < 2^{-127}$$
$$\left|t^h p^l \epsilon_4\right| < 2^{-141}$$
$$|u_2 \epsilon_4| < 2^{-127}$$
$$|u_3 \epsilon_3| < 2^{-141}$$
$$|u_4 \epsilon_3| < 2^{-126}$$
$$|u_{10} \epsilon_2| < 2^{-125}$$
$$|u_6 \epsilon_2| < 2^{-140}$$
$$|u_{11} \epsilon_1| < 2^{-124}$$
$$|u_9 \epsilon_1| < 2^{-127}$$

and therefore we have

$$|\Delta| < 2^{-122}$$

∎

These results show that this 92 cycle version of the algorithm can also be considered as correctly rounding.

It has been tested on the IMLTS system and on a potentially exhaustive list of 6344 arguments to the exponential function in double precision which the table maker's dilemma occurs with a need to know at least 100 correct bits to ensure the correct rounding. This list has been obtained using the algorithm Lefèvre has presented in [7]. No rounding error could be found neither with IMLTS neither on the list of arguments.

A more complete proof, written perhaps using a theorem proving tools such as Coq [24], could strengthen our confidence in this result. This is left to further work.

One should remark that the task of finding an optimally scheduable sequence having a appropriate accuracy for the calculation of a polynomial is very difficult for the following reasons: as Harrison remarks in [20] there are non-trivial factorizations of a polynomial that

make the search space already very large. In addition, since scheduling instructions with flow dependency constraints is $\mathcal{NP}$-complete [22], the task becomes already unfeasible even if one could argue that the length of usual polynomials and thus the maximum number of instructions needed is small enough to be exhaustively searched. But if one considers that for each appropriate sequence found, a formal accuracy proof must be written, a solution will almost always be suboptimal. In [20], accuracy issues are not considered in detail as all calculations are supposed to be ported out in native precision.

Despite, as one has been able to see throughout this article, the representation of higher precision intermediate results can be very effective. Further studies of the scheduling problem should perhaps be done taking into account such a representation.

5 Conclusion

To our opinion, several points seem to be a result of this work. Let us consider some of them:

- In contrast to others [6], we recommend the correctly rounded evaluation of the exponential function in only phase when special hardware features are available, which is nowadays commonly the case. The presented algorithm outperforms the so called 'quick' phase of the two-phase algorithm given in [6]. On some arguments, the performance gain can reach up to a 100 times speed-up and more. The overhead induced by the 'artificial' high precision arithmetic used in this competitor's algorithm and the strict limitation to double precision arithmetic even when more precision is available could be two of the reasons for this behaviour. We conceed to the authors of this algorithm that it is theoretically fully portable which our is not. Since the authors remark [6] that their function is outperformed only slightly by other libraries such as the one provided by IBM [18], one can consider the algorithm we presented to be a good concurrent to such an implementation that uses in fact Ziv's multilevel strategy which has very great impacts on the maximum timings [6, 2, 18].
- As a consequency of the single-level evaluation without any need to call other helper functions out of high-precision libraries, the implementation has not to deal with the eventual overhead of this calls. The code has tendancy to be more local which can be an advantage concerning the caching behaviour of modern processors.
- With a main path latency of 172 cycles on the Itanium 2 processor, the algorithm is only approximately 4 times slower than less accurate algorithms which are highly tuned for performance, such as the algorithm proposed by the Intel Numerics group in [17]. The implementation given in this paper could therefore be the basis of a library of correctly rounded functions whose normalization and common use would permit the formal proof of higher level algorithms [8]. Since more and more floating point units implementing a fused-multiply-add make their apparition [6] in modern CPUs further optimisation could permit to use correctly rounded algorithms in every-day-applications creating this way a new standard.

- The maximum latency of our algorithm can very easily be bounded and does not depend on the value of the input argument[16]. Other implemenations such as [18, 19] cannot provide bounds on the maximum latency of their functions independently of the input arguments, others like [6] can only give maximum bounds which are approx. 20 times higher as the mean value. For our implementation, the maximum value equals the mean time latency.

- Our function uses completely the same main path for all rounding modes which is read in the user floating point register as defined in the IEEE 754 standard [5]. Other implementations such as [19, 6] only provide different functions for the different rounding modes which must be run in addition in round-to-nearest mode in order produce correct results. This behaviour of our function makes it fully IEEE 754 compliant and faciliates the application programmer's work.

