Enhanced Floating-Point Multiply-Add with Full Denormal Support

Jongwook Sohn, David K. Dean, Eric Quintana and Wing Shek Wong
Intel Corporation
Austin, TX, USA
Email: jongwook.sohn, david.k.dean, eric.quintana, wing.shek.wong @intel.com

***Abstract*—This paper presents an enhanced floating-point multiply-add (FMA) design for the Intel E-Core processor. FMA is one of the most widely used operation in many applications. The proposed FMA is executed in 4 cycles, fully pipelined, handles SSE/AVX operations for scalar/packed IEEE single and double precision, and supports all four rounding modes. Also, the proposed FMA fully supports both denormal inputs and underflow outputs without microcode assistance. To achieve the 4-cycle FMA with full denormal support, several optimization techniques are applied: one-way alignment, radix-16 Booth encoding for the multiplier, merged J-bit correction and aligned significand with the multiply array, modified leading zero anticipation (LZA) for masking the underflow, parallel sticky and all-ones detection with the normalization, and merged two's complement with the rounding logic. As a result, the proposed FMA achieved not only full denormal support but also about 10 – 30% reduced area and about 10 – 20% reduced latency compared to the traditional FMAs.**

***Index Terms*—Floating-point multiply-add, floating-point denormal numbers, floating-point arithmetic, high-speed computer arithmetic**

I. Introduction

A steep rise of complex applications such as graphics, machine learning and artificial intelligence has increased the importance of high-speed computer arithmetic. Those advanced applications handle a wide and dynamic range of data processing, and require floating-point representation, which is specified in *IEEE Standard for floating-point arithmetic* [1]. Floating-point multiply-add (FMA) is one of the most frequently used operations in many applications. The FMA consists of complex processes like alignment, normalization and rounding, which have large latencies, areas, and power requirements. Therefore, improving design for the FMA will contribute to the next generation floating-point arithmetic unit development.

This paper presents an enhanced FMA design for the Intel E-Core processor [2]. The proposed FMA takes three 128-bit vectors of floating-point numbers and executes two 64-bit FMA units (i.e., one 64-bit double precision or two 32-bit single precision in each unit) to compute the result.

$$Z = (A \times B) \pm C \quad (1)$$

The proposed FMA is executed in 4 cycles, fully pipelined, handles SSE/AVX operations for scalar/packed IEEE single and double precision, and supports all four rounding modes as specified in the IEEE Standard 754 [1]. Since handling denormal numbers requires more complex processes, traditional floating-point units only deal with normal numbers. In this case, a microcode exception handler is required to compute the denormal numbers, which takes cycles of additional delay and significantly degrades the performance. Many researches have been conducted to develop floating-point units handling denormal numbers in hardware to avoid the penalty of the microcode assistance [3] – [5]. The proposed FMA fully supports denormal numbers with no additional delay making the microcode assistance unnecessary.

Several optimization techniques are applied to achieve the 4-cycle FMA with full denormal support:

1) One-way significand alignment is performed with the addend significand based on the exponent difference in parallel with the multiplier. The aligned significand is inserted into the multiply array to eliminate the additional CSA at the end of the multiplier. Also, the sticky logic is performed in parallel with the alignment, which is used for the rounding logic.
2) Radix-16 Booth encoding is used for area and power reduction. Although the radix-16 Booth encoding requires the pre-computations for multiples, it produces about half the partial products compared to the radix-4 Booth encoding (14 vs. 27), which reduces two levels of CSAs in the multiply array. As a result, the radix-16 Booth encoding spends a lot less area and power with about the same latency compared to the radix-4 encoding.
3) J-bit is an implicit bit above the MSB of the significand, which is zero if it is denormal. The addend J-bit is detected in parallel with the first level of the exponent difference logic so that there is no delay penalty. The other two operands, however, need to be directly passed to the multiplier, so the J-bit detections delay the critical path. To avoid the delay, the proposed FMA assumes the both J-bits are ones, then subtracts one J-bit correction line in the multiply array, which requires a more partial product line and a few bits for two's complement, but they are merged with the existing CSAs and there is no additional delay.
4) Leading zero anticipation (LZA) is applied to speed up the normalization. The LZA is performed in parallel with the main adder so that the normalization is performed right after the main adder. Also, the LZA is modified to handle the underflow when the exponent is negative after the normalization. The modified LZA stops the normalization shift if the

