Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores

Claude-Pierre Jeannerod[1,2] Guillaume Revy[1,2]

[1]INRIA Grenoble - Rhône-Alpes (Arénaire project-team),
[2]Laboratoire LIP (Université de Lyon - UMR CNRS/ÉNSL/INRIA/UCBL 5668)
email: {Claude-Pierre.Jeannerod,Guillaume.Revy}@ens-lyon.fr
École normale supérieure de Lyon — 46, allée d'Italie, 69364 Lyon cedex 07, France

Abstract

This paper presents an optimized software implementation of the reciprocal square root function $x \mapsto x^{-1/2}$, for IEEE binary32 floating-point data and with correct rounding to nearest. The main feature of this implementation is high instruction-level parallelism (ILP) exposure, which results here from an extension of the bivariate polynomial evaluation-based method of [6] as well as from the design of a specific rounding procedure. This implementation proves to be very efficient for some VLIW processor cores like STMicroelectronics' ST231 (used mainly for embedded media processing), for which a low latency of 29 cycles has been measured.

Keywords: *reciprocal square root, binary floating-point arithmetic, correct rounding (to nearest), polynomial evaluation, software implementation, VLIW processor core.*

1. Introduction

Reciprocal square roots frequently appear in digital signal processing and scientific computing [10], and correctly-rounded implementations are recommended in the latest revision of the IEEE 754 standard [1, §9.2]. Our aim here is to present such an implementation, in software, for binary32 data and rounding to nearest even.

The targeted processors are the ST231 four-issue VLIW, 32-bit cores from STMicroelectronics, whose main features are as follows: 4 parallel ALUs, 2 parallel multipliers (giving the first or last 32 bits of a 32×32 integer product), a leading-zero counter, 64 general purpose registers and 8 condition registers, partial predication via select instructions, and encoding of immediate operands up to 32 bits.

In order to fully exploit the high degree of parallelism of our target and to avoid using coefficient tables, we extend to reciprocal square roots the high-ILP, polynomial-based square rooting method introduced in [6]. This extension, which is presented in Section 2, seems to allow for more ILP exposure than the Newton-like iterations used for example in [10, 7, 8]. Section 3 then gives some details about our implementation of this extension for the binary32 format, its validation, and the performances obtained on the ST231 core. In particular, a latency of 29 cycles has been measured, for rounding to nearest even and with subnormal number support.

A typical use of correctly-rounded reciprocal square roots is for 3D vector normalization $[x, y, z] \mapsto [x/w, y/w, z/w]$, with $w = \sqrt{x^2 + y^2 + z^2}$. In the context of the FLIP library,[1] our implementation allows to replace one square root (23 cycles) and three divisions (3×32 cycles) by one reciprocal square root (29 cycles) and three products (3×21 cycles). Both cases yield an error of at most 1 ulp but, assuming the sum of squares is available, the latter reduces latency by 9% if the coordinates $x/w, y/w, z/w$ are produced simultaneously, and by over 20% if they are produced sequentially.

2. Reciprocal square root algorithm

2.1. Special operands

For reciprocal square root, special operands are ± 0, $\pm\infty$, negative numbers, and NaNs. The corresponding special results required by the IEEE 754 standard [1] are shown in Table 1. In this table "< 0" represents $-\infty$ or any negative (nonzero) finite floating-point number, and "qNaN" means any quiet Not-a-Number having the same payload as the operand (that is, only the sign is not specified).

[1] `http://flip.gforge.inria.fr/`

Table 1. Special values of function `rsqrt`.

x	$+0$	$+\infty$	-0	< 0	NaN
`rsqrt`(x)	$+\infty$	$+0$	$-\infty$	qNaN	qNaN

Since here special operands are exactly the same as those of square root, we apply the method described in [6, § 4.1] to detect all of them simultaneously. Then, classically, the special results displayed in Table 1 are computed in parallel with the generic case (described next in Section 2.2), which dominates the cost.

