TAMPERE UNIVERSITY OF TECHNOLOGY

Author(s)	Viitanen, Timo; Jääskeläinen, Pekka; Takala, Jarmo
Title	Inexpensive correctly rounded floating-point division and square root with input scaling
Citation	Viitanen, Timo; Jääskeläinen, Pekka; Takala, Jarmo 2013. Inexpensive Correctly Rounded Floating-Point Division and Square Root with Input Scaling. In: Proceedings of the 2013 IEEE Workshop on Signal Processing Systems, SiPS 2013, Taipei, October 16-18, 2013. 5p.
Year	2013
DOI	
Version	Post-print
URN	http://URN.fi/URN:NBN:fi:tty-201311251473
Copyright	© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

All material supplied via TUT DPub is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorized user.

INEXPENSIVE CORRECTLY ROUNDED FLOATING-POINT DIVISION AND SQUARE ROOT WITH INPUT SCALING

Timo Viitanen *Pekka Jääskeläinen* *Jarmo Takala*

Tampere University of Technology
PO.Box 553
FIN-33101 Tampere
Finland
{timo.2.viitanen, pekka.jaaskelainen, jarmo.takala}@tut.fi

ABSTRACT

Recent embedded DSPs are incorporating IEEE-compliant floating point arithmetic to ease the development of, e.g., multiple antenna MIMO in software-defined radio. An obvious choice of FPU architecture in DSP is to include a fused multiply-add (FMA) operation, which accelerates most DSP applications. Another advantage of FMA is that it enables fast software algorithms for, e.g., division and square root without much additional hardware. However, these algorithms are nontrivial to perform at the target accuracy to get the correctly rounded result without danger of overflow.

Previous FMA-based systems either rely on a power-hungry wide intermediate format or forego correct rounding. A wide format is unattractive in a power-sensitive embedded environment since it requires enlarged register files, wider data buses and possibly a larger multiplier. We present provably correct algorithms for efficient IEEE-compliant division and square root with only a 32-bit format using hardware prescaling and postscaling steps. The required hardware has approximately 8% of area and power footprint of a single FMA unit.

***Index Terms*—** DSP, FPU, Fused multiply-add, division, square root

1. INTRODUCTION

IEEE-754 compliant floating-point arithmetic is becoming widespread in embedded *Digital Signal Processors* (DSP). For instance, Texas Instruments has a floating-point core in all their recent DSPs, citing programmer convenience and the demands of Software Defined Radio (SDR), whose frequent matrix inversions are inefficient to implement in fixed-point [1]. DSP algorithms have typically been designed first in floating-point and then converted for fixed-point operation, so having equivalent arithmetic on the DSP promises to speed up the development cycle and reduce errors. However, DSPs typically have limited support for division and square root. In particular, IEEE-754 compliant operations are rare, and would be desirable for reasons of programmer convenience. Moreover, if only a fast approximation is avalable, as in the TI C67 floating-point core [2], some algorithms exhibiting numerical instability may work correctly on a compliant development PC and fail on the DSP. A simple example is Gaussian elimination with certain classes of matrices. These arguments may not justify adding a large dedicated hardware unit, as the operations are infrequent. Therefore, it is interesting whether there is some minimal supporting hardware that can be integrated into DSPs at a low cost and still enable reasonably fast IEEE-754 compliant division and square root. This paper attempts to present such minimal hardware.

There are two main approaches to floating-point division and square root: digit recurrence and functional iteration. Digit recurrence methods such as SRT division resemble elementary school long-form arithmetic and produce a fixed number of digits in a cycle. They are usually implemented in a dedicated hardware unit, and represent a significant hardware investment. Functional iteration refines an initial guess into a correct result with, e.g., the Newton-Raphson method. Each iteration typically doubles the number of accurate bits. These methods tend to reuse existing floating-point multiplier hardware through microcode or software, but require minimal extra hardware, often a look-up table, to obtain the initial approximation. The difficulty in functional iteration is in obtaining a correctly rounded result: a straightforward iteration fails to compute the last few significant bits of accuracy due to rounding errors, and so careful design is necessary. Almost every implementation in the literature either relies on an intermediate format with more significand and exponent bits than the target format, or gives up correct rounding. [3]

A wide intermediate format requires invasive changes to the processor including enlarged registers and data buses and, therefore, is unsuitable for minimal supporting hardware. The mathematics library designed for the Intel Itanium proces-

The work has been financially supported by the Academy of Finland (funding decision 253087).

sor [4] has, to the authors' best knowledge, the only implementation in the literature that uses a target-width significand. The algorithms rely on a *Fused Multiply-Add* (FMA) instruction, which is fortunately a natural fit for DSP, benefitting many common DSP tasks such as FIR filtering, FFT, DCT and matrix arithmetic. However, the implementation still relies on additional exponent bits to avoid some overflow-related errors. Moreover, some corner cases are handled using software traps, which may result in slowdowns by factors of up to several hundred when hit. On statically scheduled VLIW architectures used in DSP, branches and context switches are even more expensive.

