Computing Correctly Rounded Integer Powers in Floating-Point Arithmetic

Jean-Michel Muller, Peter Kornerup, Christoph Lauter, Vincent Lefèvre, Nicolas Louvet

Computing Correctly Rounded Integer Powers in Floating-Point Arithmetic*

Peter Kornerup Christoph Lauter Nicolas Louvet
Vincent Lefèvre Jean-Michel Muller †

May 12, 2008

Abstract

We introduce several algorithms for accurately evaluating powers to a positive integer in floating-point arithmetic, assuming a *fused multiply-add* (fma) instruction is available. We aim at always obtaining correctly-rounded results in round-to-nearest mode, that is, our algorithms return the floating-point number that is nearest the exact value.

1 Introduction

We deal with the implementation of the integer power function in floating-point arithmetic. In the following, we assume a radix-2 floating-point arithmetic that follows the IEEE-754 standard for floating-point arithmetic. We also assume that a fused multiply-and-add (fma) operation is available, and that the input as well as the output values of the power function are not subnormal numbers, and are below the overflow threshold (so that we can focus on the powering of the significands only).

An fma instruction allows one to compute $ax \pm b$, where a, x and b are floating-point numbers, with one final rounding only. Examples of processors with an fma are the IBM PowerPC and the Intel/HP Itanium [7].

An important case dealt with in the paper is the case when an internal format, wider than the target format, is available. For instance, to guarantee – in some cases – correctly rounded integer powers in double precision arithmetic using our algorithms based on iterated products, we will have to assume that a double-extended precision is available. The examples will consider that it has a 64-bit precision, which is the minimum required by the IEEE-754 standard.

The only example of currently available processor with an fma and a double-extended precision format is Intel and HP's Itanium Processor [22, 7]. And yet, since the fma

*This is Research Report No RR-2008-15 of Laboratoire LIP. LIP is a joint laboratory of CNRS, École Normale Supérieure de Lyon, INRIA and Université de Lyon

†Peter Kornerup is with SDU, Odense, Denmark; Vincent Lefèvre, Christoph Lauter, Nicolas Louvet and Jean-Michel Muller are with Laboratoire LIP, CNRS/ENS Lyon/INRIA/Université de Lyon, Lyon, France.

operation will be specified in the revised version of the IEEE-754 standard [1], it is very likely that more processors in the future will offer that combination of features.

The original IEEE-754 standard [2] for radix-2 floating-point arithmetic (as well as its follower, the IEEE-854 radix-independent standard [3], and the forthcoming revised standard) require that the four arithmetic operations and the square root should be correctly rounded. In a floating-point system that follows the standard, the user can choose an *active rounding mode* from:

- rounding towards $-\infty$: $RD(x)$ is the largest machine number less than or equal to x;
- rounding towards $+\infty$: $RU(x)$ is the smallest machine number greater than or equal to x;
- rounding towards 0: $RZ(x)$ is equal to $RD(x)$ if $x \geq 0$, and to $RU(x)$ if $x < 0$;
- rounding to nearest: $RN(x)$ is the machine number that is the closest to x (with a special convention if x is exactly between two machine numbers: the chosen number is the "even" one, i.e., the one whose last significand bit is a zero).

When $a \circ b$ is computed, where a and b are floating-point numbers and $\circ$ is $+$, $-$, $\times$ or $\div$, the returned result is what we would get if we computed $a \circ b$ exactly, with "infinite" precision and rounded it according to the active rounding mode. The default rounding mode is round-to-nearest. This requirement is called *correct rounding*. Among its many interesting properties, one can cite the following result (due to Dekker [13]).

Theorem 1 (Fast2Sum algorithm) *Assume the radix r of the floating-point system being considered is 2 or 3, and that the used arithmetic provides correct rounding with rounding to nearest. Let a and b be floating-point numbers, and assume that the exponent of a is larger than or equal to that of b. The following algorithm computes two floating-point numbers s and t that satisfy:*

- *$s + t = a + b$ exactly;*
- *s is the floating-point number that is closest to $a + b$.*

Algorithm 1 (Fast2Sum(a,b))

$$\begin{aligned} s &:= RN(a+b); \\ z &:= RN(s-a); \\ t &:= RN(b-z); \end{aligned}$$

Note that the information "the exponent of a is larger than or equal to that of b" cannot be checked efficiently, but if $|a| \geq |b|$, then the exponent of a will be larger than or equal to that of b.

If no information on the relative orders of magnitude of a and b is available, there is an alternative algorithm due to Knuth [15] and Møller [24], called 2Sum. It requires 6 operations instead of 3 for the Fast2Sum algorithm, but on any modern computer, the 3 additional operations cost significantly less than a comparison followed by a branching: on current pipelined architectures, an *if* statement with an wrong branch prediction may cause the instruction pipeline to drain.

Algorithm 2 (2Sum(a,b))

$$\begin{aligned} s &= RN(a+b);\\ b' &= RN(s-a);\\ a' &= RN(s-b');\\ \delta_b &= RN(b-b');\\ \delta_a &= RN(a-a');\\ t &= RN(\delta_a+\delta_b). \end{aligned}$$

The fma instruction allows one to design convenient software algorithms for correctly rounded division and square root. It also has the following interesting property. From two input floating-point numbers a and b, the following algorithm computes c and d such that $c+d=ab$, and c is the floating-point number that is nearest ab.

Algorithm 3 (Fast2Mult(a,b))

$$\begin{aligned} c &:= RN(ab);\\ d &:= RN(ab-c); \end{aligned}$$

Performing a similar calculation without a fused multiply-add operation is possible with an algorithm due to Dekker [13], but this requires 17 floating-point operations instead of 2.

Transformations such as 2Sum, Fast2Sum and Fast2Mult were called *error-free transformations* by Rump [26].

In the sequel of the paper, we examine various methods for getting very accurate (indeed: correctly rounded, in round-to-nearest mode) integer powers. We first deal with methods based on repeated multiplications (that is, we simply use the fact that x^n is $x \times x \times \cdots \times x$), where the arithmetic operations are performed with a larger accuracy using algorithms such as Fast2Sum and Fast2Mult. We then investigate methods based on the identity

$$x^n = 2^{n \log_2(x)},$$

and that use techniques we have developed when building the CRlibm library of correctly rounded mathematical functions [8, 12].

2 On correct rounding of functions

V. Lefèvre introduced a new method for finding hardest-to-round cases for evaluating a regular unary function [19, 18, 20]. That method allowed Lefèvre and Muller to give such cases for the most familiar elementary functions [21]. Recently, Lefèvre adapted his software to the case of functions x^n, where n is an integer; this consisted in supporting a parameter n.

Let us briefly summarize Lefèvre's method. The tested domain is split into intervals, where the function can be approximated by a polynomial of a quite small degree and an accuracy of about 90 bits. The approximation does not need to be very tight, but it must be computed quickly; that is why Taylor's expansion is used. For instance, by choosing intervals of length 1/8192 of a binade, the degree is 3 for $n=3$, it is 9 for $n=70$, and 12 to 13 for $n=500$. These intervals are split into subintervals where the polynomial can be approximated by polynomials of smaller degrees, down to 1. How intervals are

split exactly depends on the value of n (the parameters can be chosen empirically, thanks to timing measures). Determining the approximation for the following subinterval can be done using fixed-point additions in a precision up to a few hundreds of bits, and multiplications by constants. A filter of sub-linear complexity is applied on each degree-1 polynomial, eliminating most input arguments. The remaining arguments (e.g., one over 2^{32}) are potential worst cases, that need to be checked in a second step by direct computation in a higher accuracy.

Because of a reduced number of input arguments, the second step is much faster than the first step and can be run on a single machine. The first step (every computation up to the filter) is parallelized. The intervals are independent, so that the following conventional solution has been chosen: A server distributes intervals to the clients running on a small network of desktop machines.

All approximation errors are carefully bounded, either by interval arithmetic or by static analysis. Additional checks for missing or corrupt data are also done in various places. So, the final results are guaranteed (up to undetected software bugs and errors in paper proofs[1]).