In some implementations the reciprocal square root of -0 is $+\infty$ instead of $-\infty$. This is for example the case in GNU MPFR.[2] An advantage of such a choice is consistency with the specification of the `rootn` function $(x, n) \mapsto x^{1/n}$, for which the standard mandates that the value $+\infty$ be returned when x is -0 and n is even and negative.

2.2. Positive finite operands

When x is non special, it has the form $x = m \cdot 2^e$, where, for a binary floating-point system of precision p and extremal exponents $e_{\min}$ and $e_{\max}$,

$$m = (m_0.m_1 \dots m_{p-1})_2 \quad \text{and} \quad e_{\min} \le e \le e_{\max}.$$

More precisely, x is either such that $m_0 = 1$ (normal number) or such that $m_0 = 0$ and $e = e_{\min}$ (subnormal number). We assume as usual that $e_{\min}$ is even, $e_{\min} \le e_{\max}$, $e_{\max} = 1 - e_{\min}$, and $2 \le p \le e_{\max}$, all these constraints being satisfied by the standard binary formats of [1].

In both subnormal and normal cases one may check that, for example, $2^{e_{\min}} \le x^{-1/2} \le 2^{e_{\max}}$. This implies that the reciprocal square root of a non special number always lies in the range $[2^{e_{\min}}, (2-2^{1-p}) \cdot 2^{e_{\max}}]$ of positive normal numbers or, in other words, that neither underflow nor overflow can occur.

It follows that $x^{-1/2} = \ell \cdot 2^d$ for some real ℓ in $[1, 2]$ and some integer d such that $e_{\min} \le d < e_{\max}$. The correctly-rounded (to nearest even) value of $x^{-1/2}$ is thus given by

$$\mathrm{RN}(x^{-1/2}) = \mathrm{RN}(\ell) \cdot 2^d,$$

and, classically, $\mathrm{RN}(\ell)$ and d can be computed in parallel.

First, we provide explicit formulae for ℓ and d. Let λ be the number of leading zeros of m, and let $m' = m \cdot 2^\lambda$ and $e' = e - \lambda$. Let c be 1 if e' is even, 0 otherwise. Then one may check that

$$\ell = s\sqrt{2/(1+t)} \quad \text{and} \quad d = \lfloor -(e'+1)/2 \rfloor, \qquad (1)$$

where $s = 2^{c/2}$ and $t = m' - 1$, and with $\lfloor \cdot \rfloor$ denoting the usual floor function.

[2] `http://www.mpfr.org/mpfr-current/`; see also [4, 11].

Second, the above formula for d being already suitable for implementation, we focus on the computation of $\mathrm{RN}(\ell)$. As in [6] we proceed by correcting "one-sided truncated approximations" similar to those of [3, p. 459]: ℓ is approximated from above by v to precision p. Then v is truncated after p fraction bits into a number u. Finally $\mathrm{RN}(\ell)$ is produced by the following rounding algorithm:

if $u \ge \ell$ then
    return v truncated after $p-1$ fraction bits
else
    return $v + 2^{-p}$ truncated after $p-1$ fraction bits

This rounding algorithm is not specific to reciprocal square root and we initially used it in [6] for square root. The correctness proof given in [6, §3.2.1] only uses the fact below.

Fact 1. *ℓ cannot be exactly halfway between two consecutive floating-point numbers.*

In the case of reciprocal square root, this fact is well known to be true (see for example [5, Table 1]) and can be shown as follows.

Proof. Assume ℓ is halfway between two consecutive floating-point numbers. Then $\ell \ne 2$ and, since $\ell \in [1, 2]$, its binary expansion must have the form $1.\ell_1 \dots \ell_{p-1}1$. Consequently, $\ell = (2L+1) \cdot 2^{-p}$ for some integer L such that $2^{p-1} \le L < 2^p$. Similarly, $m' = K \cdot 2^{1-p}$ for some integer K such that $2^{p-1} \le K < 2^p$. The first identity in (1) thus gives $K(2L+1)^2 = 2^{3p+c}$, whose left-hand side has an odd factor and whose right-hand side is an integer power of two. Hence a contradiction and the proof follows. □

The main difficulties of the above approach are to compute v as fast as possible, and to evaluate the condition $u \ge \ell$ in order to decide whether u should be corrected or not:

- To maximize ILP exposure, we compute v by the method of [6], that is, as the value (up to rounding errors) of a bivariate polynomial

  $$P(s, t) = 2^{-p-1} + s \cdot a(t) \qquad (2)$$

  such that $a(t)$ is a "good enough" polynomial approximation of $\sqrt{2/(1+t)}$ over $[0, 1)$.