We extend our previous work, which produces close quotient and square root approximations on streamlined FPUs [5], to provide correct rounding on an IEEE-754 compliant single-precision FPU based on the Itanium algorithms. This is done using hardware prescaling and postscaling steps which can be implemented with lightweight additional hardware. A caveat is that we do not yet synthesize hardware with subnormal number support. We also propose and verify a division algorithm based on the proofs in [4], which requires a smaller hardware LUT by a factor of five than the original Itanium algorithm, at the cost of a worse latency.

This paper is structured as follows. Section 2 is a brief overview of related work. Section 3 describes the scaling procedures and their hardware implementation. Section 4 presents examples of IEEE-compliant division and square root algorithms using the scaling approach. In Section 5, the proposed hardware units are synthesized on 110nm ASIC. Section 6 is a summary of the work.

2. RELATED WORK

There is a large literature on implementations and correctness proofs of functional iteration, both from the academia and the industry, for example the IBM Power3, the IBM Cell and the AMD K7 [6]. However, in each case, either a wide format is used for computation or the results are not IEEE-compliant.

The TI C67 series of DSPs gives functional iteration routines for approximate division and square root, based on 8-bit hardware lookup tables [2]. Our work allows IEEE-754 compliant operation with roughly the same area, due to our smaller LUTs, but retains the option of computing fast approximations. However, we require an FMA operation which the TI DSP does not have.

Fractured Floating Point Units (FFPU) [7] is an effort similar to this work in that they use inexpensive special instructions to accelerate floating-point operations. However, they start from integer-based emulation with the *SoftFloat* library [12] whereas we assume an IEEE-compliant FMA unit and accelerate the missing operations.

According to Liu [8], SRT dividers are faster and more power-efficient than FMA-based functional iteration, but iteration is justified when divisions are sufficiently infrequent.

In [5], scaling is proposed for division and square root approximations with strict error bounds, in the context of streamlined custom-precision FPUs. This paper describes in detail and synthesizes the required hardware, and shows that the system is general enough to support correctly rounded operations, which may have wider applicability.

3. SCALING STEPS

The algorithms in [4] rely on the inputs not being too close to the edges of the *dynamic range* of the floating-point format. We illustrate the point with the well-known algorithm for computing the correctly rounded quotient $Q = a/b$ given the correctly rounded reciprocal $y = 1/b$,

$$q \longleftarrow a \times y$$
$$r \longleftarrow a - b \cdot y$$
$$Q \longleftarrow y + r \cdot y.$$

There are three issues with the algorithm if naively implemented at target precision:

- If b is very large, more than 2^{126}, y falls outside the single-precision normal dynamic range of $[2^{-126}..2^{128})$ and underflows, losing precision.
- If a is small, the remainder r, which may be up to 2^{45} times smaller than a in single-precision, may underflow with a wider range of inputs than y.
- Special case inputs are not handled correctly. For instance, a division of 1 by 0 should output ∞, but proceeds as: $q = \infty, r = 1 - 0 \times \infty = \text{NaN}, Q = \text{NaN}$.

The complete division and square root algorithms in [4] have similar issues. These errors rarely affect computations of interest, but they prevent IEEE compliance. In the original work, they are handled through using a *register float* format with a wide exponent, and branching to handle special cases. We propose to instead scale the input operands so that they lie well within the dynamic range. A scaling exponent is saved and used to postscale the result to the correct magnitude.

The technique also generalizes to handling the special case inputs of 0, -0, ∞, $-\infty$ and NaN. These always produce a special case output. For zero and infinite inputs it suffices to generate a sufficiently small or large postscaling exponent to ensure underflow or overflow, respectively. NaN special cases require an additional signal bit in the postscaling exponent. The scaling exponent manipulation turns special case inputs into meaningless small numbers with the correct sign. We use a 10-bit postscaling exponent including the signal bit.

The scaling steps are also useful for meeting other standard requirements. For instance, OpenCL requires a $\pm 2\text{ulp}$ division error across the entire dynamic range, easily leading to the issues discussed earlier, which scaling can circumvent.

