# A High-Speed Floating-Point Multiply-Accumulator Based on FPGAs

Bin Zhou, *Graduate Student Member, IEEE*, Guangsen Wang, Guisheng Jie, Qing Liu, and Zhiwei Wang

***Abstract*— In this article, a novel high-speed floating-point multiply-accumulator (FPMAC) is proposed. It comprises a signed soft multiplier and a single-cycle floating-point accumulator (FAAC). The multiplier is realized by a radix-4 Booth encoding based on sign-magnitude inputs. The FAAC contains a bidirectional shift alignment, a fast 3:1 compressor, and a three-operand leading-zero predictor (LZP). Due to the simplification of implementation steps, a short critical path, and a full-pipelined structure, our FPMAC achieves a high running speed while sustaining the good performance of delay and resource consumption. We have implemented and evaluated our FPMAC on two devices from Xilinx. Compared with two contrast FPMAC architectures, the results show that the maximum clock frequencies of our FPMAC increase by 33.4%–74.4% in single-precision floating-point (SFP) and 15.8%–86.1% in double-precision floating-point (DFP).**

***Index Terms*— Bidirectional shift, booth multiplier, floating-point, multiply-accumulate, single-cycle accumulator, ternary adder, three-operand leading-zero predictor (LZP).**

## I. Introduction

IN FIELDS of communication, multimedia, and other scientific and engineering applications, floating-point multiply-accumulators (FPMACs) are common and play an important role in digital signal processing (DSP) [1]. A multiply-accumulate can be expressed as

$$R_{ac}(n) = A(n) \times B(n) + R_{ac}(n-1). \tag{1}$$

Fig. 1 is a general architecture of an FPMAC, which can be divided into three steps: multiplying two floating-point operands $A(n)$ and $B(n)$, adding the product $P(n)$ and the feedback $R_{ac}(n-1)$, normalizing the final result $R_{ac}(n)$ [2]. Recently, due to attractive advantages in massive parallelism, deep pipeline, and rich distributed memories, field-programmable gate arrays (FPGAs) have been widely applied in high-performance computing and hardware acceleration [3], [4].

Manuscript received March 12, 2021; revised June 12, 2021 and July 28, 2021; accepted August 11, 2021. Date of publication August 27, 2021; date of current version October 6, 2021. This work was supported in part by the National Natural Science Foundation of China under Grant 61902422 and in part by the Key Laboratory of Equipment Pre-Research Foundation of China under Grant 6142217190506. *(Corresponding author: Guisheng Jie.)*

The authors are with the National Key Laboratory of Science and Technology on Vessel Integrated Power System, Naval University of Engineering, Wuhan 430033, China (e-mail: zhoubin5637@163.com; guangsenwang@sina.com; zhenyujie@sina.com; qing@alumni.hust.edu.cn; zwwang1mol@163.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2021.3105268.

Digital Object Identifier 10.1109/TVLSI.2021.3105268

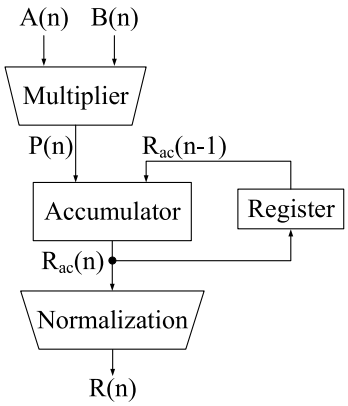


Fig. 1. General architecture of FPMACs.

As for an FPGA-based design, its performance relies on the critical path delay heavily. However, floating-point arithmetics, especially the floating-point accumulator (FAAC) in an FPMAC, are not so easy to be implemented on the hardware. The tricky feedback path in an FAAC is always the performance bottleneck of FPMAC circuits. Traditionally, an FAAC with a multiple-cycle pipeline latency, like $k$-cycle, can only accept a new input every $k$ cycles. This stall limits the design's throughput greatly. A single-cycle FAAC is an alternative, but its long critical path leads to a low clock frequency [5]. For the same reason, even though a fused multiply-add (FMA) can also be modified as an FPMAC, this usage is rare in practice [6]. Besides, Intel has offered hard floating-point DSP blocks that can natively implement an FAAC or an FPMAC, but they are only limited to Arria 10 and newer devices [7].

Recent research studies mainly focus on optimizations of FAACs and floating-point multipliers. A reduction circuit has been developed to increase the throughput of multiple-cycle pipelined FAACs [8]. However, such a reduction circuit consumes excessive resources and the adder tree inside it brings in extra delay [9]. Another multi-cycle method is to time-division multiplex the accumulator, which prevents the latest accumulation result from being fed back immediately [10]. The main limitation of this method is its high data sensitivity, i.e., it needs zero-padding operations when necessary.

Most optimizations about single-cycle accumulators concentrate on further shortening the critical path. In [11], outputs of the 4:2 compressor are returned as the accumulator's inputs. Thus, the carry propagate adder (CPA), which works as a

final adder, is excluded from the feedback path. Furtherly, the 4:2 compressor can be merged into a Wallace Tree directly [12], but it is not applicable for a high-speed FAAC. Since fixed-point accumulators are not so complicated as FAACs, it is natural to think about converting floating-point operands into fixed-point ones [13]. Actually, Intel and Xilinx both adopt this approach in their floating-point accumulation intellectual property (IP) cores [14], [15]. The cost of this conversion is a small dynamic range and a low calculation accuracy, even if the bit widths are extended.