Concerning the particular case of x^n, one has $(2x)^n = 2^n x^n$. Therefore if two numbers x and y have the same significand, their images x^n and y^n also have the same significand. So only one binade needs to be tested[2], $[1, 2)$ in practice.

For instance, in double-precision arithmetic, the hardest to round case for the function x^{458} corresponds to

$$x = 1.00001111001110001100111110101010110010110111100011010$$

we have

$$\begin{aligned} x^{458} &= \underbrace{1.00011111000010110000100001110110101110100000000100101}_{\text{53 bits}}\ 1 \\ &\quad \underbrace{00000000\cdots00000000}_{\text{61 zeros}} 1110\cdots \times 2^{38} \end{aligned}$$

which means that x^n is extremely close to the exact middle of two consecutive double-precision numbers. There is a run of 61 consecutive zeros after the rounding bit. This case is the worst case for all values of n between 3 and 733.

This worst case has been obtained by an exhaustive search using the method described above, after a total of 646300 hours of computation for the first step (sum of the times on each CPU core). The time needed to test a function x^n increases with n, as the error on the approximation by a degree-1 polynomial on some fixed interval increases. On the current network (when all machines are available), for $n \simeq 600$, it takes between 7 and 8 hours for each power function. On a reference 2.2-Ghz AMD Opteron machine, one needs an estimated time of 90 hours per core to test x^n with $n = 10$, about 280 hours for $n = 40$, and around 500 hours for any n between 200 and 600.

Table 1 gives the longest runs of identical bits after the rounding bit for $3 \leq n \leq 733$.

[1] Work has started towards formalized proofs (in the EVA-Flo project).

[2] We did not take subnormals into account, but one can prove that the worst cases in all rounding modes can also be used to round subnormals correctly.

n	k
32	48
76, 81, 85, 200, 259, 314, 330, 381, 456, 481, 514, 584, 598, 668	49
9, 15, 16, 31, 37, 47, 54, 55, 63, 65, 74, 80, 83, 86, 105, 109, 126, 130, 148, 156, 165, 168, 172, 179, 180, 195, 213, 214, 218, 222, 242, 255, 257, 276, 303, 306, 317, 318, 319, 325, 329, 342, 345, 346, 353, 358, 362, 364, 377, 383, 384, 403, 408, 417, 429, 433, 436, 440, 441, 446, 452, 457, 459, 464, 491, 494, 500, 513, 522, 524, 538, 541, 547, 589, 592, 611, 618, 637, 646, 647, 655, 660, 661, 663, 673, 678, 681, 682, 683, 692, 698, 703, 704	50
10, 14, 17, 19, 20, 23, 25, 33, 34, 36, 39, 40, 43, 46, 52, 53, 72, 73, 75, 78, 79, 82, 88, 90, 95, 99, 104, 110, 113, 115, 117, 118, 119, 123, 125, 129, 132, 133, 136, 140, 146, 149, 150, 155, 157, 158, 162, 166, 170, 174, 185, 188, 189, 192, 193, 197, 199, 201, 205, 209, 210, 211, 212, 224, 232, 235, 238, 239, 240, 241, 246, 251, 258, 260, 262, 265, 267, 272, 283, 286, 293, 295, 296, 301, 302, 308, 309, 324, 334, 335, 343, 347, 352, 356, 357, 359, 363, 365, 371, 372, 385, 390, 399, 406, 411, 412, 413, 420, 423, 431, 432, 445, 447, 450, 462, 465, 467, 468, 470, 477, 482, 483, 487, 490, 496, 510, 518, 527, 528, 530, 534, 543, 546, 548, 550, 554, 557, 565, 567, 569, 570, 580, 582, 585, 586, 591, 594, 600, 605, 607, 609, 610, 615, 616, 622, 624, 629, 638, 642, 651, 657, 665, 666, 669, 671, 672, 676, 680, 688, 690, 694, 696, 706, 707, 724, 725, 726, 730	51
3, 5, 7, 8, 22, 26, 27, 29, 38, 42, 45, 48, 57, 60, 62, 64, 68, 69, 71, 77, 92, 93, 94, 96, 98, 108, 111, 116, 120, 121, 124, 127, 128, 131, 134, 139, 141, 152, 154, 161, 163, 164, 173, 175, 181, 182, 183, 184, 186, 196, 202, 206, 207, 215, 216, 217, 219, 220, 221, 223, 225, 227, 229, 245, 253, 256, 263, 266, 271, 277, 288, 290, 291, 292, 294, 298, 299, 305, 307, 321, 322, 323, 326, 332, 349, 351, 354, 366, 367, 369, 370, 373, 375, 378, 379, 380, 382, 392, 397, 398, 404, 414, 416, 430, 437, 438, 443, 448, 461, 471, 474, 475, 484, 485, 486, 489, 492, 498, 505, 507, 508, 519, 525, 537, 540, 544, 551, 552, 553, 556, 563, 564, 568, 572, 575, 583, 593, 595, 597, 601, 603, 613, 619, 620, 625, 627, 630, 631, 633, 636, 640, 641, 648, 650, 652, 654, 662, 667, 670, 679, 684, 686, 687, 702, 705, 709, 710, 716, 720, 721, 727	52
6, 12, 13, 21, 58, 59, 61, 66, 70, 102, 107, 112, 114, 137, 138, 145, 151, 153, 169, 176, 177, 194, 198, 204, 228, 243, 244, 249, 250, 261, 268, 275, 280, 281, 285, 297, 313, 320, 331, 333, 340, 341, 344, 350, 361, 368, 386, 387, 395, 401, 405, 409, 415, 418, 419, 421, 425, 426, 427, 442, 449, 453, 454, 466, 472, 473, 478, 480, 488, 493, 499, 502, 506, 509, 517, 520, 523, 526, 532, 533, 542, 545, 555, 561, 562, 571, 574, 588, 590, 604, 608, 614, 621, 626, 632, 634, 639, 644, 653, 658, 659, 664, 677, 689, 701, 708, 712, 714, 717, 719	53
4, 18, 44, 49, 50, 97, 100, 101, 103, 142, 167, 178, 187, 191, 203, 226, 230, 231, 236, 273, 282, 284, 287, 304, 310, 311, 312, 328, 338, 355, 374, 388, 389, 391, 393, 394, 400, 422, 428, 434, 435, 439, 444, 455, 469, 501, 504, 511, 529, 535, 536, 549, 558, 559, 560, 566, 573, 577, 578, 581, 587, 596, 606, 612, 623, 628, 635, 643, 649, 656, 675, 691, 699, 700, 711, 713, 715, 718, 731, 732	54
24, 28, 30, 41, 56, 67, 87, 122, 135, 143, 147, 159, 160, 190, 208, 248, 252, 264, 269, 270, 279, 289, 300, 315, 339, 376, 396, 402, 410, 460, 479, 497, 515, 516, 521, 539, 579, 599, 602, 617, 674, 685, 693, 723, 729	55
89, 106, 171, 247, 254, 278, 316, 327, 348, 360, 424, 451, 463, 476, 495, 512, 531, 645, 697, 722, 728	56
11, 84, 91, 234, 237, 274, 407, 576, 695	57
35, 144, 233, 337, 733	58
51, 336	59
503	60
458	61

Table 1: Maximal length k of the runs of identical bits after the rounding bit (assuming the target precision is double precision) in the worst cases for n from 3 to 733.

3 Algorithms based on repeated floating-point multiplications

3.1 Using a double-double multiplication algorithm

Algorithms Fast2Sum and Fast2Mult both provide double-precision results of value $(x+y)$ represented in the form of pairs (x, y). In the following we need products of numbers represented in this form. However, we will be satisfied with approximations to the products, discarding terms of the order of the product of the two low-order terms. Given two double-precision operands $(a_h + a_\ell)$ and $(b_h + b_\ell)$ the following algorithm *DblMult* computes (x, y) such that $(x + y) = [(a_h + a_\ell)(b_h + b_\ell)](1 + \delta)$ where the relative error δ is given by Theorem 2 below.

