Improving the Effectiveness of Floating Point Arithmetic

Hossam A. H. Fahmy Albert A. Liddicoat
Michael J. Flynn
Computer Systems Laboratory, Stanford University
Stanford, California 94305, USA

Abstract

This work presents several techniques to improve the effectiveness of floating point arithmetic computations. A partially redundant number system is proposed as an internal format for arithmetic operations. The redundant number system enables carry free arithmetic operations to improve performance. Conversion from the proposed internal format back to the standard IEEE format is done only when an operand is written to memory. Efficient arithmetic units for floating point addition, multiplication and division are proposed using the redundant number system. This proposed system achieves overall better performance across all of the functional units when compared to state-of-the-art designs. The proposed internal format and arithmetic units comply with all the rounding modes of the IEEE 754 floating point standard.

1 Introduction

Many numerically intensive applications, such as signal processing, require rapid execution of arithmetic operations. Addition is the most frequent operation followed by multiplication. However, high-performance divide and other elementary functions are becoming increasingly important. This work presents several techniques to improve the effectiveness of floating point arithmetic units.

A partially redundant number system is proposed for use as an internal format within the floating point unit and the associated registers. When a number is loaded into the floating point register file it is transformed into the proposed format. The floating point functional units input and output operands in the proposed format. Therefore, all the operations occur using the partially redundant format. The redundant number format enables carry free propagation operations across all of the functional units. The proposed system with the associated algorithms and circuits achieve correctly rounded results. If a store operation is issued, the operand is transformed to the IEEE single or double precision format.

Section 2 explains the proposed format and the conversion issues. The floating point addition unit is presented in section 3 and the multiplication unit is presented in section 4. Division and other elementary functions are computed using the floating point functional unit described in section 5. Finally, in section 6 conclusions of this work are presented.

2 Proposed format

The format proposed here is developed based on the single and double precision formats of the ANSI/IEEE standard [1]. However, in the proposed format each group of 4 bits of the significand are represented redundantly as a 5 bit signed digit number using two's complement form. The fifth bit (extra bit) represents a negative value with the same weight as the least significant bit of the next higher group. This is shown for the string of bits $a4, a3, a2, a1, a0$ in Fig. 1. This extra bit, $a4$, is saved in the register to the right of least significant bit in the next higher group and to the left of $a3$. As with IEEE formats, the significand is always positive so there is no need for the extra bit in the most significant digit. The number is also always normalized in the proposed format. Denormalized IEEE numbers are normalized in the conversion process upon loading into the register file. Each group of 5 bits represents one base 16 digit and therefore, the exponent is applied to base 16 rather than base 2 as is used in the normal IEEE format. The proposed format is in the form, $(-1)^{sign} first\,digit.remaining\,digits \times 16^{exp-bias}$. The guard bits, G, and sticky bits, S, are saved in the register file with the unrounded result. The result is then rounded in the following operation when it is used or saved to the memory. This deferred rounding technique moves the rounding computation off the critical path and allows it to be overlapped with the exponent difference calculation in the adder.

The basic idea is to use a redundant representation with signed digits for an internal format instead of the standard IEEE format. Redundant representations using signed digits (SD) have been proposed for

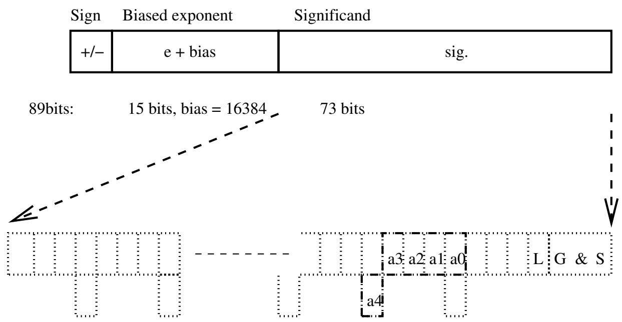

Figure 1: The proposed signed digit format for floating point numbers.

