Faster floating-point square root for integer processors

Claude-Pierre Jeannerod*, Hervé Knochel†, Christophe Monat†, Member, IEEE, and Guillaume Revy*
* Laboratoire LIP (CNRS, ENSL, INRIA, UCBL)
École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France
Email: {Claude-Pierre.Jeannerod, Guillaume.Revy}@ens-lyon.fr
†STMicroelectronics, 12 rue Jules Horowitz, 38019 Grenoble cedex 1, France
Email: {Herve.Knochel,Christophe.Monat}@st.com

***Abstract*— This paper presents some work in progress on fast and accurate floating-point arithmetic software for ST200-based embedded systems. We show how to use some key architectural features to design codes that achieve correct rounding-to-nearest without sacrificing for efficiency. This is illustrated with the square root function, whose implementation given here is faster by over 35% than the previously best one for such systems.**

I. Introduction

The STMicroelectronics ST200 is an embedded media processor derived from the Lx technology platform [7], designed to implement advanced audio and video codecs in consumer devices such as set-top boxes for HD-IPTV (High Definition Internet Protocol Television), cell phones, wireless terminals, and PDAs. The integration of several instances of the ST200 in large Systems On Chip (SOC) provides an unprecedented level of media processing ability, enabling for instance single chip H.264 solutions.

The processing power brought by these media processors allows to replace dedicated hardware by software, thus giving more flexibility to the design while sustaining a high performance level.

For instance, complex audio decoders or graphics pipeline fragments are now fully implemented in software. One of the interesting characteristics of these applications is that they are highly demanding on floating-point square root computations, mostly for vector normalization purposes. However, this kind of situation is not so frequent in actual code: application developers have traditionally avoided square root functions (as they did for the division operator) because of its high real or perceived computational cost. Instead, they have devised "square root free" algorithms or designed ad-hoc square root implementations.

We think that this burden could be removed from the application developer by providing a square root implementation that is both very fast and, as prescribed by the IEEE-754 standard [1], correctly rounded.

Motivated by the above applications, we shall focus in this paper on correct rounding-to-nearest and normalized single-precision floating-point numbers. The input argument x to square root is a *normalized single-precision floating-point number* [19, §4] when it has the form

$$x = m \cdot 2^e, \tag{1}$$

with e an integer between -126 and 127 and m a rational number having binary expansion $\pm 1.f_1 f_2 \dots f_{23}$. For $x > 0$, the real number $\sqrt{x}$ thus has the form

$$\sqrt{x} = \ell \cdot 2^d,$$

where $d = \lfloor e/2 \rfloor$, $\ell = t\sqrt{m}$ and $t = 1$ or $\sqrt{2}$ depending on the parity of e. Although d lies in the range of e, the binary expansion

$$1.\ell_1 \ell_2 \dots \ell_{23} \ell_{24} \dots \tag{2}$$

of ℓ is in general nonterminating, unlike that of m, and the exact value $\sqrt{x}$ may eventually not fit in format (1). Instead, we output a correctly-rounded value of $\sqrt{x}$; under *round-to-nearest* rounding mode, it is the unique normalized single-precision number r that is closest to $\sqrt{x}$. In fact, obtaining r from x essentially reduces to computing the exact values of the bits $\ell_1, \dots, \ell_{24}$ in (2) above (see for example [6, §8.6.3] and Subsection III-A).

Several efficient software implementations of correctly-rounded square roots have already been described for processors with or without floating-point units. On HP/Intel's Itanium the algorithms use Newton-Raphson's iteration ([4],[14, §9.1.1],[9], [3, p.238]). The same method had been used earlier for IBM's RS/6000 [16]. On ST231, the first implementation has been given in [20, §11]; there, the initial approximation is refined by a single iteration. On IBM's Power3, Newton-Raphson's method has been replaced in [21] by another algorithm, better suited to the PowerPC architecture; that faster algorithm essentially refines an initial approximation of $\sqrt{x}$ by multiplying it with a well-chosen polynomial approximation of $\sqrt{1-y}$ for some $y \simeq 0$.