A self-aligned method, which converts operands in single-precision floating-point (SFP) from base-2 (radix-2) into base-32 (radix-32), was described in a single-cycle FAAC [16], [17]. It has reduced the accumulator's complexity since the dynamic alignment process becomes four fixed and parallel alignment paths. Another advantage of this method is substituting expensive variable shifters (VSs) with constant shifters (CSs). However, it suffers from the time-consuming normalization of accumulation results and bit width extension caused by the conversion. For the sake of extending this method to applications in double-precision floating-point (DFP) and pursuing a higher speed, Bachir and David [18], [19] took several measures to optimize its accumulator, e.g., leaving out the normalization for path selections, truncating the mantissa. The optimized accumulator was applied to a commercial realtime simulator in the power electronics industry [20].

Recent research studies about optimizations of multipliers have been reported in [21]–[24]. On FPGAs and ASICs, the Booth encoding and the Wallace Tree are generally utilized to implement parallel multipliers. Improvements about these multipliers mainly focus on two aspects: cutting down the number of partial products and decreasing the delay caused by additions in the Wallace Tree [11]. Because the radix-2 Booth encoding does not really reduce partial products' amount, the modified Booth algorithm (radix-4) is a more widespread approach [25]. A high-radix Booth encoding brings higher elimination efficiency, but the calculation of relevant partial products is more challenging [12], [26]. In [24], the 4:2 compressor is based on a modified ternary adder of Xilinx FPGAs. This practice can ensure better area efficiency and lower delay. Meanwhile, FPGA vendors have provided multiplier IP cores [27], [28]. These verified, standard IP cores can take advantage of multiplier primitives to support high-performance implementations. But they are vendor-dependent, and users cannot conduct the custom tailor at a code level.

In this article, we introduce the design of a high-speed FPMAC. The rest of this article is structured as follows. Section II describes the general architecture of our FPMAC and outlines its noteworthy features. Section III presents details of our design, including a signed multiplier and a single-cycle FAAC. In Section IV, the implementation results are demonstrated and analyzed. The conclusion together with the future research plan is given in Section V.

## II. Proposed FPMAC Architecture

As shown in Fig. 2, the proposed FPMAC architecture (PRO) is full-pipelined and consists of a signed soft multiplier and a single-cycle FAAC. The soft multiplier is performed by a radix-4 Booth encoding and a Wallace Tree (S0–S2). The single-cycle FAAC is based on a 3:1 compressor (S4), together with a bidirectional shift alignment (S3) and a three-operand leading-zero predictor (LZP) (S4–S5) for accelerating the normalization (S6–S8).

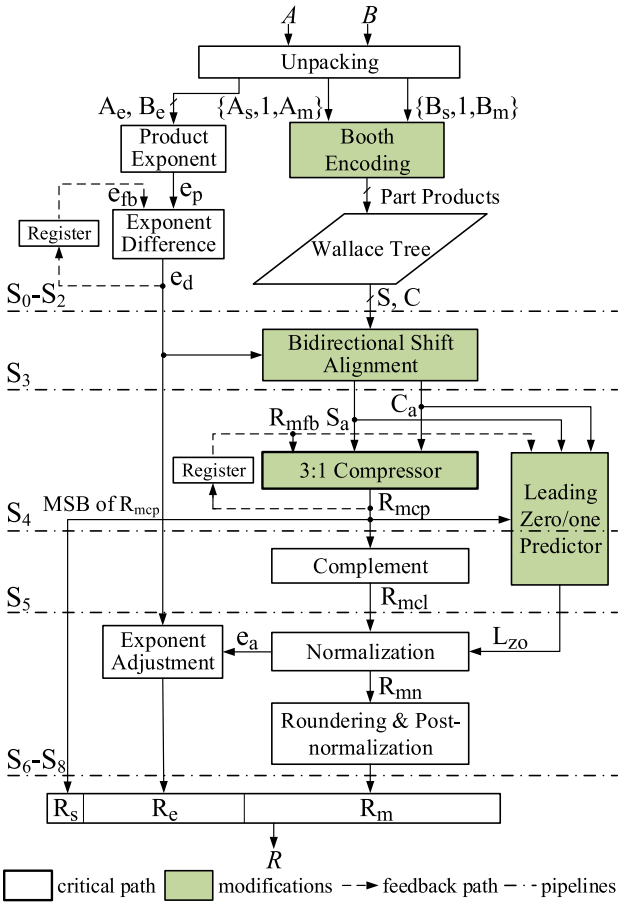


Fig. 2. Proposed FPMAC architecture.

An FPMAC architecture, which originates from basic FMAs but returns results of the final CPA as the feedback, is taken as the conventional architecture (CON) by us [6]. Vangal *et al.* [16] and Paidimarri *et al.* [17] proposed a self-aligned architecture (SEL). Bachir and David [18] and [19] optimized its accumulator by omitting the normalization of accumulation results. To ensure correct alignment paths, we choose an SEL architecture with the normalization as another contrast architecture to our PRO architecture. Note that both comparative architectures are not exactly the same as their originals. For a fair and intuitive comparison, we have also reconstructed two contrast architectures in the article. Compared with the CON architecture and the SEL architecture, the modifications of our FPMAC architecture can be summarized as follows.

1) A signed soft multiplier based on sign-magnitude inputs is realized.
2) A bidirectional shift replaces the unidirectional right-shift to complete the alignment.
3) A 3:1 compressor works as the accumulator and the final adder.

4) A three-operand LZP, rather than a two-operand LZP, accelerates the normalization.

Differences among all three architectures are presented in Table I. The steps with bold texts represent feedback paths of their accumulators, which are also their critical paths. Obviously, our PRO architecture has the shortest feedback path and the fewest implementation steps.

