Floating–Point Fused Multiply–Add under HUB Format

Javier Hormigo, Julio Villalba-Moreno, Sonia Gonzalez-Navarro
Department of Computer Architecture, Andalucia Tech, University of Malaga , Spain
Email: fjhormigo, jvillalba, sgn @uma.es

***Abstract*—The Half-Unit-Biased (HUB) format has interesting advantages for implementing floating-point arithmetic which has been proved for the four basic arithmetic operations as well as square root. Nevertheless, although Floating-point Fused Multiply-add (FMA) operation (AxB + C) is one of the most important and complex arithmetic instructions in modern processors, FMA operation for HUB numbers has not been confronted yet. In this paper, we present a design to deal with this operation under HUB format. The key points to turn the conventional FMA architecture into a HUB unit are explained. Comparing the ASIC implementation of a HUB FMA unit with the conventional one, the former reduces the required area and power up to 38% and 35%, respectively, for single-precision. For BFloat16, the HUB FMA increases the speed a 15%, and even then, reduces the area and power by 26% and 12%, respectively.**

***Index Terms*—Fused multiplication-addition, HUB format, DSP applications, Deep-learning**

I. Introduction

The spectacular growth of the deep-learning applications and their need for vast computation resources has encouraged researchers to explore new formats other than the standard IEEE in order to improve performance and reduce hardware cost. For example, Bfloat16 is an emerging floating-point format of 16 bits tailored specifically for high-performance processing of Neural Networks and it is expected to be implemented on the new generation of processors [1]. In [1] the authors propose a new SIMD instruction for accelerating matrix multiplication where the operands meet the Bfloat16 format. They implement the FMA operation and propose the use of floating-point round-to-odd mode to reduce the hardware cost of implementing the round-to-nearest mode. Similarly, in [2], DLfloat (another 16-bit floating-point format) is proposed to implement Deep-Learning accelerators. To reduce hardware cost and improve performance, DLfloat, besides changing the number of bits used to represent the exponent and the significand, basically simplifies many of the characteristics of the IEEE floating-point standard by using only round-to-nearest up and eliminating denormals and some special cases. Therefore, there is a clear tendency to use new floating-point formats that simplify the hardware implementation with the goal of improving the performance of these current emerging applications.

Following the same trend, in the literature, we can also find the HUB approach [3]. HUB is the acronym of Half-Unit-Biased format since it is based on shifting conventional numbers by half Unit-in-the-Last-Place (ULP). HUB approach can be applied to any floating-point format, which allows implementing two's complement by bit-wise inversion and the rounding to nearest by simple truncation. Hence, applying the HUB approach to an arithmetic unit simplifies the logic which reduces both area and delay, while it keeps the same precision [4]. Moreover, the HUB approach is compatible with the new proposed formats, such as BFloat16 or DLFloat, and could increase their advantages.

The advantages of the HUB approach have been validated for the four basic arithmetic operations as well as square root [4], [5], [6]. However, the fused multiply-add operation (FMA) for HUB numbers has not been studied yet. The FMA is a key operation in many current applications, from scientific computation to deep learning algorithms [7]. In fact, most modern processors have this operation in their Instruction Set Architecture (ISA) [8], [9]. This operation performs the multiplication of two floating-point operands and adds a third one to the result of the multiplication (AxB + C). The unified architecture has two main advantages, namely the operation is performed with only one rounding instead of two [10], and there is a reduction in the delay and hardware required by sharing several components [11].

In this paper, we design and analyze a floating-point FMA architecture to support HUB numbers. We show that it saves area and power consumption in comparison with conventional implementation when targeting the same clock frequency. Furthermore, it can reach higher frequencies.

From now on, we denote "conventional" to those numbers, formats, or architectures that, although not being standard compliant, follow similar rules and are not HUB.

II. The HUB format

In this section, we summarize the HUB format defined in [3] and particularize it for the floating-point normalized HUB numbers. The mathematical fundamentals and a deep analysis of the HUB format, as well as the addition and multiplication operations under this format, can be found in [3] and [4].

From a format point of view, a normalized floating-point HUB number is similar to its IEEE counterpart but the significand is HUB. Thus, the only difference is the format of the significand. Without any loss of generality, we consider radix-2 in this paper.

This work has been supported by the Ministry of Science and Innovation of Spain under project CICYT TIN2016-80920R

Let x denote a normalized floating-point HUB number such that

$$x = (-1)^s M 2^e \tag{1}$$

where s is the sign bit, e is the exponent, and the significand M is normalized ($1 < M < 2$) and has p bit of precision such that

$$M = \left[\sum_{i=0}^{p-1} m_i \cdot 2^{-i}\right] + 2^{-p} \tag{2}$$

where $m_i = \{0, 1\}$ with $m_0 = 1$ (normalized). We can see that the term 2^{-p} in expression (2) is the bias regarding the standard IEEE floating-point representation. Let us rewrite expression (2) as

$$M = M' + 2^{-p} \tag{3}$$

where M' is

$$M' = \left[\sum_{i=0}^{p-1} m_i \cdot 2^{-i}\right] \tag{4}$$

We can see that M' is the value of the significand in the standard representation so that the HUB significand is similar to that of the standard except that the HUB number has a bias of 2^{-p}.