- As the reader has seen, the task of proving the accuracy of an algorithm such as the one we give is horribly complex and time consuming. Unfortunately, no human being can work completely errorfree. Though some experience is needed to write such an accuracy proof, its final realization is a extremly complex but relatively mechanically task. A typical implementation of a transdecendental function even does not make use of high order control sequences such as `while` loops etc, which could be a possible problem in the decidability of a property of an algorithm. Before concentrating on establishing hand-written proofs for implementations of every transcedental function in common use, more research should perhaps be done on some tool offering the possibility to estimate quickly the accuracy of an algorithm. This approach would provide us also the possiblity to 'play' with an implementation in order to optimize it for performance which can only difficultly be done when one has to write and rewrite the error proof each time one changes 2 or 3 instructions.

6 Acknowledgements

This work has been achieved as a last year student research projet within Intel Nizhny Novgorod Lab Numerics Team, who sponsored the project. I would like to thank all the people that helped me in setting up the given algorithm. Among these persons I want to thank especially

- **Andrey Naraikin**, who made my research stay at Intel possible, who supported me a lot with very useful advice and who accepted to read this article with all the experience he has in the domain of the calculations of transcendental functions,

- **David Defour**, who provided me a preliminary copy of his PhD thesis and who let me benefit of a fructual discussion on some issues with the given algorithm.

[16]The cases where the call of a under-/ overflow trap must be taken, which increases slightly the latency, can be defined clearly.

- **Florent de Dinechin**, who helped me a lot in finding the opportunity to stay at Intel's, who led and supported me on the way to the final results presented here and whose advice has always been precious. Thanks to him also for the final layout of this report.

And of course, thanks to the students of the Intel Summer School in Nizhniy Novgorod with whom it was a pleasure to share my leasure time.