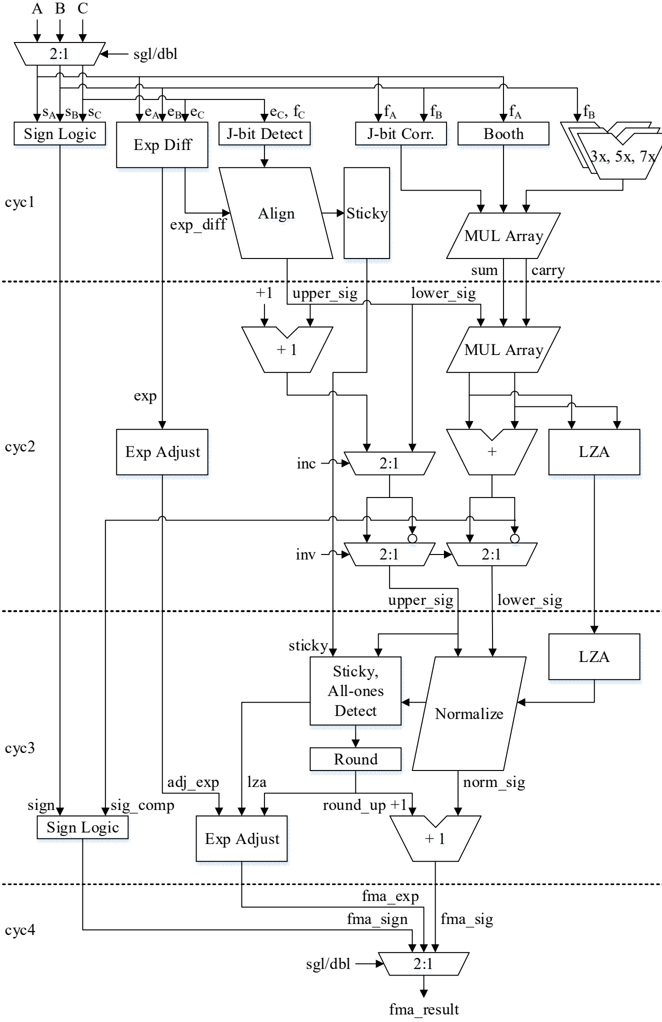

Fig. 1. Proposed Floating-Point Multiply-Add

exponent becomes zero so that the denormalization shift is unnecessary, which significantly reduce the latency.

5) The sticky and all-ones detection logic is performed in parallel with the normalization to speed up the rounding logic. The detection logic allows the early roundup decision so that it is directly passed to the incrementor for the rounding.
6) Two's complement for the main adder is merged with the rounding logic to avoid an additional MUX after the main adder. The two's complement is propagated to the rounding logic and forces the roundup of the significand result.

More details to improve the FMA design are presented in the following sections.

II. Traditional Floating-Point Multiply-Adds

A generic algorithm of the floating-point multiply-add is used for the traditional FMA designs [6], [7]. Many studies have been reported to improve the FMA design. One-way alignment with the addend significand is applied to perform it in parallel with the multiplier [6] – [8]. Variations of Booth encodings

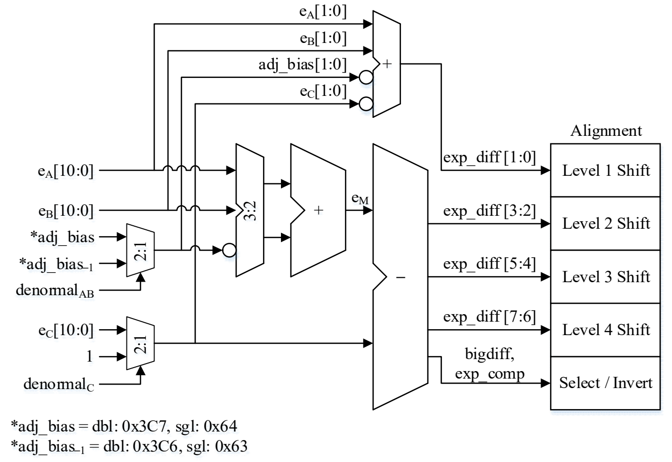

Fig. 2. Exponent Difference Logic

and reduction trees are introduced to reduce the number of partial products of the multiplier [10] – [12]. Also, LZA is applied to normalize the significand earlier by predicting the shift amount in parallel with the main adder [6] – [9]. Rounding logic is performed in parallel with the main adder to reduce the latency [8]. Finally, the FMAs are split into 4 – 8 cycles and pipelined to improve the performance [6] – [8]. The proposed FMA not only combines the optimizations previously introduced, but also applies further optimizations to speed up and support denormal numbers.

III. Enhanced Floating-Point Multiply-Add

In this section, the optimization techniques are described to achieve the 4-cycle FMA with full denormal support. Fig. 1 shows a 64-bit unit of the proposed FMA. Three 64-bit floating-point numbers are formatted into either of one double precision or two single precision values. They consist of sign, exponent and significand bits – A = (s_A, e_A, f_A), B = (s_B, e_B, f_B), and C = (s_C, e_C, f_C). The first cycle contains logic for the exponent difference, alignment, Booth encoding, and the first part of the multiply array. The second cycle contains the rest of the multiply array, main adder, incrementor and LZA logic. The normalization and rounding are performed in the third cycle, and fourth cycle contains the last MUX and bypass/writeback logic.

The proposed FMA performs the one-way significand alignment shift with the addend significand based on the exponent difference. Simultaneously, the multiplier is performed with the other two significands. The aligned addend significand is merged with the multiply array to reduce the latency. The sum and carry values from the multiply array are passed to the main adder and incrementor. The LZA is performed in parallel with the main adder and it is used for normalization. The normalized significand is then rounded and passed to the last MUX to determine the special cases and precisions.

