Implementation of Low Power and Area Efficient Floating-Point Fused Multiply-Add Unit

R. Dhanabal, Sarat Kumar Sahoo and V. Bharathi

Abstract In this paper, a modified architecture for Floating-Point Fused Multiply-Add (FMA) unit for low power and reduced area applications is presented. FMA unit is the one which computes a floating-point $(A \times B) + C$ operation as a single instruction. In this paper a bridge unit has been used, which connects the existing floating-point multiplier (FMUL) and the FMUL's add-round unit in the co-processor to perform FMA operation. The main objective of this modified FMA unit is to reuse as many components as possible to allow parallel floating-point addition and floating-point multiplication or floating-point fused multiply-add functionality by addition of little hardware into the FMUL's add-round unit. In this paper each unit is designed using Verilog HDL. The design is simulated using Altera ModelSim and is synthesized using Cadence RTL compiler in 45 nm. All the floating-point arithmetics are implemented in IEEE-754 double precision format. It is found that the proposed FMA architecture achieved 17 % improvement in power and 6 % improvement in area when compared to the existing Bridge FMA unit.

Keywords Fused multiply-add · Floating-point arithmetics · IEEE-754 double precision standard

R. Dhanabal (✉)
VLSI Division, SENSE, VIT University, Vellore, India
e-mail: rdhanabal@vit.ac.in

S.K. Sahoo
SELECT, VIT University, Vellore 632014, India
e-mail: sksahoo@vit.ac.in

V. Bharathi
VLSI Division, CSE, GGR College of Engineering,
Anna University, Vellore, India
e-mail: bharathiveerappan@yahoo.co.in

© Springer India 2016

L.P. Suresh and B.K. Panigrahi (eds.), *Proceedings of the International Conference on Soft Computing Systems*, Advances in Intelligent Systems and Computing 397, DOI 10.1007/978-81-322-2671-0_31

1 Introduction

In digital signal processing applications the floating-point fused multiply-add (FMA) operation has become one of the fundamental operations. Many of the commercial processors like IBM PowerPC, Intel Itanium have included the FMA unit in its floating-point units to execute double precision fused multiply-add operation [1]. FMA unit improves the accuracy of the floating-point $(A \times B) + C$ operation as it performs single rounding instead of two. FMA operation is very useful when a floating-point multiplication is followed by a floating-point addition.

Floating-point fused multiply-add implementation has two advantages over implementation of floating-point addition (FADD) and floating-point multiplication (FMUL) separately: (1) The FMA operation is performed with only one rounding instead of two (one for floating-point adder and other for floating-point multiplier) reducing overall error due to rounding. (2) There will be a reduction in delay and hardware required by sharing components [2, 3].

In some designs the existing FMUL unit and FADD unit is entirely replaced with a FMA unit. It performs single FMUL operation by making $C = 0$ and single FADD operation by making $A = 1$ (or $B = 1$) in $(A \times B) + C$, e.g., $(A \times B) + 0.0$ for single multiplier and $(A \times 1.0) + C$ for single adds. But due to the insertion of constants, the latencies of stand-alone FMUL, and FADD operations increase due to the complexity of FMA unit. In such designs there will not be any possibility to perform parallel FMUL and FADD instructions [3].

The first floating-point FMA unit was introduced on IBM RISC System/6000 in 1990 for single instruction execution of $(A \times B) + C$ operation as an indivisible operation [2, 4]. Executing parallel FMUL and FADD operations is not possible in basic FMA unit. In [5] the Concordia FMA architecture is designed, which uses alignment blocks before the multiplier array. So multiplier tree input range widens. Due to this larger variable multiplier tree is required. A few possible solutions have been identified in the Lang/Bruguera fused multiply-add architecture, which is designed for reduced latencies [3, 6]. But it did not reach the latency of a common FADD/FMUL instruction. A bridge FMA design is introduced in [7] to avoid the stand-alone FMUL and FADD latencies due to the insertion of constants by adding extra blocks between existing FMUL and FADD components in the processor. But the cost added to this architecture is increase in area and power consumed when compared to the basic FMA architecture.

The main objective of this work is to design a low power, area efficient FMA unit which performs FMA operation or parallel FMUL and FADD operations based on requirement. In this paper a modified add-round block is designed, which supports add-round for FMUL as well as FMA. Common add-round unit for both FMA and FMUL instructions is used to save chip area.

