SIMPLIFIED FLOATING-POINT DIVISION AND SQUARE ROOT

Timo Viitanen *Pekka Jääskeläinen* *Otto Esko* *Jarmo Takala*

Tampere University of Technology
Department of Computer Systems
Tampere, Finland
{timo.2.viitanen, pekka.jaaskelainen, otto.esko, jarmo.takala}@tut.fi

ABSTRACT

Digital Signal Processing (DSP) algorithms on low-power embedded platforms are often implemented using fixed-point arithmetic due to expected power and area savings over floating-point computation. However, recent research shows that floating-point arithmetic can be made competitive by using a reduced-precision format instead of, e.g., IEEE standard single precision, thereby avoiding the algorithm design and implementation difficulties associated with fixed-point arithmetic. This paper investigates the effects of simplified floating-point arithmetic applied to an FMA-based floating-point unit and the associated software division and square root operations. Software operations are proposed which attain near-exact precision with twice the performance of exact algorithms and resolve overflow-related errors with inexpensive exponent-manipulation special instructions.

Index Terms— Floating-point arithmetic, accelerator architectures, fused multiply-add, digital signal processing implementations, low-power design

1. INTRODUCTION

Floating-point computation is widespread in *Digital Signal Processing* (DSP). Embedded low-power applications often use fixed-point numbers, but recent studies suggest that it is possible to compete with fixed-point arithmetic by paring down the data width and dropping some features compared to, e.g., IEEE-754 single-precision floats [1], which are over-provisioned for most DSP tasks. For instance, in [2], 12-bit floats were employed on a wireless sensor platform, an application where power economy is paramount, and actually reduced power consumption compared to the 16-bit fixed-point necessary to perform the same task. Custom floating-point formats are advantageous because their arithmetic precision and dynamic range can be modified separately by adjusting the exponent and significand widths, respectively, making them customizable for the application at hand. In a fixed-point format both are controlled by the word width.

The robustness of DSP applications to accuracy reduction was studied in [3]. A suite of five DSP applications ranging from *Discrete Cosine Transform* (DCT) to speech recognition was run using emulated reduced-precision floating-point. The study found that all of the applications performed well even when the bit width of the significand was halved. Two applications could cope with a significand of as few as five bits. Video decoding and speech recognition applications were benchmarked in [4] with similar results, in the context of a design flow from C to custom-width floating-point hardware.

The previous studies have approached the topic using separate floating-point adders and multipliers. In this paper we focus on a *Fused Multiply-Add* (FMA) based architecture which, as argued in [5], appears to be a more efficient choice for most applications. A point of particular interest is to what extent can the software-based division and square root operations associated with FMA, designed for IEEE-compliant units, cope with these accuracy reductions. Many DSP applications require the use of square root and division, such as the square-root Kalman filter, the QR-RLS algorithm and the Householder transform [6]. Algorithms are often reformulated to avoid the operations altogether, but may consequently exhibit undesirable features such as overflows and numerical instability [7]. Consequently, it would be convenient for a developer to be able to use the operations with reasonable performance and without expensive dedicated hardware.

This paper explores the effects of simplification on an FMA-based floating-point unit and is organized as follows: Section 2 enumerates the advantages of an FMA-based approach and the available hardware simplifications. Section 3 discusses software-based division and square root algorithms that avoid overflows and correctly handle special-case inputs in presence of said simplifications. Section 4 evaluates the hardware and software effects of the proposed tradeoffs. Section 5 is a brief review of related literature, and concluding remarks follow in Section 6.

2. FUSED MULTIPLY-ADD HARDWARE

An FMA unit evaluates the expression $a + bc$ without rounding the intermediate product bc, and typically does not require much more hardware to implement than a multiplier.