## III. Detailed Design

### A. Signed Booth Multiplier

In this part, we propose a signed Booth multiplier with sign-magnitude inputs. Generally, if the multiplier in an FPMAC is unsigned, its product must be complemented when $A_s \oplus B_s = 1$. To remove this complement step, we implement a signed Booth multiplier that does not deploy standard IP cores. It is because these IP cores exist redundant steps and provide no access to tailor the underlying code. For the floating-point multiplier IP core, its products have already been normalized into a standard format. Meanwhile, the fixed point one requires its sign-magnitude inputs to convert into 2's complement system. Therefore, our multiplier is designed with sign-magnitude inputs directly and returns its raw products as accumulator's inputs.

We found that both the conversion from sign-magnitude to 2's complement and the generation of the negative partial product $(-X)$ are complement operations. The complement for the conversion can be omitted absolutely if we make full use of the partial product generator. Considering that it is the multiplicand that generates partial products. The negative input is exchanged as a multiplicand if $A_s \neq B_s$. To prevent overflow when generating partial products, the multiplicand and its complement are sign-extended. These extended sign bits should be reversed when operating a complement. $C2$ and $E2$ represent complement operation and sign extension, $A_s$ and $A_m$ are the sign bit and mantissa of the multiplicand, as expressed in

$$-X = C2(E2(A_m)) = \{\widetilde{A}_s, \widetilde{A}_s, C2(A_m)\}. \tag{2}$$

According to the Booth encoding rule (radix-4), whether the multiplier is signed or unsigned, their amounts of generated partial products are equal (13 in SFP, 26 in DFP). Though all partial products extend one sign bit, the resource consumption and delay they induced are negligible compared with a complement operation. The pseudo-code of the partial product generator in our signed soft multiplier is as follows.

**Algorithm 1** Partial Product Generator Based on Sign-Magnitude Inputs

**Require:** Sign bits$(A_s, B_s)$, mantissa$(A_m, B_m)$
**Ensure:** Part products$(X, -X, 2X, -2X)$, multiplier$(Y)$
//Extend sign bits and exchange the negative input as the multiplicand
**if** $A_s \oplus B_s = 1$ **then**
  $a_s \leftarrow 2'b11$
  **if** $A_s = 1$ **then**
    $a_m \leftarrow A_m$
    $b_m \leftarrow B_m$
  **else**
    $a_m \leftarrow B_m$
    $b_m \leftarrow A_m$
  **end if**
  $a_c \leftarrow \widetilde{a}_m + 1$
  $a_p \leftarrow \{a_s, a_c\}$
  $a_n \leftarrow \{\widetilde{a}_s, a_m\}$
**else**
  $a_s \leftarrow 2'b00$
  $a_m \leftarrow A_m$
  $b_m \leftarrow B_m$
  $a_c \leftarrow \widetilde{a}_m + 1$
  $a_p \leftarrow \{a_s, a_m\}$
  $a_n \leftarrow \{\widetilde{a}_s, a_c\}$
**end if**
//Generate part products and the multiplier
$X \leftarrow a_p$
$-X \leftarrow a_n$
$2X \leftarrow X << 1$
$-2X \leftarrow -X << 1$
$Y \leftarrow \{2'b00, b_m, 1'b0\}$

### B. Bidirectional Shift and Alignment

Floating-point addition or subtraction must keep two operands with the same exponent, the addend with a small exponent usually aligns to that with a large one. As depicted in Fig. 3(a), the CON architecture conducts its alignment through a unidirectional right-shift. First, the exponents of product ($e_p$) and feedback ($e_{fb}$) are subtracted. Then, the sign of their result ($e_d$) determines that it is $S$ and $C$, or $R_{mfb}$ to take a right-shift (SHR), and the size of $e_d$ decides the bits to shift.

In this part, the alignment method of FMAs is taken as a reference. In an FMA, its addend aligns to the product through a bidirectional shift while the product itself remains unchanged. Similarly, our PRO architecture performs a bidirectional shift alignment for $S$ and $C$ while keeping $R_{mfb}$ fixed. As shown in Fig. 3(b), if $e_p \leq e_{fb}$, $S$ and $C$ are shifted to right as usual. If $e_p > e_{fb}$, $S$ and $C$ take a left-shift (SHL). After a bidirectional shift, the bit widths of $S_a$ and $C_a$ are expanded by $m_w$ bits, so are the accumulator, complement, LZP, etc. However, its benefit is also obvious, the time-consuming alignment is excluded from the accumulator's feedback path.

The right-shift, the bidirectional shift alignment of $S$ and $C$ both face the mantissa truncation caused by right-shifts. Note that if the bidirectional shift is performed by $R_{mfb}$, just like the practice in FMAs, there will be no truncation. To pursue a high speed rather than high accuracy, we choose $S$ and $C$ to finish a bidirectional shift. The unidirectional right-shift in the CON architecture, the bidirectional shift in FMAs, and the bidirectional shift in our PRO architectures can be compared and analyzed from the following three aspects: bit width extension, data truncation, and critical path length. We find that their design goals correspond to resource first, accuracy first, and speed first, respectively.