All the floating-point arithmetic operations here are done using IEEE-754 double precision format. The standard IEEE-754 double precision format [8] consists of 64 bits which are divided into three sections as shown in Fig. 1. To represent any

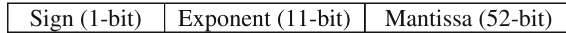


Fig. 1 IEEE-754 double precision format [8]

floating-point number, all the three sections have to be combined. The double precision floating-point number is calculated as shown in Eq. (1).

$$A = (-1)^{\text{sign}_A} \times 1 \cdot \text{fraction}_A \times 2^{\text{exp}_A - \text{bias}} \quad (1)$$

2 Architecture of Proposed FMA Unit

Block diagram for proposed floating-point fused multiply-add unit is shown in Fig. 2. The FMA unit starts with the common multiplier and adder units which can perform single stand-alone operations. The main components in this design are:

1. Floating-Point Multiplier
2. Floating-Point Adder
3. Bridge Unit
4. Add-Round unit for FMA/FMUL
5. Add-Round Unit for FADD

Our FMA unit performs parallel floating-point addition and multiplication or floating-point fused multiply-add operations based on the requirement. Suppose when a FMA operation is to be performed, this bridge architecture is connected between the existing FMUL and FMUL's add-round unit. When FMA operation is

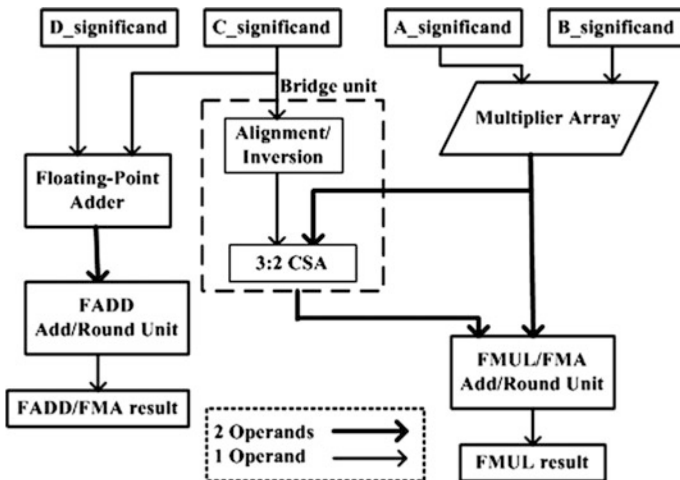


Fig. 2 Block diagram for proposed FMA unit

not needed stand-alone FMUL and FADD operation can be performed without using the intermediate bridge unit.

2.1 Multiplier

Efficient double precision floating-point multiplier using radix-4 modified booth algorithm (MBE) and Dadda algorithm has been implemented. This hybrid multiplier is designed by using the advantages in both the multiplier algorithms. MBE has the advantage of reducing partial products to be added. Dadda scheme has the advantage of adding the partial products in a faster manner [9, 10]. Our main objective is to combine these two schemes to make the multiplier design power efficient and area efficient. Finally obtained two rows (sums and carries) are added using an efficient parallel prefix adder [11].

MBE generates at most $\left\lfloor \frac{N}{2} \right\rfloor + 1$ partial products, where N is the number of bits. Radix-4 recoding is done with the digit set {−2, −1, 0, 1, 2} is shown in Table 1. Each three consecutive bits of the multiplier B represents the input to the booth recoding block. This block selects the right operation on multiplicand A which can be shift or invert ($-2B$) or invert ($-B$) or zero or no operation (B) or shift ($2B$). Figure 3 shows the generation of one partial product using MBE.

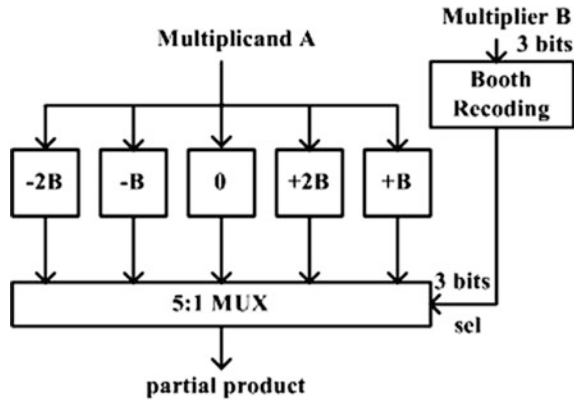
In Dadda scheme, the reduction of obtained partial products is done in stages using half adders and full adders. The reduction in size of each stage is obtained by working back from the final stage. Each preceding stage height must be not greater than $\lfloor 3 \cdot \text{successorheight}/2 \rfloor$ [10].

For a double precision floating-point multiplication two 53-bit (1 hidden bit + mantissa 52 bits) numbers are to be multiplied. If normal method is used for generation of partial products 53 partial products will be obtained. But by using MBE the partial products can be reduced to 27. Each partial product can be obtained using block shown in Fig. 3. These 27 rows of partial products are reduced to 2 rows in 7 reduction stages, where 19, 13, 9, 6, 4, 3, 2 is height of each stage as we go down in the Dadda reduction scheme. The dot diagram for 10 bit by 10 bit

Table 1 Radix-4 modified booth's recoding (for A × B)

Bits of multiplier B			Encoding operation on multiplicand A
C_{i+1}	C_i	C_{i-1}	
0	0	0	0
0	0	1	$+B$
0	1	0	$+B$
0	1	1	$+2B$
1	0	0	$-2B$
1	0	1	$-B$
1	1	0	$-B$
1	1	1	0

Fig. 3 One partial product generator using MBE



booth encoding with Dadda reduction is shown in Fig. 4. The same method in Fig. 4 is extended to 53 bit by 53 bit.

