A New Architecture For Multiple-Precision Floating-Point Multiply-Add Fused Unit Design

Libo Huang, Li Shen, Kui Dai, Zhiying Wang
School of Computer
National University of Defense Technology
Changsha, 410073, P.R.China
{libohuang, lishen, kuidai, zywang}@nudt.edu.cn

Abstract

The floating-point multiply-add fused (MAF) unit sets a new trend in the processor design to speed up floating-point performance in scientific and multimedia applications. This paper proposes a new architecture for the MAF unit that supports multiple IEEE precisions multiply-add operation (A×B+C) with Single Instruction Multiple Data (SIMD) feature. The proposed MAF unit can perform either one double-precision or two parallel single-precision operations using about 18% more hardware than a conventional double-precision MAF unit and with 9% increase in delay. To accommodate the simultaneous computation of two single-precision MAF operations, several basic modules of double-precision MAF unit are redesigned. They are either segmented by precision mode dependent multiplexers or attached by the duplicated hardware. The proposed MAF unit can be fully pipelined and the experimental results show that it is suitable for processors with floating-point unit (FPU).

1. Introduction

Floating-point multiply-add fused (MAF) operation becomes one of the key features among current processors [1, 2, 3, 4, 15, 16]. It combines two basic operations into one with only one rounding error. Besides the increased accuracy, this fused implementation can also minimize operation latency, reduce hardware cost and chip busing [4].

In this paper, we describe a new architecture for MAF unit capable of supporting one double-precision or two single-precision operations in parallel. Two observations have motivated this study.

First, single-precision floating-point operations have been widely used in multimedia and even scientific application. A number of recent commercial processors exhibit significantly higher performance for single-precision than for double-precision. Examples include the Intel Pentium, the AMD Opteron, the IBM PowerPC and CELL [2, 5]. The ratio is up to two times faster on the Pentium and up to ten times faster on the CELL. If we use double-precision unit to support two single-precision operations, the performance of single-precision will be further improved.

Second, as the development of floating-point SIMD instruction sets such as Intel's SSE, AMD's 3DNow, and PowerPC's Altivec, many FPUs add the feature of SIMD [1, 2, 3]. But most of them are realized by simply duplicating the functional units, which consume much hardware. If we replace the double-precision unit with a multiple-precision one, the hardware cost can be saved to get the same performance of single-precision.

The remainder of this paper is organized as follows. Section 2 reviews the related work on SIMD arithmetic unit and introduces a conventional implementation of high speed MAF unit. This implementation serves as a baseline for later description. In Section 3, the general architecture of the proposed multiple-precision floating-point MAF unit is presented. After that, Section 4 describes some modules of the floating-point MAF unit in details. Then, the area and delay estimation for the design is presented in Section 5. Finally, Section 6 gives the conclusion of the whole work.

2. Background

2.1. Related work

The study of SIMD arithmetic unit starts with fixed-point unit. Many fixed-point optimized subword-parallel hardware structures reducing the area and cycle delay have been developed, such as subword-parallel adders [7], multiple-precision multipliers and multiply-add (MAC) units using booth encoding [8] as well as not using booth encoding [9].

Researches on floating-point SIMD arithmetic units are

relatively rare [10, 11, 12]. This is partly because the floating-point arithmetic algorithm is somewhat complicated and the sharing between different precision units is not so easy. Pioneer work is done in [10] which extends a double-precision FPU to support single-precision interval addition/subtraction or multiplication. It proposes a method that how double-precision units can be split to compute two bounds of a single precision interval in parallel. In [11] a dual mode multiplier is presented, which uses a half-sized (i.e. 27×53) multiplication array to perform either a double-precision or a single-precision multiplication with that the double-precision operation needs two cycles. The multiplier in [12] uses two double-precision multipliers plus additional hardware to perform one quadruple precision multiplication or two parallel double-precision multiplications.

The main contribution of this paper is proposing and evaluating a new architecture for floating-point unit with SIMD feature. This is achieved by modifying several basic modules of the traditional double-precision unit. As an example, a multiple-precision MAF unit is designed and it is fully pipelined so that both precision MAF operations can be started every cycle.