- To decide whether $u \ge \ell$ or not, we evaluate the equivalent condition

  $$(1+t)u^2 \ge 2s^2, \qquad (3)$$

  whose both sides now have finite binary expansions.

3. Implementation for the binary32 format

The above algorithm has been implemented in C99 for the binary32 format of [1], where $p = 24$ and $e_{\min} = -126$. The lines of code for handling special operands, computing d, and evaluating the rounding condition (3) have been written and optimized by hand. However, the polynomial $a(t) = \sum_i a_i t^i$ has been computed as a truncated Remez approximant using Sollya,[3] and a parallel and accurate evaluation code for v has been automatically written by a generator under development called CGPE (Code Generation for Polynomial Evaluation [9]).

For each of these subtasks the goal is to achieve low latency by exposing as much ILP as possible. Sections 3.1 and 3.2 illustrate this for, respectively, special-operand handling and polynomial evaluation. Section 3.3 further details our optimized implementation of the evaluation of condition (3). Section 3.4 concludes with the validation of the resulting code and with performance results obtained on the ST231 core.

3.1. Implementation of two specifications for special operands

Our C implementation of the first specification (Table 1) is given below. Here and hereafter `X` is the 32-bit unsigned integer that carries the standard encoding of the binary32 floating-point operand x. (Also, unless otherwise specified the variables of our C codes are always of type `uint32_t`.)

```
1 if ((X + 0xFFFFFFFF) >= 0x7F7FFFFF)
2 {
3    if (X <= 0x7F800000 || X == 0x80000000)
4       return X ^ 0x7F800000; // +-inf or +0
5    return X | 0x7FC00000;    // qNaN having the
6                              // same payload as X
7 }
```

Line 1 allows to filter out all special inputs, while line 3 further isolates inputs of the form ± 0 or $+\infty$; these two conditions are exactly the same as for square root in [6, § 4.1]. The only difference is then the bitwise XOR of `X` and the hexadecimal constant `0x7F800000`, giving an overhead of only one instruction (involving an extended immediate).

To implement the second specification of Section 2.1, it suffices to replace line 4 in the code above with:

```
4       return ~X & 0x7F800000;  // +inf or +0
```

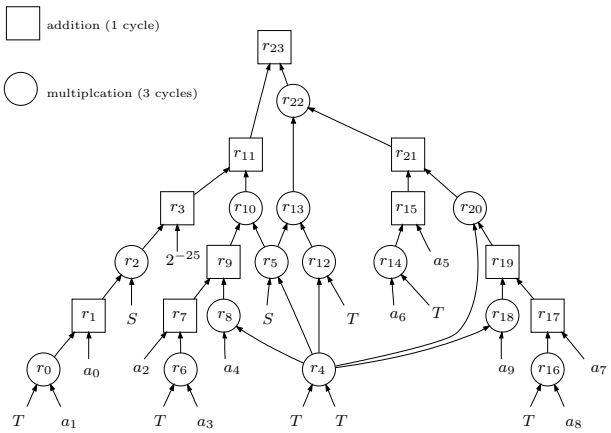
The bitwise negation of `X` allows in particular to exchange the exponent fields of zero and infinity. Masking then sets both the sign bit and the fraction field to zero.

[3] `http://sollya.gforge.inria.fr/`

On ST231, both specifications can be implemented at exactly the same cost using the codes above. This comes from the availability of an instruction `andc` that complements the input and then applies a bitwise AND, and which has the same latency (1 cycle) as the bitwise XOR instruction.