A. Exponent Difference and Alignment

The first cycle begins with the exponent difference logic to determine the significand alignment shift amount as shown in Fig. 2. Exponent difference is implemented by subtracting e_C

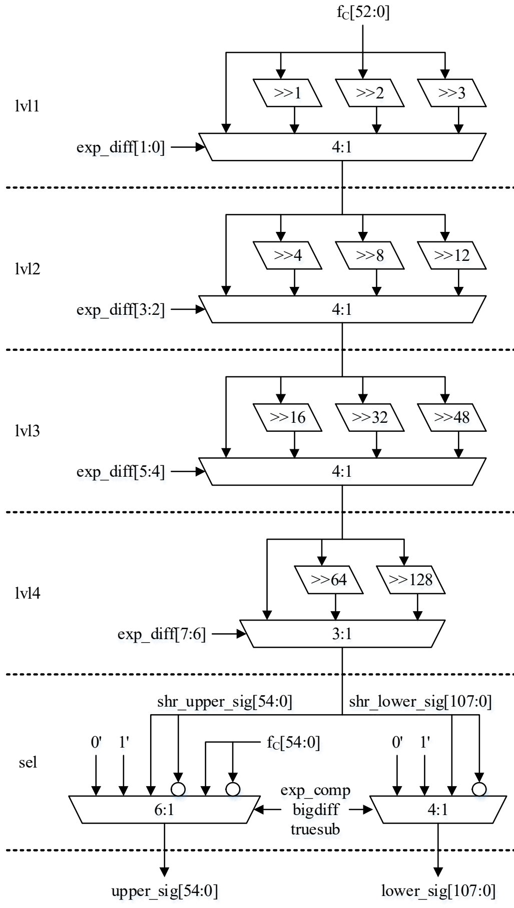

Fig. 3. Alignment Logic

from the exponent of the product e_M. The e_M is computed by adding e_A and e_B, then subtracting the bias, which is 0x3ff for double and 0x7f for single precision, respectively. Since the addend significand is right shifted in one direction, it needs to be placed leftmost of the alignment bit range. To correct the gap between the addend and product significands, the bias is adjusted by subtracting 56 for double and 27 for single precision, respectively. The adjusted bias is subtracted by one if either of the two operands to the multiplier is denormal to handle the 1-bit denormal bias. Likewise, the e_C is adjusted to one if it is denormal. Also, the *exp_comp* is determined by the MSB of the exponent difference, which means the e_M is larger than e_C.

$$\begin{aligned} e_M &= e_A + e_B - adj_bias \\ exp_diff &= e_M - e_C \end{aligned} \tag{2}$$

$$exp_comp = \begin{cases} 1 & if\ e_M > e_C \\ 0 & otherwise \end{cases} \tag{3}$$

The exponent difference is computed in four levels with 2 bits in each level – 1st level [1:0], 2nd level [3:2], 3rd level [5:4], and 4th level [7:6]. Those 2 bits in each level represent the shift amount of the significand alignment. A 2-bit exponent difference for the first level exponent difference is performed separately so that the first level significand alignment starts earlier

TABLE I
UPPER ALIGNED SIGNIFICAND SELECTION

bigdiff	exp_comp	truesub	upper_sig
0	-	0	aligned f_C
0	-	1	aligned & inverted f_C
1	0	0	f_C
1	0	1	inverted f_C
1	1	0	'0
1	1	1	'1

TABLE II
LOWER ALIGNED SIGNIFICAND SELECTION

bigdiff	truesub	lower_sig
0	0	aligned f_C
0	1	aligned & inverted f_C
1	0	'0
1	1	'1

before the entire exponent difference is completed. Also, *bigdiff* is detected, which means the exponent difference is large enough and all the significand bits are shifted out. In this case, all the smaller significand bits are shifted out and the sticky bit is set.

$$bigdiff = \begin{cases} 1 & if\ exp_diff \leq 0\ or\ exp_diff \geq maxdiff \\ 0 & otherwise \end{cases} \tag{4}$$

where *maxdiff* is 192 for double and 128 for single precision, respectively.

The J-bit is an implicit one for the normal numbers. To handle denormal numbers, however, the J-bit needs to be treated as zero. The J-bit is determined by checking if the exponent is nonzero.

$$Jbit = \begin{cases} 1 & if\ exp \neq 0 \\ 0 & otherwise \end{cases} \tag{5}$$

The J-bit of the addend significand is detected in parallel with the first level of the exponent difference so that there is no additional delay to handle denormal numbers. Then, it is right shifted based on the exponent difference. The significand alignment consists of four levels of shifters and the last selection MUX as shown in Fig. 3. In each level, one of three or four shift amounts is selected based on the exponent difference – 1st level [0, 1, 2, or 3], 2nd level [0, 4, 8, or 12], 3rd level [0, 16, 32, or 48], and 4th level [0, 64, or 128]. The aligned significand is split into the upper 55 bits and lower 108 bits. The upper and lower aligned significands are determined based on *bigdiff*, *exp_comp* and *truesub* as shown in TABLE I and TABLE II. The *truesub* is generated by XORing the three signs and the subtraction operation.

$$truesub = \begin{cases} 1 & if\ s_A \oplus s_B \oplus s_C \oplus sub_op \\ 0 & otherwise \end{cases} \tag{6}$$

The selected upper significand is passed to the incrementor, and the lower significand is passed to the multiply array to be