2.2. Conventional MAF unit

In this paper, the MAF operation A×B+C is implemented for IEEE floating-point format [13]. In this format, a floating-point number X represents the value $X = (-1)^s \times f \times \beta^{e-p}$, where s, f, β, e and p are integers with s, the sign bit; f, the normalized mantissa in the range [1, 2); β, the radix, 2 for binary; e, the biased exponent; and p, the biased number, 127 for single-precision and 1023 for double-precision. These fields with different precisions are shown in Figure 1. Our numbering convention starts with zero and with the least significant bit (LSB) first; that is the LSB is numbered as bit zero.

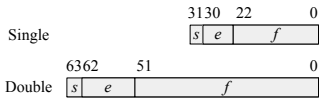


Figure 1. IEEE Single/double-precision format

We first make a brief introduction to the conventional MAF unit design as it servers as baseline for proposed MAF unit. Figure 2 illustrates the basic architecture of its implementation [14], which is fairly typical and used in some recent floating-point units of commercial processors [2, 15, 16]. It consists of three pipelined dataflow stage:

1. Multiply stage produces an intermediate product of A×B in carry-save representation; aligns the inversed C as a right shift by placing C to two bits left of the most significant bit (MSB) of A×B. The two extra bits is to allow correct rounding when A is not shifted.
2. Add stage carries on the addition of C and A×B product using a 3-2 CSA and a carry-propagate adder, determines the normalize shift count by means of LZA.
3. Normalize/Round stage involves normalizing and rounding the intermediate result to get the final result.

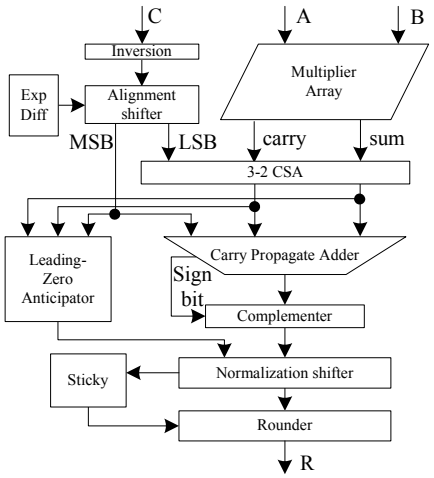


Figure 2. Basic structure of double-precision floating-point MAF unit

Table 1 lists the wordlengths of operands inside conventional MAF unit datapath for single/double-precision. Note that in exponent processing, two extra bits added in both precision modes are used for indicators of negative result and overflow.

Table 1. Wordlengths in the Single/double-precision MAF unit

modules	single	double
Multiply array	24	53
3-2 CSA	48	106
Alignment&Adder&Normalization	74	161
Exponent Processing	10	13

Several optimization techniques for this basic structure have been introduced [6, 17]. The modules of these MAF implementations do not have distinct differences, so in this paper we only discuss how to modify the basic MAF architecture to support multiple-precision MAF operations but the sharing method can be generalized to other MAF architectures.

3. General architecture of multiple-precision MAF unit

We now give an overview of the proposed multiple-precision MAF unit structure, and leave the implementation details to the next section. Figure 3(a) illustrates the 64-bit double-precision register which can be used to store two single-precision numbers and Figure 3(b) shows the generated results when performing two single-precision MAF operations.

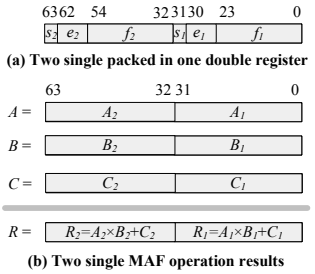


Figure 3. Two single-precision numbers packed in one double-precision register

Using one double-precision unit to support two single-precision parallel operations, we need a modification for each module. The MAF architecture can be considered as an exponent and a mantissa unit. From table 1, we can see that for exponent processing, the wordlength of 13-bit double-precision exponent units should be extended to 20 bits to deal with two single-precision computations (10 bits$\times$2). But for speed reason, a duplicated separate single-precision exponent processing unit is used in our design. Even thus, it has slightly influence on the MAF unit because the exponent processing consumes little area and delay. The double-precision mantissa units are always two times wider than single-precision mantissa units, so it is possible to share most of the hardware by careful design.