The final sums and carries are added using parallel prefix adders as it offer a highly efficient solution to the binary addition and suitable for VLSI implementations [11, 12]. Among the parallel prefix adders, Kogge–Stone architecture is the widely used and the popular one. Kogge–Stone adder is considered as the fastest

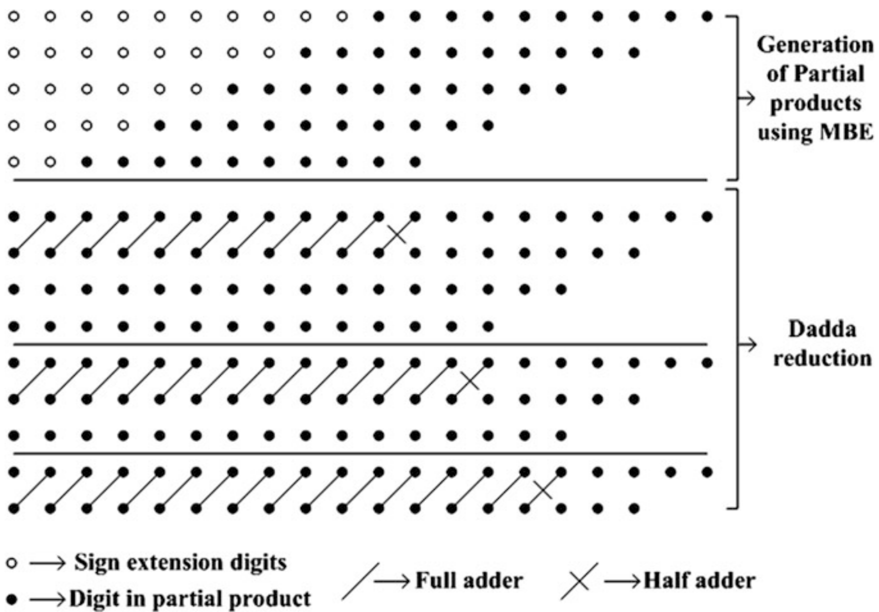


Fig. 4 10 bit by 10 bit booth encoding with Dadda reduction

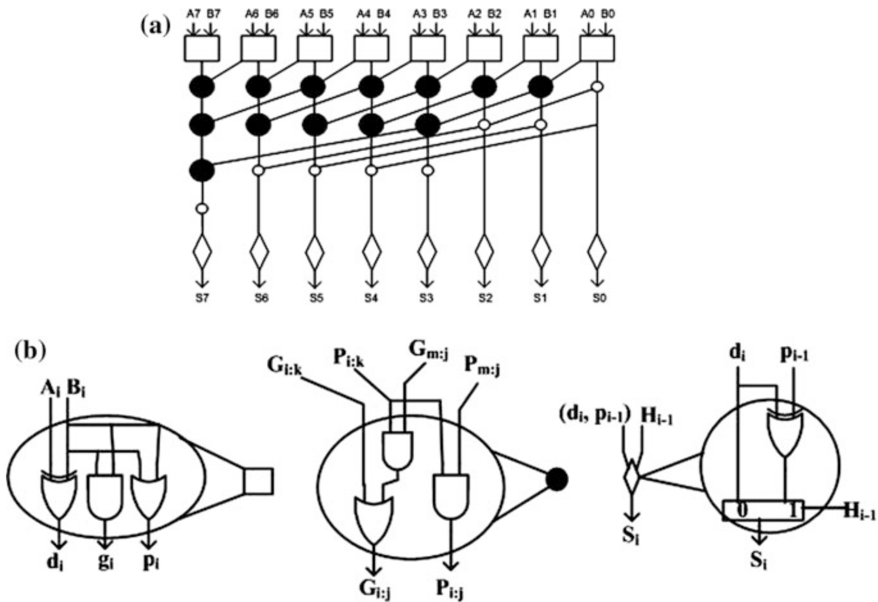


Fig. 5 **a** 8-bit Kogge–Stone adder. **b** Logic implementation of each block of Kogge–Stone adder [11]

adder design possible [11]. Architecture for 8-bit Kogge–Stone adder is as shown in Fig. 5. In this design to add the final sums and carries a 109-bit Kogge–Stone adder is used.