Algorithm 4 (DblMult(a_h,a_ℓ,b_h,b_ℓ))

$$\begin{array}{lcl} t_{1h} & := & RN\,(a_h b_h); \\ t_2 & := & RN\,(a_h b_\ell); \\ t_{1\ell} & := & RN\,(a_h b_h - t_{1h}); \\ t_3 & := & RN\,(a_\ell b_h + t_2); \\ t_4 & := & RN\,(t_{1\ell} + t_3); \\ c_h & := & RN\,(t_{1h} + t_4); \\ t_5 & := & RN\,(c_h - t_{1h}); \\ c_\ell & := & RN\,(t_4 - t_5); \end{array}$$

The result to be returned is (c_h, c_ℓ).

Theorem 2 *Let* $\varepsilon = 2^{-p}$*, where* p *is the precision of the radix-*2 *floating-point system used. If* $|a_\ell| \leq 2^{-p}|a_h|$ *and* $|b_\ell| \leq 2^{-p}|b_h|$ *then the returned value* (x, y) *of function DblMult*$(a_h, a_\ell, b_h, b_\ell)$ *satisfies*

$$x + y = (a_h + a_\ell)(b_h + b_\ell)(1 + \eta),$$

with

$$|\eta| \leq 7\varepsilon^2 + 18\varepsilon^3 + 16\varepsilon^4 + 6\varepsilon^5 + \varepsilon^6.$$

Notes:

1. as soon as $p \geq 5$, we have $|\eta| \leq 8\varepsilon^2$;
2. in the case of single precision ($p = 24$), $|\eta| \leq 7.000002\varepsilon^2$;
3. in the case of double precision ($p = 53$), $|\eta| \leq 7.000000000000002\varepsilon^2$.

Proof:

Note that $(t_{1h}, t_{1\ell})$ is Fast2Mult(a_h, b_h) and (c_h, c_ℓ) is Fast2Sum(t_{1h}, t_4), so that:

- $t_{1h} + t_{1\ell} = a_h b_h$ exactly;
- $c_h + c_\ell = t_{1h} + t_4$ exactly.

Now, let us analyze the other operations. In the following, the ε_i's are terms of absolute value less than or equal to $\varepsilon = 2^{-p}$. First, notice that $a_\ell = \varepsilon_4 a_h$ and $b_\ell = \varepsilon_5 b_h$. Since the additions and multiplications are correctly rounded (to the nearest) operations:

1. $t_2 = a_h b_\ell (1+\varepsilon_1)$;

2.
$$\begin{aligned} t_3 &= (a_\ell b_h + t_2)(1+\varepsilon_2) \\ &= a_h b_\ell + a_\ell b_h + a_h b_h(\varepsilon_1\varepsilon_5 + \varepsilon_2\varepsilon_4 + \varepsilon_2\varepsilon_5 + \varepsilon_1\varepsilon_2\varepsilon_5) \\ &= a_h b_\ell + a_\ell b_h + a_h b_h(3\varepsilon_6^2 + \varepsilon_7^3) \end{aligned}$$

3.
$$\begin{aligned} t_4 &= (t_{1\ell} + t_3)(1+\varepsilon_8) \\ &= t_{1\ell} + a_h b_\ell + a_\ell b_h + a_h b_h(3\varepsilon_6^2 + \varepsilon_7^3) \\ &\quad + t_{1\ell}\varepsilon_8 + a_h b_h(\varepsilon_4\varepsilon_8 + \varepsilon_5\varepsilon_8) \\ &\quad + a_h b_h(3\varepsilon_6^2 + \varepsilon_7^3)\varepsilon_8 \end{aligned}$$

but, from $(t_{1h}, t_{1\ell}) = \text{Fast2Mult}(a_h, b_h)$, we deduce $t_{1\ell} = a_h b_h \varepsilon_9$, therefore

$$\begin{aligned} t_4 &= t_{1\ell} + a_h b_\ell + a_\ell b_h \\ &\quad + a_h b_h(3\varepsilon_6^2 + \varepsilon_7^3 + \varepsilon_8\varepsilon_9 + \varepsilon_4\varepsilon_8 + \varepsilon_5\varepsilon_8 + 3\varepsilon_6^2\varepsilon_8 + \varepsilon_7^3\varepsilon_8) \\ &= t_{1\ell} + a_h b_\ell + a_\ell b_h + a_h b_h(6\varepsilon_{10}^2 + 4\varepsilon_{11}^3 + \varepsilon_{12}^4). \end{aligned}$$

4.
$$\begin{aligned} c_h + c_\ell &= t_{1h} + t_4 \\ &= a_h b_h + a_h b_\ell + a_\ell b_h + a_h b_h(6\varepsilon_{10}^2 + 4\varepsilon_{11}^3 + \varepsilon_{12}^4) \\ &= a_h b_h + a_h b_\ell + a_\ell b_h + (a_\ell b_\ell - \varepsilon_4\varepsilon_5 a_h b_h) + a_h b_h(6\varepsilon_{10}^2 + 4\varepsilon_{11}^3 + \varepsilon_{12}^4), \\ &= (a_h + a_\ell)(b_h + b_\ell) + a_h b_h(7\varepsilon_{10}^2 + 4\varepsilon_{11}^3 + \varepsilon_{12}^4). \end{aligned}$$

Now, from $a_h = (a_h + a_\ell)(1+\varepsilon_{14})$ and $b_h = (b_h + b_\ell)(1+\varepsilon_{15})$, we get

$$a_h b_h = (a_h + a_\ell)(b_h + b_\ell)(1 + \varepsilon_{14} + \varepsilon_{15} + \varepsilon_{14}\varepsilon_{15}),$$

from which we deduce

$$c_h + c_\ell = (a_h + a_\ell)(b_h + b_\ell)(1 + 7\varepsilon_{16}^2 + 18\varepsilon_{17}^3 + 16\varepsilon_{18}^4 + 6\varepsilon_{19}^5 + \varepsilon_{20}^6).$$

□

3.2 The IteratedProductPower algorithm

Algorithm 5 (IteratedProductPower(x, n), $n \geq 1$)

$i := n$;
$(h, \ell) := (1, 0)$;
$(u, v) := (x, 0)$;
while $i > 1$ **do**
  if $(i$ **mod** $2) = 1$ **then**
    $(h, \ell) :=$ *DblMult*(h, ℓ, u, v);
  end;
  $(u, v) :=$ *DblMult*(u, v, u, v);
  $i := \lfloor i/2 \rfloor$;
end do;
return *DblMult*(h, ℓ, u, v);

Due to the approximations performed in algorithm *DblMult*, terms corresponding to the product of low order terms are not included. A thorough error analysis is performed below. The number of floating-point operations used by the IteratedProductPower algorithm is between $8(1 + \lfloor \log_2(n) \rfloor)$ and $8(1 + 2\lfloor \log_2(n) \rfloor)$.

3.3 Error of algorithm IteratedProductPower

Theorem 3 *The two values c_h and c_ℓ returned by algorithm IteratedProductPower satisfy*

$$h + \ell = x^n(1 + \alpha),$$

with

$$(1 - |\eta|)^{n-1} \leq 1 + \alpha \leq (1 + |\eta|)^{n-1}$$

where $|\eta| \leq 7\varepsilon^2 + 18\varepsilon^3 + 16\varepsilon^4 + 6\varepsilon^5 + \varepsilon^6$ is the same value as in Theorem 2.