Algorithm 1 shows the simplified multiple-precision floating-point MAF algorithm which focuses on the mantissa datapath. As the implementation of multiple-precision MAF unit should select carefully the position of the single-precision operands inside the double-precision datapath, we give detailed information about datapath wordlengths. In the algorithm, s_a, e_a,f_a denote the sign, exponent and mantissa of input operand A respectively. This is the same as input operands B and C. The control signal *double* is employed, and for double-precision operation, *double* is set. Signal $x[m:n]$ represents the binary numbers of x from index n to m. *s.sub*, $s.sub_1$ and $s.sub_2$ in Step 3 indicate the signs of the effective mantissa addition operations for one double and two single-precision operations respectively. *s.sub*, for example, is computed by *s.sub*=$s_a \oplus s_b \oplus s_c$.

Algorithm 1 Multiple-Precision MAF Algorithm

Require: *A, B, C must be normalized numbers*
◇ Step 1 Exponent Difference : $\delta[19:10]$
if $double = 1$ **then**
$\delta[12:0] = e_a[12:0] + e_b[12:0] - e_c[12:0] - 967$
else $\{double = 0\}$
$\delta[9:0] = e_a[9:0] + e_b[9:0] - e_c[9:0] - 100$
$\delta[19:10] = e_a[19:10] + e_b[19:10] - e_c[19:10] - 100$
end if
◇ Step 2 Mantissa Product : $fprod[105:0]$
if $double = 1$ **then**
$fprod[105:0] = f_a[52:0] \times f_b[52:0]$
else $\{double = 0\}$
$fprod[47:0] = f_a[23:0] \times f_b[23:0]$
$fprod[96:49] = f_a[48:25] \times f_b[47:24]$
end if
◇ Step 3 Alignment and negation : $fca[160:0]$
if $double = 1$ **then**
$fca[160:0] = (-1)^{s.sub} \times f_c[52:0] \times 2^{-\delta[12:0]}$
else $\{double = 0\}$
$fca[73:0] = (-1)^{s.sub_1} \times f_c[23:0] \times 2^{-\delta[9:0]}$
$fca[148:75] = (-1)^{s.sub_2} \times f_c[47:24] \times 2^{-\delta[19:10]}$
end if
◇ Step 4 Mantissa Addition : $facc[160:0]$
$facc[160:0] = fprod[105:0] + fca[160:0]$
◇ Step 5 Complementation : $faccabs[160:0]$
if $double = 1$ **then**
$faccabs[160:0] = |facc[160:0]|$
else $\{double = 0\}$
$faccabs[73:0] = |facc[73:0]|$
$faccabs[148:75] = |facc[148:75]|$
end if
◇ Step 6 Normalization : $faccn[160:0]$
if $double = 1$ **then**
$faccn[160:0] = norm\_shift(faccabs[160:0])$
else $\{double = 0\}$
$faccn[73:0] = norm\_shift(faccabs[73:0])$
$faccn[148:75] = norm\_shift(faccabs[148:75])$
end if
◇ Step 7 Rounding : $fres[51:0]$
if $double = 1$ **then**
$fres[51:0] = round(faccn[160:0])$
else $\{double = 0\}$
$fres[22:0] = round(faccn[73:0])$
$fres[45:23] = round(faccn[148:75])$
end if

With above algorithm, we get the hardware architecture of the proposed MAF unit, as shown in Figure 4. It is pipelined for a latency of three cycles, confirming to traditional three steps. For simplicity, the exponent computation block, the sign production block, and the exception handling block are not illustrated in the figure.

The resulting implementation have some similar characteristics with the conventional MAF unit except for the following aspects:

1. Precision mode-dependent multiplexers. These multiplexers in our design are controlled by the signal *double* to select the corresponding precision operands.