3.1. Division

A scaling division procedure is shown in Algorithm 1. $a^{30..24}$ denotes the bits 30 down to 24 of a, in this case the exponent, interpreted as an unsigned integer. a' and b' are the scaled inputs, y_0 the scaled reciprocal approximation, and k the postscaling exponent which represents the exponents and possible special cases of the inputs. k^9 is the NaN signal bit. *Divide* is any software division algorithm that would compute the significand at the desired accuracy with a wide exponent. Since significand computation is independent of exponents in floating-point division, each input can be simply scaled to the range $[1..2)$ by setting the exponent to 127, which represents 0 in the biased format.

Algorithm 1: DivideWithScaling

Data: Single-precision float dividend a and divisor b; n-bit reciprocal LUT
Result: The quotient a/b
begin
- $a' \longleftarrow a$
- $a'^{30..23} \longleftarrow 127$
- $b' \longleftarrow b$
- $b'^{30..23} \longleftarrow 127$
- $y_0^{30..23} \longleftarrow 126$
- $y_0^{22..22-n} \longleftarrow LUT[b^{22..22-n}]$
- $k \longleftarrow a^{31..24} - b^{30..23}$
- **if** $b^{22..0} =$ "11..1" **then**
  - $y_0^0 \longleftarrow 1$
- **if** $a^{30..23} = 0 \lor b^{30..23} = 255$ **then**
  - $k \longleftarrow -256$
- **else if** $a^{30..23} = 255 \lor b^{30..23} = 0$ **then**
  - $k \longleftarrow 255$
- **if** $(a^{30..23} = 255 \land a^{22..0} \neq 0)$ $\lor(b^{30..23} = 255 \land b^{22..0} \neq 0)$ $\lor(a^{30..23} = 255 \land b^{30..23} = 255)$ **then**
  - $k^9 \longleftarrow 1$
- $Q' \longleftarrow \text{Divide}(a', b', y_0)$
- $Q \longleftarrow Q' \cdot 2^k$
- **if** $k^9 = 1$ **then**
  - $Q \longleftarrow \text{NaN}$
- **return** Q

3.2. Square root

A scaling square root procedure is shown in Algorithm 2. b' is the scaled input, y_0 the scaled reciprocal square root approximation, k the postscaling exponent, and *SquareRoot* the underlying square root algorithm. The square root significand depends on the LSB of the input exponent, which must be left intact, therefore, the input is scaled to the range $[1, 4)$. The computations of k and the exponent $y_0^{30..23}$ ensure a correct result exponent.

Algorithm 2: SquareRootWithScaling

Data: Single-precision number b; n-bit reciprocal square root LUT
Result: The square root $\sqrt{b}$
begin
- $b' \longleftarrow b$
- $b'^{30..24} \longleftarrow 63$
- $y_0^{30..24} \longleftarrow 63$
- $y_0^{23} \longleftarrow$ **not** b^{23}
- $y_0^{22..22-n} \longleftarrow LUT[b^{23..23-n}]$
- $k \longleftarrow (b^{30..23} - 1)/2 - 63$
- **if** $b^{30..23} = 0$ **then**
  - $k \longleftarrow -256$
- **else if** $b^{30..23} = 255$ **then**
  - $k \longleftarrow 255$
- **if** $(b^{30..23} = 255 \land b^{23..0} \neq 0)$ $\lor(b^{31} = 1 \land b^{23..0} \neq 0)$ **then**
  - $k^9 \longleftarrow 1$
- $g' \longleftarrow \text{SquareRoot}(b', y_0)$
- $g \longleftarrow g' \cdot 2^k$
- **if** $k^9 = 1$ **then**
  - $g \longleftarrow \text{NaN}$
- **return** g

3.3. Gradual underflow

Gradual underflow is a feature of IEEE-754 in which very small numbers can be represented as *subnormal* numbers with progressively fewer significant bits of accuracy. Subnormal support is expensive in terms of hardware. Most commodity hardware either flushes subnormal inputs and results to zero in violation of the standard, or raises a software trap. The analysis up to now assumes one of these approaches.

However, a DSP might opt to provide hardware support, as branches and context switches are expensive to implement in statically scheduled VLIW processors. The shown algorithms work correctly with subnormal inputs and outputs as long as the FMA unit supports them, with one caveat: postscaling needs to be performed before the rounding stage in the final FMA operation. Otherwise the operation incurs two rounding errors and the result may be off by one. Moreover, the prescaling instructions require normalization and become expensive to implement. An efficient design would share the normalization hardware of the FMA unit.