7 Appendix: Complete listing of the algorithm

```
.data
.align 16
// First table entries for the constants needed.
exp_constants:
data8  0xb17217f7d1cf79ac , 0x00003ff0  // l_h
data8  0xd871319ff0342543 , 0x0000bfae  // l_l
data8  0xb60b60b649ed8281 , 0x00003ff5  // a_7
data8  0x8888888893775ffe , 0x00003ff8  // a_6
data8  0xaaaaaaaaaaaaaaab , 0x00003ffa  // a_5
data8  0xaaaaaaaaaaaaaaab , 0x00003ffc  // a_4_h
data8  0xabfad83a09aabd5c , 0x0000bfbb  // a_4_l
data8  0x40862e42fefa39f0               // smallest dbl overflow
data8  0xc0874910d52d3052               // largest dbl RTN for 0 res
data8  0x40862e42fefa39ef               // largest dbl for normal dbl res
data8  0xc086232bdd7abcd2               // smallest dbl for normal dbl res

exp_table_1:  // values exp(j*ln(2)/2^14) j=0..127
<<< Here is normally the first table >>>

exp_table_2:  // values exp(j*ln(2)/2^7) j=0..127
<<< Here is normally the second table >>>

.section .text
.global crexp#
.proc  crexp#
.align 32
.explicit
crexp::
{ .mlx
  addl      rADD         = @ltoff(exp_constants), gp       // Load the constants' table's addresses's address
  movl      rInv_ln2     = 0xb8aa3b295c17f0bc              // load significant of inv_ln2
}
{ .mlx
  addl      rExp_2_M_56  = 0xffff-49, r0                   // Form exponent for rescaling fk
  movl      rRSHF_2_56   = 0x46f8000000000000              // Form shift constant
} ;;
{ .mlx
  setf.d    fRSHF_2_56   = rRSHF_2_56                      // Form shift constant
  movl      rRSHF        = 0x43e8000000000000              // Form shift constant
}
{ .mfi
  setf.sig  fInv_ln2     = rInv_ln2                        // form inv_ln2 = 1/ln(2) * 2^63
  fnorm.s1  fx           = f8                              // Normalize the input value to register format
  addl      rExpMinArg   = 0xffff-54, r0                   // Form exponent for no calc testf
}
;;
{ .mfi
  setf.exp  f2_M_56      = rExp_2_M_56                     // Form rescale constant
  fclass.m  p9, p0       = f8, 0x07                        // Test if x == 0
  nop.i     0
}
{ .mfi
  ld8       rADD         = [rADD]                          // Get the address of the constants' table
}
;;
{ .mfi
  setf.exp  fMin_Calc_Arg = rExpMinArg                     // Form max not 1+x test constant
  fclass.m  p8, p0        = f8, 0x22                       // Test if x == -inf
}
{ .mfi
  setf.d    fRSHF         = rRSHF                          // Form shift constant
}
;;
{ .mfi
  add       rADD_tmp      = 16, rADD                       // Trick to overcome the 5 cycle latency of the offset-add of the load
  fclass.m  p10, p0       = f8, 0x1e1                      // Test if x == +inf, nan, NaT
  adds      rADD_tmp2     = 64, rADD                       // Trick
}
{ .mfb
  ldfe      fl_h          = [rADD], 32                     // Load high part of ln(2)/2^14
(p9) fma.d.s0 f8          = f1, f1, f0                     // Fast exit for 0
(p9) br.ret.spnt b0
} ;;
{ .mfi
  ldfe      fl_l          = [rADD_tmp], 32                 // Load low part of ln(2)/2^14
  fabs      fAbsX         = fx                             // Calculate absolute value of x for test |x| < 2^-54
}
{ .mfb
  nop.m     0
(p8)  fma.d.s0 f8         = f0, f0, f0                     // Fast exit for -inf
(p8)  br.ret.spnt b0
} ;;
```

```
// We are now in the range reduction phase
// We calculate
// w = x * inv_ln2
// k = int(w)
// x = k * ln2_2_14 + x + x_l + \tau
// The error \tau is less than 2^-118
// In the same time, we load the polynomial,
// calculate the integer version of k and
// start the evaluation of the polynomial.
{ .mfi
  nop.m     0
  fma.s1    fq_1          = fx, fInv_ln2, fRSHF_2_56       // Inv_ln2 is scaled by 2^63, shift by adding 1.1 * 2^(63 + (63 - 14))
  addl      rADD_2        = @ltoff(exp_table_2), gp        // Get 2nd table entries' address's address
}
{ .mfb
  nop.m     0
(p10) fma.d.s0 f8         = f8, f8, f0                     // Fast exit for +inf, nan, NaT
(p10) br.ret.spnt b0                                       // Fast exit for +inf, nan, NaT
}
{ .mfi
  nop.m     0
  fcmp.le.s1 p11, p0      = fAbsX, fMin_Calc_Arg           // Test if |x| < 2^-53
}
;;
{ .mfi
  fms.s1    fk            = fq_1, f2_M_56, fRSHF           // Form float k by rescaling and substracting the shift constant
  nop.i     0
}
{ .mfi
  ld8       rADD_2        = [rADD_2]                       // Get the address of the 2nd table's entries
  nop.i     0
} ;;