All those implementations rely on the same approach: compute an initial approximation to square root or square root reciprocal, and refine it by one or two iterations, typically Newton-Raphson's. The initial approximation is obtained either by table lookup (RS/6000, Power3) or by calling an instruction that approximates the square root reciprocal to about 8 bits (`frsqrta` instruction on IA-64), or by evaluating small degree polynomial approximants (two degree-3 polynomials for ST231, see [20, p.113]).

In this paper, we propose for the ST231 some new implementations: they are based exclusively on the evaluation of

polynomial approximants and thus avoid Newton-like iterative refinements. Although the polynomials we use have degree 5 or 6, they can be evaluated very fast and accurately thanks to some key architectural features of the ST231. As a result, the square root latency of [20] has been reduced by over 35%.

Notice that, for IA-64, a similar approach is used in [11] and [9], but for transcendental functions. Also, [12] studies floating-point software support for Intel's XScale processor which, like the ST231, has no floating-point unit.

The paper is organized as follows. Section II describes some features of the ST231 architecture and compiler that have been crucial for speeding up square root. Section III outlines our square root implementation and reports experimental results. Finally Section IV gives concluding remarks and future plans.

II. Overview of the ST231 processor and compiler

A. Architectural features

The ST231 is a 32-bit, four-issue member of the ST200 VLIW core family. VLIW (Very Long Instruction Word) [8] processors use an architectural technique where instruction level parallelism (ILP) is explicitly exposed to the compiler.

RISC-like operations (syllables) are grouped into bundles (wide words). The operations in a bundle are issued simultaneously and complete simultaneously. While the delay between issue and completion is the same for all operations, some results are available for bypassing to subsequent operations prior to completion.

A hardware implementation of a VLIW is significantly simpler than a corresponding multiple issue superscalar CPU. This is due principally to the simplification of the operation grouping and scheduling hardware. This complexity is moved to the ILP extractor (compiler) and the instruction scheduling system (compiler and assembler) in the software toolchain.

The ST200 family of cores uses a scalable technology that allows variation in instruction issue width, the number and capabilities of functional units and register files, and the instruction set.

The ST200 family includes the following features:

1) parallel execution units, including multiple integer ALUs and 32x32 bit multipliers,
2) a large register file of 64 general purpose registers and 8 condition registers,
3) predicated execution through select operations,
4) efficient branch architecture with multiple condition registers,
5) encoding of immediate operands up to 32 bits.

All these features are key to the square root function implementation, that uses efficiently all the resources exposed by the machine.

B. Compiler

The ST231 VLIW compiler is based on the Open64 technology (www.open64.net) retargeted for the ST200 processor family by STMicroelectronics since 2001. The compiler has been improved to support development for high performance embedded targets. Most important to the square root code efficiency are the if-conversion optimization [2] and the Linear Assembly Optimizer (LAO) [5]. The if-conversion optimization generates mostly straight-line code by emitting efficient sequences of `select` instructions instead of costly control flow. The LAO generates a schedule for the instructions that is very close to the optimal.

III. Square root implementation

As usual for IEEE-754 arithmetic [19, §4], a number x as in (1) is stored in a 32-bit register $R = [R_{31} \dots R_0]$ as follows: R_{31} contains the sign bit, $R_{30}, \dots, R_{23}$ contain the 8 bits of the biased exponent $e + 127$, and the last 23 bits of R contain the fraction bits $f_1, \dots, f_{23}$. The correctly-rounded value r of $\sqrt{x}$ will be returned using the same three-field representation. This representation further allows to store special values like ± 0, $\pm\infty$ and NaN (Not a Number).

A. General algorithm outline

Using the above machine representation of single-precision floating-point numbers, we get the following steps for computing a square root with correct rounding:

1) Unpack x into three fields (sign, exponent, fraction)
2) Handle special values of x (zeros, infinities, NaNs)
3) Compute the 8-bit exponent field of r
4) Compute the bits $\ell_1, \dots, \ell_{24}$ in (2)
5) Round to get the 23-bit fraction field of r
6) Pack the three fields of r

For special values, the following behavior is mandated by the standard: $\sqrt{x} = x$ for $x \in \{\pm 0, +\infty, \text{NaN}\}$ and $\sqrt{x} = \text{NaN}$ for either x negative and nonzero, or $x = -\infty$. Filtering such special values can clearly be done in parallel with the rest of the computation, and step 2) has been implemented efficiently as in [20, §11.2.1].