Proof: Algorithm IteratedProductPower computes approximations to powers of x, using $x^{i+j} = x^i x^j$. By induction, one easily shows that the approximation to x^k is of the form $x^k(1 + \beta_k)$, where $(1 - |\eta|)^{k-1} \leq (1 + \beta_k) \leq (1 + |\eta|)^{k-1}$. If we call η_{i+j} the relative error (obtained from Theorem 2) when multiplying together the approximations to x^i and x^j, the induction follows from

$$(1-\eta)^{i-1}(1-\eta)^{j-1}(1-\eta) \leq \Big(x^i(1 + \beta_i)\Big)\Big(x^j(1 + \beta_j)\Big)(1+\eta_{i+j}) \leq (1+\eta)^{i-1}(1+\eta)^{j-1}(1+\eta).$$

□

Table 2 gives bounds on $|\alpha|$ for several values of n (note that the bound is an increasing value of n), assuming the algorithm is used in double precision.

Define the *significand* of a non-zero real number u to be

$$\frac{u}{2^{\lfloor \log_2 |u| \rfloor}}.$$

Define α_{max} as the bound on $|\alpha|$ obtained for a given value of n. From

$$x^n(1 - \alpha_{max}) \leq h + \ell \leq x^n(1 + \alpha_{max}),$$

we deduce that the significand of $h + \ell$ is within $2\alpha_{max}$ from $x^n/2^{\lfloor \log_2 |h+\ell| \rfloor}$. From the results given in Table 2, we deduce that for all practical values of n the significand of $h+\ell$ is within much less than 2^{-53} from $x^n/2^{\lfloor \log_2 |h+\ell| \rfloor}$ (indeed, to get $2\alpha_{\max}$ larger that 2^{-53}, we need $n > 2^{49}$). This means that $RN(h + \ell)$ is within less than one ulp from x^n, more precisely,

Theorem 4 *If algorithm IteratedProductPower is implemented in double precision, then $RN(h + \ell)$ is a faithful rounding of x^n, as long as $n \leq 2^{49}$.*

Moreover, for $n \leq 10^8$, $RN(h+\ell)$ is within 0.50000008 ulps from the exact value: we are very close to correct rounding (indeed, we almost always return a correctly rounded result), yet we cannot guarantee correct rounding, even for the smallest values of n. This requires a much better accuracy, as shown in Section 3.4. To guarantee a correctly rounded result in double precision, we will need to run algorithm IteratedProductPower in double-extended precision.

Table 3 gives bounds on $|\alpha|$ for several values of n assuming the algorithm is realized in double-extended precision. As expected, we are 22 bits more accurate.

n	$-\log_2(\alpha_{max})$	n	$-\log_2(\alpha_{max})$
3	102.19	1000	93.22
4	101.60	10,000	89.90
5	101.19	100,000	86.58
10	100.02	1,000,000	83.26
20	98.94	10,000,000	79.93
30	98.33	100,000,000	76.61
40	98.90	2^{32}	71.19
50	97.57		
100	96.56		
200	95.55		

Table 2: Binary logarithm of the relative accuracy ($-\log_2(\alpha_{max})$), for various values of n assuming algorithm IteratedProductPower is used in double precision.

n	$-\log_2(\alpha_{max})$	n	$-\log_2(\alpha_{max})$
3	124.19	1000	115.22
4	123.60	10,000	111.90
5	123.19	100,000	108.58
10	122.02	1,000,000	105.26
20	120.94	10,000,000	101.93
30	120.33	100,000,000	98.61
40	120.90	2^{32}	93.19
50	119.57		
100	118.56		
200	117.55		
586	116.0003		
587	115.9978		

Table 3: Binary logarithm of the relative accuracy ($-\log_2(\alpha_{max})$), for various values of n assuming algorithm IteratedProductPower is implemented in double-extended precision.

3.4 Getting correctly rounded values with IteratedProductPower

We are interested in getting correctly rounded results in double precision. To achieve this we assume that algorithm IteratedProductPower is executed in double extended precision. The algorithm returns two double-extended numbers h and $c\ell$ such that

$$x^n(1-\alpha_{max}) \le h + \ell \le x^n(1+\alpha_{max}),$$

where α_{max} is given in Table 3.

In the following we shall distinguish two roundings: RN_e means round-to-nearest in extended double precision and RN_d is round-to-nearest in double precision. Let $\mathrm{ulp}(\cdot)$ denote "unit-in-last-position" such that $|x - RN(x)| \le \frac{1}{2}\mathrm{ulp}(x)$.

Define a *breakpoint* as the exact midpoint of two consecutive double precision numbers. $RN_d(c_h + c_\ell)$ will be equal to $RN_d(x^n)$ if and only if there is no breakpoint between x^n

and $c_h + c_\ell$.

The worst cases obtained (given in Table 1, the very worst case for $n \leq 733$ being attained for $n = 458$) show that:

- if x is a double-precision number, and if $3 \leq n \leq 586$, then the significand y of x^n is always at a distance larger than 2^{-115} from the breakpoint μ (see Figure 1) where the distance $|y - \mu| \geq 2^{-(53+61+1)} = 2^{-115}$;
- if $587 \leq n \leq 733$ then the significand y of x^n is always at a distance larger than $2^{-(53+55+1)} = 2^{-109}$ from a breakpoint.

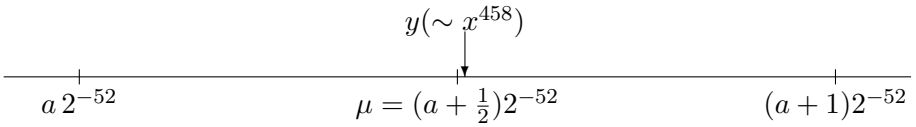

Figure 1: Position of the hardest to round case $y = x^{458}$ within rounding interval $[a2^{-52}; (a + 1)2^{-52}]$ with breakpoint $\mu = (a + \frac{1}{2})2^{-52}$, for significand defined by integer a.

We know that the significand of $h + \ell$ is within $2\alpha_{max}$ from that of x^n, where α_{max} (as given by its binary logarithm) is listed in Table 3. For all values of n less than or equal to 586, we have $2\alpha_{max} \leq 2^{-115}$, and for $587 \leq n \leq 733$, we have $2\alpha_{max} \leq 2^{-109}$. Thus $RN_d(h + \ell) = RN_d(x^n)$. We therefore get the following result:

Theorem 5 *If algorithm IteratedProductPower is performed in double-extended precision, and if* $3 \leq n \leq 733$, *then* $RN_d(h + \ell) = RN_d(x^n)$*: Hence by rounding* $h + \ell$ *to the nearest double-precision number, we get a correctly rounded result.*

Now, two important remarks:

- We do not have the worst cases for $n > 733$, but from probabilistic arguments we strongly believe that the lengths of the largest runs of consecutive bits after the rounding bit will be of the same order of magnitude (i.e., around 50) for some range of n above 733. However, it is unlikely that we will be able to show correct rounding in double precision for values of n larger than 1000.
- On an Intel Itanium processor, it is possible to directly add two double-extended precision numbers and round the result to double precision without a "double rounding" (i.e., without having an intermediate sum rounded to double-extended precision). Hence Theorem 5 can directly be used. Notice that the draft revised standard IEEE 754-R (see `http://754r.ucbtest.org/`) includes the fma as well as rounding to any specific destination format, independent of operand formats.

3.5 Two-step algorithm using double-extended precision

Now we suggest another approach: first compute an approximation to x^n using double-extended precision and a straightforward, very fast, algorithm. Then check if this approximation suffices to get $RN(x^n)$. If it does not, use the IteratedProductPower algorithm presented above.

Let us first give the algorithm. All operation are done in double extended precision.

Algorithm 6 (DbleXtendedPower(x, n), $n \geq 1$)

$i := n$;
$pow := 1$;
$u := w$;
while $i > 1$ **do**
  if $(i \bmod 2) = 1$ **then**
    $pow := RN_e(pow \cdot u)$;
  end;
  $u := RN_e(u \cdot u)$;
  $i := \lfloor i/2 \rfloor$;
end do;
$pow := RN_e(pow \cdot u)$;
return pow;

Using the very same proof as for Theorem 3, one easily shows the following result

Theorem 6 *The final result* pow *of Algorithm DbleXtendedPower satisfies*

$$pow = x^n(1 + \alpha),$$

where

$$(1 - 2^{-64})^{n-1} \leq 1 + \alpha \leq (1 + 2^{-64})^{n-1}.$$

n	$-\log_2(\lvert\alpha_{max}\rvert)$
3	63
4	62.41
5	62
6	61.67
10	60.83
15	60.19
20	59.41
30	59.14
32	59.04
33	58.9999999999999987
40	58.71
50	58.38
100	57.37
1000	54.03
1024	54.001
1025	53.99999999999999959996

Table 4: Maximum value of the parameter $|\alpha|$ of Theorem 6, for various values of n.