{ .mfb
  getf.sig  rk            = fq_1                           // Form int version of k, Attention: we have still two leading ones too much
(p11) fma.d.s0 f8         = f1, fx, f1                     // Fast exit for |x| < 2^-54
(p11) br.ret.spnt b0                                       // Fast exit
}
;;
{ .mfi
  ldfe      fa_7          = [rADD], 48                    // Load polynomial coefficient a_7
  fnma.s1   fr_h          = fk, fl_h, fx                  // Calculate high part of preliminary reduced arge
  nop.i     0
}
{ .mfi
  ldfe      fa_6           = [rADD_tmp], 32               // Load polynomial coefficient a_6
  fnma.s1   fr_l           = fk, fl_l, f0                 // Calculate low part of preliminary reduced arg
  nop.i     0
}
{ .mfi
  mov       rExp_h         = 0xffff-1                     // Form exp for h = 1/2
  nop.f     0
  nop.i     0
}
;;
{ .mfi
  ldfe      fa_5           = [rADD_tmp2]                  // Load polynomial coefficient a_5;;
  fadd.s1   fr             = fr_h, fr_l                   // Normalize the reduced arg (r + x_l) = (r_l + r_l) with |x_l| < 2^-64*|r_l|
  nop.i     0
}
;;
// The exponential of the reduced argument is approximated by a 6 degree polynomial, as follows:
//
// p_h + p_l = (r + ((1/2*(r*r)) + (((r*r)*r)*((a_4_h + a_4_l) + ((a_5 + (a_6 + a_7*r)*r)*r))))) + \Delta, |\Delta| < 2^-124
//
// The polynomial is chosen so that it approximates exp(x) in |x|<ln(2)/2^15 with a maximal error of 2^-123
//
// The exponential exp(x_l) of the lower part of the reduced argument, x_l, is approximated by (1 + x_l).
// This is sufficient, since |x_l|<2^-79 and so x_l^2 < 2^-158 and the Taylor series of exp(x) converges very fast.
// Start of 128 bit poly. approx. phase
// We load at the same time the address of the second table,
// generate the indices and calculate the value
// M for the final 2^(bias + M) constant
// We prepare the calculation of the lower part the reduced argument.
// At the same time, we load the address of the address of the second table and branch perhaps to the certain under-/ overflow cases
{ .mfi
  fma.s1    fz_1           = fa_7, fr, fa_6               // Start evaluation of the polynomial
  nop.i     0
}
;;
{ .mfi
  ldfe      fa_4_h         = [rADD], 16                   // Load the high part of the polynomial coefficient a_4_h
  shr       rk_128         = rk, 7                        //  Get bits 8 to 14 for the 2nd index
};;
{ .mfi
```

```
  setf.exp  fh            = rExp_h                    // Form constant h = 1/2
  fma.s1    fz_2          = fz_1, fr, fa_5            // Polynomial
  and       rInd_1        = 0x7f, rk                  // Mask the other parts of the integer
}
{ .mfi
  nop.m     0
  fma.s1    fz_11_h       = fr, fr, f0                // z_11_h + z_11_l = r^2 exactely by means of fma;;
  shl       rM            = rk, 2                     // Get rid of the two leading ones (1.1)
} ;;
{ .mfi
  nop.m     0
  fma.s1    fz_3_h        = fz_2, fr, f0              // z_3_h + z_3_l = (a_5 + (a_6 + (a_7*r)*r)*r + \Delta_1, |\Delta_1| < 2^-82
  and       rInd_2        = 0x7f, rk_128              // Mask the other parts of the integer
}
{ .mfi
  nop.m     0
  fma.s1    fz_12         = fz_11_h, fr, f0
  nop.i     0
}
{ .mfi
  nop.m     0
  fms.s1    fz_11_l       = fr, fr, fz_11_h           // Low part of z_11_h/l
  shr       rM            = rM, 16                    // Get bits 15, 16, ... for M, we had shifted to the left by 2 thus 16
}
{ .mfi
  nop.m     0
  fma.s1    fz_23         = fz_11_h, fh, f0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fz_4          = fa_4_h, fz_3_h
  shl       rInd_1        = rInd_1, 5                 // Multiply index by 32 (32 bytes per index)
}
{ .mfi
  nop.m     0
  fms.s1    fz_3_l        = fz_2, fr, fz_3_h          // Low part of z_3_h/l
  shl       rInd_2        = rInd_2, 5                 // Multiply by 32
}
{ .mfi
  nop.m     0
  fms.s1    fz_13         = fz_11_h, fr, fz_12
  nop.i     0
}
{ .mfi
  nop.m     0
  famin.s1  fq_5_0        = fr_h, fr_l                // Get min of r_h and r_l (in magnitude)
  nop.i     0
}
{ .mfi
  nop.m     0
  fms.s1    fz_24         = fz_11_h, fh, fz_23
}
;;
{ .mfi
  ldfe      fa_4_l        = [rADD], 16                // Load the low part of the polynomial coefficient a_4_l
  fsub.s1   fz_5          = fz_4, fa_4_h
  nop.i     0
}
{ .mfi
  nop.m     0