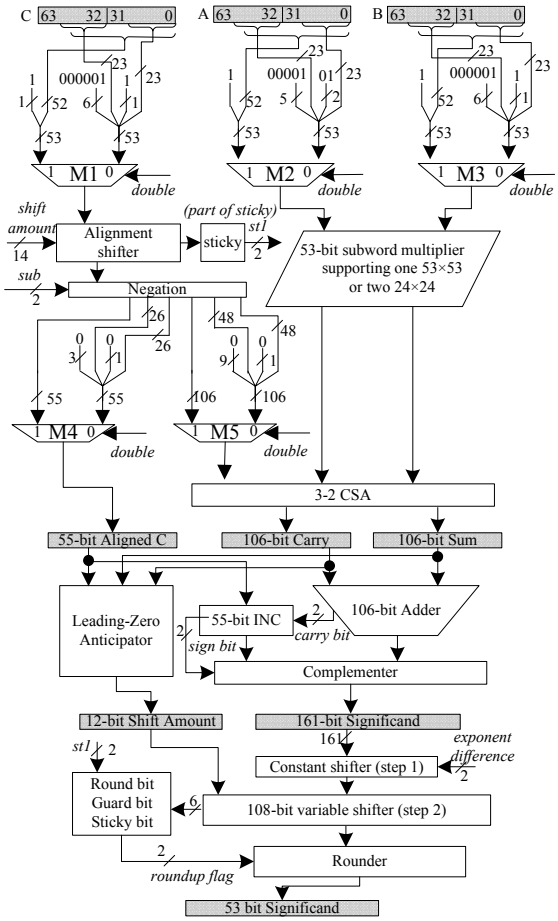


Figure 4. General structure of multiple-precision MAF unit

2. Vectorized modules. All the modules in the MAF unit are able to handle two parallel single-precision datapath, so the double-precision modules should be vectorized.

3. Two's complement. In our design, operands in the datapath are represented by two's complement. This allows us to simplify the vectorization of many modules such as adder. Moreover, the end around carry adjustment needed for effective subtraction in one's complement adder can be avoided.

4. Special LZA with concurrent position correction [19] is used for handling both positive and negative adder results.

5. Two-step normalization. Since operand C is a normalized number, the normalization shifter can be implemented as a first constant shift (53 for double-precision, 24 for single-precision) followed by a variable shifter (108-bit for double-precision, 50-bit for single-precision) in such a way that the first shift is controlled by exponent difference and the second shift is controlled by LZA [6].

The main building blocks of proposed architecture are: exponent processing, multiplier, alignment shifter, adder, LZA, normalization shifter, rounder. Section 4 will give the detail description of these individual modules.

4. Modules implementation

4.1. Sign and exponent processing

We use two bits to deal with the sign processing. They have either 1-bit zero followed by 1-bit double-precision sign or 2-bit single-precision signs in the datapath according to the *double* signal. For example, the sign of the product A×B is determined by $s_a \oplus s_b$, as shown in Figure 5.

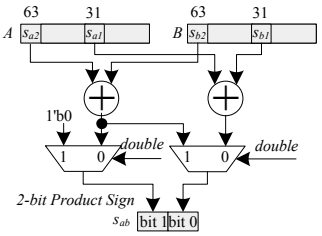


Figure 5. Method of sign determination

The exponent difference d can be denoted as $d = e_c - (e_a + e_b - p)$, where p=1023 for double-precision, p=127 for single-precision. Then, the shift amount is $\delta = L - d$, where L is the number of C left of A×B, 56 for double-precision, 27 for single-precision. So shift amount equals to $e_a + e_b - e_c - 967$ for double-precision, and $e_a + e_b - e_c - 100$ for single-precision.

We expand the exponent processing unit by one single-precision datapath. Figure 6 shows the modification of original exponent processing unit. The additional single-precision datapath which is similar to the double-precision is not shown. In the figure, additional 1 is added to the Adder 2 to complete the two's complement of e_c. This is achieved by adding two 1s in the right position of the adder. The shift adjust module selects the right shift amount which must be in the range [0, MAXSHIFT], where MAXSHIFT is 161 for double-precision and 74 for single-precision.

4.2. Alignment shifter

The C alignment shifter can be implemented as a variable right-shifter by positioning the B operand mantissa left of the binary point of the A×B product before the shift count is calculated. For double-precision, the shifter is 161