2.2 *Floating-Point Adder*

The modified FMA architecture uses the Farmwald's dual-path floating-point adder design [7]. FADD design is shown in Fig. 6. It uses two paths close path and far path to handle different data cases. Far path is used for significand addition and subtraction, when exponent difference is more than 1. Close path is used for significand addition and subtraction, when the exponents are equal or differ by $\pm$ 1. In far path both the significands are passed through swap multiplexers. When the larger significand is detected it is passed through far_op_greater and the smaller significand is aligned till the exponents match. Smaller significand is passed through far_op_smaller. In close path the two significands are pre-shifted by 1. The original significands and the pre-shifted significands are given to swap multiplexers and based on the exponent difference the significands are swapped. Meanwhile when the exponents are equal the comparator compares the two significands. The greater significand in close path is passed through close_op_greater and the smaller significand is passed through close_op_smaller.

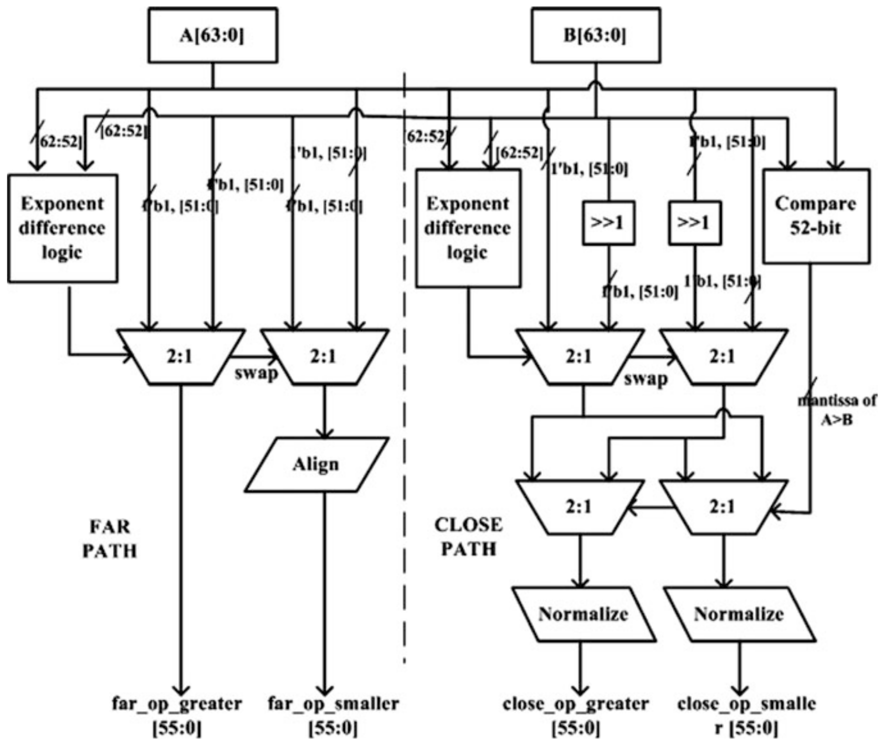


Fig. 6 Floating-point adder unit [7]

2.3 Bridge Unit

The bridge unit is as shown in Fig. 7. This bridge unit is capable of carrying data from multiplier array to FMUL's add-round unit to perform FMA operation $((A \times B) + C))$ efficiently. Inputs to this unit are mantissa of the operand C and the carry save format product of $A \times B$ from multiplier array. The operand C is aligned based on the exponent difference between exponent of C and exponent of the product. After alignment, the select line 'sub' decides whether to perform inversion or not. This inversion provides effective 2's compliment for effective subtraction. If sub = 1, it performs inversion on the aligned data else the aligned data is buffered.

Bridge unit adds the product (i.e., mul_sum and mul_carry) $A \times B$ along with a part ([108:0]) of pre-aligned 161-bit addend (operand C) using 3:2 CSA as shown in Fig. 7. The remaining 52 ([161:109]) bits of the 161 added is given to the incrementer in FMA/FMUL's add-round unit. The 109-bit sum and carry obtained from 3:2 CSA is given to multiplexer stage in FMA/FMUL's add-round unit.