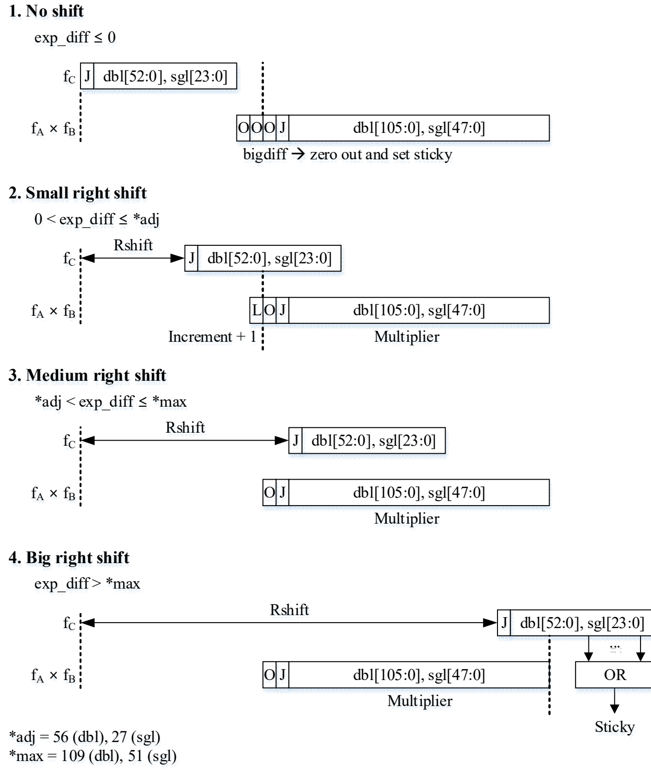

Fig. 4. Four Alignment Cases

merged with the significand product, then passed to the main adder.

There are four cases of alignment as shown in Fig. 4: 1) no shift is needed if the e_C is large enough so that all the product significand bits are shifted out and sticky bit is set, 2) a small right shift is needed if the e_C is smaller than e_M and some of the f_C bits are overlapped with the product bits, and the upper f_C bits are passed to the incrementor and the lower product bits are passed to the main adder, 3) a medium right shift is needed if the e_C is larger than the e_M and the f_C bits are completely overlapped with the product bits, and all those bits are passed to the main adder, and 4) a big right shift is needed if the e_C is smaller than the e_M and some or all the f_C bits are shifted out below the LSB of the product, and those shifted bits are ORed to determine the sticky bit. The sticky logic is performed in parallel with the alignment. The sticky bit is set only in cases 1 and 4 of the alignment cases described above. In case 4, the sticky bit is set if the f_C is right shifted more than the maximum shift range.

B. Multiplier and J-bit Correction

While the addend significand is aligned, in the first cycle, the other two significands are passed to the multiplier. Since the multiplier is on the critical path, the two significands need to be directly passed to the multiplier with no delay of the J-bit detection. To avoid the delay, the proposed FMA assumes both J-bits are ones, then subtracts one J-bit correction line in the multiply array, which requires one more partial product line and a few bits for two's complement. If the operand A is denormal, the f_B is subtracted, and if the operand B is denormal, the f_A is subtracted, which is called a J-bit correction line. The case that both operands are denormal is ignored, since it results in a tiny number with the underflow.

The proposed FMA uses the radix-16 Booth encoding to reduce the area and power. The radix-16 Booth encoding produces about half the partial products compared to the radix-4 Booth encoding (14 vs. 27). The radix-16 Booth encoding, however, requires the pre-computations to obtain 1x, 2x, …, and 8x multiples of the significand f_B, which needs three adders in parallel.

- 1x = f_B
- 2x = f_B << 1
- 3x = 1x + 2x (need an adder)
- 4x = f_B << 2
- 5x = 1x + 4x (need an adder)
- 6x = 3x << 1
- 7x = 8x – 1x (need an adder)
- 8x = f_B << 3

The f_A is encoded to select the multiples based on the radix-16 Booth encoding [10]. The selected multiples form 14 partial products, and they are compressed by 6 levels of 3:2 CSA tree. The J-bit correction line and aligned addend significand are added to the CSA tree with no additional level of CSA. The 16 partial products are grouped by the number of partial products that need the same levels of CSA to reduce the number of CSAs.

- 3 – 4 partial products (2 levels)
- 5 – 6 partial products (3 levels)
- 7 – 9 partial products (4 levels)
- 10 – 13 partial products (5 levels)
- 14 – 16 partial products (6 levels)

Modified 4:2 CSAs with optimal interconnections are used for a faster partial product reduction. While the 4:2 CSA is implemented with back-to-back 3:2 CSAs, it takes 3 XOR delay by connecting the sum of the first 3:2 CSA to the second XOR in the second 3:2 CSA [11]. Sum and carry from the CSA tree are passed to the main adder. For the timing balance, the first four levels of CSAs are performed in the first cycle, and the last two levels of CSAs are performed in the second cycle.

C. Main adder and Incrementor

The significand sum and carry from the multiplier are passed to the main adder. The main adder computes the sum of the two significands for a set of double precision, or two sets of single precision as shown in Fig. 5. Also, the upper significand from the alignment is passed to the incrementor. The incrementor adds one to the upper significand only if the main adder produces carry-out. The result of the main adder and incrementor needs to be two's complemented if it is positive. On the other hand, the result of the main adder and incrementor needs to be inverted if it is negative.

$$\begin{aligned} X - Y &= X + \bar{Y} + 1 \quad (X > Y) \\ Y - X &= \overline{X + \bar{Y}} \qquad (X < Y) \end{aligned} \tag{7}$$