Step 3) can also be done independently. This is essentially due to some known properties of the square root function. First, $\sqrt{x} = \ell \cdot 2^d$ cannot lie in the middle of two consecutive numbers in format (1) (see for example [14, p.139]). Therefore knowing the bits $\ell_1, \dots, \ell_{24}$ is enough for rounding correctly to nearest: $r = (1.\ell_1 \dots \ell_{23} + \ell_{24} \cdot 2^{-23}) \cdot 2^d$. Second, when $\ell_{24} = 1$, it can be checked that $1.\ell_1 \dots \ell_{23} + 2^{-23}$ is always < 2 and thus d needs not be incremented. Hence step 3) can be performed at any time once the biased exponent $E = e + 127$ of x is available; since $d = \lfloor e/2 \rfloor$, the biased exponent $D = d + 127$ of r is obtained very fast by shifting and truncating.

We are thus left with steps 4) and 5), which are the most time-consuming. Several algorithmic choices are possible as well shown for example in [17]. Subsection III-B below reviews the methods that we have implemented; their latencies will be given in Subsection III-C.

B. Implemented methods

We have implemented seven square root algorithms, each of which falling into one of the three categories below.

1) Restoring and nonrestoring algorithms: These two are the most basic algorithms based on *digit recurrence* (see [6] for a detailed and comprehensive study). Both consist of 24 iterations: in the restoring algorithm, the ith iteration outputs exactly ℓ_i, possibly after a restoration step involving one more addition; the nonrestoring algorithm avoids restoration by using $\{-1, 1\}$ instead of $\{0, 1\}$ as a digit set. Although a final correction step is needed for that algorithm to return $\ell_1, \ldots, \ell_{24}$, it is in general faster than the previous one.

These two methods output $\ell_1, \ldots, \ell_{24}$ and thus rounding to nearest (step 5) in Subsection III-A) is straightforward. However, these methods are highly sequential (iteration $i+1$ cannot start before iteration i is completed) and thus poorly suited to the parallelism available on ST231. Hence we use them mostly for comparison with the next, faster methods.

2) Newton-Raphson and Goldschmidt iterations: Those methods are based on *iterative approximation* [6, §7]: they start with an approximation v_0 of $1/\sqrt{m}$ over $[1, 2)$ and refine it by successive iterations into a very good approximation v of $\sqrt{m}$, from which $\ell_1, \ldots, \ell_{24}$ can be deduced. For ℓ as in (2), it can be shown that a sufficient condition on v is

$$|v - \ell| \leq 2^{-25}. \tag{3}$$

As in [20], initial approximations are obtained by evaluating a low-degree truncated minimax polynomial [18, p.36] that approximates the function $1/\sqrt{m}$ over a small interval. Also, to exploit parallelism while keeping degrees small, the interval $[1, 2)$ is split into two or three subintervals, each of them corresponding to a possibly different approximation polynomial.

Newton-Raphson's or Goldschmidt's iteration is known to roughly double the number of bits of accuracy of the current approximation. Starting from an approximation of $1/\sqrt{m}$ rather than $\sqrt{m}$ ensures that the iterations contain multiplications (for which the ST231 is well suited) but no division. Newton-Raphson's iteration converges to $1/\sqrt{m}$ and a final multiplication by m gives the desired value v; Goldschmidt's version can be seen as performing that multiplication in the first iteration and then converging directly to $\sqrt{m}$. If the refinement consists of a single iteration then both methods are essentially the same.

We have implemented the following three variants:

- *Newton-2*: three subintervals associated with three polynomials of degree 1 giving initial values v_0 such that $|v_0 - 1/\sqrt{m}| \leq \epsilon$ and $\epsilon \in \{2^{-10}, 2^{-8}\}$ depending on the subinterval. Then two Newton-Raphson iterations are enough to get v as in (3).
- *Goldschmidt-2*: similar to *Newton-2* except that the two iterations are now Goldschmidt iterations.
- *Goldschmidt-1* [20, §11.2.3]: two subintervals associated with two degree-3 polynomials giving initial values v_0 such that $|v_0 - 1/\sqrt{m}| \leq \epsilon$ and $\epsilon \in \{2^{-14.5}, 2^{-13.5}\}$ depending on the subinterval. Then one iteration suffices to get v as in (3).