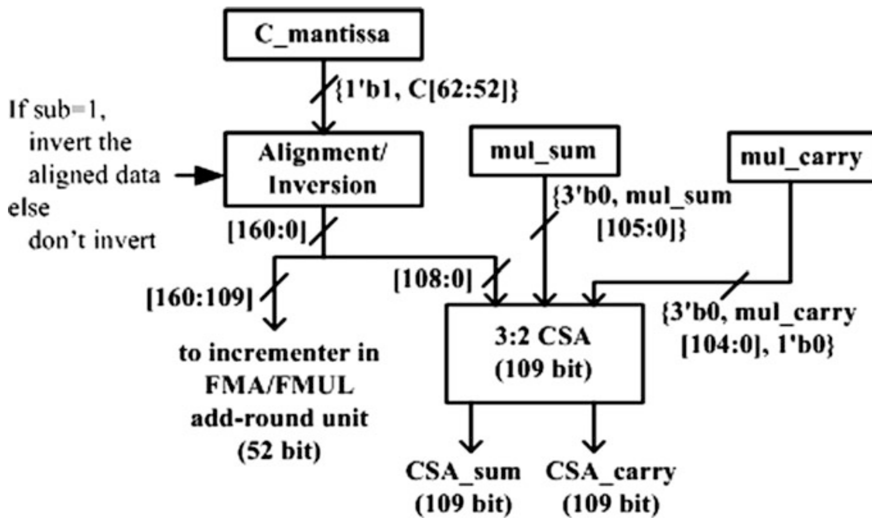


Fig. 7 The bridge unit

Consider 'D' as the exponent difference between exponent of C and exponent of product $A \times B$, its value is $D = \exp(C) - (\exp(A) + \exp(B))$, where exp($A$), exp ($B$), and exp ($C$) are the exponents of operands A, B, and C, respectively.

When $D \geq 0$ (i.e.,$\exp(C) > (\exp(A) + \exp(B))$), the normal aligner will shift exponent of $A \times B$ right by D bits or shift 'C' left by D bits until the exp ($A \times B$) = exp (C). When $D \geq 56$, the sum and carry are placed right of LSB of operand 'C'

When $D < 0$, the operand 'C' will be shifted right by D bits until exp ($A \times B$) = exp (C). For the right shift greater than 105 (i.e., $D < -105$), the operand C is placed to the right of the LSB of the sum and carry (product).

To avoid bidirectional shifter, the alignment is totally implemented as right shift by placing operand 'C' left to that of sum and carry and by placing two extra bits (guard bit and round bit) between the two. Combining both the cases the shift amount will be in the range of 161-bit right shifter. Figure 8 shows how to align operand 'C' in different cases in detail.

- In case of $D \geq 0$, the shift amount is shift amount = max $\{0, 56 - D\}$
- In case of $D < 0$, the shift amount is shift amount = min $\{161, 56 - D\}$

2.4 FMA/FMUL Add-Round Unit

FMA/FMUL add-round unit is shown in Fig. 9. This same add-round unit is used for both FMA and FMUL operation. When a stand-alone FMUL is required it acts

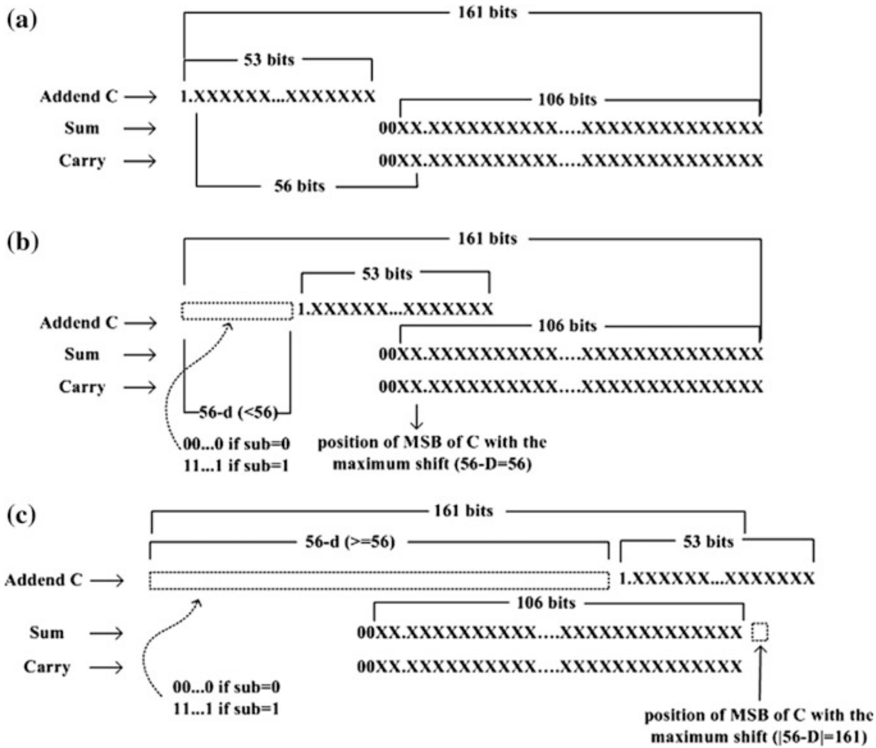


Fig. 8 Alignment of operand C. **a** Before alignment. **b** Alignment with $D \geq 0$. **c** Alignment with $D < 0$

as FMUL add-round unit and when FMA is required it acts as FMA add-round unit. Multiplexer stage is used to select FMA or FMUL. 109 bit Kogge–Stone adder is used to add the data from the mux stages. In parallel to this part of aligner output (52 MSB's) from the bridge unit is given to incrementer. Based on the carry from 109-bit adder the 2:1 mux will select the aligner output or the incrementer output. Compliment the output if necessary. After normalizing the data is sent to perform rounding.