3.2. Generating an optimized code for polynomial evaluation

The polynomial $a(t)$ used has degree 9, and CGPE found the following scheme for evaluating $P(s,t)$ in (2):



In this scheme, all the variables are 32-bit unsigned integers, and addition and multiplication respectively mean $(A \pm B) \bmod 2^{32}$ and $\texttt{mul}(A,B) := \lfloor AB/2^{32} \rfloor$ for two such integers A and B. On the ST231, an addition takes 1 cycle and a multiplication takes 3 cycles. Furthermore, it turns out that the above parenthesization can be evaluated in 14 cycles compared to 38 cycles using Horner's rule. (One of the a_i's is a power of two, which allows to trade off one multiplication against one shift, and eventually to save two cycles. Hence 38 cycles instead of 40 cycles.) The accuracy (rounding errors) of this evaluation scheme has been checked using interval arithmetic from Gappa.[4]

3.3. Implementing the condition for correct rounding

Assume that t and u are exactly represented by the k-bit unsigned integers $T = t \cdot 2^k$ and $U = u \cdot 2^{k-2}$, respectively. To represent $1 + t$ exactly, we define a third k-bit unsigned integer $Y = (1+t) \cdot 2^{k-1}$. It follows that $Y = 2^{k-1} + T/2$, and condition (3) thus reads

$$YU^2 \geq Z, \qquad Z = 2^{3k-4+c}, \qquad c \in \{0,1\}.$$

Note that YU^2 is a $3k$-bit positive integer that can be seen as a $Q5.(3k-5)$ number. However, since Z is a large power of

[4] `http://gappa.gforge.inria.fr/` and [2].

two, evaluating the condition $YU^2 \geq Z$ does not require the knowledge of the full product YU^2. Indeed, the property below shows that it suffices to evaluate $P \geq Q$, where P and Q are two k-bit unsigned integers.

Property 1. *Let $P = \lfloor YU^2 \cdot 2^{-2k} \rfloor$ and $Q = 2^{k-4+c}$. One has $YU^2 \geq Z$ if and only if $P \geq Q$.*

Proof. One has $Z = Q \cdot 2^{2k}$. From $P \leq YU^2 \cdot 2^{-2k}$ it follows that $P \geq Q$ implies $YU^2 \geq Z$. Conversely, $P < Q$ implies $P \leq Q - 1$ and, using $YU^2 \cdot 2^{-2k} - 1 < P$, it follows that $YU^2 \cdot 2^{-2k} < Q$, that is, $YU^2 < Z$. □

It remains to evaluate condition "$P \geq Q$" as efficiently as possible in our context. If $k = 32$ then Q equals 2^{28+c} and can be implemented simply by shifting left by c:

```
Q = 0x10000000 << c;
```

To implement the computation of $P = \lfloor YU^2 \cdot 2^{-2k} \rfloor$, we may proceed as follows. Let A_{hi} and A_{lo} be the two k-bit, positive integers such that $U^2 = A_{hi} \cdot 2^k + A_{lo}$. Then $YU^2 = YA_{hi} \cdot 2^k + YA_{lo}$ and, defining B_{hi}, B_{lo}, C_{hi}, C_{lo} as the k-bit positive integers such that $YA_{hi} = C_{hi} \cdot 2^k + C_{lo}$ and $YA_{lo} = B_{hi} \cdot 2^k + B_{lo}$, we obtain

$$YU^2 = C_{hi} \cdot 2^{2k} + (B_{hi} + C_{lo}) \cdot 2^k + B_{lo}.$$

Consequently,

$$P = C_{hi} + \big(\text{carry generated by } B_{hi} + C_{lo}\big). \qquad (4)$$

(The carry is 1 if addition overflows, 0 otherwise.) This is illustrated by Figure 1, where $D_{hi} = P$ and $D_{lo} = B_{lo}$.