The main advantage of the FMA architecture is the ability to execute all arithmetic instructions in a single compact unit. Addition may be emulated simply by multiplying by one, and multiplication by adding zero. An FMA instruction can be also used for efficient software-based implementation of division and square root, and can replace separate hardware units for these operations. [5]

Furthermore, many important computations consist of sums of products, e.g., matrix multiplication or polynomial evaluation. In particular, multiply-add is an important operation in many DSP algorithms, e.g., linear filtering and discrete trigonometric transforms. With such computations an accelerated FMA instruction will roughly halve the code size, improve performance, and reduce rounding errors. For these reasons, commercial processors such as the Intel Itanium [8], the AMD Bulldozer [9] and the IBM Cell Broadband Engine [10] incorporate FPUs built on an FMA architecture.

As with other FPUs, an FMA unit can be cut down by narrowing the data width, removing support for subnormal numbers, and switching to *Round-toward-Zero* (RtZ) rounding. Subnormal numbers are a special case in the IEEE-754 standard for representing very small quantities of less than 2^{-125} [1], which will not affect the result of most practical computations. They require siginificant extra logic to support, e.g., in [5] adding subnormals caused an FMA unit to grow by 30%.

RtZ rounding is a less obvious tradeoff, producing a more incremental hardware benefit of, e.g., 15% of the area of a multiplier unit in [3] while doubling the maximum round-off error of each basic arithmetic operation from $\pm 0.5\mathrm{ulp}$ to $\pm 1\mathrm{ulp}$. Moreover, as a final measure, the 'fused' property of the FMA could be dropped, reducing it into a plain multiply-adder. Such an unit would forego the accuracy benefit of FMA, but exhibit further improved area and performance. It should be noted that some FMA architectures are difficult to reconcile with this idea, for instance [5], where the addition is inserted into the multiplier CSA tree. Furthermore, this reduction only makes sense when using RtZ rounding, which requires no extra logic between the operations.

3. SOFWARE-BASED DIVISION AND SQUARE ROOT OPERATIONS USING FMA

3.1. Division

Computation of FMA-accelerated software division a/b usually starts with a fast approximation of the reciprocal of the divisor, $y \approx 1/b$. This can be performed by means of, e.g, a small lookup table. After this, one approach is to approximate the result by multiplying $a \times y$, and then iteratively refine the approximate result. This can be done by a Newton-Raphson iteration or a Goldschmidt power series iteration, the difference being that Goldschmidt makes no reference to the original inputs, allowing rounding errors to accumulate. Algorithm 1, paraphrased from [11], is an example of this approach. It originally performs the final iteration in double-precision in order to attain a correctly rounded result.

Algorithm 1: DivideFast

Data: Single-precision float dividend a and divisor b
Result: The quotient a/b
begin
$a', b', c \longleftarrow \mathrm{InitializeDivision}(a, b)$
$y \longleftarrow \mathrm{ReciprocalApproximation}(b)$
$q_0 \longleftarrow a' \cdot y; \quad e \longleftarrow 1 - b' \cdot y$
$q_1 \longleftarrow q_0 \cdot e + q_0; \quad e_1 \longleftarrow e \cdot e$
$q_2 \longleftarrow q_1 \cdot e_1 + q_1$
$q_2' \longleftarrow \mathrm{MultiplyByPowerOfTwo}(q_2, c)$
return q_2'

Another approach is to estimate $1/b$ by means of Goldschmidt iterations, and then use a single Newton-Raphson iteration to compute the quotient while compensating for rounding errors. An example of this approach is the division procedure used on the Intel Itanium processor [8]. The original algorithm computes four successively more accurate reciprocal approximations $y_1 ... y_4$ with Goldschmidt iterations for a total of 11 FMA operations. We consider variations of the algorithm that are reduced to one and two iterations, which require 5 and 7 operations, respectively. The two-iteration version is shown in Algorithm 2.