Those methods are more suited to the ST231 architecture than the (non)restoring algorithm: multiplications are done accurately enough thanks to the 32x32 bit multipliers, and polynomials are evaluated concurrently thanks to the parallel execution units. However, further speed-ups can be obtained on ST231 by a third approach, which we outline now.

3) Evaluation of polynomial approximants: This method corresponds to *approximation by a real function* [17, §5]. Unlike the previous method, it approximates directly $\sqrt{m}$ by one or several truncated minimax polynomials, and no refinement is needed. We propose two implementations of this approach:

- *Poly-5*: three subintervals associated with three degree-5 polynomials giving values v as in (3).
- *Poly-6*: two subintervals associated with two degree-6 polynomials giving values v as in (3).

The 32x32 bit multipliers of the ST231 allow to evaluate those polynomials very accurately, so that (3) holds. In order to take full advantage of the parallelism available on that architecture, we further use Estrin-like algorithms rather than the classical, but highly sequential Horner's scheme ([13, p.488], [11]): for example polynomials of degree 5 are evaluated as

$$(a_5 \cdot x + a_4) \cdot z + \big((a_3 \cdot x + a_2) \cdot y + (a_1 \cdot x + a_0)\big),$$

where $y = x \cdot x$ and $z = y \cdot y$. As we shall see in the next subsection, on ST231 this simpler approach turns out to be also the fastest one.

To achieve further speed-up, we have also improved the rounding procedure of [20, §11]: instead of first deducing $1.\ell_1 \ldots \ell_{24}$ from v and then rounding according to the value of ℓ_{24}, we compute the 23-bit fraction of r directly from the 26 most significant bits of v.

C. Experimental results

The implementation of each of the above square root algorithms has been verified by exhaustive tests (bit-exactness comparison with the values returned by the `sqrtf` function of the gcc libm). To measure timings (in numbers of clock cycles), non-special generic values have been used. As stated in Section II, the ST231 is a four-issue machine, but to evaluate the impact of the instruction parallelism, performance has been measured for three issue width values (2, 3, 4), which is a feature of the ST200 toolset (compiler and simulator). Timings are detailed in Table I, whereas Table II shows the average IPB (instructions per bundle) and IPC (instructions per cycle) numbers computed for the same issue width values. We see that best performance is obtained with the polynomial

	2	3	4
Restoring	170	153	148
Nonrestoring	233	159	133
Newton-2	53	49	45
Goldschmidt-2	50	46	42
Goldschmidt-1	45	42	36
Poly-5	53	45	33
Poly-6	49	38	**30**

TABLE I

TIMINGS (CLOCK CYCLES) FOR ISSUE WIDTH VALUES 2, 3, 4.

	2		3		4	
	IPB	IPC	IPB	IPC	IPB	IPC
Newton-2	1.46	1.30	1.59	1.37	1.92	1.61
Goldschmidt-2	1.39	1.33	1.63	1.52	2.15	1.73
Goldschmidt-1	1.51	1.45	1.66	1.58	2.03	1.91
Poly-5	1.44	1.44	1.75	1.67	2.50	2.34
Poly-6	1.43	1.43	1.89	1.85	2.45	**2.45**

TABLE II

AVERAGE IPB (INSTRUCTIONS PER BUNDLE) AND IPC (INSTRUCTIONS PER CYCLE) NUMBERS FOR ISSUE WIDTH VALUES 2, 3, 4.

evaluation methods of III-B-3), achieving a timing of 30 cycles for the implementation of *Poly-6*, and efficiently exploiting the instruction parallelism of the ST231. Compared to the Newton-Raphson/Goldschmidt-based method [20], currently implemented in the latest official ST231 toolset (and whose latency is of 48 cycles), a speed-up of 37.5% is observed; compared to the implementation of the naive nonrestoring algorithm, the speed-up is by a factor of almost 5.