Design metrics of alignment shifters in three architectures are listed in Table II by two situations: not truncated and

TABLE I
DIFFERENCES AMONG THREE ARCHITECTURES

| Architecture | CON | SEL | PRO |
|---|---|---|---|
| Steps(partial) | Unsigned multiplication | Unsigned multiplication | Signed multiplication |
| | Not used | Radix-32 conversion | Not used |
| | Complement | Complement | Not used |
| | **3-operand unidirectional right-shift** | **4-operand self-alignment** | 2-operand bidirectional shift |
| | **3:2 compressor** | **4:2 compressor** | **3:1 compressor** |
| | **CPA** | CPA | Not used |
| | Not used | Radix-2 conversion | Not used |
| | 2-operand LZP | 2-operand LZP | 3-operand LZP |

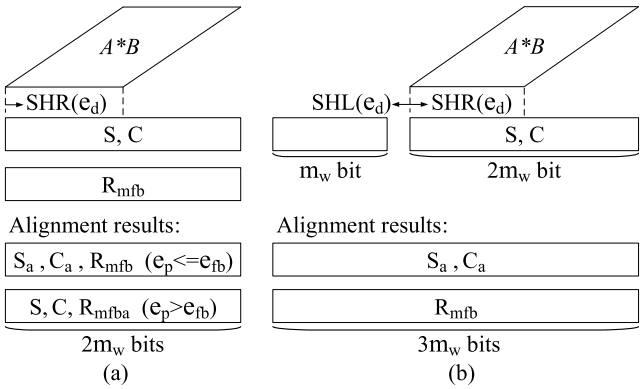


Fig. 3. Unidirectional right-shift versus bidirectional shift. (a) Unidirectional right-shift. (b) Bidirectional shift.

TABLE II
DESIGN METRICS OF ALIGNMENT SHIFTERS

| Architecture | CON | SEL | PRO |
|---|---|---|---|
| Type | VS | CS | VS |
| Amount(not truncated) | 3 | $4*(2m_w \backslash (2^k-1))$ | 2 |
| Size(not truncated) | $2m_w$ | $2m_w+2^k-1$ | $3m_w+1$ |
| Amount(truncated) | 3 | $4*((m_w+g)\backslash(2^k-1))$ | 2 |
| Size(truncated) | $m_w+g$ | $m_w+2^k-1+g$ | $m_w+1+2g$ |

truncated. Parameters are defined as follows: $m_w$, the mantissa width; $k$, the power of a high-radix ($2^k$); $g$, guard bits [19]. Alignment shifters of the SEL architecture are different from the other two. The CSs in it are much simpler but exist a dilemma in settings of the amount and the size. For simplicity and preventing overflow, the mantissa of accumulation results in the SEL architecture is truncated according to the parameter $g$. For an impartial comparison among different architectures, we perform the same truncation on the CON architecture and our PRO architecture. Table III shows settings of alignment shifters. Parameter settings are $m_w = 24$, $k = 4$, and $g = 16$ in SFP, $m_w = 53$, $k = 5$, and $g = 32$ in DFP. Only SFP and DFP are considered because of their widespread usage. As is shown, the CON architecture has the smallest shifters. Though the SEL architecture has the most shifters, its CSs are much simpler than those VSs. Shifters in our PRO architecture are the largest in the size but the least in amount.

### C. 3:1 Compressor

As the latest FPGA families of Intel and Xilinx all offer ternary adders [29], [30]. We take a ternary adder as the single-cycle accumulator to gain an efficient compression, namely a 3:1 compressor. As analyzed in [22], these ternary adders are area efficient because they consume the same resource as two-input adders. Intel FPGAs can support ternary additions directly because their adaptive logic modules (ALMs) have two outputs. For ternary adders in Xilinx FPGAs, they rely on a specific primitive (LUT6-2) and suffer from a decrease in speed. As depicted in Fig. 4(a), an LUT6 in the configurable logic blocks (CLBs) acts as a dual LUT5 and outputs two values ($co_i$ and $s_i$). As a result, the LUT6 has replaced the dedicated carry logic to be the ternary adder's critical path [bold lines in Fig. 4(a)]. To implement a fast ternary adder on Xilinx FPGAs, we refer to the practice in Intel FPGAs. Fig. 4(b) shows the structure of our 3:1 compressor, the carry propagation of $co_i$ through LUTs is cut off, the carry propagation of $c_i$ through the dedicated carry logic ($muxcy$ and $xorcy$) is retained. On this condition, the delay of a 3:1 compressor is equal to that of a CPA approximately.

TABLE III
SETTINGS OF ALIGNMENT SHIFTERS

| Architecture | CON | SEL | PRO |
|---|---|---|---|
| Amount | 3 VSs | 8 CSs | 2 VSs |
| Size(SFP) | 40 | 55 | 57 |
| Size(DFP) | 85 | 116 | 118 |

Actually, the structure of ternary adders can be explained as two cascaded 3:2 compressors. In Fig. 4(b), an LUT3 and an LUT4, whose logic is expressed as

$$co_i = R_{mfb}[i] * S_a[i] + R_{mfb}[i] * C_a[i] + S_a[i] * C_a[i] \tag{3}$$

$$s_i = R_{mfb}[i] \oplus S_a[i] \oplus C_a[i] \oplus co_{i-1} \tag{4}$$

construct a 3:2 compressor of the first stage, a $muxcy$ and a $xorcy$ construct a 3:2 compressor of the second stage. First, three inputs $R_{mfb}[i]$, $S_a[i]$, and $C_a[i]$ generate a carry ($co_i$) through the LUT3, their remainder is denoted as $I_r[i] = R_{mfb}[i] \oplus S_a[i] \oplus C_a[i]$. Based on this, the LUT4 can be equivalent to an "XOR" operation between $I_r[i]$ and $co_{i-1}$. Then, $I_r[i]$, $co_{i-1}$, and $c_{i-1}$ are configured as three inputs of the second stage 3:2 compressor, while $R_{mcp}[i]$ and $c_i$ are obtained according to