  famax.s1  fq_5_1        = fr_l, fr_h                // Get max of r_l and r_h (in magnitude) Do not change the order of r_h and r_l
  add       rTbl_2        = rInd_2, rADD_2            // Index the 2nd table
}
;;
{ .mfi
  nop.m     0
  fsub.s1   fz_6          = fz_3_h, fz_5
  nop.i     0
}
{ .mfi
  mov       rExp_bias     = 0xffff                    // Load bias for 2^M float generation
  fma.s1    fz_14         = fz_11_l, fr, fz_13
  nop.i     0
}
{ .mfi
  ldfe      ft_2_h        = [rTbl_2], 16              // Load high part of 2nd table entry, add 16 to addr
  fma.s1    fz_25         = fz_11_l, fh, fz_24
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fq_5_2        = fq_5_1, fr                // Prepare final calculation of the reduced argument
  nop.i     0
}
```

```
;;
{ .mmf
  ldfpd     fMin_dbl_oflow_arg, fMax_dbl_zero_arg = [rADD], 16 // Load overflow, underflow, denormal etc. test constants
  fadd.s1   fz_7            = fz_3_l, fz_6
} ;;
{ .mfi
  add       rM_bias         = rM, rExp_bias                  // Form biased M
  fadd.s1   fz_8            = fa_4_l, fz_7
  nop.i     0
}
{ .mfi
  ldfe      ft_2_l          = [rTbl_2]                       // Load low part of 2nd table entry
  fadd.s1   fz_15_h         = fz_12, fz_14                   // z_15_h + z_15_l = (z_11_h + z_11_l) * r + \Delta_5, |\Delta_5| < 2^-173
  nop.i     0
}
{ .mfi
  nop.m     0
  fadd.s1   fz_26_h         = fz_23, fz_25                   // z_26_h + z_26_l = h * (z_11_h + z_11_l) + \Delta_6, |\Delta_6| < 2^-159
}
{ .mfi
  nop.m     0
  fadd.s1   fx_l            = fq_5_2, fq_5_0                 // Calculate the lower part of the reduced argument
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fcmp.ge.s1 p12, p0        = fx, fMin_dbl_oflow_arg         // Test if we have a certain overflow cas
  nop.i     0
}
{ .mfi
  ldfpd     fMax_dbl_norm_arg, fMin_dbl_norm_arg = [rADD], 16 // Load constants, after this load, rADD points to the first table
  fadd.s1   fz_9_h          = fz_4, fz_8                     // z_9_h + z_9_l = ((a_4_h + a_4_l) + (z_3_h + z_3_l)) + \Delta_2, |\Delta_2| < 2^-130
  nop.i     0
}
{ .mfi
  nop.m     0
  fcmp.le.s1 p13, p0        = fx, fMax_dbl_zero_arg          // Test if we have a certain underflow case
  nop.i     0
}
;;
{ .mfb
  setf.exp  f2M             = rM_bias                        // Form 2^M final multiplication value
  fma.s1    fz_17           = fz_9_h, fz_15_h, f0
}
{ .mfb
  nop.m     0
  fsub.s1   fz_10           = fz_9_h, fz_4
(p12) br.cond.spnt EXP_CERTAIN_OVERFLOW
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fz_16           = fz_15_h, fz_12
  nop.i     0
}
{ .mfb
  nop.m     0
  fsub.s1   fz_27           = fz_26_h, fz_23
(p13) br.cond.spnt EXP_CERTAIN_UNDERFLOW
} ;;
{ .mfi
  nop.m     0
  fms.s1    fz_18           = fz_9_h, fz_15_h, fz_17
  add       rTbl_1          = rInd_1, rADD                   // Index the first table
}
{ .mfi
  nop.m     0
  fsub.s1   fz_9_l          = fz_8, fz_10
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_15_l         = fz_14, fz_16                   // Low part of z_15_h/l
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_26_l         = fz_25, fz_27                   // Low part of z_26_h/l
  nop.i     0
};;
{ .mfi
  nop.m     0
  fma.s1    fz_19           = fz_9_l, fz_15_h, fz_18
  nop.i     0
}
```

```
{ .mfi
  nop.m     0
  fcmp.gt.s1 p14, p0       = fx, fMax_dbl_norm_arg          // Test for possible overflow (it might be but it needs not)
  nop.i     0
} ;;
{ .mfi
  ldfe      ft_1_h         = [rTbl_1], 16                   // Load high part of the first table entry, add 16 for the next value
  fma.s1    fz_20          = fz_9_h, fz_15_l, fz_19
  nop.i     0
}
{ .mfi
  nop.m     0
  fcmp.lt.s1 p15, p0       = fx, fMin_dbl_norm_arg          // Test for possible underflow (it might be but it needs not)
  nop.i     0
};;
{ .mfi
  nop.m     0
  fadd.s1   fz_21_h        = fz_17, fz_20
  nop.i     0
} ;;
{ .mfi
  ldfe      ft_1_l         = [rTbl_1]                       // Load low part of the first table entry
  fadd.s1   fz_28          = fz_26_h, fz_21_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_22          = fz_21_h, fz_17
  nop.i     0
}
{ .mfi
  nop.m     0
  fma.s1    fu_1           = ft_1_h, ft_2_h, f0
  nop.i     0
}
;;
{ .mfi
  nop.m     0
  fsub.s1   fz_29          = fz_28, fz_26_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_21_l        = fz_20, fz_22                   // Low part of z_21_h/l
  nop.i     0
}
{ .mfi