All special values are handled in at most 22 cycles and are thus never slower than non-special values. Notice also that *Goldschmidt-2* is always faster than *Newton-2*, thus confirming the observation already made in [15] that Goldschmidt's method can be useful not only in hardware but also in software.

As highlighted in Tables I and II, the polynomial evaluation methods are also more sensitive to the issue width value, because of their high degree of parallelism; for example, the *Poly-6* method achieves an IPC value of 2.45.

IV. CONCLUSIONS

As we have observed, a software implementation of a correctly rounded-to-nearest single-precision square root operator using polynomial evaluation algorithms, and exploiting at best the instruction parallelism, turns out to be very efficient on a VLIW architecture such as the ST231, achieving a cost of no more than 30 cycles.

Furthermore, as it was formerly noticed in the IA-64 context [10], letting the compiler generate open code for the square root can even be more efficient when multiple calls to this function are done (either by the programmer or because of compiler transformations such as loop unrolling or software pipelining). In that case, all the constants needed to evaluate the polynomials are factorized by the compiler, thus increasing the computation issue rate.

In this paper we have focused on square root only: although simple, this function already illustrates well how to adapt to a target like the ST231 in order to achieve both speed and accuracy; also, in processor performance bottlenecks, square roots come first among basic floating-point operations [22, Table 2]. But we are currently extending our approach to other functions like square root reciprocal (that is even more critical to graphics pipelines), reciprocal, and division.

Finally, automating the generation of fast and accurate C code for computing algebraic functions beyond the four above will provide a way to explore various code generation tradeoffs, and to evaluate our techniques on other architectures.

ACKNOWLEDGMENT

The authors would like to thank the "Pôle de Compétitivité Mondial Minalogic" (www.minalogic.org) for its support.

REFERENCES

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[2] Christian Bruel. If-conversion SSA framework for partially predicated VLIW architectures. In *Digest of the 4th Workshop on Optimizations for DSP and Embedded Systems (Manhattan, New York, NY)*, March 2006.

[3] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium-Based Systems*. Intel Press, 2002.

[4] Marius Cornea-Hasegan and Bob Norin. IA-64 floating-point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, 1999-Q4:1–16, 1999.

[5] Benoît Dupont de Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2), September 2004. http://cri.ensmp.fr/classement/2003.

[6] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[7] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA'00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 203–213. ACM Press, 2000.

[8] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[9] Bruce Greer, John Harrison, Greg Henry, Wei Li, and Peter Tang. Scientific computing on the Itanium processor. In *Proceedings of the 2001 Conference on Supercomputing*, 2001.

[10] John Harrison. Formal verification of square root algorithms. *Formal Methods in Systems Design*, 22:143–153, 2003.

[11] John Harrison, Ted Kubaska, Shane Story, and Peter Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4:1–7, 1999.

[12] Cristina Iordache and Ping Tak Peter Tang. An overview of floating-point support and math library on the Intel XScale™ architecture. In *IEEE Symposium on Computer Arithmetic*, pages 122–128, 2003.

[13] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

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

[15] Peter Markstein. Software division and square root using Goldschmidt's algorithms. In *6th Conference on Real Numbers and Computers*, pages 146–157, 2004.

[16] Peter W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, 1990.

[17] P. Montuschi and P. M. Mezzalama. Survey of square rooting algorithms. *Computers and Digital Techniques, IEE Proceedings-*, 137(1):31–40, 1990.

[18] Jean-Michel Muller. *Elementary functions: algorithms and implementation*. Birkhäuser, second edition, 2006.

[19] Michael L. Overton. *Numerical computing with IEEE floating point arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[20] Saurabh-Kumar Raina. *FLIP: a floating-point library for integer processors*. PhD thesis, École Normale Supérieure de Lyon, 2006.

[21] Martin S. Schmookler, Ramesh C. Agarwal, and Fred G. Gustavson. Series approximation methods for divide and square root in the Power3™ processor. In *ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 116–123. IEEE Computer Society, 1999.

[22] Joshua J. Yi, Ajay Joshi, Resit Sendag, Lieven Eeckhout, and David J. Lilja. Analyzing the processor bottlenecks in SPEC CPU2000. In *Proceedings of the 2006 SPEC Benchmark Workshop*, page on CD, Austin, TX, 1 2006.