parallel arithmetic and studied in detail in the literature [2, 3]. Likewise, the use of an IEEE compliant hexadecimal-based internal format is not new [4]. The novelity of this proposal stems from the way conversion to and from the SD numbers is achieved, postponed rounding as well as extending a number of known techniques to improve the performance and provide a comprehensive arithmetic system. In general, SD numbers allow carry free addition by using redundant number representations. Eliminating the carry propagation significantly reduces the latency of arithmetic operations. The conversion from binary to SD form is trivial since the binary format is usually a valid SD representation. However, converting a SD number back into a non-redundant form involves a carry propagation. SD numbers are not commonly used in arithmetic circuits since the SD to binary conversion requires a carry propagation. The proposed system overlaps the SD number to binary conversion with memory store operations, thus removing it from the critical path.

2.1 Conversion issues

To convert from the IEEE binary format to the proposed format both the significand and the exponent must be modified. The method used has been adapted from the method proposed by Schwarz *et al.* [4]. Effectively, the bias from the original base 2 exponent must be subtracted, then the exponent should be divided by 4 to account for the radix change, and finally the bias of the proposed hex exponent must be added. In addition to the exponent change, the significand must be adjusted by the two least significant bits of the original base 2 exponent since a fractional exponent is not allowed.

A function named *bthe* (as a short notation for Binary To Hex Exponent) is applied to the most significant $n-2$ bits of the biased base 2 exponent. The *bthe* function complements the biased base 2 exponent's most significant bit and extends the result throughout the higher bit positions in the hex exponent. The most significant bit of the hex exponent is set equal to the most significant bit of the biased base 2 exponent.

Let R be the remainder of the biased base 2 exponent, exp_2, divided by 4. Alternately stated, R is equal to the value of the least significant two bits of the biased base 2 exponent, exp_2. If the original number is normalized in the IEEE format then the conversion follows one of the following four possible cases. Note that for the case of $R = 3$ there is a 1 added to the exponent to account for a one digit shift to normalize the significand.

$$\begin{array}{lcl}
R = 0 & 001x.xxxx\cdots & \times 16^{(bthe(exp_2)-2^{14})} \\
R = 1 & 01xx.xxx\cdots & \times 16^{(bthe(exp_2)-2^{14})} \\
R = 2 & 1xxx.xx\cdots & \times 16^{(bthe(exp_2)-2^{14})} \\
R = 3 & 0001.xxxxx\cdots & \times 16^{(bthe(exp_2)-2^{14}+1)}
\end{array}$$

If the original binary number is denormalized then the leading non-zero digit must first be identified and the significand left shifted to normalize it. The new exponent would thus be equal to $bthe(exp_2 = 0) - shift\ amount$. Conversion from a binary format to the proposed SD hex format is quite simple. At most, one short addition (15 bits) may be required for the exponent computation. Since the numbers are stored in the SD hex format in the register file, the binary to SD conversion only occurs when operands are loaded from memory.

The add, multiply and divide/elementary functions units are all designed to work on unrounded SD numbers, therefore, conversion out of the internal format only occurs when the operands are written to memory. Rounding is postponed until the start of the subsequent operation or when an operand must be stored to memory. Rounding does not require a carry propagation since the operand is rounded to a SD number. By using SD numbers and SD addition rules the long carry propagation required within carry propagate adders (CPA's) is eliminated.

To convert the SD significand back to a binary number the negative extra bits of the SD number are subtracted from the positive bits. This subtraction requires a full carry propagation. The carry propagation delay required in the conversion is overlapped with the memory write operation. The exponent must be converted back to the binary base as well. This exponent conversion is the reverse of the conversion into the SD Hex format.