Basically three bits after the LSB decides the rounding. The three bits next to LSB are guard bit (g), round bit (r), and sticky bit (s), respectively. Sticky bit is the logical OR of all bits beyond the guard bit. In the Fig. 9, $R[2{:}0]$ represents $\{g, r, s\}$, respectively. Round-up method which is in [13] is used for rounding purpose, result and result + 1 need to be generated for rounding up. By using the rounding table given in [13] the result is rounded. Finally mantissa of the FMA/FMUL output will be obtained. In parallel to this the exponent is to be adjusted accordingly whenever the normalization or shifting is done.

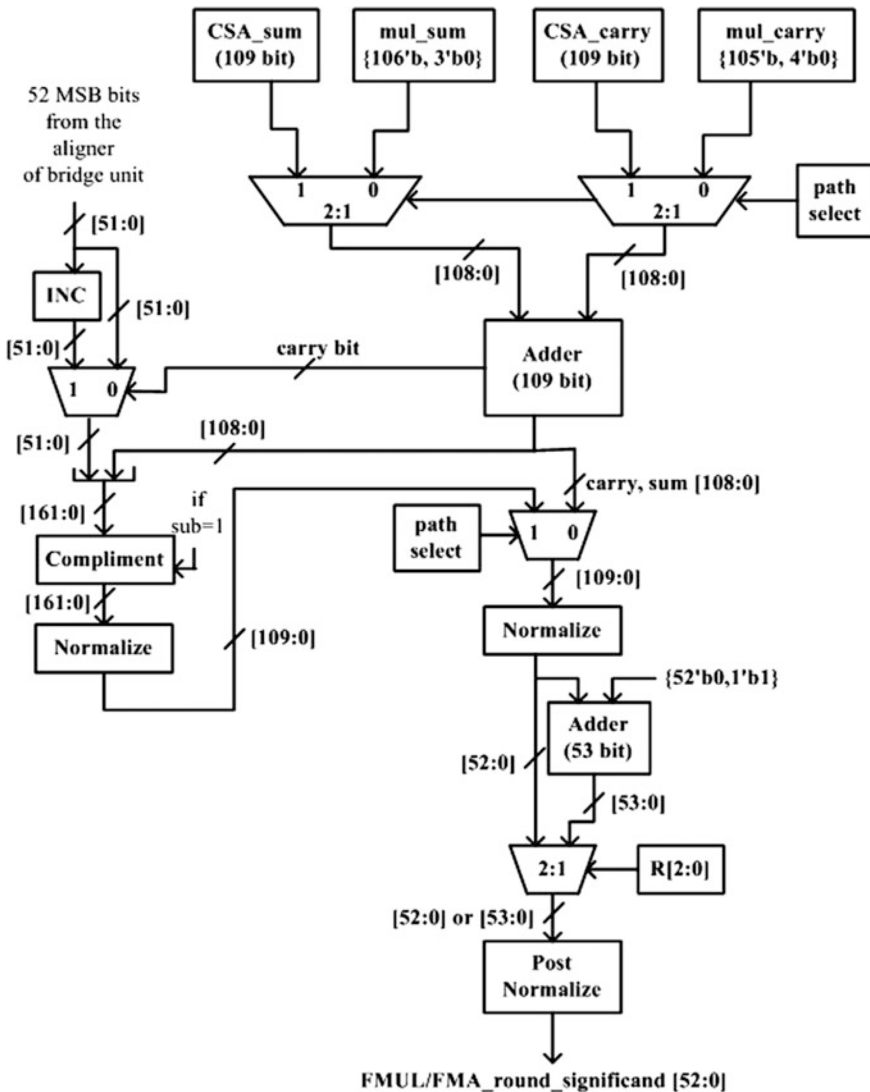


Fig. 9 FMA/FMUL add-round unit

2.5 FADD Add-Round Unit

The add-round unit which is shown in Fig. 10 is exclusively used for FADD operation. The far path and close path operands from floating-point adder are given to FADD add-round unit. The two selected inputs are passed through 56 bit Kogge–Stone adder and the 56-bit 3:2 CSA. In order to perform round-up we are taking third input of the CSA as {55′b0, 1′b1}. The rounding is done in the same way as