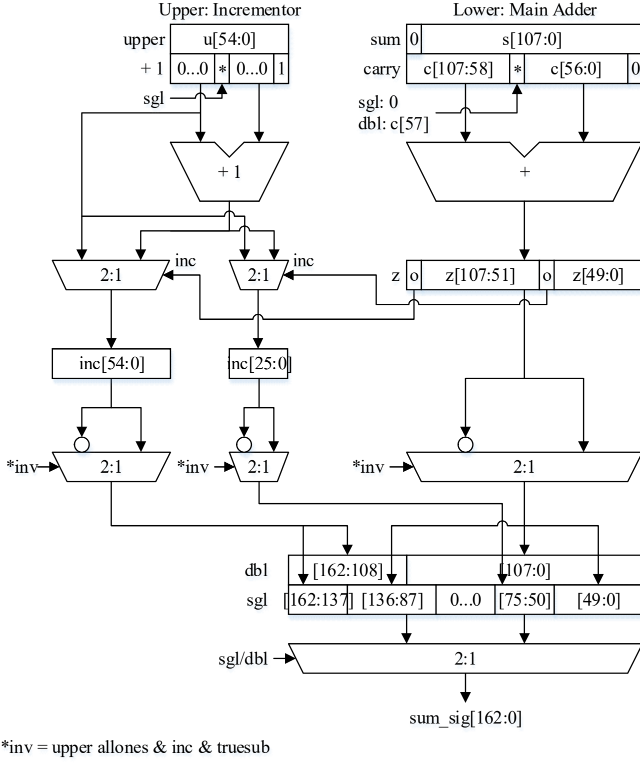

Fig. 5. Main Adder and Incrementor

Adding one for the two's complement is merged with the rounding logic to speed up the critical path – the details will be described in the rounding section. Inversion can be detected by checking the carry-out of the incrementor, which needs a delay of the incrementor to be completed. To avoid the delay, it is detected by checking if the upper significand bits are all ones, incremented, and *truesub*. The inverted result of the main adder and incrementor is re-organized based on the precision, then passed to the third cycle for the normalization and rounding.

D. Modified Leading Zero Anticipation

The result of the main adder needs to be normalized. To speed up the normalization, the LZA is performed in parallel with the main adder. The LZA takes sum and carry from the multiply array and generates *f* vectors for both the cases that the result is positive and negative, then one of them is selected based on the inversion, which is described in the main adder section.

$$\begin{aligned} sum &= \{s_n, s_{n-1}, \ldots, s_0\} \\ carry &= \{c_n, c_{n-1}, \ldots, c_0\} \\ g_i &= s_i \wedge c_i \\ z_i &= \overline{s_i \vee c_i} \\ f^{pos}{}_i &= (g_i \vee z_i) \wedge \overline{z_{i-1}} \\ f^{neg}{}_i &= (g_i \vee z_i) \wedge \overline{g_{i-1}} \end{aligned} \tag{8}$$

The basic equations are same as the traditional LZA [12]. The LZA is modified to handle the underflow that occurs when the exponent becomes negative after the normalization. The modified LZA stops the normalization shift by masking the *f* vectors if the exponent is less than the normalization shift amount. The mask vector is generated in four levels based on the exponent – 1st level [0, 64, or 128], 2nd level [0, 16, 32, or 48], 3rd level [0, 4, 8, or 12], and 4th level [0, 1, 2, or 3].

$$\begin{aligned} m_{lvl1} &= {m_0}^{64} {m_{64}}^{64} {m_{128}}^{64} \\ m_{lvl2} &= {m_0}^{16} {m_{16}}^{16} {m_{32}}^{16} {m_{48}}^{16} \\ m_{lvl3} &= {m_0}^4 {m_4}^4 {m_8}^4 {m_{12}}^4 {m_0}^4 {m_4}^4 \ldots {m_8}^4 {m_{12}}^4 \\ m_{lvl4} &= m_0 m_1 m_2 m_3 m_0 m_1 m_2 m_3 \ldots\ m_2 m_3 \\ m &= m_{lvl1} \wedge m_{lvl2} \wedge m_{lvl3} \wedge m_{lvl4} \wedge (exp < 128) \end{aligned} \tag{9}$$

where $m_k{}^n$ is set if the exponent is less than or equal to k and repeated n times. In each level, two bits of the exponent are used to generate the mask bits – 1st level [7:6], 2nd level [5:4], 3rd level [3:2], and 4th level [1:0]. The *f* vectors are ORed with the mask vector m and it is used to count the leading zeros. The LZA consists of four levels, which is same as the normalization. In each level, the LZA vector is split into four chunks and the bits in each chunk are ORed to search the ones. Then, one of the four chunks with the first one from the MSB is selected to determine the shift amount. The levels of the LZA are organized from coarse to fine – 1st level 64 bits, 2nd level 16 bits, 3rd level 4 bits, and 4th level 1 bit per chunk.

E. Normalization, Sticky, and All-ones Detection

The result from the main adder and incrementor is passed to the normalization logic in the third cycle. The normalization logic consists of four levels of shifters as shown in Fig. 6. In each level, one of three or four shift amount is selected based on the LZA result – 1st level [0, 64 or 128], 2nd level [0, 16, 32 or 48], 3rd level [0, 4, 8 or 12], and 4th level [0, 1, 2 or 3]. Since the LZA may have a 1-bit error, a 1-bit right shift is needed, which is called post-normalization. It is detected in parallel with the last level of the LZA so that there is no additional delay.

$$post_norm = \begin{cases} norm_sig_{lvl3}[MSB] & if\ lza_{lvl4} = 0 \\ norm_sig_{lvl3}[MSB-1] & if\ lza_{lvl4} = 1 \\ norm_sig_{lvl3}[MSB-2] & if\ lza_{lvl4} = 2 \\ norm_sig_{lvl3}[MSB-3] & if\ lza_{lvl4} = 3 \end{cases} \tag{10}$$

The normalization requires an exponent adjustment by subtracting the shift amount, and it causes underflow if the exponent becomes less than zero after the adjustment. In this case, the denormalization shifter is needed to recover the negative exponent to zero, which requires additional delay. To avoid the extra process, the modified LZA stops the normalization if the exponent is less than the shift amount so that the denormalization is unnecessary. Also, underflow is detected if the J-bit after normalization is zero, which means a denormal significand result, and the exponent is set to zero.