Algorithm 2: DivideSlow

Data: Single-precision float dividend a and divisor b
Result: The quotient a/b
begin
$a', b', c \longleftarrow \mathrm{InitializeDivision}(a, b)$
$y_0 \longleftarrow \mathrm{ReciprocalApproximation}(b)$
$e \longleftarrow 1 - b' \cdot y_0$
$y_1 \longleftarrow y_0 \cdot e + y_0; \quad e_1 \longleftarrow e \cdot e$
$y_2 \longleftarrow y_1 \cdot e_1 + y_1$
$q \longleftarrow a' \cdot y_2$
$r \longleftarrow b' \cdot q + a'$
$Q \longleftarrow r \cdot y_2 + q$
$Q' \longleftarrow \mathrm{MultiplyByPowerOfTwo}(Q, c)$
return Q'

Special case handling and large exponents are points of difficulty in this class of algorithms. For instance, computing $1/\infty$ should result in 0, but Algorithm 2 without preprocessing would attempt to subtract $\infty - \infty$ when evaluating $y1$, and, therefore, output a Not-a-Number special case. The Itanium handles special cases by branching, and overflows by advising developers to use extended-precision *register floats*.

Branches are expensive especially on VLIW architectures which are popular in DSP. Moreover, an extended-precision

format would be counterproductive when the objective is to decrease the floating-point precision. Therefore, we propose an alternative system that includes preprocessing and postprocessing steps which are cheap to implement on hardware as single-cycle special instructions.

The preprocessing step takes advantage of the fact that significands in division can be computed separately from exponents. The divisor and the dividend can, therefore, be multiplied by any power of two without affecting the significand computation of the final result. Since the overflow problems are associated with large exponents, the preprocessing step normalizes the inputs a, b such that:

$$a' = a \times 2^i, \qquad 1 \leq a' < 2$$
$$b' = b \times 2^j, \qquad 1 \leq b' < 2.$$

The postprocessing step adjusts the exponent of the computed quotient to reverse the earlier normalization:

$$a/b = (a'/b') \times 2^{i-j}.$$

This step can be implemented as a multiply-by-power-of-two instruction which may have wider applicability. Both steps involve mostly exponent manipulation, and adjust the significand only to zero it in order to output a special-case float. They are, therefore, cheap to implement on hardware. A disadvantage of this scheme is the need to store the scaling factor 2^{i-j}, increasing pressure on the register file.

Special case inputs are handled by passing the corresponding special case as the scaling term. The preprocessing stage can zero the exponents as normal. The resulting computation will produce a meaningless small number with the correct sign, which the scaling term converts into the expected special case output. Likewise, due to the orthogonality of significand and exponent calculation, the preprocessing instruction can detect over- and underflows, and output a suitable scaling term.

3.2. Square Root

The overflow and special case issues in the square root operation can be handled in similar fashion as in division, except that the significand computation is sensitive to the parity of the exponent, that is, the result significand does not change as long as the result is multiplied by a power of four. Therefore, the input is normalized to the range $[1, 4)$ with a similar preprocessing step and identical postprocessing step compared to division:

$$a' = a \times 2^{2i}, \qquad 1 \leq a' < 4$$
$$\sqrt{a} = \sqrt{a'} \times 2^{-i}.$$

A square root algorithm paraphrased from [11] is shown in Algorithm 3. It consists of a single Goldschmidt iteration to approximate the inverse square root of an input, followed by a Newton-Raphson iteration that generates the final output.

Algorithm 3: SquareRoot

Data: A single-precision float a
Result: The square root of a
begin
  $b', c \longleftarrow \mathrm{InitializeSquareRoot}(b)$
  $y_0 \longleftarrow \mathrm{InverseSquareRootApproximation}(b)$
  $g \longleftarrow b \cdot y; \quad h \longleftarrow 1/2 \cdot y$
  $r \longleftarrow 1/2 - h \cdot g$
  $g_1 \longleftarrow g \cdot r + g; \quad h_1 \longleftarrow h \cdot r + h$
  $d \longleftarrow g_1 \cdot g_1 + b$
  $g_2 \longleftarrow h_1 \cdot d + g_1$
  $g_2' \longleftarrow \mathrm{MultiplyByPowerOfTwo}(g_2, c)$
return g_2'

4. EVALUATION

4.1. Accuracy

The accuracy of the proposed FMA-based software operations was evaluated by simulation. Square root takes only a single operand and, therefore, it can be verified with exhaustive search over all possible inputs. The division algorithms were evaluated by dividing $10,000,000$ randomly chosen pairs of floats using software emulation, and comparing the results to the correctly rounded quotient. The results are shown in Table 1.