From expression (2) we deduce that the form of the significand of a HUB number is

$$M = 1.m_1 m_2 \cdots m_{p-1} 1 \tag{5}$$

where the least significant bit (LSB) is always 1 due to the bias. This LSB is denoted ILSB (Implicit Least Significant Bit) since it is a constant and implicit when represented (similar to the implicit leading bit of the standard). Consequently, the HUB format has two implicit bits: the MSB and the LSB.

The set of the exactly represented numbers (ERN) for the standard representation and for its counterpart HUB one (that is, with the same precision) are disjoint. This is shown in Figure 1.a where we can see the relative position of consecutive ERNs for both the standard (white circle) and the HUB formats (black circle). We can observe that both representations keep the same distance between consecutive numbers (that is, the same precision) and that the distance between a standard and a HUB number is just half ulp. Moreover, the amount of ERNs for both representations is also the same.

Figure 1.b shows a simple example for $ulp = 2^{-3}$, where we can see that consecutive HUB numbers (5 bits, 1.000**1**, 1.001**1**, 1.010**1**, ...) have all of them their LSB=1 (the ILSB). In comparison with the standard (4 bits, 1.000, 1.001, 1.010, ...) shown in this figure, the HUB numbers have one extra bit (just the ILSB=1). This means that one extra bit is required to operate with HUB numbers in general (for some operations like addition with round to nearest this extra bit is compensated with the lack of the rounding bit for HUB approach which is not required to achieve round to nearest [3]). In short, each HUB number is just in the middle point of two standard numbers and both systems have the same precision.

Despite having an extra bit (the ILSB), this is not required for storing since it is implicit. Thus, for the same precision, both representations require the same storage. For example, the single-precision FP IEEE has 24 bits of precision (significand of 24 bits), 23 out of them are used for storage and 24 for operating. The counterpart HUB format for the same precision (24 bits) has 25 bits for operating and 23 bits for storage (that is, the same storage requirement [3]).

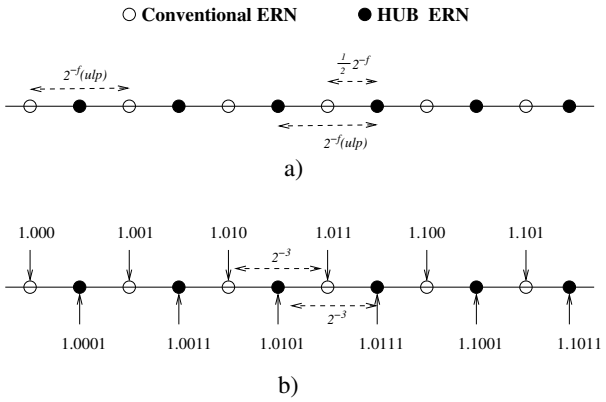


Fig. 1. ERNs for the conventional and the HUB systems, STEP=2^{-3}

Some of the main advantages of the HUB format are:

- Round to nearest mode is carried out by truncation. This is one of the most important features of the format. Let x denote a non HUB number with n significant bits ($p < n$). We want to obtain the nearest p-bit HUB number to x. The nearest HUB number is obtained by truncation of the p MSBs of x. Thus, the round to nearest mode is implemented by truncation of the original number (see [3]). Now we present a simple example:

```
x is a positive non HUB number (9 bits)
RN(x):Nearest HUB number, 4 bit precis.
x=1.0110 1000 -> RN(x)=1.01101
Operation:
x      -> 1.0110 0110
trunc. -> 1.0110
RN(x) ->  1.01101
```

- The two's complement of a HUB number is obtained by inversion of all bits of M', which is a bit-wise operation. Thus, carry propagation is prevented when obtaining the two's complement. This is due to the fact that the ILSB is 1. When obtaining the two's complement, first we invert all bits and then add 1 to the LSB position. In our case, after inverting the ILSB we have a 0, and the addition of 1 at this position changes it to 1 again with no carry propagation to the rest of the bits. Next example shows it:

```
x is a HUB number
x=01.0111 -> 2'c(x)=10.1001
2'C(x) operation:
invert ->    1 0.100 0
add 1      +         1
           -----------
             1 0.100 1
```

III. FMA ARCHITECTURE FOR CONVENTIONAL REPRESENTATION

The first widely accepted FMA that was implemented on an extended general-purpose processor was that of the IBM RS/6000 [12]. The corresponding algorithm and architecture for the standard representation are widely described and analyzed in [13] and [14]. In this section, we present a basic FMA architecture for conventional floating-point numbers based on the guideline presented in [13] and [14]. The design and implementation of this architecture are carried out for comparison purposes since it is the base of that proposed in Section IV for the HUB representation. We are aware that there are some improved FMA architectures in the literature, but in this work, we use a basic one to better illustrate the differences between the conventional and the HUB approaches.