$$R_{mcp}[i] = s_i \oplus c_{i-1} \tag{5}$$

$$c_i = I_r * co_{i-1} + s_i * c_{i-1}. \tag{6}$$

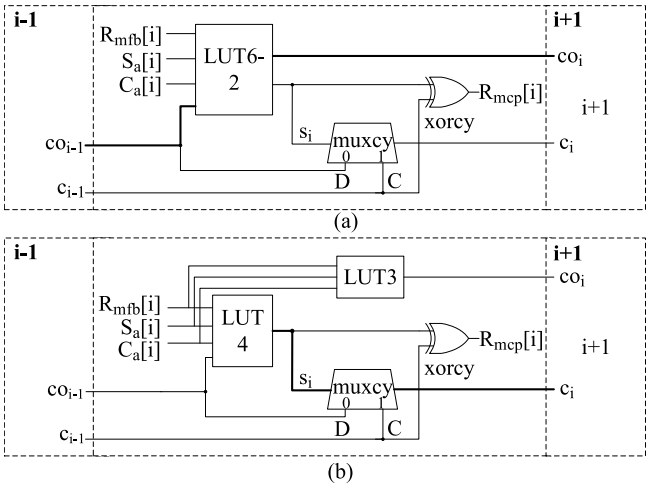


Fig. 4. Structures of two ternary adders at $i$ th bit. (a) Area efficient structure from Xilinx [29]. (b) High-speed structure in our 3:1 compressor.

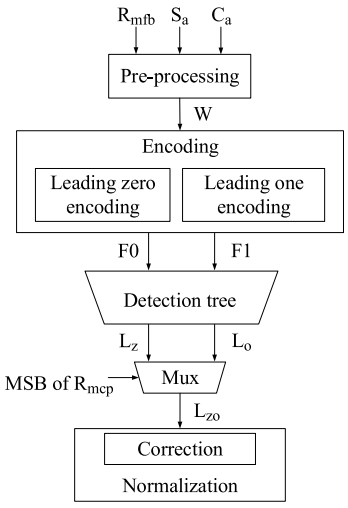


Fig. 5. Proposed three-operand LZP.

TABLE IV
ACCUMULATORS AND FINAL ADDERS OF THREE ARCHITECTURES

| Accumulator & final adder | Slice LUTs | Total delays |
|---|---|---|
| 3:2 compressor & CPA | $2n+n$ | $t_{LUT}+t_{CPA}$ |
| 4:2 compressor & CPA | $4n+n$ | $2t_{LUT}+t_{CPA}$ |
| 3:1 compressor | $2n$ | $t_{LUT}+t_{CPA}$ |

If $I_r[i] = co_{i-1}$, then $s_i = 0$, $R_{mcp}[i] = c_{i-1}$, and $c_i = co_{i-1}$. Similarly, if $I_r[i] \neq co_{i-1}$, then $s_i = 1$, $R_{mcp}[i] = \tilde{c}_{i-1}$, and $c_i = c_{i-1}$. Significantly, $co_i$ in $i$-bit is yielded in parallel among different bits and only affects the output of $i$ th bit ($R_{mcp}[i+1]$), while $c_i$ can be passed to higher bits all the time through the carry chain.

A 3:2 compressor, a 4:2 compressor, and a 3:1 compressor are used as accumulators in three FPMAC architectures, respectively. To implement them and a CPA, the corresponding numbers of LUTs are 2n, 4n, 2n, and n. The delay of 3:2 compressors and 4:2 compressors are fixed, their logic layers are 1 and 2, so we denote their total logic delay as $t_{\text{LUT}}$ and $2t_{\text{LUT}}$. The logic delay of 3:1 compressors and CPAs vary linearly with their bit widths. In addition, the 3:1 compressor experiences an LUT3 at the beginning moment of additions. Thus, its delay should add a $t_{\text{LUT}}$ to $t_{\text{CPA}}$. The resource consumption and delay of accumulators and final adders of three architectures are listed in Table IV.

In terms of the design of accumulators and final adders, the 3:1 compressor requires the least resource and delay, whereas its corresponding three-operand LZP consumes more resources than the one with two operands. The SEL architecture gets the most resource consumption and delay. But its fixed accumulation delay will outstand under large bit widths. The delay of a 3:1 compressor varies linearly with its bit widths. In this case, a segmented addition or a simple one-level carry-select adder can be taken for optimizations, especially in DFP or larger bit widths.

### D. Three-Operand LZP

An LZP and a leading-one predictor (LOP) can predict the position of the first "1" (positive sums) or the first "0" (negative sums) from the inputs. They are executed in parallel with the final adder and accelerate the normalization. For simplicity, we take the LZP to represent them, unless otherwise specified. In general LZPs, their inputs should be compressed into two operands. In this part, a three-operand LZP is proposed and cooperates with the 3:1 compressor in Section III-C.

Similar to that of two-operand, a three-operand LZP comprises four units: pre-processing, encoding, detection, and correction (see Fig. 5) [16], [31]–[33]. In the pre-processing unit, $R_{mfb}[i]$, $S_a[i]$, and $C_a[i]$ are added in the form of

TABLE V
LEADING ZERO ENCODING OF THREE-OPERAND

| Pattern of $W$ | 1st "1" position | Substring |
|---|---|---|
| $1^k3(x)$ | $k/k+1$ | $P_{i+1}T_i$ |
| $1^k23(x)$ | $k/k+1$ | $G_iT_{i-1}$ |
| $1^k22(x)$ | $k/k+1$ | $G_iG_{i-1}$ |
| $1^k21(x)$ | $k+1/k+2$ | $G_{i+1}P_i$ |
| $1^k20^j3(x)$ | $k+j/k+j+1$ | $Z_iT_{i-1}$ |
| $1^k20^j2(x)$ | $k+j/k+j+1$ | $Z_iG_{i-1}$ |
| $1^k20^j1(x)$ | $k+j+1/k+j+2$ | $Z_{i+1}P_i$ |