  nop.m     0
  fms.s1    fu_2           = ft_1_h, ft_2_h, fu_1
  nop.i     0
}
;;
{ .mfi
  nop.m     0
  fsub.s1   fz_30          = fz_21_h, fz_29
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fz_31          = fz_21_l, fz_30
  nop.i     0
}
{ .mfi
  nop.m     0
  fma.s1    fu_3           = ft_1_h, ft_2_l, fu_2
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fz_32          = fz_26_l, fz_31
  nop.i     0
}
{ .mfi
  nop.m     0
  fma.s1    fu_4           = ft_1_l, ft_2_h, fu_3
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fz_33_h        = fz_28, fz_32                   // z_33_h + z_33_l = (z_26_h + z_26_l) + (z_21_h + z_21_l) + \Delta_7, |\Delta_7| < 2^-158
}
// Start of the table value multiplication phase and end of polynomial approx. phase
{ .mfi
  nop.m     0
```

```
  fadd.s1   ft_h           = fu_1, fu_4                    // t_h + t_l = 2^(ind_1/2^7) * 2^(ind_2/2^14) + \Delta, |\Delta| < 2^-123
  nop.i     0
}
;;
{ .mfi
  nop.m     0
  fadd.s1   fz_35          = fz_33_h, fr
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_34          = fz_33_h, fz_28
  nop.i     0
}
{ .mfi
  nop.m     0
  fma.s1    fv_1_h         = fx_1, ft_h, f0
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fu_5           = ft_h, fu_1
  nop.i     0
}
;;
// Start of the reconstruction phase
// We calculate:
// s_h + s_l = (t_h + t_l) * (1 + (p_h + p_l)) * (1 + x_l) + \Delta, |\Delta| < 2^-124
// Therefore, we have multiplied by hand the equation above and evaluate only the necessary terms.
// At the very end, the result is normalized, so that |s_l| < 2^-64*|s_h|.
// The different intermediate values have no simply understandable meaning :-( => see proof
// At the same time, some other values are still calculated
// the two tests for possible under-/ overflow are done.
{ .mfi
  nop.m     0
  fsub.s1   fz_36          = fz_35, fr
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fz_33_l        = fz_32, fz_34
  nop.i     0
}
{ .mfi
  nop.m     0
  fadd.s1   fv_3_h         = ft_h, fv_1_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fms.s1    fv_1_l         = fx_1, ft_h, fv_1_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   ft_l           = fu_4, fu_5                    // Lower part of reconstructed table lookup value t_h/l
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fz_37          = fz_33_h, fz_36
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fv_10          = fv_3_h, ft_h
  nop.i     0
};;
{ .mfi
  nop.m     0
  fadd.s1   fz_38          = fz_33_l, fz_37
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fp_h           = fz_35, fz_38                  // p_h + p_l = (z_33_h + z_33_l) + x + \Delta_8, |\Delta_8| < 2^-142
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fma.s1    fv_4_h         = fp_h, ft_h, f0
  nop.i     0
}
{ .mfi
```

```
  nop.m     0
  fsub.s1   fz_39          = fp_h, fz_35
  nop.i     0
};;
{ .mfi
  nop.m     0
  fadd.s1   fv_8_h         = fv_3_h, fv_4_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fms.s1    fv_4_l         = fp_h, ft_h, fv_4_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fp_l           = fz_38, fz_39                  // Low part of the polynomial approx. value p_h/l
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fv_11          = fv_8_h, fv_3_h
  nop.i     0
}
{ .mfi
  nop.m     0
  fma.s1    fv_5           = fx_l, fv_4_h, fv_4_l
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fma.s1    fv_6           = fp_l, ft_h, fv_5
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fv_8_l         = fv_4_h, fv_11
  nop.i     0
}
{ .mfi
  nop.m     0
  fsub.s1   fv_3_l         = fv_1_h, fv_10
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fma.s1    fv_7           = ft_l, fp_h, fv_6
  nop.i     0
}
{ .mfi
  nop.m 0
  fadd.s1   fv_2           = ft_l, fv_1_l
  nop.i 0
}
;;
{ .mfi
  nop.m     0
  fadd.s1   fv_9           = fv_8_l, fv_7
  nop.i     0
}
{ .mfi
  nop.m     0
  fadd.s1   fv_15          = fv_3_l, fv_2
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fv_13          = fv_9, fv_15
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fadd.s1   fs_h           = fv_8_h, fv_13                 // s_h + s_l = (t_h + t_l) * (1 + (p_h + p_l)) * (1 + x_l) + \Delta, |\Delta| < 2^-124
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fv_14          = fs_h, fv_8_h
  nop.i     0
} ;;
{ .mfi
  nop.m     0
  fsub.s1   fs_l           = fv_13, fv_14                  // Lower part of s_h/l