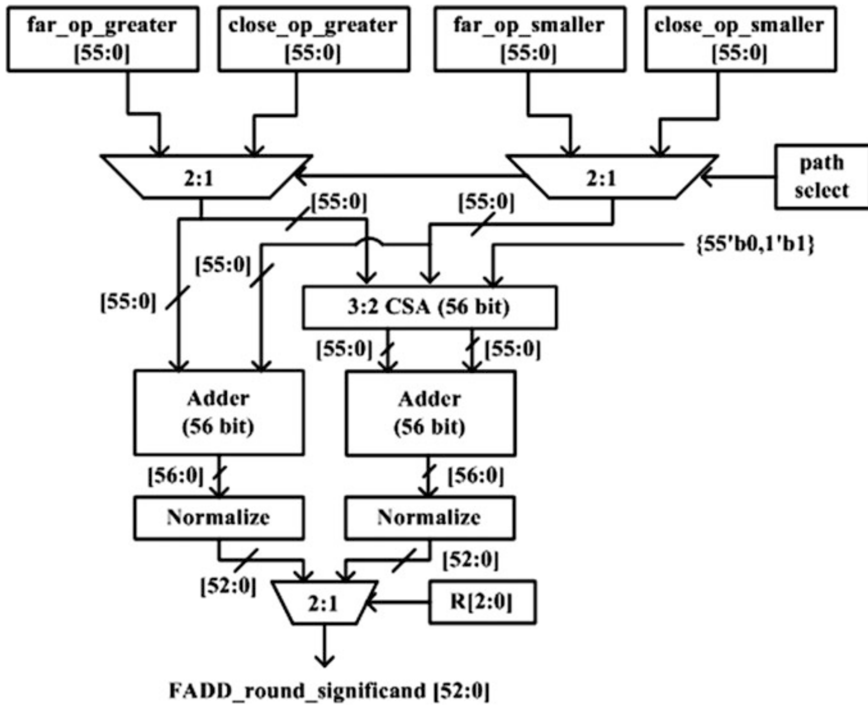


Fig. 10 FADD add-round unit

the multiplier. Then sum and carry from CSA is added with one more 56-bit Kogge–Stone adder. Finally mantissa of the adder output will be obtained. In parallel to this the exponent is to be adjusted accordingly whenever the normalization or shifting is done.

3 Results and Discussion

Simulation results for floating-point multiply-add operation is shown in Fig. 11. Simulation result for parallel floating-point addition and multiplication operation is shown in Fig. 12. Synthesis report for proposed FMA design and FMA design in [7] using Cadence RTL Compiler in 45 nm technology is given in Table 2.

From Table 2 we found that delay and power consumed for stand-alone FADD and FMA operation decreased. The comparison charts for delay, power, and area is shown in Figs. 13, 14 and 15 respectively.

The proposed FMA unit has achieved 7 and 18 % improvement in delay for FADD and FMA instructions respectively, 19 and 17 % improvement in power consumption for FADD and FMA instructions, respectively, and 6 % improvement

Fig. 11 Simulation result for floating-point multiply-add operation

Fig. 12 Simulation result for parallel floating-point addition and multiplication operation

Table 2 Delay, area, power report in 45 nm technology

FMA	Delay (ps)	Area (μm^2)	Power (μW)
Bridge FMA [7]	FADD: 7498.90	34271.51	FADD: 522.22
	FMUL: 6998.90		FMUL: 3510.21
	FMA: 9156.10		FMA: 4582.01
Proposed FMA design	FADD: 6995.20	32542.21	FADD: 422.40
	FMUL: 7499.00		FMUL: 3564.82
	FMA: 7499.00		FMA: 3823.64

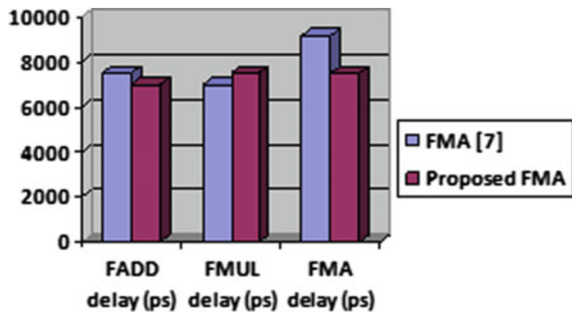


Fig. 13 Comparison chart for delay of proposed FMA with FMA in [7]

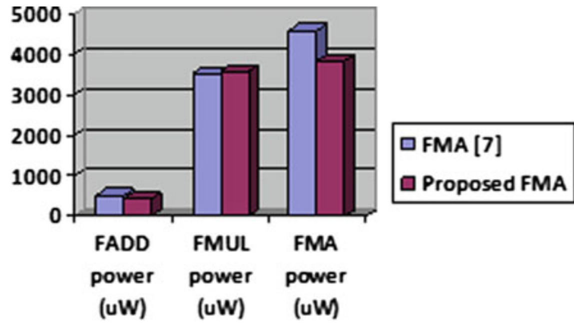


Fig. 14 Comparison chart for power of proposed FMA with FMA in [7]

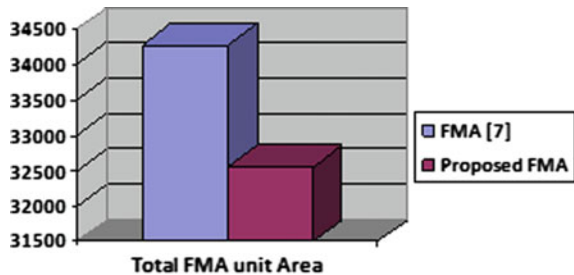


Fig. 15 Comparison chart for delay of proposed FMA with FMA in [7]

in total area when compared to FMA in [7]. Proposed FMA unit for FMUL instruction consumes almost same power as that of the FMA in [7]. But the drawback of proposed FMA unit is that, it has 7 % degradation in timing for FMUL instruction when compared to FMA in [7].

