Low Power Floating-Point Multiplication and Squaring Units with Shared Circuitry

Jason Moore, Mitchell A. Thornton, and David W. Matula
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX

***Abstract*—An architecture for a combinational floating point multiplier and squarer is described for the purpose of producing a low power floating point square with small area requirements. The floating-point multiplier and squarer architecture are compared to demonstrate the power advantage of the squarer. The multiplier and squarer are combined into one circuit in order to take advantage of the squarer power improvements with a minimal increase in area. Shared circuitry among the units provides justification for inclusion of a dedicated squarer since a small amount of additional circuitry is required and the power savings for squaring computations is significant as compared to the use of a general-purpose multiplier to generate a squared value.**

Keywords: Floating-point, squarer, combined multiplier and squarer

I. Introduction

The floating point squarer design presented in this paper is a full precision squarer. The squarer was constrained to be low power and require small area in order to justify the benefits of including both a multiplier and squarer. Furthermore, the design utilizes shared circuitry among the floating-point squarer and multiplier circuits to minimize impact due to area increases. If a unit-in-the-last-place (ulp) accurate approximation is the accepted rounding method for the system, a left-to-right approximation squarer such as the one presented in [8, 9, 10] would be the best choice since floating point circuits only use the *n* most significant bits if truncation is used. This approach however is not a feasible option because all the IEEE rounding methods are implemented in the floating-point multiplier in [2]. By requiring the squarer to also implement the same rounding methods as the multiplier, combining the two circuits allows for increased resource sharing as well as providing a more fair comparison. However, the combinational circuit presented here can easily be modified to use only truncation resulting in an approximate squarer (ie. most-significant bit first).

To keep area cost low, the right-to-left low-order-bit-first radix-4 squarer presented in [7] is used. The squarer takes advantage of Booth radix-4 folding, an idea first presented in [11,12,13] that uses symmetry and radix-4 Booth recoding to reduce the number of partial products. The overall architecture presented here is capable of utilizing any full-precision squaring implementation as a core design.

II. Background

The IEEE 754 floating point standard [1] can be interpreted as follows:

$$\begin{cases} 0.F * 2^{-127} & E = 0 \\ \begin{cases} NaN & F \neq 0 \\ \pm Infinity & F = 0 \end{cases} & E = 255 \\ -1^s * 1.F * 2^{E-127} & 0 < E < 255 \end{cases}$$

Conceptually, floating point multiplication is performed through addition of the exponents and an accompanying multiplication of the significands with normalization governing the value of the product exponent. If both a and b are normalized, the equation, $(a \times 2^{e1}) \times (b \times 2^{e2}) = ab \times 2^{e1+e2}$, is all that is needed to calculate the floating-point product. The above equation does not account for the exponent bias. The true value of the exponent is the value represented by the exponent bit field minus the bias. In order to preserve the bias, it must be subtracted from the sum of the exponents as $(a \times 2^{e1}) \times (b \times 2^{e2}) = ab \times 2^{e1+e2-bias}$.

Since the values for a and b are in the range [1,2), the product range is [1,4). The output from the circuit must be within the range [1,2), thus, the product undergoes normalization. This is accomplished by shifting the significand to the right by one bit position if the significand is greater than or equal to two. If the significand is divided by two, the sum of the exponents must accordingly be incremented by one.

The algorithm must account for inputs that are denormalized as well as the special values defined in the standard. Normalizing an input value is accomplished in a similar manner to normalizing the output. The significand shifted left to one place pass the most significant one and exponent is accordingly decremented by the number of times the significand is shifted left. In the IEEE standard, the special values that the input can represent are 'not a number' (NaN), positive infinity, and negative infinity. The equation presented above cannot be used to calculate the product if one or more of the inputs is a special value. Therefore, special values must be detected and the appropriate output selected.

Since only a certain range of values are represented, the output is not always an exact representation of the product. In this case, flags are set to indicate that output is not exact. Flags are required for the cases of divide by zero, invalid, underflow, overflow, and inexact. The remaining sections describe how a floating-point multiplier accomplishes these tasks for the purpose of comparison to our design.