Table 4 could be used to show that up to $n = 1024$, algorithm 6 (run in double-extended precision) can be used to guarantee faithful rounding (in double precision). What interests us here is correct rounding. From Table 4, it follows that if $n \leq 32$,

then $\alpha < 2^{-59}$, which means that the final result *pow* of Algorithm DbleXtendedPower is within $2^{53} \times 2^{-59} = 1/64$ ulp from x^n. This means that if the bits 54 to 59 of *pow* are not 100000 or 011111, then rounding *pow* to the nearest floating-point number will be equivalent to rounding x^n. Otherwise, if the bits 54 to 59 of *pow* are 100000 or 011111 (which might occur with probability close to 1/32), we will have to run a more accurate yet slower algorithm, such as Algorithm IteratedProductPower.

3.6 When n is a constant

Very frequently n is a constant, i.e., n is known at compile-time. In such a case it is possible to simplify the iterated product algorithm, as well as the 2-step algorithm (that first uses Algorithm DbleXtendedPower and uses the other algorithm only if the double-extended result does not make it possible to deduce a correctly rounded value). The possible simplifications are:

- the loops can be unrolled, there is no longer any need to perform the computations "$i := \lfloor i/2 \rfloor$", nor to do tests on variable i;
- moreover, for the first values of n, addition chains to obtain the minimal number of multiplications needed to compute a power are known. This can be used for optimizing the algorithm. For instance, for n up to 10001, such addition chains can be obtained from `http://www.research.att.com/~njas/sequences/b003313.txt`.

4 Algorithm based on logarithms and exponentials

With binary asymptotic complexity in mind [4], it might seem silly to compute x^n by

$$x^n = 2^{n \cdot \log_2 x}.$$

However, in this section we are going to show that on actual superscalar and pipelined hardware, if n is large enough, the situation is different. For that purpose we consider an implementation on the Itanium architecture. Itanium offers both double extended precision and the FMA instruction, as well as some other useful operations. These features permit achieving high performance. In the example, for $n \leq 735$, the measured average evaluation time for x^n is equivalent to about 21 sequential multiplications on Itanium 2.

4.1 Basic layout of the algorithm

We combine the scheme for x^n based on logarithms and exponentials with a two-step approximation approach. This approach has already been proven efficient for common correctly rounded elementary functions [14, 30, 9]. It is motivated by the rarity of hard-to-round cases. In most cases, an approximation which is just slightly more accurate than the final precision, suffices to ensure correct rounding. Only in rare cases, the function's result must be approximated up to the accuracy demanded by the worst cases [14, 30, 9]. There is a well-known and efficient test whether correct rounding is already possible with small accuracy [30, 8].

We propose the scheme shown in Figure 2 for correctly rounding x^n. The function $2^{n \cdot \log_2(x)}$ is first approximated with an accuracy of $2^{-59.17}$. These 6.17 guard bits with

respect to double precision make the hard-to-round-case probability as small as about $2 \cdot 2^{-6.17} \approx 2.8\%$. If rounding is not possible correct rounding is ensured by the second step that provides an accuracy of 2^{-116}.

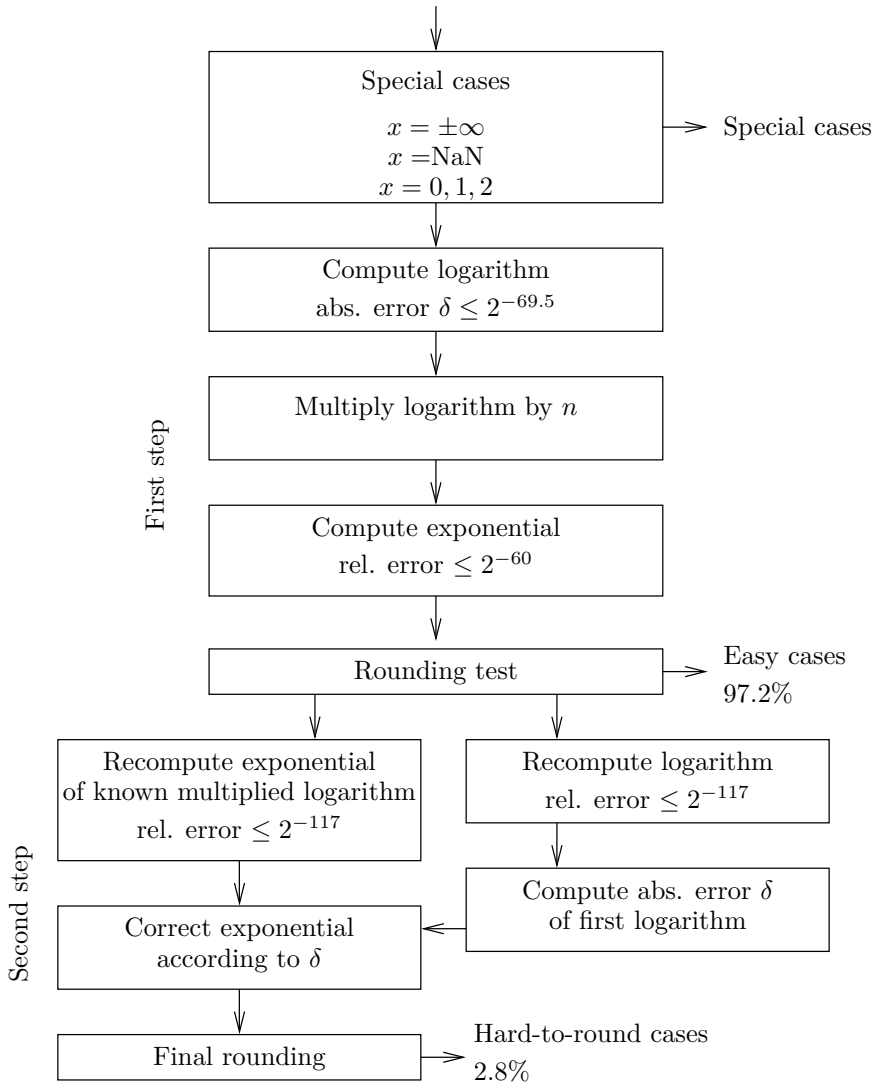

Figure 2: Two step exponential of logarithm approach

The design of the second step is particularly adapted to superscalar hardware. As the approximation to the logarithm $\ell = \log_2 x + \delta$ computed at the first step is already available, it is possible to perform the computation of the accurate logarithm and exponential in parallel. Considering

$$x^n = 2^{n \cdot \log_2 x} = 2^{n \cdot \ell} \cdot 2^{n \cdot (\log_2 x - \ell)},$$

we see that we can accurately approximate $2^{n \cdot \ell}$ and $\ell' = \log_2 x$ in parallel. The correction step, the multiplication of the exponential by $2^{n \cdot (\log_2 x - \ell)} = 2^{n \cdot (\ell' - \ell)} = 2^{n \cdot \delta}$ can be performed easily with a first order approximation: $2^{n \cdot \delta} \approx 1 + c \cdot n \cdot \delta$.

4.2 Implementation details and error estimates

Both the logarithm and the exponential approximation sub-algorithms follow the well-known principles of table look-up and polynomial approximation. The algorithms implemented are variants of the techniques presented in [29, 7, 16, 11]. Our implementation uses about 8 kbytes of tables. The approximation polynomials have optimized floating-point coefficients [5].