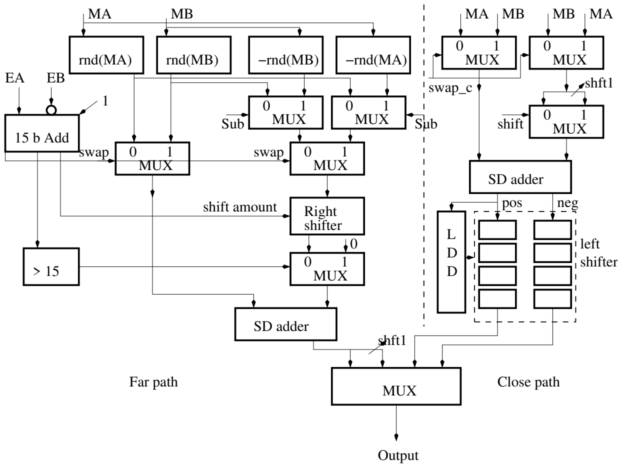

Figure 2: Block diagram of the two-path adder.

In current floating point designs, the store operation does not use the execution stage of the pipeline except for the address calculation of the target memory location. In the proposed system, the integer execution unit is used to convert an operand from the internal format back into the IEEE format. Another alternative is to have an additional integer adder in the FPU unit for the conversion. This conversion does not add any extra delay to the store operation. Therefore, contrary to previous designs using SD numbers, the proposed system effectively hides the delay of the conversion from SD numbers back to the non-redundant representation.

3 Floating point addition

In state-of-the-art high-performance floating point adders, two-path algorithms are used with integrated rounding similar to the designs proposed by Farmwald [5] and Quach [6]. Two different designs for the adder using the proposed format were devised. The first design is a one-path sequential algorithm and the second design is a two-path algorithm as shown in the block diagram of Fig. 2. In both designs, postponed rounding occurs in parallel with the exponent subtraction in order to reduce latency. In the two-path design, the far path is used for all effective additions and for effective subtractions with exponent differences larger than one. The close path is only used for the case of effective subtractions with an exponent difference of zero or one.

The far path of the proposed adder is similar to the far path of other algorithms presented in the literature. The unique aspects of the proposed adder are, first, the use of signed digit numbers in the significand and second the location of the rounding logic in parallel with the exponent difference. In the close path the exact exponent difference is not calculated so the rounding must be done in conjunction with the significand addition. A round digit is computed while the significands are steered into the adder and the round digit must be applied as a carry in to the significand adder.

In an effective subtraction in the close path one or more of the leading digits in the result maybe zero. Then, in order to normalize the result, the leading non-zero digit must be detected and the result must be normalized by left shifting the significand by the number of leading zeros. All floating point adders include circuits to either detect or predict the position of the leading non-zero digit after the subtraction is performed. The prediction circuits operate on the adder's operands in parallel with the significand addition. Since in the current scheme signed numbers are used, there is a slightly more complicated situation for normalization. The leading zeros may be expressed directly as zeros or indirectly as a leading 1 followed by -15's or by a leading -1 followed by 15's. In the indirect cases the leading 1 (-1) can be converted to a zero and borrowed into the neighbor -15 (15) digit position as a 16 (-16). Since $16 - 15 = 1$ ($-16 + 15 = -1$) the zero propagation may continue into lower significance digits. The following example illustrates how leading non-zero digits may be leading insignificant digits. Assuming $\mid l \mid < 15$,

$$
\begin{array}{cccccccc}
1 & -15 & -15 & \cdots & -15 & l & \cdots & \\
= & 0 & 0 & 0 & \cdots & 1 & l & \cdots \\
-1 & 15 & 15 & \cdots & 15 & l & \cdots & \\
= & 0 & 0 & 0 & \cdots & -1 & l & \cdots
\end{array}
$$

Because of the added complications in the problem at hand, instead of trying to predict the number of leading zeros in parallel with the significand addition, a leading digit detection scheme is applied to the significand adder output. Details of the technique is described elsewhere [7].