  nop.i     0
```

```
} ;;
// End of the reconstruction phase, start of the multiplication by 2^M and final rounding
{ .mfi
  nop.m     0
  fma.s1    fs_l_2M        = fs_l, f2M, f0
  nop.i     0
} ;;
{ .mfb
  nop.m     0
  fma.d.s0  f8             = fs_h, f2M, fs_l_2M
(p14) br.cond.spnt EXP_POSSIBLE_OVERFLOW                    // Branch if there could be an overflow
}
{ .mbb
  nop.m     0
(p15) br.cond.spnt EXP_POSSIBLE_UNDERFLOW                   // Branch if there could be an underflow
  br.ret.sptk b0 ;;                                         // Normal path exit
}
// General remark: do not change the order of the labeled parts of the code below. We rely on the conditional
// continuation in the next section in some cases. Do not change the order of the code sequences neither, they are
// all scheduled for optimal performance, additionally we reuse some physical registers.
//
// This case should not happen if the input is a double number, because with a double
// we would be in the certain overflow case.
// With a higher precision input, there is a possibility.
// We recalculate the final result with a higher exponent range in which overflow does not occur.
// If this result is greater than the greatest double, we have a certain overflow.
// Otherwise we return the already known result in f8.
// For this calculations, we use status field number 2.
//
EXP_POSSIBLE_OVERFLOW:
{ .mfi
  mov       rGtln          = 0x103ff                        // Exponent for largest dbl + 1 ulp
  fsetc.s2  0x7F, 0x42                                      // Set wre (widest range enable)
  nop.i     0 ;;
}
{ .mfi
  setf.exp  fGtpln         = rGtln                          // Form greatest dbl + 1 ulp
  fma.d.s2  fWRE_f8        = fs_h, f2M, fs_l_2M             // Recalculate the result
  nop.i     0
}
{ .mfi
  nop.m     0
  fsetc.s2  0x7F, 0x40                                      // Turn wre off
  nop.i     0 ;;
}
{ .mfi
  nop.m     0
  fcmp.ge.s1 p0, p6        = fWRE_f8, fGtpln                // Test if there would be an overflow in double prec.
  nop.i     0 ;;
}
{ .mfb
  nop.m     0
  fma.d.s0  f8             = fs_h, f2M, fs_l_2M             // Recalculate common result to have all flags correctly set
(p6) br.ret.sptk b0 ;;                                      // Return if the test is false to the exp() caller
                                                            // otherwise continue with the certain overflow case below.
}
EXP_CERTAIN_OVERFLOW:
{ .mmi
  mov       rExpBig        = 0x1ffff-1 ;;                   // Do not remove the stop ! We use a bypass
  setf.exp  fBig           = rExpBig                        // Form big normal
  nop.i     0 ;;
}
{ .mfb
  nop.m     0
  fma.d.s0  f8             = fBig, fBig, f0                 // Form +inf + flags
  br.ret.sptk b0 ;;                                         // Return
}
EXP_POSSIBLE_UNDERFLOW:
// We are in an underflow case iff we flush tiny (underflowed) numbers to zero when the ftz (flush to zero) flag is set.
// So it suffices to recalculate the final result with ftz set in status register two.
{ .mfi
  nop.m     0
  fsetc.s2  0x7F, 0x41                                      // Set ftz in status reg s2
  nop.i     0 ;;
}
{ .mfi
  nop.m     0
  fma.d.s2  fFTZ_f8        = fs_h, f2M, fs_l_2M             // Recalculate the result with ftz
  nop.i     0
}
{ .mfi
  nop.m     0
  fsetc.s2  0x7F, 0x40                                      // Turn ftz in s2 off
  nop.i     0 ;;
```

```
}
{ .mfi
  nop.m     0
  fcmp.eq.s1 p6, p7        = fFTZ_f8, f0                    // Check if result == 0
  nop.i     0 ;;
}
{ .mfb
  nop.m     0
  fma.d.s0  f8             = fs_h, f2M, fs_l_2M             // Recalculate the common result
(p6) br.cond.spnt EXP_UNDERFLOW_RETURN ;;                   // If we are in an underflow case, we branch,
}
{ .mib
  nop.m     0
  nop.i     0
(p7) br.ret.sptk b0 ;;                                      // otherwise we return to caller function
}
EXP_CERTAIN_UNDERFLOW:
{ .mmi
  mov       rExpSmall      = 1 ;;
  setf.exp  fSmall         = rExpSmall                      // Form small normal
  nop.i     0 ;;
}
{ .mfi
  nop.m     0
  fma.d.s0  f8             = fSmall, fSmall, f0             // Form tiny, inexact, underflow
  nop.i     0
}
EXP_UNDERFLOW_RETURN:
{ .mib
  nop.m     0
  nop.i     0
  br.ret.sptk b0 ;;                                         // And back to the roots
}
.default
.endp crexp#
```