4.2.1 Logarithm

In both first and second step, the logarithm $\log_2 x$ is based on the following argument reduction:

$$\begin{aligned}
\log_2 x &= \log_2\left(2^E \cdot m\right) \\
&= E + \log_2(m \cdot r) - \log_2 r \\
&= E + \log_2(1 + (m \cdot r - 1)) - \log_2 r \\
&= E + \log_2(1 + z) + \log_2 r \\
&= E + p(z) + logtblr[m] + \delta
\end{aligned}$$

In this argument reduction, the decomposition of x into E and m can be performed using Itanium's `getf` and `fmerge` instructions.

The value r is produced by Itanium's `frcpa` instruction. This instruction gives an approximate to the reciprocal of m with at least 8.886 valid bits [7]. The instruction is based on a small table indexed by the first 8 bits of the significand (excluding the leading 1) of x. This makes it possible to tabulate the values of $\log_2 r$ in a table indexed by these first 8 bits of the significand of m.

The reduced argument z can exactly be computed with an FMS:

$$z = RN_e(m \cdot r - 1).$$

Indeed, as can easily be verified on the 256 possible cases, the `frcpa` instruction [7] returns its result r on floating-point numbers with at most 11 leading non-zero significand bits. Since x is a double, $x \cdot r$ holds on $53 + 11 = 64$ bits, hence a double-extended precision number. No rounding occurs on the subtraction $x \cdot r - 1$ as per Sterbenz' Lemma [28].

The exactness of the reduced argument z makes it possible to reuse it in the second, more accurate step of the algorithm. It is worth to remark that the latency for obtaining the reduced argument is small. It is produced by only 3 depended operations: `fmerge`, `frcpa` and `fms`.

The tabulated values $logtbl[m]$ for $\log_2 r$ are stored as double-double-extended numbers $logtblr_{hi}[m] + logtblr_{lo}[m]$. The absolute error of the entries with respect to the exact value $\log_2 r$ is bounded by 2^{-130}. Both double-extended numbers of an entry are read in the first step of the algorithm. The second step can reuse the values directly.

The magnitude of the reduced argument z is bounded as follows: we have $r = \frac{1}{m} \cdot (1 + \varepsilon_r)$ with $|\varepsilon_r| \leq 2^{-8.886}$. Hence

$$z = m \cdot r - 1 = \frac{1}{m} \cdot (1 + \varepsilon_r) \cdot m - 1 = \varepsilon_r$$

is bounded by $2^{-8.886}$.

The function $\log_2(1+z)$ is approximated using a polynomial of degree 6 for the first step and of degree 12 for the second step. The corresponding absolute approximation errors are bounded by $2^{-69.49}$ respectively $2^{-129.5}$. The polynomials have optimized floating-point coefficients. We minimize the number of double-extended and double precision numbers because Itanium can load doubles and single precision numbers faster [7, 23].

The approximation polynomials are evaluated using a mixture of Estrin and Horner scheme [7, 25, 11, 27]. In the first step, double-extended precision is sufficient for obtaining an absolute round-off error less than 2^{-70}. In the second, accurate step, double-double-extended arithmetic, based on variants of the *DblMult*, is used. This yields an absolute round-off error less than 2^{-130}.

Reconstruction of the value of $\log_2$ out of the polynomial approximation and the table values is performed in double-double-extended arithmetic in both steps. The value of $\log_2 x$ is returned in three registers E, ℓ_{hi} and ℓ_{lo}. In the first step, a modified version of the Fast2Sum algorithm is used that ensures that ℓ_{hi} is written only on 53 bits (double precision).

4.2.2 Exponential

The following argument reduction is used for the exponential 2^t:

$$\begin{aligned} 2^t &= 2^M \cdot 2^{t-M} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{t-\left(M+i_1 \cdot 2^{-7}+i_2 \cdot 2^{-14}\right)} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u-\Delta} \\ &= 2^M \cdot exptbl_1[i_1] \cdot (1 + exptbl_2[i_2]) \cdot q(u) \cdot (1+\varepsilon) \end{aligned}$$

Herein the values M, i_1 and i_2 are integers. They are computed out of

$$t = n \cdot E + n \cdot \ell_{hi} + n \cdot \ell_{lo}$$

using the FMAs, shifts and Itanium's `getf` instruction giving the significand of a number:

$$\begin{aligned} s &= RN_e\left(n \cdot \ell_{hi} + \left(2^{49}+2^{48}\right)\right) \\ a &= RN_e\left(s - \left(2^{49}+2^{48}\right)\right) = \left\lfloor n \cdot \ell_{hi} \cdot 2^{14} \right\rceil \\ b &= RN_e\left(n \cdot \ell_{hi} - a\right) \\ u &= RN_e\left(n \cdot \ell_{lo} + b\right) \\ k &= \texttt{getf}(s) \\ M &= k \div 2^{14} \\ i_1 &= \left(k \div 2^7\right) \bmod 2^7 \\ i_2 &= k \bmod 2^7 \end{aligned}$$

In that sequence, all floating-point operations but those producing s and u are exact by Sterbenz' lemma [28]. The error in s is compensated in the following operations; actually, it is b. The absolute error Δ the value u is affected of is bounded by 2^{-78} because u is upper-bounded by 2^{-15}.

For approximating 2^u for $u \in [-2^{-15}; 2^{-15}]$, a polynomial of degree 3 is used in the first step and a polynomial of degree 6 is the second, accurate step. The polynomials provide a relative accuracy of $2^{-62.08}$ respectively $2^{-118.5}$.

The table values $exptbl_1[i_1]$ and $exptbl_2[i_2]$ are all stored as double-double-extended numbers. Only the higher parts are read in the first step. The second step reuses these higher parts and reads the lower parts. The reconstruction is performed with double-extended precision multiplications in the first step and with *DblMult* in the second step.

The first step delivers the final result $2^{n\cdot\log_2 x}\cdot(1+\varepsilon_1)$ as two floating-point numbers r_{hi} and r_{lo}. The value r_{hi} is a double precision number; r_{lo} hence a round-off error estimate of rounding x^n to double precision.

In the second step, the exponential $2^{n\cdot\ell}$ is corrected by

$$2^{\delta''} = 2^{n\cdot\left(E+i_1\cdot 2^7+i_2\cdot 2^{14}+u\right)-n\cdot(E+\ell')} = 2^{\delta-\Delta},$$

where ℓ' is a the accurate approximation of the logarithm. The correction approximates first $\delta'' = n\cdot\left(E+i_1\cdot 2^7+i_2\cdot 2^{14}+u\right)-n\cdot(E+\ell')$ up to 58 bits and then uses a polynomial of degree 1 for approximation the correction $2^{\delta''}$. The final result is delivered as double-double-extended value.

The function x^n has some arguments for which it is equal to the midpoint of two consecutive double precision numbers. An example is 9^{17}. For rounding correctly in that case, the ties-to-even rule must be followed. The final rounding after the second accurate approximation step must hence distinguish the two cases. The separation is easy because the worst cases of the function are known: if and only if the approximation is nearer to a midpoint than the worst case of the function, the infinitely exact value of the function is a midpoint. See [17] for details on the technique.

4.2.3 Complete error bounds

A complete, perhaps formal proof of the error bounds for the two steps of the algorithm goes beyond the scope of this article. Following the approach presented in [10, 8], the Gappa tool can be used for this task. Bounds on approximation errors can safely be certified using approaches found in [6]. Let us give just the general scheme of the error bound computation and proof for the first step of the presented algorithm.

We are going to use the following notations:

- $E+\ell = E+\ell_{hi}+\ell_{lo}$ stands for the approximation to the logarithm $\log_2 x$,
- δ is the associated total absolute error,
- δ_{table}, δ_{approx}, δ_{eval} and $\delta_{reconstr}$ are the absolute errors of the tables, the approximation, the evaluation and reconstruction of the logarithm.
- $r_{hi}+r_{lo}$ stands for the approximation to $x^n = 2^{n\cdot\log_2 x}$,
- $\varepsilon_{firststep}$ is the associated total relative error,
- ε_1 the total relative error due only to the approximation of the exponential without the error of the logarithm,
- ε_{table}, ε_{approx}, ε_{eval} and $\varepsilon_{reconstr}$ are the relative errors of the tables, the approximation, the evaluation and reconstruction of the exponential, and
- Δ stands for the absolute error the reduced argument u of the exponential is affected with.