TABLE VI
LEADING ONE ENCODING OF THREE-OPERAND

| Pattern of $W$ | 1st "0" position | Substring |
|---|---|---|
| $1^k013^j(x), j=1$ | $k+2/k+3$ | $P_{i+1}T_i\widetilde{T}_{i-1}$ |
| $1^k013^j(x), j>1$ | $k+j+1/k+j+2$ | $T_{i+1}T_i\widetilde{T}_{i-1}$ |
| $1^k012(x)$ | $k+2/k+3$ | $P_{i+1}G_i$ |
| $1^k011(x)$ | $k+1/k+2$ | $Z_{i+1}P_iP_{i-1}$ |
| $1^k010(x)$ | $k+1/k+2$ | $Z_{i+1}P_iZ_{i-1}$ |
| $1^k00(x)$ | $k+1/k+2$ | $Z_{i+1}Z_i$ |
| $1^k02^j0(x)$ | $k+j+1/k+j+2$ | $G_{i+1}Z_i$ |
| $1^k02^j10(x)$ | $k+j+1/k+j+2$ | $G_{i+1}P_iZ_{i-1}$ |
| $1^k02^j11(x)$ | $k+j+1/k+j+2$ | $G_{i+1}P_iP_{i-1}$ |
| $1^k02^j12(x)$ | $k+j+2/k+j+3$ | $P_{i+1}G_i$ |

TABLE VII
EVALUATION RESULTS OBTAINED FROM XILINX VIRTEX-6 FPGA (XC6VLX240T)

| Architecture | CON | SEL | PRO | CON | SEL | PRO |
|---|---|---|---|---|---|---|
| Precision | | SFP | | | DFP | |
| Maximum clock frequency(MHz) | 215.979 | 267.924 | 375.023 | 171.674 | 265.058 | 307.748 |
| Pipeline stages | 5 | 7 | 9 | 5 | 7 | 9 |
| Total daley(ns) | 23.150 | 23.604 | 24.093 | 29.125 | 26.411 | 29.241 |
| Number of slice registers | 1062(0%) | 1781(0%) | 956(0%) | 3456(1%) | 4507(1%) | 2914(0%) |
| Number of slice LUTs | 2175(0%) | 2349(1%) | 2693(1%) | 7190(4%) | 8016(5%) | 7770(5%) |
| Used LUT-FF pairs | 2686 | 3347 | 2757 | 8909 | 10104 | 8564 |

TABLE VIII
EVALUATION RESULTS OBTAINED FROM XILINX VIRTEX-7 FPGA (XC7VX485T)

| Architecture | CON | SEL | PRO | CON | SEL | PRO |
|---|---|---|---|---|---|---|
| Precision | | SFP | | | DFP | |
| Maximum clock frequency(MHz) | 223.470 | 287.405 | 383.039 | 181.028 | 279.496 | 337.393 |
| Pipeline stages | 5 | 7 | 9 | 5 | 7 | 9 |
| Total daley(ns) | 22.375 | 24.353 | 23.499 | 27.620 | 25.046 | 26.676 |
| Number of slice registers | 1062(0%) | 1769(0%) | 979(0%) | 3455(0%) | 4640(0%) | 3045(0%) |
| Number of slice LUTs | 2226(0%) | 2544(0%) | 2869(0%) | 7204(2%) | 8203(2%) | 8105(2%) |
| Used LUT-FF pairs | 2738 | 3552 | 2946 | 8924 | 10573 | 8917 |

$w_i = R_{mfb}[i] + S_a[i] + C_a[i]$, where carry is out of consideration and $w_i \in \{0, 1, 2, 3\}$ are denoted as $Z$, $P$, $G$ and $T$, respectively. The final sum is $W = w_{m-1}w_{m-2}, \ldots, w_1 w_0$. We take the $i$ th bit to explain how to encode. It is assumed that there are $k$ consecutive "1" in front of $w_i$ and $j$ consecutive $w_j \in \{0, 2, 3\}$ behind $w_i$. When $w_i = 2$ or $w_i = 3$, all "1" in front of $w_i$ become "0" due to the carry, this case corresponds to the LZP. When $w_i = 1$, $i$ increases by one, the encoding task goes to the $(i + 1)$ th bit straightway. When $w_i = 0$, it is corresponding to the LOP. Encoding results of the LZP and the LOP based on three big-endian operands are listed in Tables V and VI. The detection unit is a general binary tree circuit, more details of which can be found in [31] and [32].

There may be a deviation of one bit in the detection results. We take the $(i + 1)$ th bit as the default result temporarily and correct it during the normalization. The identification substrings of the LZP and the LOP can be unified as

$$F0 = P_{i+1}T_i + (G_{i+1} + Z_{i+1})P_i + (G_i + Z_i)(T_{i-1} + G_{i-1}) \quad (7)$$

$$F1 = ((Z_{i+1} + G_{i+1})(P_i(P_{i-1} + Z_{i-1}) + Z_i) + P_{i+1}G_i + (P_{i+1} + T_{i+1})T_i\widetilde{T}_{i-1}. \quad (8)$$

$L_{zo}$ is ultimately determined between $L_z$ and $L_o$ according to the sign of $R_{mcp}$. The correction logic will induce extra complexity and double an LZP circuit's resource consumption approximately. We adopt an easy and feasible aftermath correction. That is, after $R_{mcl}$ finishing a left-shift by $L_{zo}$ bits, its most two significant bits are used to decide whether to correct or not. If they are "01" (positive sums) or "11" (negative sums), a correction is necessary. Undoubtedly, this will increase the delay of normalization logic slightly, but it does not affect the critical path of the whole circuit.