3.4. Hardware implementation

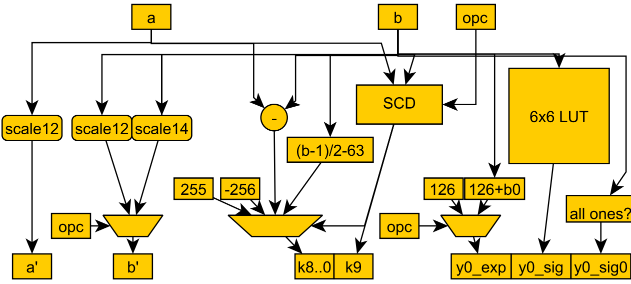
As visible in Algorithms 1 and 2, the prescaling stages mostly contain wiring and small-integer arithmetic, making them suitable for hardware implementation. A straightforward design which produces all initial values for both division and square root is shown in Figure 1. Note that a non-scaling approximation unit would require an additional multiplexer in

the LUT datapath to output, e.g., zero or infinity. The scaling design handles special cases in the shorter, parallel k datapath for a shorter overall delay. The design appears feasible to implement as two single-cycle special instructions.

Our research platform TCE [9] handles multi-output instructions and function units well, but in another architecture a four-output instruction might be unreasonable, and the hardware unit might need to provide the values one at a time through different instructions. k and a' (for division) are only required late in the algorithm, so there is no major effect on latency. To further simplify the unit, a' and b' could be executed as two integer ALU operations each.

We integrate the postscaling operation into the FMA unit in order to reuse its special case handling logic, reduce latency and instruction count, and prepare for future subnormal number support. It might also be implemented in a separate *multiply-by-power-of-two* instruction as in [5].

Fig. 1. Block diagram of the prescaling operation. SCD detects special cases in a and b. Scale12 scales the input between [1..2) by wiring, and Scale14 between [1..4).



4. ALGORITHMS

The scaling steps presented in the previous section can wrap any division or square root algorithm that computes the significand at the desired accuracy, but to verify the approach, we show one example of each. The square root algorithm is paraphrased from the double-precision algorithm in [10] by removing iterations. The division algorithm is based on proofs in [4], but has the advantage of requiring a smaller LUT. Performance characteristics are summarized in Table 1.

4.1. Division

Algorithm 3 computes correctly-rounded floating-point quotient of scaled inputs a' and b'. Operations that can be executed in parallel are shown on the same line.

The algorithm start from a reciprocal approximation y_0 fetched from a 6×6-bit LUT, refines it into a closer reciprocal approximation y_2 with two Goldschmidt iterations, computes the correctly rounded scaled reciprocal y_3, and finally computes the correctly rounded scaled quotient Q'.

Algorithm 3: Divide

Data: Scaled single-precision dividend a' and divisor b', reciprocal approximation y_0

Result: The quotient a'/b'

begin

$e \longleftarrow 1 - b' \cdot y_0$

$y_1 \longleftarrow y_0 \cdot e + y_0; \quad e_1 \longleftarrow e \cdot e$

$y_2 \longleftarrow y_1 \cdot e_1 + y_1$

$r \longleftarrow 1 - b' \cdot y_2$

$y_3 \longleftarrow r \cdot y_1 + y_1$

$q \longleftarrow a' \cdot y_3$

$r_1 \longleftarrow b' \cdot q + a'$

$Q' \longleftarrow r_1 \cdot y_2 + q$

return Q'

As shown in [4], given b and a close reciprocal approximation $y_0 \approx \frac{1}{b}$, the correctly rounded reciprocal y_1 can be computed as

$$r \longleftarrow 1 - b \cdot y$$
$$y_1 \longleftarrow y_0 + r \cdot y_0,$$

except possibly when the significand of b is all-ones. This case is handled by having the approximation unit output the correct reciprocal $1.00...1$ at cost of an and tree and a one-bit multiplexer. The correctly rounded quotient is computed with the algorithm discussed in Section 2.

In many cases, the algorithm presented in [4] may be more useful if modified to use our scaling steps. It achieves a better latency at the cost of requiring more parallel operations and an approximately 5 times larger LUT.

4.2. Square root

Algorithm 4 [10] computes the correctly rounded square root of b' starting from a reciprocal square root approximation $y_0 \approx \frac{1}{b'}$ fetched from a 6×6-bit lookup table. The algorithm uses three Goldschmidt iterations and a final Newton-Raphson iteration.