The proposed algorithm and other algorithms from the literature [8, 9, 10, 11] are compared in the following table.

Comparative gate delays for double precision

Proposal	Nielsen	Oberman	Smith	Seidel
21	**40**	**27**	**28**	**27**

The table lists the estimated number of equivalent gate delays for each of the designs. All of the designs listed in the table are for double precision operands and consistent assumptions about the subcomponents delays are used in the evaluation of each design.

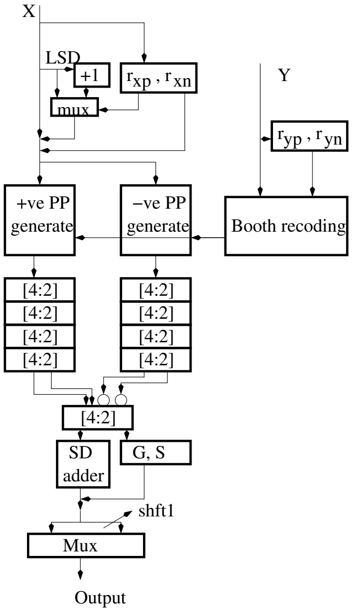

Figure 3: General block diagram of the multiplier.

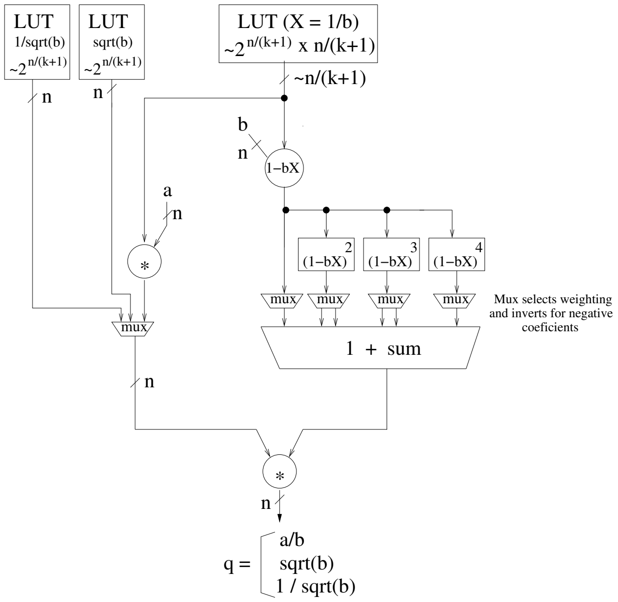

Figure 4: Division and elementary functions unit.

4 Floating point multiplication

The design of the proposed multiplier is shown in Fig. 3. As shown in the figure, while the rounding decision is being made for operand X, $X + 1$ is calculated. Then based on the proper rounding decision the correctly rounded least significant digit is selected. The $X + 1$ computation does not require a carry propagation since SD numbers are being used. For the multiplier operand Y, the Booth 2 recoding scheme is modified to take into consideration the extra negative bits, the guard bits, and the sticky bits of the multiplier in order to produce a correctly rounded result. The extra negative bits of the multiplicand, X, are dealt with in a slightly different way. The significand of X is taken as having two components: P the positively valued bits and E the negatively valued extra bits.

The output of the Booth recoders are used as select lines in multiplexers to generate the required partial products. The positive vectors are then summed by a tree of [4 : 2] compressors while the negative vectors are summed by a separate tree of [4 : 2] compressors. The output of each tree is in carry save format. The positive and negative vectors are then added using a [4 : 2] compressor and signed digit adder to form the final result.

A gate delay estimation similar to that of the adder was performed to determine the multiplier latency. The latency of the proposed multiplier was compared to the system presented by Oberman [12]. In order to perform an accurate comparison, the delay in the final stage of the comparison multiplier incurred to share the multiplier with the divide unit was not included. The comparison multiplier requires 39 gate delays while the proposed multiplier requires 32 gate delays for extended precision results. The proposed multiplier can compute double precision results in 29 gate delays.