## IV. IMPLEMENTATION AND EVALUATION

We have described our PRO architecture and two contrast architectures in Verilog HDL. Afterward, they were evaluated by Xilinx ISE 14.7 software, targeting the Virtex-6 XC6VLX240T and the Virtex-7 XC7VX485T from Xilinx, respectively. The code can be easily ported to devices from different vendors because it does not rely on any IP cores.

### A. Overall Performance

A VLSI circuit's performance is mainly reflected in its clock frequency, delay, and resource consumption, etc. Tables VII and VIII summarize these evaluation results of three architectures in SFP and DFP. We can find that the performance of each architecture is consistent between the two tables. Overall, our PRO architecture achieves the highest clock frequency, the CON architecture has the lowest speed but consumes relatively little resource, the SEL architecture increases the speed at the expense of consuming more resource. For our PRO architecture, though it has the maximum bit widths (see Table III), its performance in resource consumption is acceptable, even best in DFP. This is because of the simplification of implementation steps and the truncation of mantissa. Moreover, its speeds outperform the other two architectures by 33.4%–74.4% in SFP and 15.8%–86.1% in DFP, which means higher throughput. As mentioned previously, our PRO architecture can also be further optimized for a faster speed through a segmented addition or a simple one-level carry-select adder.

For the CON architecture, its bit widths are only approximately 70% of the other two architectures (see Table III). So it has the advantage to acquire lower delay and less resource consumption. This is why it gets the minimum total delay and occupies the least resources in SFP. But this advantage

TABLE IX
RESOURCE CONSUMPTION OF DIFFERENT COMPONENTS IN PRO

| Blocks | Multiplier | Accumulator | LZA | Normalization | Multiplier | Accumulator | LZA | Normalization |
|---|---|---|---|---|---|---|---|---|
| Precision | SFP | | | | DFP | | | |
| Number of slice registers | 396 | 269 | 119 | 201 | 1311 | 702 | 595 | 360 |
| Number of slice LUTs | 1358 | 442 | 674 | 406 | 5922 | 1101 | 835 | 929 |
| Used LUT-FF pairs | 1382 | 470 | 677 | 422 | 6042 | 1320 | 1160 | 929 |
| Proportion(%) | 46.91 | 15.95 | 22.98 | 14.32 | 63.93 | 13.97 | 12.27 | 9.83 |

TABLE X
PERFORMANCE COMPARISONS OF MULTIPLIERS FROM PRO AND IP CORES

| Bit widths | 25 | | | 54 | | |
|---|---|---|---|---|---|---|
| Multipliers | PRO | IP cores | IP cores | PRO | IP cores | IP cores |
| Pipeline stages | 3 | 3 | 5 | 3 | 3 | 6 |
| Number of slice registers | 396 | 494 | 717 | 1311 | 2036 | 3068 |
| Number of slice LUTs | 1358 | 809 | 824 | 5922 | 3287 | 3444 |
| Used LUT-FF pairs | 1382 | 1303 | 1541 | 6042 | 5323 | 6512 |
| Maximum clock frequency(MHz) | 473.252 | 408.213 | 661.463 | 365.631 | 244.391 | 493.997 |

will be weakened with large bit widths. As a result of large size or big amount alignment shifters and the normalization for selecting alignment paths, the SEL architecture consumes the most resource. Its speeds, however, only experience a tiny drop in DFP due to its fixed accumulation delay. This result is consistent with our previous analysis in Section III-C.

Total delay of each architecture is close in most cases. Although bit widths of our PRO architecture are the maximum, its delay is not outstanding since its implementation steps have been simplified. In DFP, the SEL architecture has almost the same bit widths as our PRO architecture, but its delay is much less. The reason is that our PRO architecture has more pipeline stages than contrast architectures, which means less possibility to be automatically optimized by the synthesizer.

All three architectures have made no use of hardware DSP blocks and have only consumed up to 5% (XC6VLX240T) and 2% (XC7VX485T) of the total slice logics. It is the soft multiplier that occupies the majority of these resources. This will be discussed in the next part.

### B. Partial Analysis

Table IX lists the resource consumption of four components of our PRO architecture. The evaluation results are obtained based on Xilinx Virtex-7 XC7VX485T. The resources occupied by multipliers account for 46.91% in SFP and 63.93% in DFP. A radix-4 Booth encoding has low elimination efficiency and constructs a huge Wallace tree, especially in DFP. However, in comparison with the multipliers generated by IP cores (LogiCORE IP Multiplier, speed optimized), our multipliers perform better.

As shown in Table X, our multipliers are divided into three pipeline stages in both SFP and DFP. For the ones generated by IP cores, they fail to reach close speeds under the same condition. Only the pipeline stages are set to 5 and 6, respectively, can their speeds exceed our multipliers at the cost of more resources and longer delay.

## V. CONCLUSION

In this article, an FPGA-based high-speed FPMAC is proposed. We design a signed soft multiplier based on sign-magnitude inputs and take a 3:1 compressor as the single-cycle FAAC. The soft multiplier and the 3:1 compressor have simplified implementation steps of the FPMAC. The bidirectional shift alignment together with the 3:1 compressor shortens the critical path. A three-operand LZP is developed to cooperate with the 3:1 compressor and accelerate the normalization. Due to these modifications and a full-pipelined structure, our FPMAC obtains a good overall performance. It can achieve a high running speed while sustaining delay and resource consumption close to contrast FPMAC architectures.