The stand-alone FMUL and FADD operations in existing floating-point units and ALUs [14, 15, 16] can be replaced by floating-point fused multiply-add, if a floating-point addition is followed by a floating-point multiplication. Further this FMA design can be extended and implemented using Residue Number System as it is gaining popularity for fast arithmetic operations [17].

4 Conclusion

This paper presents a low power and area efficient double precision floating-point fused multiply-add unit. The use of common add-round unit for FMUL and FMA instruction is the main reason for reduction in area occupied by the unit. By this the overall power consumption of the unit also decreased. The design has been compared with existing bridge FMA and it is found to be efficient in terms of power and area. But the only drawback is the degradation in timing for FMUL instruction. The proposed FMA can perform FMA operation or it can perform stand-alone FMUL

and FADD operations parallely with out any need for insertion of constants. This is not possible with the classic FMA unit. This FMA design is suitable for high performance floating-point units of the co-processors.

References

1. Schmookler M, Trong SD, Schwarz E, Kroener M (2007) P6 Binary floating-point unit. In: Proceedings of the 15th IEEE symposium on computer arithmetic, Montpellier, pp 77–86, June 2007
2. Hokenek E, Montoye R, Cook PW (1990) Second-generation RISC floating point with multiply-add fused. IEEE J Solid-State Circuits 25(5):1207–1213
3. Lang T, Bruguera JD (2004) Floating-point multiply-add-fused with reduced latency. IEEE Trans Comput 53(8):988–1003
4. Montoye RK, Hokenek E, Runyon SL (1990) Design of the IBM RISC System/6000 floating point execution unit. IBM J Res Dev 34:59–70
5. Pillai RVK, Shah SYA, Al-Khalili AJ, Al-Khalili D (2001) Low power floating point MAFs-a comparative study. In: Sixth international symposium on signal processing and its applications, 2001, vol 1, pp 284–287, 2001
6. Lang T, Bruguera JD, Floating-point fused multiply-add: reduced latency. In: Proceedings of the 2002 IEEE international conference on computer design: VLSI in computers and processors, pp 145–150, 2002
7. Quinnell E, Swartzlander EE, Lemonds C (2008) Bridge floating-point fused multiply-add design. IEEE Trans Very Large Scale Integr (VLSI) Syst 16(12):1727–1731
8. IEEE Standard for Binary Floating-Point Arithmetic (1985) ANSI/IEEE Standard 754–1985, Reaffirmed 6 Dec 1990, 1985
9. Dadda L (1964) Some schemes for parallel multipliers. IEEE Trans Comput 13:14–17
10. Waters RS, Swartzlander EE (2010) A reduced complexity wallace multiplier reduction. IEEE Trans Comput 59(8):1134–1137
11. Dimitrakopoulos Giorgos, Nikolos Dimitris (2005) High-speed parallel-prefix VLSI ling adders. IEEE Trans Comput 54(2):225–231
12. Anitha RV, Bagyaveereswaran (2012) High performance parallel prefix adders with fast carry chain logic. Int J Adv Res Eng Technol 3(2):01–10
13. Quach N, Takagi N, Flynn M, (1991) On fast IEEE rounding, Stanford University, Stanford, CA, Technical Report CSL-TR-91-459, Jan 1991
14. Dhanabal R, Bharathi V, Shilpa K, Sujana DV, Sahoo SK (2014) Design and implementation of low power floating point arithmetic unit. Int J Appl Eng Res 9(3):339–346, 2014. ISSN: 0973-4562
15. Ushasree G, Dhanabal R, Sahoo SK (2013) VLSI implementation of a high speed single precision floating point unit using verilog. In: Proceedings of IEEE conference on information and communication technologies (ICT 2013), pp 803–808, 2013
16. Dhanabal R, Bharathi V, Salim S, Thomas B, Soman H, Sahoo SK (2013) Design of 16-bit low power ALU-DBGPU. Int J Eng Technol 5(3):2172–2180
17. Dhanabal R, Sarat Kumar Sahoo, Barathi V, Samhitha NR, Cherian NA, Jacob PM (2014) Implementation of floating point mac using residue number system. J Theor Appl Inf Technol 62(2), April 2014