5 Floating point division and other elementary functions

To perform division and other elementary functions, a design from the literature is adapted [13, 14]. This arithmetic unit (shown in Fig. 4) provides rapid convergence based on higher-order Newton-Raphson and series expansion techniques. The architecture provides fast and efficient function evaluation while allowing high-throughput. The arithmetic unit achieves fast computation by using parallel squaring, cubing, and powering units. These units compute the higher-order terms significantly faster than the traditional approach of serial multipliers. All of the terms are computed in parallel further reducing the latency. To adapt the original design to the format proposed here, a short adder is used to eliminate the redundancy from the most significant part of the divisor operand by subtracting the extra bits. This non-redundant part is used to access the lookup table while the rest of the operand is fully transformed into a non-redundant form. In parallel, another adder is used to convert the dividend into a non-redundant form as well. The unit then works on those two operands as in the original design and at the end a signed digit adder is used instead of the regular carry propagate adder. The delay of the proposed unit is not much different from the original design since the extra delay of the short adder at the start is compensated by the reduced delay in the final addition.

6 Conclusions

The proposed internal format with the proposed algorithms and arithmetic units provide a complete IEEE compatible arithmetic system. The elimination of carry propagation from the arithmetic operations enhances the performance of the functional units. The proposed arithmetic unit architecture includes further enhancements that increase the floating point performance such as a hex based number format and postponed rounding techniques. The proposed system pushes the performance boundary of the design space and provides a means to achieve the computational demands of numerically intensive applications. The proposed addition and multiplication algorithms are about 20% faster than the corresponding state of the art.

References

[1] "IEEE standard for binary floating-point arithmetic," Aug. 1985. (ANSI/IEEE Std 754-1985).

[2] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, Sept. 1961.

[3] B. Parhami, "On the implementation of arithmetic support functions for generalized signed-digit number systems," *IEEE Transactions on Computers*, vol. 42, pp. 379–384, Mar. 1993.

[4] E. M. Schwarz, R. M. Smith, and C. A. Krygowski, "The S/390 G5 floating point unit supporting hex and binary architectures," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 258–265, Apr. 1999.

[5] P. M. Farmwald, *On the Design of High Performance Digital Arithmetic Units.* PhD thesis, Stanford University, Aug. 1981.

[6] N. T. Quach, *Reducing the latency of floating-point arithmetic operations.* PhD thesis, Stanford University, Dec. 1993.

[7] H. Fahmy and M. Flynn, "Leading digit detection for floating point adders using signed digit numbers." Not yet published.

[8] A. M. Nielsen, D. W. Matula, C. N. Lyu, and G. Even, "An IEEE compliant floating-point adder that conforms with the pipelined packet-forwarding paradim," *IEEE Transactions on Computers*, vol. 49, pp. 33–47, Jan. 2000.

[9] S. F. Oberman and M. J. Flynn, "Reducing the mean latency of floating-point addition," *Theoretical Computer Science*, vol. 196, pp. 201–214, 1998.

[10] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced latency ieee floating-point standard adder architectures," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 35–42, Apr. 1999.

[11] P.-M. Seidel and G. Even, "How many logic levels does floating-point addition require?," in *Proceedings of the International Conference on Circuit Design*, pp. 142–149, 1998.

[12] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7^{TM} microprocessor," in *Proceedings of the 14th IEEE Sympsoium on Computer Arithmetic, Adelaide, Australia*, pp. 106–115, Apr. 1999.

[13] A. A. Liddicoat and M. J. Flynn, "High-performance floating point divide," in *Proceedings of the Euromicro Symposium on Digital System Design*, pp. 354–361, Sept. 2001.

[14] A. A. Liddicoat, *High-Performance Arithmetic for Division and The Elementary Functions.* PhD thesis, Stanford University, 2002. To be published.