Sticky and all-ones detection is performed in parallel with the normalization to speed up the rounding logic. The sticky bit in each level of the normalization is set by ORing the bits under the guard bit. The sticky bits from the four levels of the normal-

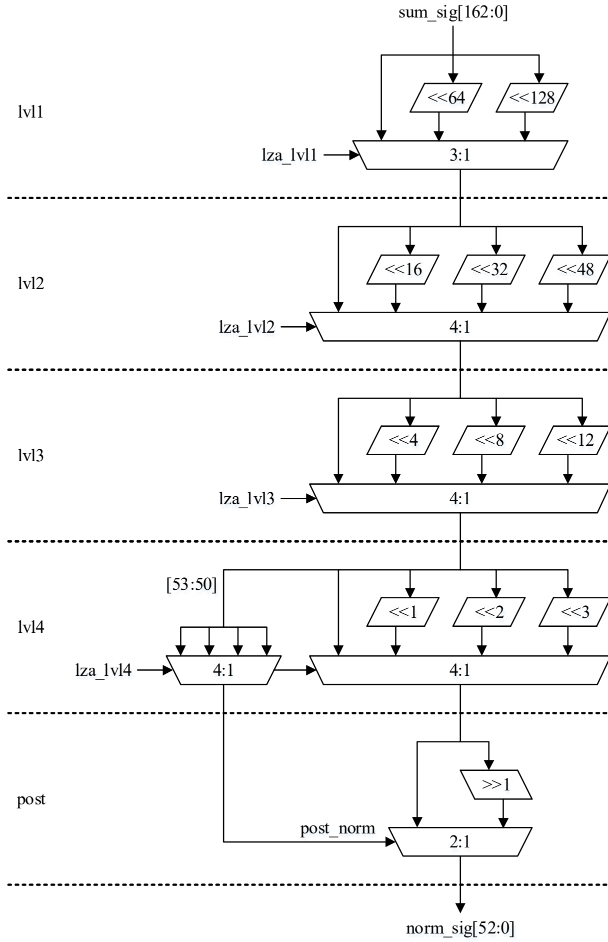

Fig. 6. Normalization Logic

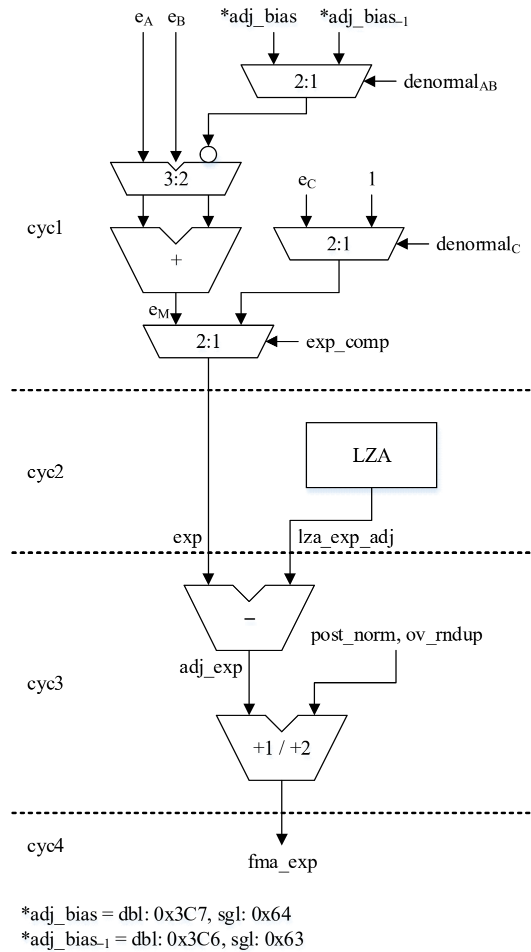

Fig. 7. Exponent Logic

ization and the sticky bit from the alignment are ORed to generate the final sticky bit. Likewise, all-ones in each level is set by ANDing all the bits under the LSB. The final all-ones is generated by ANDing the all-ones from the four levels of normalization. The sticky and all-ones are used in the rounding logic.

F. Rounding

The normalized significand is passed to the rounding logic. The regular rounding is determined by the rounding mode, LSB, guard, sticky and sign bit as described in [13]. The rounding modes are simplified by merging round to +infinity and round to –infinity. Also, round to zero can be omitted by using AOI MUX.

$$round\ to\ infinity = \begin{cases} round\ to + infinity & if\ sign = 0 \\ round\ to - infinity & if\ sign = 1 \end{cases} \quad (11)$$

$$round_up = \begin{cases} G \wedge (L \vee S) & if\ round\ to\ nearest\ even \\ G \vee S & if\ round\ to\ infinity \end{cases} \quad (12)$$

where L is the LSB, G is guard bit, and S is sticky bit. As mentioned in the main adder section, the two's complement is merged with the rounding logic. The two's complement is propagated only if all the bits under the LSB are ones, which is detected in parallel with the normalization. The propagated two's complement forces the roundup. Thus, the normalized significand is rounded by either the regular roundup or the forced roundup by two's complement.