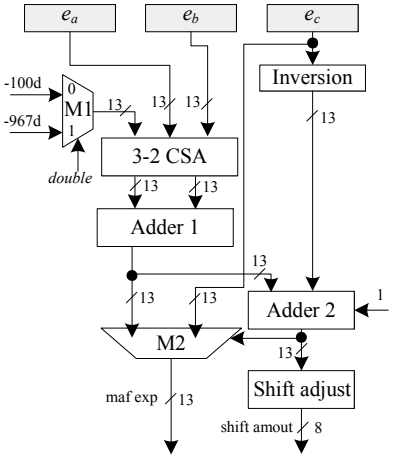


Figure 6. The implementation of exponent processing for double-precision part

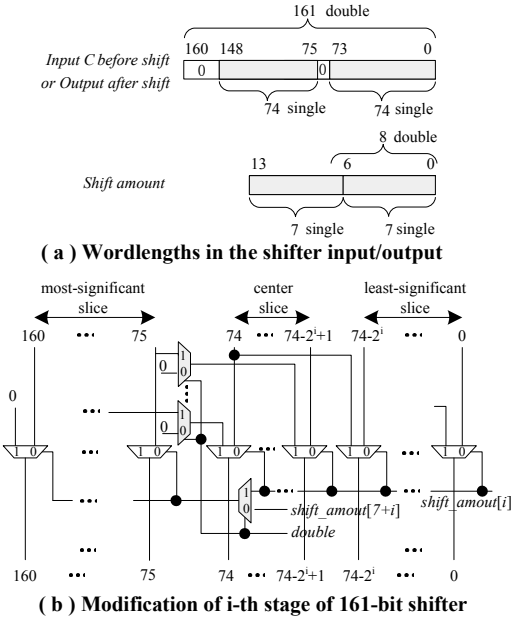


Figure 7. 161-bit modified shifter supporting two 74-bit shifts

bits, and for single-precision, the shifter is 74 bits. As the alignment shifter is not on the critical datapath, we can use one 161-bit shifter to support two parallel 74-bit shifts [10]. Figure 7(a) illustrates the wordlengths in the input/output of shifter and Figure 7(b) shows how it works for one stage of the modified shifter. It is split by some multiplexers. As can be seen from the figure, all wires are divided into three slices: a most significant slice (bits $[160 : 75]$), a center slice (bits $[74 : 74 - 2^i + 1]$), and a least significant slice (bits $[74-2^i : 0]$), where *i* is the index of shifter stages ($0 \sim log_2 161$). The shifting of most significant slice is controlled by $shift\_amount[i]$ or $shift\_amout[7 + i]$ according to precision mode, so it needs one more multiplexer; The shifting of center slice is controlled by $shift\_amount[i]$, but the shift in bits should be selected according to precision mode, so additional 2^i multiplexers are required; the least significant slice requires no more multiplexers. Since the last shift stage is not needed in the single-precision mode, we therefore need $\sum_{i=0}^{6} 2^i + 1 = 134$ additional multiplexers. Compared with usual 161-bit shifter consists of $8 \times 161 = 1288$ multiplexers, the overhead is about 10%.

4.3. Multiplier

The multiplier in the proposed MAF unit is able to perform either one 53-bit or two parallel 24-bit multiplications. Two methods can be used to design the multiplier, one is booth encoding [8], and the other is array multiplier [9]. Although booth encoding can reduce the number of partial products to half and make the compression tree smaller, it adds the complexity of control logic when handling two precision multiplications, which increases the latency. Furthermore, it requires detection and suppression of carries across subword boundaries in reduction tree and final carry-propagate adder (CPA). This is not suitable for the multi-precision MAF unit design because the product of booth encoding in carry-save representation contains the subword carry bits which should be prevented at single-precision mode. One way to eliminate the contradiction is to calculate the final multiplication result first, and then add with aligned C. But this requires more delay and hardware. So array multiplier is used for the proposed multiple-precision MAF unit. Although it produces a larger compression tree, it does not require booth decoding and detection and suppression in reduction tree and CPA, which avoids the contradiction encountered in the booth encoding scheme.

The block diagram of the multiplier A×B is shown in Figure 8. In this figure, input operands A and B are selected according to the method shown in Figure 4. One bit zero is inserted between the two 24-bit operands A_2 and A_1, which is used to preserve carry bit in later addition. The white region of the partial product can be written as $double \cdot a_i b_j$, and the gray region can be written as $a_i b_j$, where a_i, b_j denote the i-th bits of A and j-th of B respectively. When double-precision is performed, the partial product is generated similar to that of a traditional 53-bit multiplier; when the single-precision is performed, the white region is set to zero and the gray region are two partial products of 24-bit multiplication respectively. This enables two single-precision multiplications working in parallel.