4.3. Large look-up table

Given 12-bit LUT for initial approximation, some steps can be omitted in each algorithm. Namely, $e1$ and y_2 can be skipped for division and r_1, g_2 and h_2 for square root. A straightforward implementation would be expensive in terms of area. However, the cost can be reduced by using a *piecewise linear approximation*. We found that the piecewise linear approximation produced an equal precision to the 12-bit LUT with only two 6×12-bit LUTs and a 12×15-bit multiplier. If the multiplier hardware of an FMA unit were reused, the LUT itself would be four times larger than in the 6-bit case, and could be further halved using the optimized approach in [11].

Table 1. Performance characteristics for each algorithm. Throughput is expressed as operations per cycle.

LUT size	6×6			8×8	12×12		
Operation	a/b	$1/b$	$\sqrt{b}$	a/b [4]	a/b	$1/b$	$\sqrt{b}$
FP operation count	9	7	10	10	7	4	7
Latency (FMA latencies)	8	6	7	5	7	4	5
Latency (Cycles, 6-cycle FMA)	49	37	43	31	43	25	31
Throughput per FMA unit	0.11	0.14	0.1	0.1	0.14	0.25	0.14

Algorithm 4: SquareRoot

Data: Scaled single-precision float b', reciprocal square root approximation y_0
Result: The square root of b'
begin

$g \longleftarrow b' \cdot y_0; \quad h \longleftarrow 1/2 \cdot y_0$
$r \longleftarrow 1/2 - h \cdot g$
$g_1 \longleftarrow g \cdot r + g; \quad h_1 \longleftarrow h \cdot r + h$
$r_1 \longleftarrow 1/2 - h_1 \cdot g_1$
$g_2 \longleftarrow g_1 \cdot r_1 + g_1; \quad h_2 \longleftarrow h_1 \cdot r_1 + h_1$
$d \longleftarrow g_2 \cdot g_2 + b'$
$g_3 \longleftarrow h_2 \cdot d + g_2$

return g_3

4.4. Performance

Performance characteristics of the algorithms are shown in Table 1. Performance is low compared to, e.g., the AMD K-7 divider [6] which computes a single-precision division or square root in four multiplications and 16 cycles using a very wide format. A more accurate initial guess improves performance somewhat, but many of the operations go into ensuring correct rounding, and are unaffected. However, the performance of 8 FP operation latencies for division and 6 for square root should be sufficient for applications where the operations are infrequent, and represent a large improvement over integer-based emulation with, e.g., *SoftFloat* [12], which may require hundreds of integer operations.

For instance, a straightforward implementation of the Cholesky decomposition takes $N^3/6$ multiply-subtractions and N square roots [13]. With $N = 16$, using the proposed system, square roots would account for 20% of the executed FP operations. If all operations were executed sequentially, FP operations took six times as many cycles as integer operations, and a *softfloat* square root took 316 cycles [7], then switching to *softfloat* would increase the runtime by 81%. These are generous assumptions since the algorithm allows for FP operations to be pipelined, and DSP processors are inefficient for branch-heavy code.

An advantage of the software approach is that precision can be traded off for speed at compile time. If the programmer allows unsafe optimizations, or uses a framework such as OpenCL, CUDA or OpenGL ES with loose error bounds, performance can be improved by eliminating the rounding computations. For instance, an OpenCL compliant division (within error bounds of $\pm 2.5\text{ulp}$) can be obtained in five FMA operations with a small LUT [5], a speedup of 44%. Also, a reciprocal is faster to compute than full division.

4.5. Verification

Floating point division is difficult to verify since it is a two-output operation, and there are too many combinations of inputs for exhaustive search. Due to the mathematical identities used to construct Algorithm 3, if the reciprocal y_3 is correctly rounded, so is the quotient. Accordingly, the algorithm was verified by testing all 2^{23} possible divisor significands. Algorithm 4 was similarly verified by testing 2^{24} combinations of input significand and exponent LSB in software simulation. Double-precision algorithms would require more sophisticated verification. Tests were carried out with both the 6×6-bit and 12×12-bit LUT versions of each algorithm. Moreover, correct operation with subnormal numbers was tested in software simulation with 10^7 random inputs. The algorithms produced the correctly rounded-to-nearest-even result in each case.