The rounded significand needs to be shifted right by one bit if the significand overflow occurs after the rounding. Such a case occurs only if the significand bits are all ones and it is rounded up, which is detected in parallel with the normalization.

$$ov_rndup = allones \wedge round_up \quad (13)$$

If *ov_rndup* is detected, significand becomes zero and the exponent is adjusted accordingly, which eliminates the re-normalization after the rounding. The rounded significand is passed to the last MUX in the fourth cycle to determine precision and special cases, then passed to the bypass and writeback.

G. Exponent and Sign Logic

The exponent logic computes the exponent of the FMA as shown in Fig. 7. It computes e_M and e_C, as described in the exponent difference section, and selects one of them based on the

exp_comp. The selected exponent is adjusted by subtracting the normalization shift amount from LZA. Then, it is adjusted again by adding one or two based on the *post_norm* and *ov_rndup*, which is described in the normalization section and rounding section, respectively.

$$fma_exp = \begin{cases} adj_exp + 2 & if\ post_norm \wedge ov_rndup \\ adj_exp + 1 & if\ post_norm \oplus ov_rndup \\ adj_exp & otherwise \end{cases} \quad (14)$$

The sign logic is performed in the first and third cycles based on the three signs and comparison result. The sign of the product s_M and effective sign of C s_{Ceff} is determined in the first cycle.

$$\begin{aligned} s_M &= s_A \oplus s_B \\ s_{Ceff} &= s_C \oplus sub_op \end{aligned} \quad (15)$$

The sign of the FMA is determined in the third cycle, since it requires to check if the result is inverted, which is described in main adder section. The sign result is set to negative if one of the following four cases, and set to positive, otherwise.

- $s_M = 1$ and $s_{Ceff} = 1$
- $s_M = 1$ and not inverted
- $s_{Ceff} = 1$ and inverted
- $s_M \oplus s_{Ceff} = 1$ and round to –infinity

IV. RESULTS

In this section, the latency and area comparison between the traditional and proposed FMA designs are described. Since the latency and area vary depending on the process technology, synthesis, placement, and routing, they are analyzed based on the logic gate comparisons. The latency is estimated by the gate levels, where the unit gate delays 3 – 4 FO4 inverters depending on the complexities. The area is estimated by the gate count assuming a full adder is composed of 6 gates. TABLE III and TABLE IV show the latency and area comparisons for double precision. The proposed design is compared with two traditional designs [7], [8], that are the most well-known FMA designs. As described in the previous sections, the proposed FMA applies the radix-16 Booth encoding, which requires pre-computations, but produces a lot fewer partial products. Considering the placement and routing benefits, the radix-16 Booth encoding results in about the same latency with much less area compared to the radix-4 Booth encoding. The aligned significand is inserted into the CSA tree, which eliminates the CSAs at the end of the multiply array. Also, the incrementor for the two's complement after the main adder is merged with the rounding logic. Additionally, the sticky and all-ones detection is performed in parallel with the normalization so that the incrementor for the rounding is performed right after the normalization. All the extra logic gates for denormal support are completely merged or performed in parallel with the other logic to avoid the additional delay. The J-bit detection for the addend is detected in parallel with the first level of the exponent difference. Since the other two operands need to be directly passed to the multiplier, J-bits are not detected. Instead, a J-bit correction line is added into the CSA tree, which is merged with the existing CSAs with no additional delay. Also, the 1-bit denormal bias for the denormal inputs are handled in the exponent difference logic by adjusting the bias. Although the modified LZA needs additional OR gates for masking, they are not timing critical. As a result, the proposed FMA reduces about 10 – 20% of the latency and about 10 – 30% of the area over the traditional FMAs [7], [8]. Moreover, the proposed FMA fully supports denormal numbers with no additional delay, which eliminates the additional cycle delays of the microcode assist handler for denormal numbers.

V. CONCLUSION

An enhanced FMA design for the Intel E-Core processor [2] is presented. FMA is one of the most frequently used operations, so improving FMA design will contribute to the next generation floating-point unit development. The proposed FMA is fully

TABLE III
LATENCY COMPARISON (DOUBLE PRECISION)

Logic	PowerPC FMA [7]	Lang's FMA [8]	Proposed FMA	Comment
Multiplier (gate levels)	Radix-4 Booth (6)	Radix-4 Booth (6)	Radix-16 Booth & 53-bit Adder (10)	Radix-16 Booth and pre-computations are in parallel.
	8 Levels of 3:2 CSAs (12)	8 Levels of 3:2 CSAs (12)	6 Levels of 3:2 CSAs (9)	
Main Adder (gate levels)	3:2 CSA (2)	3:2 CSA (2)	No 3:2 CSA (0)	Aligned addend is merged with the multiply array. Two's complement is merged with the rounding.
	106-bit Adder (12)	Part of 162-bit Adder (10)	106-bit Adder (12)	
	106-bit Incrementor (10)	No Complement (0)	No Complement (0)	
Normalization (gate levels)	162-bit Shifter (8)	162-bit Shifter (8)	162-bit Shifter (8)	Sticky and all-ones detection are in parallel with the normalization.
Rounding (gate levels)	53-bit Incrementor (8)	Rest of 162-bit Adder & Rounding (13)	53-bit Incrementor (8)	Post-normalization after the rounding is detected in parallel with the rounding.
	1-bit Shifter (2)	1-bit Shifter (2)	No Shifter (0)	
Total Gate Levels	60	53	47	