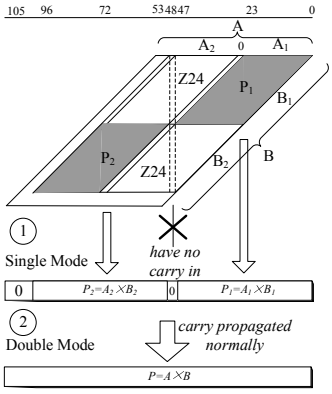


Figure 8. 53-bit multiple-precision mantissa multiplier

4.4. Two's complement of aligned C

The two's complement of aligned C must be performed in case of effective subtraction. As shown in Figure 4, we place the inversion after alignment which is different from the conventional MAF unit completing inversion before alignment. This change allows the shifter only to shift in 0 during alignment while in conventional design, 0 or 1 may need to be shifted in. After inversion, an additional 1 has to be added to the LSB. Since after the CSA tree of multiplier there will be one empty slot of carry word, we could add 1 in that position. Figure 9 shows the wordlengths. We can see that the only case needs to add 1 is when $sub = 1$ (effective subtraction) and $st1 = 0$ (partial sticky).

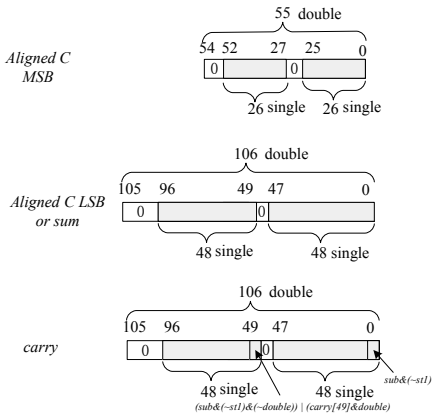


Figure 9. Wordlength before 3-2 CSA

4.5. Mantissa adder and LZA

The 161-bit mantissa adders can be implemented as one upper 55-bit incrementer and one low order 106-bit CPA. The carry-in of upper incrementer is fed by the carry-out of CPA. Note that this CPA does not need change because one extra bit is inserted between two single-precision addends. Figure 10 illustrates this mantissa adder.

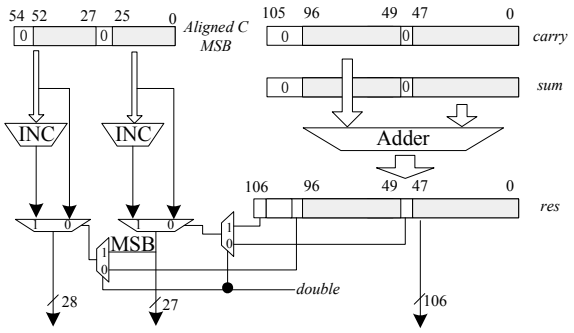
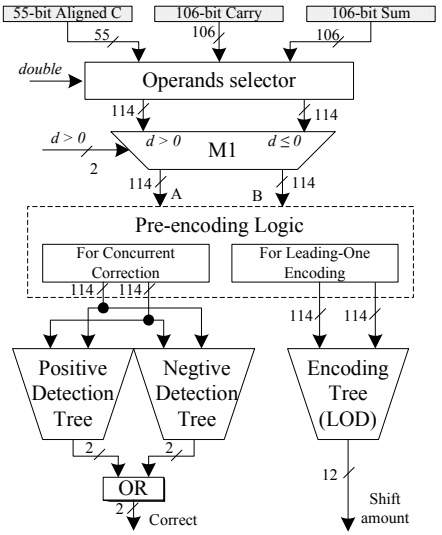
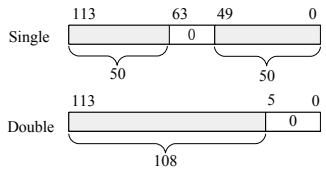


Figure 10. Structure of mantissa adder

The LZA predicts the leading digit position in parallel with the addition step so as to enable the normalization shift to be started as the addition completes. But the prediction result might have an error of one bit. To correct this error, some concurrent position correction methods have been introduced [18, 19].



(a) Structure of LZA



(b) Wordlengths in the operands A and B

Figure 11. Structure of LZA

The proposed LZA supports either 108-bit prediction or two 50-bit predictions. Figure 11(a) shows the structure of the LZA. Its operands are input from carry, sum and aligned C registers and then selected by operands selector and multiplexer M1 to get the 114-bit addend A and B. Every addend is divided into two sections, from which we can obtain two 50-bit addition leading-zero positions, and when the two sections are gathered together we can obtain 108-bit addition leading-zero position, as shown in Figure 11(b). This is because tail zeros of the operands does not influence the leading zero prediction results.