III. Floating Point Multiplier

The floating-point multiplier implemented in [2] decomposes the multiplier architecture into the following components: pre-processer, special case detector, pre-normalizer, multiplier, exponent adder, normalizer, shifter, rounder, flagger, and assembler. The components are connected as shown in Fig. 2. Each of the components are discussed in detail in the following paragraphs.

The preprocessor in Fig. 2 implements the following Boolean equations to determine whether either of the inputs are a special case. The variables *Asig*, *Bsig, Aexp*, and *Bexp* represent the width of the significands and exponents of the inputs *A* and *B* respectively.

$$zero = (\sim|Asig\&\sim|Aexp)|(\sim|Bsig\&\sim|Bexp)$$

$$aisnan = \sim(\sim|Asig)\ \&\ (\&Aexp)$$

$$bisnan = \sim(\sim|Bsig)\ \&\ (\&Bexp)$$

$$infinity = \big((\sim|Asig)\&(\&Aexp)\big)|$$

$$\big((\sim|Bsig\ )\&(\&Bexp)\big).$$

$$aisdenorm = \sim(\sim|Asig)\&\ (\sim|Aexp)$$

$$bisdenorm = \sim(\sim|Bsig)\ \&\ (\sim|Bexp)$$

The outputs *aisnan, bisnan, zero,* and *infinity* from the pre-processor are inputs to the special case detector as well as *A* and *B*. The *specialsigncase* is defined as $aisnan \mid bisnan \mid (zero\ \&\ infinity)$ and *specialcase* is defined as $aisnan \mid bisnan \mid infinty \mid zero$. *Invalid* equates to $zero\ \&\ infinity$. If *invalid* is set, the *specialsign* is the most significant bit *A* if *aisnan* is true, (referred to as *aishighernan* in [2]) *and* the significant bit of *A* is greater than or equal to the significant bit of *B* or *bisnan* is false and is the most significant bit of *B* if it is false.

The inputs *A* and *B* and the outputs from the pre-processor *aisdenorm* and *bisdenorm* are used to normalize *A* and *B* such that the significand value range is between [1,2). If the input is denormalized, the significand is left shifted by the width minus the bit position of the most significant one. If the significand is shifted, the exponent must be adjusted such that the number continues to represents the same value.

The multiplier circuit used here is a carry-save array multiplier. A carry-save array multiplier is a tree multiplier consisting of a one-sided reduction tree of carry-save adders with a ripple-carry adder for the final addition [14]. The normalized significands are input to the multiplier. The most significant bit of the product is used to determine if the product is equal to or greater than two.

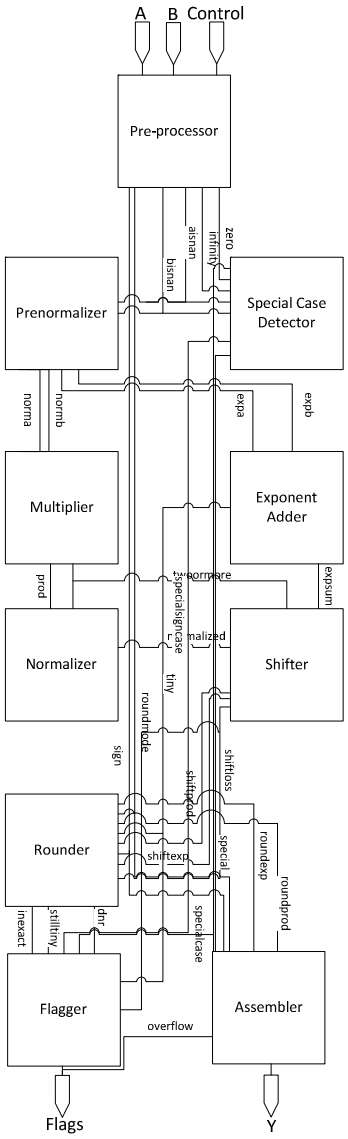

Fig. 2: Floating Point Multiplier

The exponent adder calculates the exponent of the product and the *tiny* bit by the equation, $expsum = expa + expb - (bias - twoormore)$. If *expsum* is less than one, *tiny* is set to one.