The following error bound can hence be given

$$\begin{aligned} r_{hi} + r_{lo} &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot p(u) \cdot \\ &\quad \cdot (1 + \varepsilon_{reconstr}) \cdot (1 + \varepsilon_{table}) \cdot (1 + \varepsilon_{eval}) \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u-\Delta} \\ &\quad \cdot 2^{\Delta} \cdot (1 + \varepsilon_{reconstr}) \cdot (1 + \varepsilon_{table}) \cdot (1 + \varepsilon_{eval}) \cdot (1 + \varepsilon_{approx}) \\ &= 2^{n \cdot (E + \ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1) \end{aligned}$$

Herein, ε_1 is bounded by

$$|\varepsilon_1| \leq \varepsilon_{reconstr} + \varepsilon_{table} + \varepsilon_{eval} + \varepsilon_{approx} + 2 \cdot \Delta + \mathcal{O}\left(\varepsilon^2\right)$$

With $|\varepsilon_{reconstr}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{table}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{eval}| \leq 4 \cdot 2^{-64}$, $|\varepsilon_{approx}| \leq 2^{-62.08}$ and $|\Delta| \leq 2^{-78}$, this gives

$$|\varepsilon_1| \leq 2^{-60.5}.$$

Additionally, we obtain for $E + \ell_{hi} + \ell_{lo}$:

$$\begin{aligned} E + \ell_{hi} + \ell_{lo} &= E + logtblr_{hi}[m] + logtblr_{lo}[m] + p(z) + \delta_{eval} + \delta_{reconstr} \\ &= E + \log_2(r) + \log_2(1 + z) + \delta_{table} + \delta_{approx} + \delta_{eval} + \delta_{reconstr} \\ &= \log_2(x) + \delta_{table} + \delta_{approx} + \delta_{eval} + \delta_{reconstr} \\ &= \log_2(x) + \delta \end{aligned}$$

Since $|\delta_{approx}| \leq 2^{-69.49}$, $|\delta_{eval}| \leq -\log_2\left(1 - 2^{-8.886}\right) \cdot 3 \cdot 2^{-64} \leq 2^{-70.7}$, $|\delta_{table}| \leq 2^{-128}$ and $|\delta_{reconstr}| \leq 2^{-117}$, we get

$$|\delta| \leq 2^{-68.9}.$$

These bounds eventually yield to

$$\begin{aligned} r_{hi} + r_{lo} &= 2^{n \cdot (\ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1) \\ &= 2^{n \cdot \log_2(x)} \cdot 2^{n \cdot \delta} \cdot (1 + \varepsilon_1) \\ &= x^n \cdot (1 + \varepsilon_{firststep}) \end{aligned}$$

With $n \leq 735$, this gives

$$|\varepsilon_{firststep}| \leq 2^{735 \cdot 2^{-68.9}} \cdot \left(1 + 2^{-60.5}\right) - 1 \leq 2^{-59.17}.$$

For the second step, a similar error bound computation can be performed. One deduces that the overall relative error ε_2 of the second step is bounded by $|\varepsilon_2| \leq 2^{-116}$. This is sufficient for guaranteeing correct rounding for n up to 735.

5 Comparisons and tests

In this section, we report timings for the various algorithms described above. The algorithms have been implemented on the Intel/HP Itanium architecture, using the Intel ICC compiler [3]. We compare the following programs.

[3] We used ICC v10.1, on an Itanium 2 based computer.

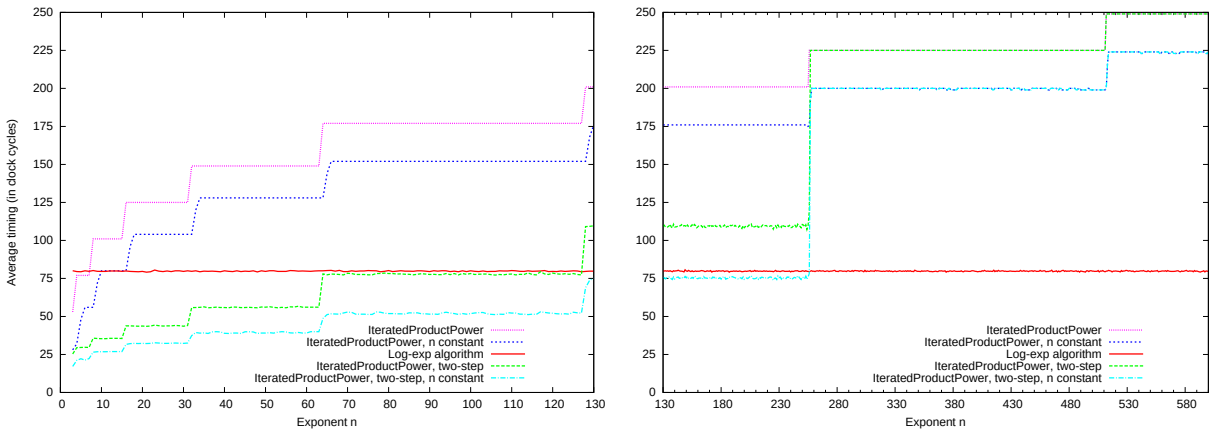

Figure 3: Average timings.

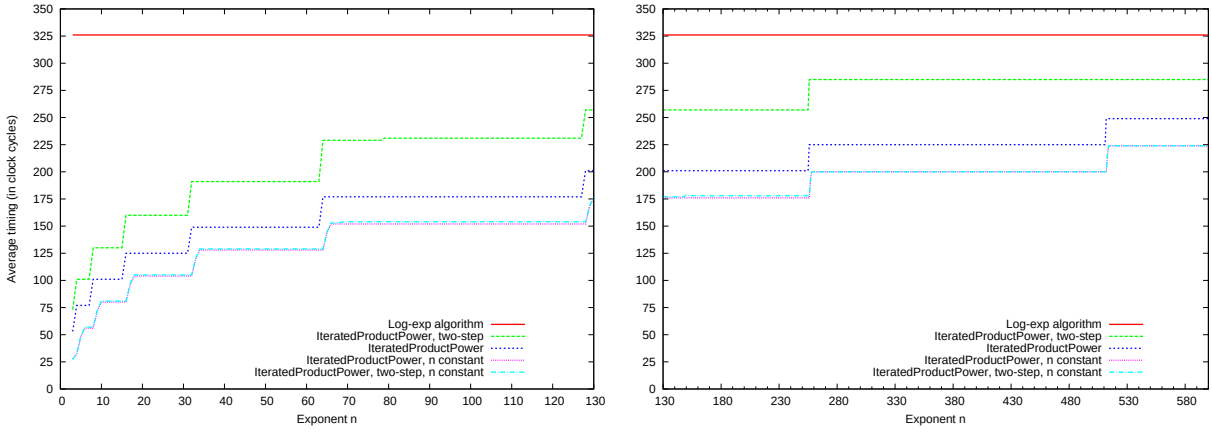

Figure 4: Worst timings.

- IteratedProductPower: our implementation follows Algorithm 5 strictly.
- Two-step IteratedProductPower: for the first, "fast" step we use Algorithm 6, and if needed, for the "accurate" step we use Algorithm 5 (see Subsection 3.5).
- IteratedProductPower and two-step IteratedProductPower with constant n: these implementations are the same as the two previous ones, except that the exponent n is fixed *a priori* (see Subsection 3.6). To take this information into account, a different function has been implemented for each exponent n considered. In the experiments reported hereafter, we consider exponents n ranging from 3 to 600: we have used a code generator to automatically generate the C code for these 571 functions. Since all the branches are resolved *a priori*, it also allows us to perform some obvious optimizations, without any (possibly costly) additional branch in the generated C code. For instance, at the first iteration of the loop in 5, since we know that $v = 0$, the statement $(u, v) := \mathit{DblMult}(u, v, u, v)$ is replaced by $(u, v) := \mathit{Fast2Mult}(u, v)$.
- Log-exp algorithm described in Section 4.