References

[1] Tang, Ping Tak Peter: *Table-Driven Implementation of the Exponential Function in IEEE Floating - Point Arithmetic*, ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989

[2] Muller, Jean-Michel: *Elementary Functions Algorithms and Implementation*, Birkhäuser, 1997

[3] Markstein, Peter: *IA-64 and Elementary Functions*, Prentice Hall 2000

[4] Cornea, Marius et al: *Scientific Computing on Itanium-based Systems*, Intel Press, 2002

[5] IEEE Association: *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York 1985

[6] Defour, David: *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*, Thèse de doctorat, E.N.S. de Lyon, 2003 (to be published)

[7] Lefèvre, Vincent: *Moyens arithmétiques pour un calcul fiable*, Thèse de doctorat, E.N.S de Lyon, 2000

[8] Defour, David, Dinechin, Florent de, Muller, Jean-Michel: *Correctly Rounded Exponential Function in Double Precision Arithmetic*, Rapport de rechèrche, CNRS, INRIA, ENS de Lyon Nr. 01-26 (july 2001)

[9] Lefèvre, Vincent, Muller, Jean-Michel, Tisserand, Arnaud: *Toward Correctly Rounded Transcendentals*, in: IEEE Transactions on Computers, Vol. 47, Nr. 11, 1998

[10] Defour, David, Muller, Jean-Michel: *Evaluation des fonctions élémentaires*, in: CNRS, RSR-CP - 13/2001 Arithmétique des ordinateurs, p. 449-464

[11] Intel: *Intel Itanium Architecture, Software Developer's Manual, Volume 3: Instruction Set Reference*, Rev. 2.0, December 2001 (Intel Document Number: 245319-003)

[12] Møller, O.: *Quasi double-precision in floating-point addition*, BIT, vol. 5, 1965, p. 37-50

[13] Dekker, T. J.: *A Floating-Point Technique for Extending the Available Precision*, Numerical Mathematics, vol. 18, 1971, p. 224-242

[14] Cody, W. J.: *Software Manual for Elementary Functions*, Prentice-Hall, 1980

[15] Cody, W. J.: *Implemenation and testing of function software*, in: Messina, P. C. and Murli, A., editors, *Problems and Methodologies in Mathematical Software Production*, Lecture Notes in Computer Science 142, Springer Verlag, Berlin.

[16] Sterbenz, P. H.: *Floating point computation*, Prentice-Hall, Englewood Cliffs, NJ, 1974

[17] Intel: *Highly Optimized Mathematical Functions for the Intel Itanium Architecture*, Application Note, Dec. 2002 (Intel Document Number: 245410-009)

[18] IBM: *IBM accurate portable mathematical library*, URL: `http://oss.software.ibm.com/mathlib/`

[19] *MPFR, the Multiprecision Precision Floating-Point Reliable library*, URL: `http://www.loria.fr/equipes/polka/software.html`

[20] Harrison, J., Kubaska, T., Story, S., Tang, P., *The Computation of Transcendental Functions on the IA-64 Architecture*, Intel Technical Journal, 1999, Q4: 1-7

[21] Knuth, D. E.: *The art of computer programming, Volume 2 Seminumerical Algorithms*, Addison-Wesley, 1969

[22] Garey, M. R., et al.: *Some simplified NP-complete graph problems*, Theoretical Computer Science, 1 (1976), p. 237-267

[23] Lefèvre Vincent, Muller, Jean-Michel: *Worst Cases for Correct Rounding of the Elementary Functions in Double Precision*, in: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, 2001, p. 111-118

[24] Bertot, Yves, Castéran, Pierre: *Coq'Art*, URL: `ftp://ftp-sop.inria.fr/lemme/Yves.Bertot/bouquin/coqart.ps.gz`



Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399