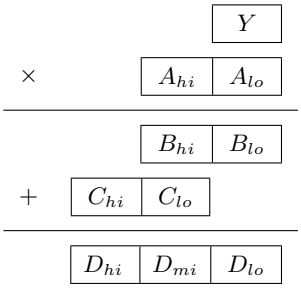


Figure 1. Decomposing the exact value of the product YU^2 into three chunks of k bits each.

In particular, the lower half B_{lo} of the product $A_{lo}Y$ is not needed, nor is the middle term D_{mi}. All we must know is whether adding B_{hi} to C_{lo} overflows or not. The property below shows how to detect this.

Property 2. *The carry generated by the sum $B_{hi} + C_{lo}$ is equal to 1 if and only if $(B_{hi} + C_{lo}) \bmod 2^k < B_{hi}$.*

Proof. Recall that $B_{hi}, C_{lo} \in \{0, 1, \ldots, 2^k - 1\}$, and let Q and R be the unique non-negative integers such that

$$B_{hi} + C_{lo} = Q \cdot 2^k + R \quad \text{and} \quad 0 \leq R < 2^k. \qquad (5)$$

Since $B_{hi} + C_{lo} < 2^{k+1}$, Q is either 0 or 1. Hence the sum overflows if and only if $Q = 1$. If $Q = 1$ then $R = B_{hi} + C_{lo} - 2^k$ and, since $C_{lo} < 2^k$, we obtain $R < B_{hi}$. Conversely, let us show that $R < B_{hi}$ implies $Q = 1$. Form (5), $R < B_{hi}$ implies $C_{lo} < Q \cdot 2^k$. Since $C_{lo} \geq 0$ and $Q \in \{0, 1\}$, we deduce that $Q = 1$. □

Combining (4) and Property 2 leads to the following implementation for computing P.

```
1 A_hi = mul(U, U);       A_lo = U * U;
2
3
4 B_hi = mul(Y, A_lo);    C_lo = Y * A_hi;
5                         C_hi = mul(Y, A_hi);
6
7 S = B_hi + C_lo;
8 carry = S < B_hi;
9 P = C_hi + carry;
```

Here, all variables are 32-bit unsigned integers. Furthermore, we assume available `+` and `mul` defined in Section 3.2, and as well as $\star$ such that $A \star B := AB \bmod 2^{32}$ for two 32-bit unsigned integers A and B. On ST231 cores, $<$ has the same latency as $+$ (1 cycle) and $\star$ has the same latency as `mul` (3 cycles). Therefore, the above code yields P in 8 instructions and 9 cycles.

To sum up, we have shown so far that "$P \geq Q$" can be evaluated in 10 cycles. Since in (4) getting the carry is 1-cycle more expensive than getting C_{hi} (and, of course, than getting Q), we may try to evaluate instead the equivalent condition "$\texttt{carry} \geq Q - C_{hi}$." Unfortunately, how to implement this comparison in fewer than 10 cycles is unclear.

However, on ST231 one can save 1 cycle by using the operation `addcg` which performs integer addition with carry propagation. More precisely, `addcg` takes two 32-bit unsigned integers A, B and one bit c_{in} (carry in) as inputs, and produces as outputs the 32-bit unsigned integer C and the bit c_{out} (carry out) such that

$$A + B + c_{in} = c_{out} \cdot 2^{32} + C.$$

From Figure 1, we can thus obtain $P = D_{hi}$ as follows:

- $(D_{mi}, c_{out}) = \texttt{addcg}(B_{hi}, C_{lo}, 0)$,
- $(P, *) = \texttt{addcg}(0, C_{hi}, c_{out})$.

Notice that the carry $*$ produced by the second call to the `addcg` instruction is in fact known to be zero.