First we consider the average timings for computing x^n rounded to the nearest floating-point value. For each n from 3 to 600, we compute the average execution time for

16384 double-precision arguments randomly generated in the interval $[1, 2]$. We report on Figure 3 the average timings over these 16384 arguments with respect to n.

We also report on Figure 4 the worst case timings measured over the 16384 arguments tested. These worst case timings are mainly relevant for functions based on the two-step approach. Anyway we also recall on the figure the timing for the other functions for comparison purpose.

Table 5: Timings (in clock cycles) for tested functions.

n	Iterated Product Power	Two-step Iterated		Log-exp		Iterated (n fixed)	Two-step Iterated (n fixed)	
		avg case	wst case	avg case	wst case		avg case	wst case
3	53	25.3	73	80.1	326	28	17.1	27
4	77	29.5	101	79.5	326	32	21.1	33
5	77	29.6	101	79.4	326	48	22.2	49
6	77	29.7	101	80.0	326	56	21.3	57
7	77	29.6	101	79.7	326	56	22.3	57
8	101	35.7	130	80.2	326	56	26.5	57
9	101	35.5	130	79.7	326	72	26.8	73
10	101	35.4	130	79.6	326	80	26.8	81
15	101	35.5	130	79.7	326	80	26.9	81
16	125	43.8	160	79.9	326	80	31.6	81
17	125	43.8	160	79.6	326	96	32.1	97
18	125	43.7	160	79.5	326	104	32.3	105
31	125	44.0	160	80.0	326	104	32.5	105
32	149	55.8	191	79.7	326	104	37.4	105
33	149	55.9	191	80.0	326	120	39.4	121
34	149	56.0	191	79.4	326	128	39.0	129
63	149	56.1	191	80.0	326	128	40.0	129
64	177	78.0	229	80.0	326	128	48.7	129
65	177	77.4	229	80.2	326	144	51.2	145
66	177	77.9	229	80.3	326	152	51.7	153
127	177	77.3	231	79.4	326	152	52.2	154
128	201	109.1	257	79.6	326	152	68.9	154
129	201	109.3	257	79.8	326	168	73.8	169
130	201	109.6	257	79.8	326	176	75.2	177
255	201	109.8	257	79.6	326	176	76.1	178
256	225	119.5	285	79.4	326	176	78.2	178
257	225	225.0	285	79.8	326	192	192.0	192
258	225	225.0	285	80.1	326	200	200.0	200
511	225	225.0	285	79.4	326	200	199.0	200
512	249	249.0	285	79.8	326	200	200.0	200
513	249	249.0	285	79.9	326	216	216.0	216
514	249	249.0	285	80.3	326	224	224.0	224
600	249	249.0	285	79.6	326	224	224.0	224

Finally, the timings for some typical values of the exponent n are reported in Table 5: the timings for each "step" observed on the graphics of Figures 3 and 4 can be read in this table.

From these timings, we can see that in the worst cases, the implementations based on iterated products are always more efficient than the one based on the log-exp. Moreover, if we consider the average timings reported on Figure 3 and in Table 5, we can do the following observations.

- The average and worst case timings for the log-exp implementation are constant with respect to n, with an average execution time of about 80 clock cycles and a worst case execution time of 236 cycles.
- The straightforward (one-step) IteratedProductPower algorithm is more efficient on average than the log-exp only for $n \leq 9$
- The implementations of the two-step IteratedProductPower algorithm with n fixed are significantly more efficient than the log-exp approach as long as $n \leq 128$.

Conclusions

We have introduced several algorithms for computing x^n, where x is a double-precision floating-point number, and n is an integer, $3 \leq n \leq 626$. Our multiplication-based algorithms require the availability of a fused multiply-add (FMA) instruction, and a double-extended format for intermediate calculations.

After our experiments, the best choice, depending on the order of magnitude of the exponent n, on whether n is known at compile-time or not, and on whether one is interested in minimizing the average execution time or the worst case execution time:

- if one is interested in worst case performance, then Algorithm 5 (IteratedProductPower) is preferable (at least, up to $n = 626$, since we do not have a proof that our algorithms work for larger values);
- if one is interested in average case performance and n is *not* a constant then the two-step IteratedProductPower algorithm should be used for n less than around 60, and the log-exp algorithm should be used for larger values of n;
- if one is interested in average case performance and n is a constant then the "specialized" two-step IteratedProductPower algorithm should be used for n less than around 250, and the log-exp algorithm should be used for larger values of n.

References

[1] 754-R Committee. DRAFT standard for floating-point arithmetic p754 1.2.1. Technical report, September 2006. Available at `http://754r.ucbtest.org/nonabelian.com/754/754r.pdf`.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. New York, 1985.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic, ANSI/IEEE Standard 854-1987*. New York, 1987.

[4] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *J. ACM*, 23(2):242–251, 1976.

[5] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞ approximations. In *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Washington, DC, USA, 2007. IEEE Computer Society.

[6] S. Chevillard and Ch. Q. Lauter. A certified infinite norm for the implementation of elementary functions. In A. Mathur, W. E. Wong, and M. F. Lau, editors, *Proceedings of the Seventh International Conference on Quality Software*, pages 153–160, Portland, OR, 2007. IEEE Computer Society Press, Los Alamitos, CA.

[7] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-Based Systems*. Intel Press, Hillsboro, OR, 2002.

[8] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, Ch. Q. Lauter, and J.-M. Muller. Cr-libm, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arenaire team, Available at `https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf`, December 2006.

[9] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic*, Cape Cod, Massachussets, June 2005.

[10] F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In P. Langlois and S. Rump, editors, *Proceedings of the 21st Annual ACM Symposium on Applied Computing - MCMS Track*, volume 2, pages 1318–1322, Dijon, France, April 2006. Association for Computing Machinery, Inc. (ACM).

[11] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *RAIRO, Theoretical Informatics and Applications*, 41:85–102, 2007.

[12] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 41:85–102, 2007.

[13] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 3 1971.

[14] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations. Lecture Notes in Computer Science*, volume 235, pages 1–16. Springer-Verlag, Berlin, 1986.

[15] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison-Wesley, Reading, MA, 1998.

[16] Ch. Q. Lauter. A correctly rounded implementation of the exponential function on the Intel Itanium architecture. Research Report RR-5024, INRIA, November 2003. Available at `http://www.inria.fr/rrrt/rr-5024.html`.

[17] Ch. Q. Lauter and V. Lefèvre. An efficient rounding boundary test for `pow(x,y)` in double precision. Research Report RR2007-36, Laboratoire de l'Informatique du Parallélisme, Lyon, France, September 2007.

[18] V. Lefèvre. *Developments in Reliable Computing*, chapter An Algorithm That Computes a Lower Bound on the Distance Between a Segment and $\mathbb{Z}^2$, pages 203–212. Kluwer Academic Publishers, Dordrecht, 1999.

[19] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[20] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[21] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In Burgess and Ciminiera, editors, *Proc. of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. IEEE Computer Society Press, Los Alamitos, CA, 2001.

[22] R.-C. Li, P. Markstein, J.P. Okada, and J.W. Thomas. The libm library and floating-point arithmetic in HP-UX for Itanium 2. Technical report, Hewlett-Packard Company, 2002. `http://h21007.www2.hp.com/dspp/files/unprotected/libm.pdf`.

[23] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, Englewood Cliffs, NJ, 2000.

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

[25] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, 2nd edition, 2006.

[26] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[27] G. Revy. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Master's thesis, École Normale Supérieure de Lyon, 2006.

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

[29] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[30] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.