If *tiny* and *twoormore* are true, the product is already normalized and *normalized* is set to *prod*. Since *twoormore* represents an overflow of the product and *tiny* represents an overflow of the sum of the exponents, the product is shifted left by one position if either is true or it is shifted left by two if neither one is true. This results in placing the most significant bit of the possible output in the most significant bit of the product. If *tiny* is true, the product is denormalized and the significand will need to be shifted right by the magnitude of *expsum*. If *expsum* is greater than zero, *shiftexp* is set to *expsum*, otherwise *shiftexp* is set to zero. If any accuracy is lost due to the shifting operations, *shiftloss* is set to one.

The multiplier produces an output with a bit-width twice as large as the bit widths of the inputs. However, the IEEE standard only allows for the output to be stored in the same bit width as the inputs. Since it sometimes is not possible to store the exact result, the output must be rounded. Rounding in the floating-point multiplier implementation in [2] is based on the rounding methods presented in [3]. The multiplier allows four different types of rounding modes: round to nearest even (00), round to zero (01), round to positive infinity (10), and round to negative infinity (11). The rounding mode is determined by the two least significant bits of *control*. The upper half of the shifted product is used as the basis for the answer that is referred to as *unrounded*. The purpose of the Rounder in Fig. 2 is to determine whether or not to add one to *unrounded*. The simplest rounding method is round to zero because one is never added to *unrounded* as the least significant bits are always truncated which is referred to as *lowerBits*. *Unrounded* is incremented if *unrounded* is an odd value, any of the *lowerBits* are 1, and the rounding method is round to nearest even. Using the round to positive infinity rounding method, *unrounded* is incremented when the sign bit is 0 and any of the *lowerBits* are 1. The rounding method round to negative infinity requires that *unrounded* is incremented when the sign bit is 1 and any of the *lowerBits* are 1. This leads to following Boolean expression:

$$roundUp = |lowerBits \,\&\, \begin{pmatrix} \sim round[1] \sim round[0] \sim unrounded[0] | \\ round[1](round \oplus sign) \end{pmatrix}.$$

The exponent will need to be adjusted if the significand needs to be rounded up and adding one to the significand could cause it to be greater than or equal to two. If this is the case, the exponent will need to be incremented and the significand left shifted by one. Since the exponent might be changed, it must be rechecked to determine if the number is still *tiny* and it must also be checked for overflow. In compliance with the IEEE standard [1], once the value has been rounded, it is no longer exact. If the value is inexact, it must be detected which is accomplished by computing the value $|lowerBits\,|roundUp|round[0]$. Just as the initial inputs are checked in to determine if they are denormalized, the rounded product must also be checked.

The flagger implements the Boolean equations: $dividebyzero = 0, overflow = overflow,\ underflow = shiftloss\ \&\ (shiftloss \mid inexact)$, and $inexact = inexact\,|underflow\,|overflow$.

The Assembler in Fig. 2 is composed mainly of multiplexers. The inputs *specialsigncase*, *sign*, *roundmode,* and *overflow* are used to select the correct output. If *specialsigncase* is 1, the most significant bit or sign bit is *specailsign,* otherwise it is set to *sign*.

IV. Dual Recoded Radix-4 Square

Several modifications are made to the floating-point multiplier discussed in the previous section. The first change is in the pre-processor. The pre-processor is simplified and represented by the Boolean expressions:

$$zero = (\sim|Asig)\&(\sim|Aexp)$$

$$aisnan = \sim(\sim|Asig)\ \&\ (\&Aexp)$$

$$infinity = \big((\sim|Asig\,)\&(\&Aexp)\big).$$

The simplified special case detector is also modified. The *specialsigncase* is defined as $aisnan \mid (zero\ \&\ infinity)$ and *specialcase* is defined as$aisnan \mid infinty \mid zero$. *Invalid* equates to $zero\ \&\ infinity$. If *invalid* is set, the *specialsign* becomes the most significant bit *A* if *aisnan* is true (referred to as *aishighernan* in [2]).