When the addition result is negative, the number of leading ones is predicted. But the number of leading zeros of its absolute value is actually needed. In order to make the leading one predicted result match the number of leading zeros of its absolute value, the increment by 1 for two's complement of the addends is left behind the normalization shifting.

We use the string $F = f_{n-1} \cdots f_0$ to identify the leading-one position. The pre-encoding of the LZA has the unified form of positive and negative number as shown in following equations [18].

$$T = A \oplus B,\ G = AB,\ Z = \overline{A}\,\overline{B}$$

$$f_i = \begin{cases} \overline{T_i} T_{i-1}, & \text{if } i = n-1 \\ T_{i+1}(G_i \overline{Z_{i-1}} + Z_i \overline{G_{i-1}}) & \\ \quad + \overline{T_{i+1}}(Z_i \overline{Z_{i-1}} + G_i \overline{G_{i-1}}), & \text{if } 0 \le i < n-1 \end{cases}$$

Once the string F has been obtained, the position of the leading one has to be encoded by means of a LOD tree [19]. Figure 12 shows the structure of LOD tree. For double-precision, it can encode the 108 most significant bits of F; and for single-precision, it can encode the 50 most significant bits of F and 50 least significant bits of F respectively. Using the method described in LOD tree, we can also design the error detection module [19] which determines one 108-bit or two 50-bit detection results as not shown in this paper.

4.6. Normalization and rounding

We can use the same sharing method for alignment shifter to implement 108-bit normalization shifter. Figure 13 illustrates the block diagram of normalization and rounding unit. To reduce the rounding delay, another duplicated rounding unit is used for single-precision.

5. Performance evaluation

With respect to the hardware cost, we consider the additional resource needed for the main building blocks of

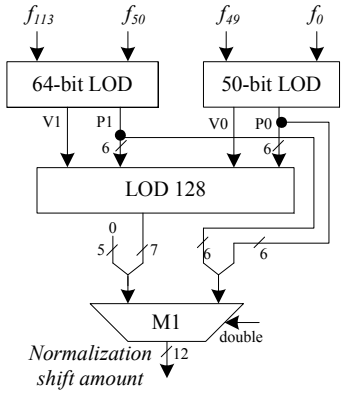


Figure 12. Structure of LOD tree

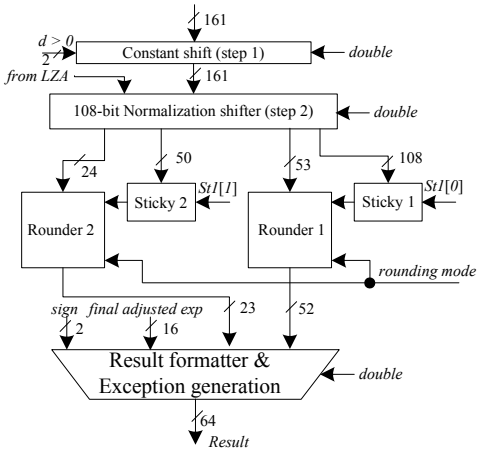


Figure 13. Diagram of rounding unit

MAF. Table 2 shows additional resources which helps to understand the complexity of the resulting MAF unit.

For comparison, single/double-precision MAF units are also designed. They use the same performance-enhancement techniques as the multi-precision MAF. All the MAF units are pipelined for a latency of three cycles and a maximum throughput of one result per cycle. They are implemented in Verilog HDL at a structural level and then synthesized using the Synopsys Design Compiler and the TSMC 0.18 micron CMOS standard cell library. The designs were optimized for speed with an operating voltage of 1.8V, and a temperature of 25°C.

The area and delay comparison results of the MAF units are given in table 3 and table 4 separately. In the two tables, area and worst delay estimates are given for each pipeline stage, along with the total area and overall worst case delay. The area of single-precision MAF unit is about 32% of double-precision MAF unit. Compared to the conventional general purpose double-precision floating-point MAF unit, the multi-precision floating-point MAF unit has roughly

18% more area cost and its worst case delay is roughly 9% longer.