Because of relying on no IP cores, our FPMAC is not limited by devices. In addition, it can function as building blocks of FPGA-based high-performance computing systems, such as matrix multiplication and matrix-vector multiplication. It is our future research work.

## REFERENCES

[1] D. Esposito, D. De Caro, E. Napoli, N. Petra, and A. G. M. Strollo, "On the use of approximate adders in carry-save multiplier-accumulators," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.
[2] J. J. F. Cavanagh, *Digital Computer Arithmetic: Design and Implementation*. New York, NY, USA: McGraw-Hill, 1984.
[3] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating SpMV on FPGAs by compressing nonzero values," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2015, pp. 64–67.
[4] M. Psarakis, A. Dounis, A. Abdoalnasir, S. Stavrinidis, and G. Gkekas, "An FPGA-based accelerated optimization algorithm for real-time applications," *J. Signal Process. Syst.*, vol. 92, no. 2, pp. 1–22, Feb. 2020.
[5] D. Wilson and G. Stitt, "The unified accumulator architecture: A configurable, portable, and extensible floating-point accumulator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 1–23, 2016.
[6] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida, "Ultra-low-power adder stage design for exascale floating point units," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3s, pp. 1–24, Mar. 2014.
[7] Intel. (2014). *The Industry's First Floating-Point FPGA*. [Online]. Available: https://www.intel.com

[8] L. Zhuo and V. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. FPGA*, Feb. 2005, pp. 63–74.
[9] S. Sun and J. Zambreno, "A floating-point accumulator for FPGA-based high performance computing applications," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2009, pp. 493–499.
[10] J. D. Cappello and D. Strenski, "A practical measure of FPGA floating point acceleration for high performance computing," in *Proc. IEEE 24th Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jun. 2013, pp. 160–167.
[11] Y.-H. Seo and D.-W. Kim, "A new VLSI architecture of parallel multiplier–accumulator based on Radix-2 modified booth algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 2, pp. 201–208, Feb. 2010.
[12] M. MohamedAsanBasiri and N. M. Sk, "An efficient hardware-based higher radix floating point MAC design," *ACM Trans. Design Autom. Electron. Syst.*, vol. 20. no. 1, p. 15, 2014.
[13] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2008, pp. 33–40.
[14] Intel. (2016). *Floating-Point IP Cores User Guide*. [Online]. Available: https://www.altera.com
[15] Xilinx. (2016). *Logicore IP Floating-Point Operator v7.1*. [Online]. Available: https://www.xilinx.com
[16] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar, and A. Alvandpour, "A 6.2-GFlops floating-point multiply-accumulator with conditional normalization," *IEEE J. Solid-State Circuits*, vol. 41, no. 10, pp. 2314–2323, Oct. 2006.
[17] A. Paidimarri, A. Cevrero, P. Brisk, and P. Ienne, "FPGA implementation of a single-precision floating-point multiply-accumulator with single-cycle accumulation," in *Proc. 17th IEEE Symp. Field Program. Custom Comput. Mach.*, Apr. 2009, pp. 267–270.
[18] T. O. Bachir and J.-P. David, "Performing floating-point accumulation on a modern FPGA in single and double precision," in *Proc. 18th IEEE Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2010, pp. 105–108.
[19] T. Ould-Bachir and J. P. David, "Self-alignment schemes for the implementation of addition-related floating-point operators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 1, pp. 1:1–1:21, 2013.
[20] *Opal-RT*. Accessed: 2018. [Online]. Available: https://wiki.opal-rt.com/display/HDGD/OP5600V2
[21] M. Kumm, S. Abbas, and P. Zipf, "An efficient softcore multiplier architecture for Xilinx FPGAs," in *Proc. IEEE 22nd Symp. Comput. Arithmetic*, Jun. 2015, pp. 18–25.
[22] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2009, pp. 250–255.
[23] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf, and U. Meyer-Baese, "Multiple constant multiplication with ternary adders," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–8.
[24] M. Kumm and P. Zipf, "Efficient high speed compression trees on Xilinx FPGAs," in *Proc. MBMV*, 2014, pp. 171–182.
[25] X. Cui, W. Liu, X. Chen, E. E. Swartzlander, and F. Lombardi, "A modified partial product generator for redundant binary multipliers," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1165–1171, Apr. 2016.
[26] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi, "Approximate hybrid high radix encoding for energy-efficient inexact multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 421–430, Mar. 2018.
[27] Xilinx. (2015). *Multiplier v12.0 Logicore IP Product Guide*. [Online]. Available: https://www.xilinx.com
[28] Intel. (2020). *Intel FPGA Integer Arithmetic IP Cores User Guide*. [Online]. Available: https://www.intel.com
[29] J. M. Simkins and B. D. Philofsky, "Structures and methods for implementing ternary adders/subtractors in programmable logic devices," U.S. Patent 7 274 211, Sep. 25, 2007.
[30] G. Baeckler, M. Langhammer, J. Schleicher, and R. Yuan, "Logic cell supporting addition of three binary words," Patent 7 565 388, Jul. 21, 2009.
[31] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 1, pp. 124–128, Mar. 1994.
[32] J. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction," *IEEE Trans. Comput.*, vol. 48, no. 10, pp. 1083–1097, Oct. 1999.
[33] R. Ji *et al.*, "Comments on 'leading-one prediction with concurrent position correction,'" *IEEE Trans. Comput.*, vol. 58, pp. 1726–1727, 2009.