A first way of implementing this computation of P is to use the intrinsic function `__st200addcg`, whose operands

are three 32-bit unsigned integers A, B, c_{in}, and whose output is the 64-bit unsigned integer $c_{out} \cdot 2^{32} + C$. Hence, the lines 7, 8, 9 of the previous code can be replaced with:

```
7 uint64_t S = __st200addcg(B_hi, C_lo, 0);
8 P = (uint32_t) __st200addcg(0, C_hi, S >> 32);
```

The assembly code generated by the ST200 compiler (-O3, ST231) on the above piece of code then looks like

```
convib $b0 = $r0;;
addcg $r0, $b0 = $r17, $r20, $b0;;
addcg $r8, $b0 = $r0, $r21, $b0;;
```

where reading `$r0` returns 0, and where `$r8` is a 32-bit register containing the value of P. Since the latency of `addcg` is of 1 cycle, we finally get P in 8 cycles instead of 9. The main drawback of this approach is the explicit use of an intrinsic function, that makes the C code ST231-dependent.

A second way of getting P in 8 cycles through portable C code is as follows. The above `addcg`-based approach computes D_{hi} and D_{mi}, that is, performs the addition $B_{hi} + C$ exactly, where $C = Y A_{hi}$ is a full, 64-bit product. Hence P can be computed as follows, with $\star$ now denoting the lower half of the exact product of two 64-bit unsigned integers:

```
1 A_hi = mul(U, U);       A_lo = U * U;
2
3
4 B_hi = mul(Y, A_lo);    C = (uint64_t) Y
5                           * (uint64_t) A_hi;
6
7 S = B_hi + C; // S and C are of type uint64_t
8 P = S >> 32;
```

Interestingly, the assembly generated by the ST200 compiler for this portable C code contains exactly the sequence of two `addcg` shown above, thus yielding P in 8 cycles. This code is the one we have kept for our implementation.

3.4. Validation and performances

Our implementation, called `rsqrt`, has been compared exhaustively to the power functions of the glibc and MPFR.

We have also compiled it with the ST200 VLIW compiler, in -O3 and for the ST231 core. Without subnormal support, the latency of the generated assembly code is of 28 cycles. For comparison, the previously best available code for the ST231 was an implementation of Goldschmidt's method with initial approximation by a degree-3 polynomial [8, §12] and has a latency of 56 cycles. Our approach is thus twice faster. Also, our code offers full subnormal support at the cost of only 1 extra cycle, the latency then being of 29 cycles.

Finally, the table below shows the advantage of using our specialized operator `rsqrt` rather than simply compounding division/inversion and square root. (Brackets contain the results obtained when subnormals are not supported.)

Code sequence used for computing $x^{-1/2}$	Number N of instructions	Latency L (cycles)	N/L
`div(1.0f,sqrt(x))`	164 [141]	57 [48]	2.9 [2.9]
`inv(sqrt(x))`	123 [110]	48 [43]	2.6 [2.6]
`rsqrt(x)`	68 [63]	29 [28]	2.3 [2.2]

Although each of the operators `div`, `inv`, and `sqrt` used here is highly optimized for the ST231, full specialization yields significantly smaller and faster codes. In fact, such codes are also more accurate since only one rounding error occurs instead of two.

Acknowledgements

This research was supported by "Pôle de compétitivité mondial" Minalogic and by the ANR project EVA-Flo. Many thanks to Christophe Monat and Philippe Théveny for useful discussions and comments on a draft of this paper.

References

[1] IEEE standard for floating-point arithmetic. IEEE Std. 754-2008, pp.1-58, Aug. 29 2008.

[2] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1), 2009.

[3] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007.

[5] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240, 1999.

[6] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, Oct. 2008.

[7] J.-A. Piñeiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root and inverse square root. *IEEE TC*, 51(12):1377–1388, 2002.

[8] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, ÉNS Lyon, France, 2006.

[9] G. Revy. CGPE - Code Generation for Polynomial Evaluation. `http://cgpe.gforge.inria.fr/`.

[10] M. J. Schulte and K. E. Wires. High-speed inverse square roots. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 124–131, 1999.

[11] P. Zimmermann. Implementation of the reciprocal square root in MPFR. In A. Cuyt, W. Krämer, W. Luther, and P. Markstein, editors, *Numerical Validation in Current Hardware Architectures*, number 08021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008.