Assume that we want to perform the operation $x \cdot y + z$ where x, y and z are conventional floating-point numbers. Consider that the floating-point numbers have p bit precision such that

$$\begin{aligned} x &= (-1)^{s_x} M_x 2^{e_x} \\ y &= (-1)^{s_y} M_y 2^{e_y} \\ z &= (-1)^{s_z} M_z 2^{e_z} \end{aligned} \tag{6}$$

where the significand M is normalized ($1 \leq M < 2$) and has p bits. The operation $x \cdot y + z$ requires that the product $x \cdot y$ and the addend z are aligned. From now on, "product" denotes $x \cdot y$ and "addend" denotes z. The alignment is performed based on the exponent difference as

$$d = e_z - (e_x + e_y) \tag{7}$$

The basic FMA architecture that we have implemented is presented in Figure 2 where the shaded elements highlight the main differences with the HUB design (explained in next section). Now we explain the different elements of the architecture as well as the operations that are carried out.

First, we have a multiplier that multiplies the significands of the operands x and y (p-bit each one), which is $M_x \cdot M_y$. This is implemented using partial product generation and a compression tree. The output ($2p$ bits) is obtained in carry-save representation, as shown in Figure 2.

In parallel with the multiplier, the shifter of Figure 2 aligns the addend. It operates on the significand of the addend M_z (in fact the product is never shifted; all the involved shifts are carried out over the addend). Now we analyze the different alignment that this shifter has to perform.

Figure 3 shows the different cases of alignment for $p = 5$ bits. On the one hand, when the dominant term is the addend, a left shift of the addend up to $p+3$ positions is possible. We can see that there is a maximum gap of two bits, namely the rounding bit (round-to-nearest is considered in this paper) and the guard bit (used when normalization involves a left shift of one position). Figure 3.a shows the maximum left shift when normalization of the result is not required. Figure 3.b shows the maximum left shift when the result needs a normalization (guard bit is needed). If the addend is placed more than $p+2$ positions to the right, the whole of the product is condensed in a sticky bit.