5. SYNTHESIS

In order to evalute the hardware cost of the proposed method, the 6×6 prescaler-LUT unit and the FMA-postscaler described in Section 3.2 were synthesized on 110nm ASIC at 300MHz. For the FMA-postscaler, we modified the multiply-adder function in [14] for six pipeline stages and fused operation. The FMA lacks special-case handling and subnormal support, but the LUT unit includes all logic necessary for special cases. The implemented units verified in RTL simulation with a small testbench of division and square root operations. We synthesized as reference a simple 8-bit LUT approximation unit similar to [2].

Synthesis results are shown in Table 2. A combinational LUT unit has 4% the area and maximum power of a multiply-adder. Inserting input registers in the style of our research platform [9] doubles the area and power footprint. A sequential SRT implementation would be expected to take up more area and power, since it requires registers for at least the partial quotient, partial remainder and divisor.

Table 2. Synthesis results on 110nm ASIC, 300MHz, 1.5V.

Unit	Latency	Area (NAND)				Power (mW)			
		Comb.	NC	Total	ratio	Switch	Int	Total	ratio
6×6 LUT+ prescaler	1	504	538	1042	0.09	0.188	0.349	0.537	0.09
8×8 LUT	1	889	184	1073	0.09	0.094	0.133	0.227	0.03
FMA	6	8627	3200	11827	1.00	2.33	3.73	6.06	1.00
FMA + postscaler	6	8845	3516	12362	1.04	2.37	3.99	6.36	1.05

The postscaling FMA instruction increases the footprint of the FMA unit by approximately 4%, mainly due to additional registers which propagate the postscaling exponent to the rounding stage of the pipeline. It may be more economical to have a separate multiply-by-power-of-two instruction, though this increases the latency of each operation by one cycle, and is not extensible to subnormal numbers. Alternatively, the postscaler could be written later than other inputs, removing most of the overhead. Both additions combined take up approx. 150% as much area as the reference 8-bit LUT, which is incapable of IEEE-compliant results.

6. CONCLUSIONS

We presented and verified a system for IEEE-compliant single-precision division and square root based on hardware scaling operations. The system has a hardware footprint slightly larger than LUT instructions similar to [2], which computes only approximate results. The iteration is done without software traps or wide intermediate formats which are prevalent in the literature, making the system inexpensive to integrate into a FMA-based DSP. Performance is low compared to larger units, but a major improvement over integer-based emulation. Future work will include implementing subnormals and benchmarking the system against sequential SRT units and extended-format datapaths, which are the main alternatives for a lightweight implementation.

7. REFERENCES

[1] G. Maur, "Efficient fixed- and floating-point code execution on the TMS320C674x core delivers faster code development and reduces system cost with improved performance," Texas Instruments, 2009. [Online]. Available: http://www.ti.com/lit/wp/spry127/spry127.pdf

[2] S. Poland, "TMS320C67xx divide and square root floating-point functions," Texas Instruments, 1999. [Online]. Available: http://www.ti.com/lit/an/spra516/spra516.pdf

[3] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 833–854, 1997.

[4] J. Harrison, "Formal verification of IA-64 division algorithms," in *Theorem Proving in Higher Order Logics*. Springer, 2000, pp. 233–251.

[5] T. Viitanen, P. Jääskeläinen, O. Esko, and J. Takala, "Simplified floating-point division and square root," *Proc. IEEE Int. Conf. Acoustics Speech and Signal Process.*, pp. 2707–2711, May 26–31 2013.

[6] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor," in *Proc. IEEE Symp. on Comput. Arithmetic*. IEEE, 1999, pp. 106–115.

[7] N. Hockert and K. Compton, "FFPU: fractured floating point unit for fpga soft processors," in *Int. Conf. Field-Programmable Technology*. IEEE, 2009, pp. 143–150.

[8] W. Liu and A. Nannarelli, "Power dissipation challenges in multicore floating-point units," in *IEEE Int. Conf. Application-specific Systems Architectures and Processors*. IEEE, 2010, pp. 257–264.

[9] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, San Jose, CA, January 2007, pp. 65 070X–1 – 65 070X–11.

[10] P. Markstein, "Software division and square root using Goldschmidt's algorithms," in *Conf. Real Numbers and Computers*, Schloß Dagstuhl, Germany, Nov. 15–17 2004, pp. 146–157.

[11] S. Yajima, "Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification," *IEEE Trans. Computers*, vol. 46, no. 4, p. 495, 1997.

[12] J. Hauser, "SoftFloat," 2007. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[14] D. W. Bishop, "VHDL-2008 support library," 2011. [Online]. Available: http://www.eda.org/fphdl/