TABLE IV
AREA COMPARISON (DOUBLE PRECISION)

Logic	PowerPC FMA [7]	Lang's FMA [8]	Proposed FMA	Comment
Exponent Difference & Alignment (gate count)	11-bit Adder × 2 (200)	11-bit Adder × 2 (200)	11-bit Adder × 2 (200)	
	162-bit Shifter (2,400)	162-bit Shifter (2,400)	162-bit Shifter (2,400)	
	Sticky (100)	Sticky (100)	Sticky (100)	
Multiplier (gate count)	Radix-4 Booth (200)	Radix-4 Booth (200)	Radix-16 Booth (400) 53-bit Adder × 3 (1,500)	Radix-16 Booth requires the pre-computations. Radix-16 needs 3 level CSAs with 14 partial products, while radix-4 needs 4 level CSAs with 27 partial products.
	1,500-bit 3:2 CSAs (9,000)	1,500-bit 3:2 CSAs (9,000)	900-bit 3:2 CSAs (5,400)	
Main Adder (gate count)	106-bit 3:2 CSAs (600)	106-bit 3:2 CSAs (600)	No 3:2 CSA (0)	Aligned addend is merged with the multiply array. Two's complement is merged with the rounding.
	106-bit Adder (1,200)	Part of 162-bit Adder × 2 (1,200)	106-bit Adder (1,200)	
	56-bit Incrementor (200)	No Upper Incrementor (0)	56-bit Incrementor (200)	
	106-bit Incrementor (400)	No Complement (0)	No Complement (0)	
LZA & Normalization (gate count)	LZA (2,000)	LZA (2,000)	Modified LZA (2,400)	Modified LZA has masking logic to stop the normalization.
	162-bit Shifter (2,400)	162-bit Shifter × 2 (4,800)	162-bit Shifter (2,400) Sticky & All-ones Detection (200)	
Rounding (gate count)	56-bit Incrementor (200)	Rest of 162-bit Adder × 2 (1,200) Rounding (1,400)	56-bit Incrementor (200)	Post-normalization after the rounding is detected in parallel with the rounding.
	1-bit Shifter (100)	1-bit Shifter (100)	No Shifter (0)	
Total Gate Count	19,000	23,200	16,600	

pipelined, executes SSE/AVX operations for scalar/packed IEEE single and double precisions, and supports all four rounding modes. Also, the proposed FMA fully supports denormal numbers with no additional delay so that the microcode assistance is unnecessary. Several optimization techniques are applied to achieve the proposed FMA with full denormal support – one-way alignment, radix-16 Booth encoding for the multiplier, merged J-bit correction and aligned significand with the multiply array, modified LZA for masking the underflow, parallel sticky and all-ones detection with the normalization, and merged two's complement with the rounding logic. As a result, the proposed FMA achieved not only full denormal support but also about 10 – 20% reduced latency and about 10 – 30% reduced area over the traditional FMAs.

ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, IEEE, 2019.
[2] *How 13th Gen Intel® Core™ Processors Work*, https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html, Intel Corp., 2022.
[3] E. M. Schwarz, M. Schmookler, S. Dao Trong, "FPU Implementations with Denormalized Numbers," *IEEE Trans. on Computers*, vol.54, no. 7, pp. 825-836, July 2005.
[4] D. R. Lutz, "Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines," *Proc. 20th IEEE Symp. Computer Arithmetic*, pp. 123-128, July 2011.
[5] J. Sohn, D. K. Dean, E. Quintana and W. S. Wong, "Enhanced Floating-Point Adder with Full Denormal Support," *Proc. 29th IEEE Symp. Computer Arithmetic*, pp. 35-42, September 2022.
[6] E. Hokenek, R. Montoye, and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1207-1213, 1990.
[7] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener, "P6 Binary Floating-Point Unit", *Proc. 18th IEEE Symp. Computer Arithmetic*, pp. 77-86, 2007.
[8] T. Lang, J.D. Bruguera, "Floating-Point Multiply-Add-Fused with Reduced Latency", *IEEE Trans. on Computers*, Vol. 53, No. 8, pp. 988-1003, August 2004.
[9] M. Schmookler and K. Nowka, "Leading Zero Anticipation and Detection – A Comparison of Methods", *Proc. 15th IEEE Symp. Computer Arithmetic*, pp. 7-12, 2001.
[10] G. W. Bewick, *Fast Multiplication: Algorithms and Implementation*, PhD dissertation, Stanford University, 1994.
[11] V.G. Oklobdzija, D. Villeger, and S.S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Partial Multipliers Using an Algorithmic Approach," *IEEE Trans. on Computers*, vol. 45, no. 3, pp. 294–306, Mar. 1996.
[12] R. M. Jessani and M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units," *IEEE Trans. on Computers*, vol. 47, no. 9, pp. 927-937, 1998.
[13] G. Even and P.M. Seidel, "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication," *IEEE. Trans. on Computers*, vol. 49, no. 7, pp. 638-650, July 2000.