The adder circuit in the exponent adder is replaced with a left shift by one. The multiplier is replaced with a full squarer as described in [7]. The squarer is comprised of *n*/2 partial square generators, a sign extension circuit, and an adder tree. The three least significant bits represent a radix-4 digit, -2, -1, 0, 1, or 2. Based on the radix-4 digit value, all but the three least significant bits are negated, shifted, or zeroed. The three least significant bits are defined as:

$$y_2 = \overline{(x_0 \oplus x_1)(x_0 \oplus x_2)(x_0 \oplus x_1)}$$

$$y_1 = 0$$

$$y_0 = \overline{(x_0 \oplus x_1)(x_0 \oplus x_2)} + \overline{(x_0 \oplus x_1)}.$$

The sign extension circuit implements the equation: $1Z_{n-3}1Z_{n-1}\cdots 1Z_5\beta\alpha q_1 q_1$ where $Z_x = \overline{q}_x + q_{x-1}q_{x-2}$, $\beta = \overline{q_3 + q_1}$, and $\alpha = (q_1 \oplus q_3) + q_1 q_2 q_3$. Finally, the output of the partial square generators and sign extension are added together using an adder tree. The normalizer, shifter, rounder, flagger, and assembler are unchanged.

V. Results and Comparison

The floating-point squarer and a floating-point multiplier are both implemented in Verilog and mapped to the OSU standard cell library described in [6]. Both circuits are constrained to a 50ns clock cycle and are implemented for 16, 32, and 64 bit operands. Fig. 3 shows the simulated power characteristics as compared to a multiplier performing the squaring operation.

The pre-processor and pre-normalizer used in the multiplier can easily be split into two square pre-processors and pre-normalizers with no area penalty. The reason that area does not increase is that the dataflow paths of the two inputs in the pre-processor and pre-normalizer are independent of each other. The only exception is the calculation of the *sign* in the pre-processor. Therefore the XOR gates for calculating the sign are moved outside of the pre-processor.

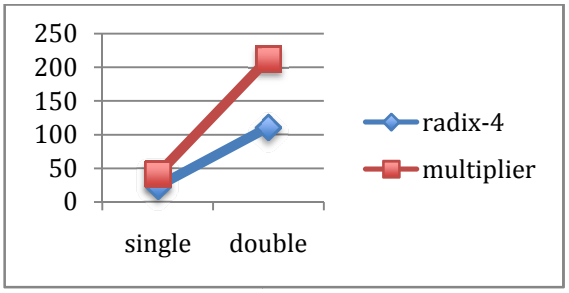

Fig. 3: Power (mW) vs Bit-Width chart

For the special case detector, the inputs *bzero* and *azero* are produced by a 2×1 multiplexer (MUX) to produce *zero*. The MUX is controlled by the input *mult* that is set when the circuit functions as a multiplier and is zero when the circuit functions as squarer. Therefore, the output of MUX is *zero* when *mult* is 0. The same circuit with *ainf* and *binf* is used to calculate the *infinity* flag. Input *B* is ignored if *mult* is 0. The size of the special case detector increases by two 1-bit 2×1 MUXs, two 2-input OR gates, and one *n*-bit 2×1 MUX.

The other modifications to the multiplier circuit to allow it include a squaring unit are shown in Figure 6. Four 2×1 MUXs, a full precision integer squarer, an exponent adder for the square, and an AND gate are required. While a MUX could have been used on one of the inputs of the multiplier exponent adder to eliminate the need for square exponent adder, it was not used because it would have offered only minimal area gains while sacrificing power gains from using a shift left by one instead of an adder to compute the exponent. The AND gate is used in lieu of a 2×1 MUX since the sign of square is always positive.

I. Conclusion

The floating-point squarer implementation results in a significant savings in total power. Since many circuits require a multiplier, this causes the usage of the squarer to become a tradeoff between power and area. The solution to the area issue is the inclusion of both a multiplier and squarer unit that utilizes common circuitry. The shared circuitry overcomes excessive area penalties due to the presence of both units. The components are only enabled as needed for the selected operation thus maintaining most of the power reductions that would occur from using separate circuits. This architectural arrangement justifies the inclusion of individual squaring and multiplication circuitry since the power savings for the squaring operation is achieved with a minimal increase in overall area.