As expected, the plain multiply-adder suffers an accuracy hit compared to the FMA. Interestingly, in *DivideSlow* further Goldschmidt iterations after the first increase the average error; therefore the only reasonable algorithms for such an unit are those with five operations.

Notably, the proposed algorithms require only five multiply-add operations to achieve reasonable results, while an exact algorithm using an IEEE-compliant FPU requires 11. For example, if only 10% of arithmetic operations required by an application were divisions, this would reduce the final instruction count by more than one-fourth.

A point of interest is whether the operations are accurate enough to comply to the OpenCL Embedded Profile specification [12], which specifies error bounds for square root and division instead of exact results. The bounds are $\pm 3\mathrm{ulp}$ and $\pm 3.5\mathrm{ulp}$, respectively. All algorithms except for DivFast conform to these limits. The advantage of DivFast is that several of its component operations can be performed in parallel, reducing the runtime of some programs.

Possibly the closest available point of comparison is the MASS arithmetic library for the IBM Cell processor which is built around a four-way SIMD FMA unit with RtZ rounding and disabled subnormals, similarly to the 'FMA' column

Table 1. Accuracy and performance characteristics of each algorithm. Accuracy is measured with 10,000,000 random float pairs for division and with exhaustive search for square root. Errors are reported relative to the correctly rounded (RtZ) result.

	FMA				MA			
Procedure	DivFast	DivSlow 1	DivSlow 2	Sqrt	DivFast	DivSlow 1	DivSlow 2	Sqrt
Avg. Error (ulp)	0.70	7.0×10^{-5}	4.0×10^{-7}	0.13	0.50	0.41	0.56	0.26
Error Bounds (ulp)	-3 .. 0	-1 .. 0	-1 .. 0	-1 .. 1	-2 .. 4	0 .. 2	0 .. 2	-1 .. 0
Error Rate	60%	$< 1\%$	$< 1\%$	13%	43%	40%	53%	26%
OpenCL EP compliant	no	yes	yes	yes	no	yes	yes	yes
FMA count	5	5	7	7	5	5	7	7
FMA latency	3	5	6	5	3	5	6	5
Latency ($D_{FMA} = 6$)	21	33	39	33	21	33	39	33
Throughput w/ 2 FMA	0.4	0.4	0.29	0.29	0.4	0.4	0.29	0.29

Table 2. Synthesis results on a Xilinx Virtex-6 FPGA. The baseline is an IEEE-compliant single precision multiply-adder with a 23-bit significand and 8-bit exponent.

Unit	Cycle (ns)	Slices	LUTs
FMA, subnormals, RtN	20.123	592	1872
FMA, RtN	15.119	492	1523
FMA	14.353	420	1276
MA	14.107	372	1011
FMA, -1 sig. bit	14.007	387	1104
FMA, -2 sig. bits	14.440	416	1254
FMA, -3 sig. bits	13.963	328	944

in Table 1 [10]. It achieves looser error bounds than the proposed one on the 'MA' unit. The throughput is similar, assuming that the proposed one can also exploit four parallel execution units and sufficient registers to utilize them. The Cell can also be used with the SIMDmath library which is more accurate, but slower by a factor of four. [13, 14]

4.2. Hardware Benchmark

In Section 2, we suggested that reducing an FMA unit into a non-fused multiply-adder may be an useful technique to trade precision for hardware complexity. In order to test this hypothesis, we prepared hardware units using a single-precision multiply-adder from Bishop [15] as a base and synthesized them on a Xilinx Virtex-6 FPGA. Automatic register retiming produced very different results depending on the original placement of the pipeline registers, and, therefore, pipelining was omitted entirely in the interests of a fair benchmark. Results are shown in Table 2.

