ensl-00391185, version 1 - 3 Jun 2009

Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores (Extended Summary) - *LIP Research Report 2009-21*

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

[1]INRIA Rhône-Alpes (Arénaire project-team), [2]Université de Lyon

email: {Claude-Pierre.Jeannerod,Guillaume.Revy}@ens-lyon.fr

Laboratoire LIP (CNRS/ÉNSL/INRIA/UCBL)

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 polynomial evaluation-based method of [3] 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), where a low latency of 29 cycles has been measured.

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

1. Introduction

Reciprocal square roots frequently appear in digital signal processing and scientific computing [6], and correctly-rounded implementations are recommended in the latest revision of the IEEE 754 standard [1]. Our aim here is to present such an implementation, in software, for binary32 data (formerly called "single precision") and rounding to nearest even. The targeted processors are the ST231 four-issue VLIW, 32-bit cores from STMicroelectronics, whose main features are: 4 parallel ALUs, 2 parallel multipliers (giving the first or last 32 bits of a 32×32 product), a leading zero counter, 64 general purpose registers and 8 condition registers, partial predication through 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 [3]. This extension, which is presented in Section 2, seems to allow for more ILP exposure than the Newton-like iterations used for example in [6, 4, 5]. 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.

Application. 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 the latter reduces latency by over 20%.

2. Reciprocal square root algorithm

Special operands. Operands like ± 0, $\pm\infty$, negative numbers, and NaNs are filtered out as in [3]. Then the special results required by the standard [1] are computed in parallel with the generic case described next, which dominates the cost.

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 $p \geq 2$ and extremal exponents $e_{\min}$ and $e_{\max} = 1 - e_{\min}$,

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

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

ensl-00391185, version 1 - 3 Jun 2009

In this case, we can show that $x^{-1/2}$ always falls in the normal range and that it can not be exactly halfway between two consecutive floating-point numbers. (Therefore, neither under/overflow nor rounding ties can occur, which makes the implementation simpler and faster.) 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, RN(ℓ) and d are 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

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

where $s = 2^{c/2}$ and $t = m' - 1$.

Second, the above formula for d being already suitable for implementation, we focus on the computation of RN(ℓ). As in [3] we proceed by correcting “one-sided truncated approximations” [2]: ℓ is approximated from above by v to precision p. Then v is truncated after p fraction bits into a number u. Finally RN(ℓ) is obtained by adding a small correction to u and truncation after $p - 1$ fraction bits. The main difficulties 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 as the value (up to rounding errors) of a bivariate polynomial

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

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

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

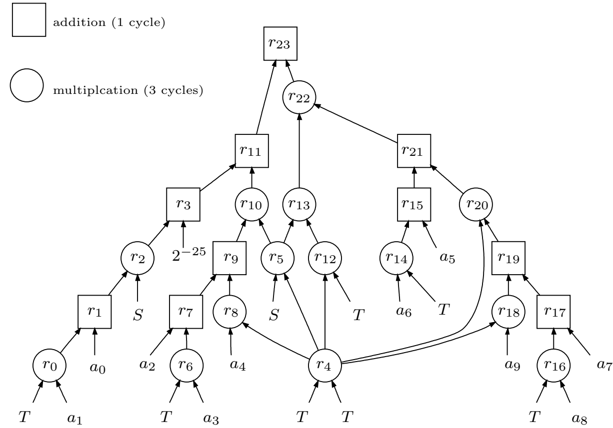
whose both sides now have finite binary expansions.

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 correcting u 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,[2] and a parallel and accurate evaluation code for v has been written automatically by a generator under development.

Generating polynomial evaluation codes. The polynomial $a(t)$ used has degree 9 and our generator found the following scheme for evaluating $P(s, t)$ in (1):

[2] http://sollya.gforge.inria.fr/



Its latency is of 14 cycles for the ST231 (compared to 38 cycles using Horner’s rule). Its accuracy (rounding errors) has been checked by Gappa.[3]

Implementing the condition for correct-rounding. Three 32-bit words are needed for representing the left hand side of (2). However, we can show that the first 64 bits are enough for evaluating (2) exactly, thus reducing the overall latency of the rounding step.

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 [5, §12] and had a latency of 67 cycles. Our approach is thus more than 2.3 times faster. Also, our code offers full subnormal support at the cost of only 1 extra cycle, since the latency then is 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 indicate that 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))`	147 [124]	53 [47]	2.7 [2.6]
`inv(sqrt(x))`	121 [115]	49 [47]	2.5 [2.4]
`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.

[3] http://lipforge.ens-lyon.fr/www/gappa/

References

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

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

[3] 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.

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

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

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

ensl-00391185, version 1 - 3 Jun 2009