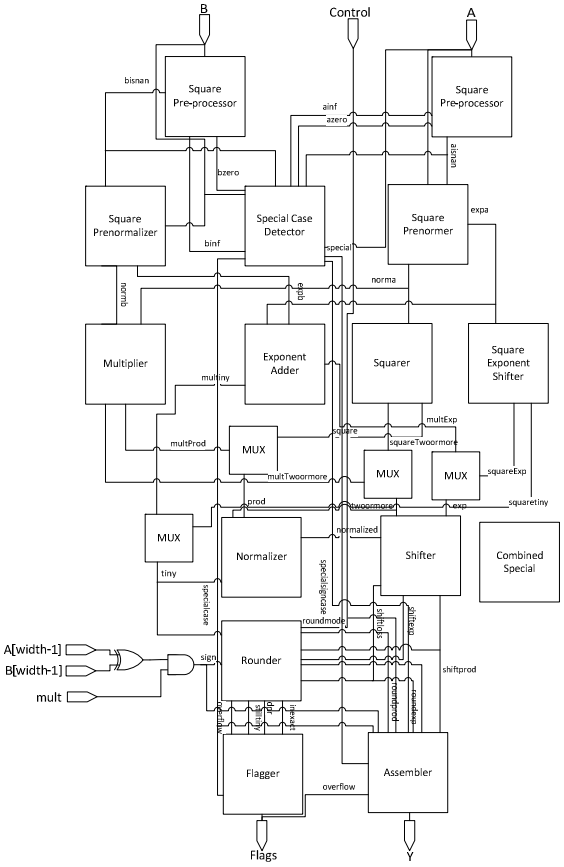

Figure 6: Multiplier/Squarer Combination Circuit

References

[1] IEEE *Standard for Binary Floating-Point Arithmetic*. New York: ANSI/IEEE 754-1985, 1985.

[2] M. E. Phair, "Free Floating-Point Madness: Multiplier", http://www.hmc.edu/chips/fpmul.html, May 14, 2002.

[3] N. Quach, N. Takagi, and M. Flynn, "On Fast IEEE Rounding," *Technical Report CSL-TR-91-459*, Stanford University, January 1991.

[4] G. Even and P-M. Seidel, "A comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication." *IEEE Transactions on Computers*, vol. 49 no. 7 July 2000.

[5] J. Moore, M.A. Thornton, and D.W. Matula, "A Low Power Radix-4 Dual Recoded Integer Squaring Implementation for use in Design of Application Specific Arithmetic Circuits", *IEEE Asilomar Conference on Signals, Systems, and Computers* (ASILOMAR), October 26-29, 2008, pp. 1819-1822.

[6] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, FreePDK: An Open-Source Variation-Aware Design Kit. Proceedings of the *2007 IEEE International Conference on Microelectronic Systems Education* (MSE), 2007, p. 20.

[7] J. Moore, M.A. Thornton, and D.W. Matula, "Dual Recoded Radix-4 Multiplier Circuits", *technical report*, *Dept. of Computer Science and Engineering*, Southern Methodist University, 2013.

[8] S.R. Datla, M.A. Thornton, and D.W. Matula, "A Low Power High Performance Radix-4 Approximate Squaring Circuit", in Proc. of *IEEE Int. Conf. on Application Specific Systems, Architectures, and Processors* (ASAP), 2009, pp. 91-97.

[9] D.W. Matula, "Higher Radix Squaring Operations Employing Left-to-Right Dual Recoding," pp.39-47, 2009 19th *IEEE Symposium on Computer Arithmetic* (ARITH), 2009

[10] P. Kornerup and D.W. Matula, **Finite Precision Number Systems and Arithmetic**, Cambridge Univ. Press, 2010, pp.262-269.

[11] A.G.M. Strollo and D. De Caro, "Booth Folding Encoding for High Performance Squarer Circuits" *IEEE Trans. Circuits and Systems-I1 Analog and Digital Signal Processing,* 50(5):250-254, 2003.

[12] D. De Caro and A.G.M. Strollo, "Parallel squarer using Booth-folding technique," *Elec. Lett.*, vol. 37, no. 6, pp. 346–347, Mar. 2001.

[13] A.G.M. Strollo, E. Napoli, and D. De Caro, "New design of squarer circuits using Booth encoding and Folding techniques," in *Proc. 8th IEEE Int. Conf. on Electronics, Circuits and Systems* (ICECS), 2001, pp. 193–196.

[14] B. Parhami, **Computer Arithmetic: Algorithms and Hardware Designs**, Oxford University Press, 2000, pp. 181-182.