Disabling subnormal numbers and rounding produce benefits in line with the literature. The multiplier-adder optimization produces an area reduction similar to a reduction in significand width between 2 and 3 bits.

5. PRIOR WORK

The work presented here was related to the use of reduced-precision floating-point arithmetic in low-power digital signal processing. There is some literature concerning the use of reduced-precision floats [2, 3, 4] and fused multiply-adders [5] in this setting. The present paper combines these ideas by considering the effects of reduced precision on a multiply-adder datapath, and especially the associated software operations. A system is proposed for software division and square root which, e.g., inexpensively resolves overflow-related errors without resorting to a wider floating-point format. The errors are associated with inputs close to the limits of the dynamic range, and become relevant when the dynamic range is customized to the minimum necessary for the application. In addition, an incremental hardware optimization is proposed.

6. CONCLUSIONS

This paper evaluated a multiply-adder based, reduced-precision floating-point unit for embedded digital signal processing. Non-fused multiply-addition was proposed as one option for trading accuracy for efficiency. The tradeoff appears to save more space than would be consumed by adding one or two bits of significand accuracy, which would suffice in most situations to compensate for the lost accuracy.

Software-based division and square root operations were proposed that handle special cases without branching (thus are efficient on common VLIW architectures), avoid overflows without resorting to a higher-precision format, and exhibit close-to-exact accuracy while outperforming exact algorithms by a factor of two. The software-based operations are more accurate than the MASS library on the IBM Cell processor, which is similarly structured except for the multiplier-adder tradeoff, and ideally reach a similar or higher throughput.

Acknowledgements. This work was funded by the Academy of Finland (funding decision 253087).

7. REFERENCES

[1] *Standard for Floating-Point Arithmetic*, IEEE Std. 754, 2008.

[2] J. Janhunen, P. Salmela, O. Silvén, and M. Juntti, "Fixed- versus floating-point implementation of MIMO-OFDM detector," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, Prague, Czech Republic, May 22–27 2011, pp. 3276–3279.

[3] J. Tong, D. Nagle, and R. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. VLSI Systems*, vol. 8, no. 3, pp. 273 –286, June 2000.

[4] F. Fang, T. Chen, and R. Rutenbar, "Floating-point bit-width optimization for low-power signal processing applications," in *IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 3, 2002, pp. III–3208.

[5] T. M. Bruintjes, K. H. G. Walters, S. H. Gerez, B. Molenkamp, and G. J. M. Smit, "Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 41:1–41:22, Jan. 2012.

[6] S. Haykin, *Adaptive Filter Theory*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[7] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

[8] B. Greer, J. Harrison, G. Henry, W. W. Li, and P. Tang, "Scientific computing on the Itanium processor," *Scientific Programming*, pp. 329–337, 2002.

[9] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *Micro, IEEE*, vol. 31, no. 2, pp. 6–15, March-April 2011.

[10] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong, "The vector floating-point unit in a synergistic processor element of a CELL processor," in *Proc. IEE Int. Symp. Comput. Arithmetic*, Cape Cod, MA, USA, June 27–29 2005, pp. 59–67.

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

[12] A. Munshi, "The OpenCL specification version: 1.2 document revision: 15," Khronos, 2011.

[13] "Accuracy information for the MASS libraries for Cell/B.E. SPU," Tech. Rep., 2009. [Online]. Available: http://www-01.ibm.com/support/docview.wss?uid=swg27009549

[14] "Performance information for the MASS libraries for Cell/B.E. SPU," Tech. Rep., 2010. [Online]. Available: http://www-01.ibm.com/support/docview.wss?uid=swg27009548

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