Table 2. Additional resources to the main blocks of MAF

Main blocks	Additional resources
Multiply	Several multiplexers
Alignment shifter	Several multiplexers
Adder	Several multiplexers
LZA	Width expands to 114-bit
Normalize	Several multiplexers
Rounder	One single-precision rounder
Exponent Processing	One single-precision datapath

Table 3. Area Estimation (μm^2)

Stages	Multiple	Double	Single
Multiply & Align	531947	469514	137233
Add & LOP	107669	82081	35369
Normalize & Round	68974	49098	20815
Total Area	708590	600693	193417

Table 4. Delay Estimation (ns)

Stages	Multiple	Double	Single
Multiply & Align	3.40	3.11	2.56
Add & LOP	3.34	3.10	2.53
Normalize & Round	3.38	3.04	2.54
Worst Case Delay	3.40	3.11	2.56

6. Conclusion

This paper presents a new architecture for multiple-precision floating-point MAF unit design capable of performing either one double-precision floating-point MAF operation or two single-precision floating-point MAF operations in parallel. Each module of the traditional double-precision MAF unit is vectorized by sharing between multiple precision operations or duplicating hardware resources when the module is on the critical datapath. This method can also be extended to other floating-point operations, such as multiple precision floating-point addition or multiplication for implementing SIMD instruction sets like 3D NOW! or SSE.

References

[1] Cornea M., et. al., "Intel Itanium^{TM} Floating-Point Architecture", *WCAE*, San Diego, 2003.

[2] S. M. Mueller, et. al., "The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor", *ARITH-17*, 2005

[3] S. Chatterjee, L. R. Bachega, "Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L", *IBM J. of Research and Development*, vol. 49, pp. 377-392, 2005

[4] R. K. Montoye, et. al., "Design of the IBM RISC System / 6000 Floating-Point Execution Unit", *IBM J. of Research and Development*, Vol. 34, pp. 61-62, 1990

[5] J., Langou, et. al., "Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy", *University of Tennessee Computer Science Tech Report*, UT-CS-06-574, 2006

[6] Chen, L. and Cheng, J. "Architectural Design of a Fast Floating-Point Multiplication-Add Fused Unit Using Signed-Digit Addition". *Proceedings of the Euromicro Symposium on Digital Systems Design*. pp.346, 2001.

[7] M. Senthilvelan and M. J. Schulte, "A Flexible Arithmetic and Logic Unit for Multimedia Processing", *Proceedings of SPIE : Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, pp. 520-528, August, 2003.

[8] D. Tan, A. Danysh, and M. Liebelt, "Multiple-Precision Fixed-Point Vector Multiply-Accumulator using Shared Segmentation", *ARITH-16*, pp. 12-19, 2003.

[9] S. Krithivasan and MJ Schulte, "Multiplier Architectures for Media Processing", *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, pp. 2193-2197, 2003.

[10] R. Kolla, et. al., "The IAX Architecture : Interval Arithmetic Extension", *Technical Report 225*, Universitat Wurzburg, 1999

[11] G. Even, S. Mueller, and P.-M. Seidel, "A Dual Mode IEEE multiplier". *Proc of the 2nd IEEE Int. Conf. on Innovative Systems in Silicon*, pp. 282-289, 1997.

[12] A. Akkas and M. J. Schulte, "A Quadruple Precision and Dual Double Precision Floating-Point Multiplier", *Proceedings of the 2003 Euromicro Symposium on Digital System Design*, pp. 76-81, 2003.

[13] ANSI/IEEE standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[14] Romesh M. Jessani, Michael Putrino, "Comparison of Single- and Dual-Pass Multiply-Add Fused Floating-Point Units", *IEEE Transactions on Computers*, v.47 n.9, p.927-937, September 1998.

[15] Hokenek E, Montoye R, Cook P W. "Second-Generation RISC Floating Point with Multiply-Add Fused". *IEEE J.Solid-State Circuits*, 25(5): 1207-1213, 1990.

[16] O'Connell F P, White S W. "POWER3: The Next Generation of PowerPC Processors", *IBM J.Research and Development*, 44(6): 873-884, 2000.

[17] T. Lang and J. Bruguera, "Floating-point fused multiply-add wth reduced latency", *ICCD*, 2002.

[18] Martin S. Schmookler , Kevin J. Nowka, "Leading Zero Anticipation and Detection A Comparison of Methods", *ARITH-15*, p.7, June 11-13, 2001

[19] J. Bruguera, and T. Lang, "Leading-One Prediction with Concurrent Position Correction", *IEEE Trans. on Computers*, vol. 48, pp. 298-305, Oct. 1999.