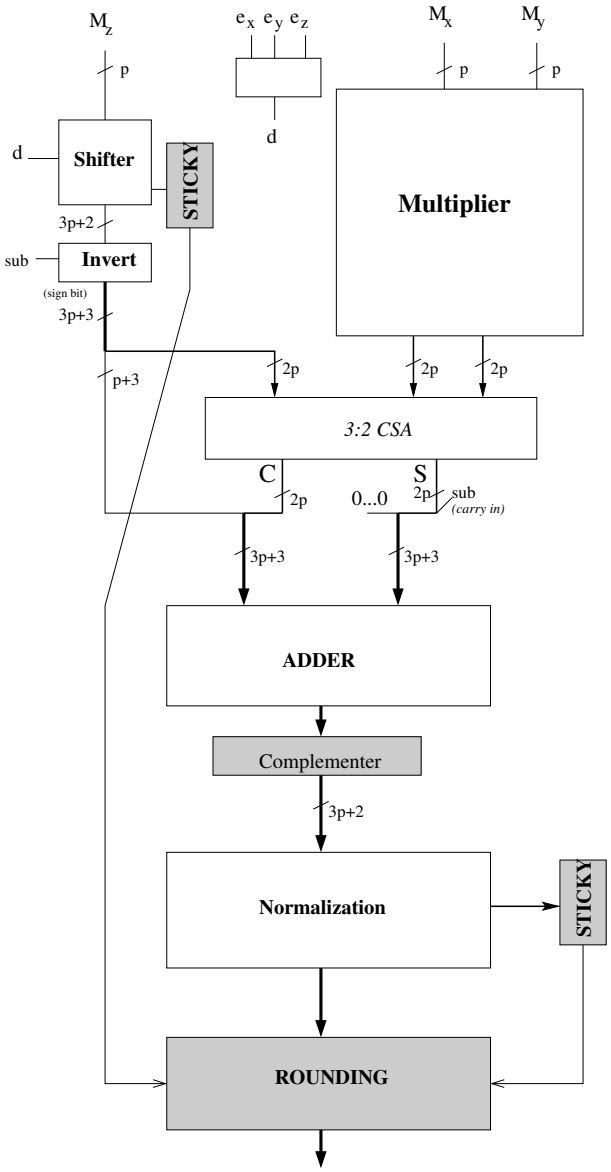


Fig. 2. FMA basic architecture

On the other hand, when the dominant term is the product, a right-shift of the addend up to $2p-1$ positions is involved, as shown in Figure 3.c. Notice that all bits beyond the bound of the product are condensed in the sticky bit, as shown in Figure 3.c. Thus, a shift of the addend up to $(p+3)+(2p-1) = 3p+2$ positions is possible, as shown in Figure 3.d. Normally, to avoid the complexity of the shifter with left/right shifts, a hardwired left shift of $p+3$ of the addend is carried out, and then the shifter operates only to the right.

After the shifter of Figure 2, an inverter operates if an effective subtraction is required, and the sign bit is incorporated to the bus ($p+3$ bits, as shown in Figure 2).

The carry and sum words of the output of the multiplier

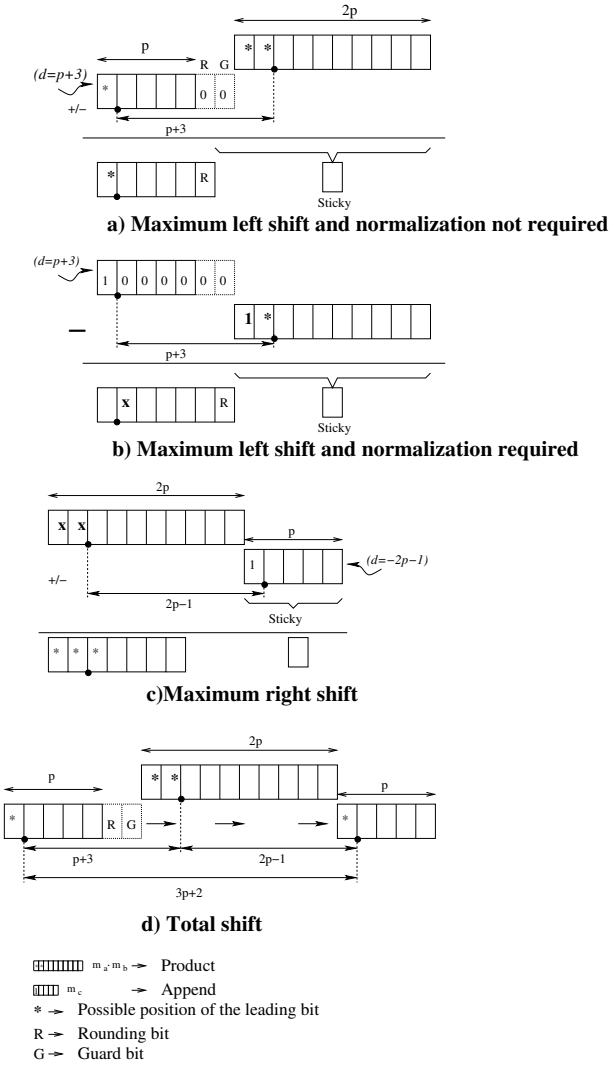


Fig. 3. Alignment for the conventional representation

are added with the $2p$ LSBs of the output of the conditional inverter to generate a new carry-save number, which is converter into conventional representation using a fast adder, as shown in Figure 2.

The carry word of the 3:2 CSA is extended with the p+3 MSBs of the output of the conditional inverter and the sum word of the 3:2 CSA is extended with 0s. The value of the carry-in depends on the effective operation to complete the two's complement conversion of the addend.

At the output of the adder, the complementer is in charge of obtaining the sign-magnitude version of the number in case of having a negative number. After this, the normalization is performed (and a possible sticky bit is generated). Finally, we have the rounding process, which may require an addition for rounding up. Moreover, in the worst case, this rounding up may require a renormalization and an update of the exponent.

IV. FMA ARCHITECTURE FOR HUB NUMBERS

Assume that we want to perform the operation $a \cdot b + c$ where a, b and c are floating-point HUB numbers. Let us consider that the floating-point HUB numbers have p-bit precision such that

$$\begin{aligned} a &= (-1)^{s_a} M_a 2^{e_a} = (-1)^{s_a}(M'_a + 2^{-p})2^{e_a} \\ b &= (-1)^{s_b} M_b 2^{e_b} = (-1)^{s_b}(M'_b + 2^{-p})2^{e_b} \\ c &= (-1)^{s_c} M_c 2^{e_c} = (-1)^{s_c}(M'_c + 2^{-p})2^{e_c} \end{aligned} \tag{8}$$

where the significand M is the normalized HUB significand with p bits of precision (see equations (2), (3) and (4)). Figure 4 shows the proposed FMA HUB architecture.

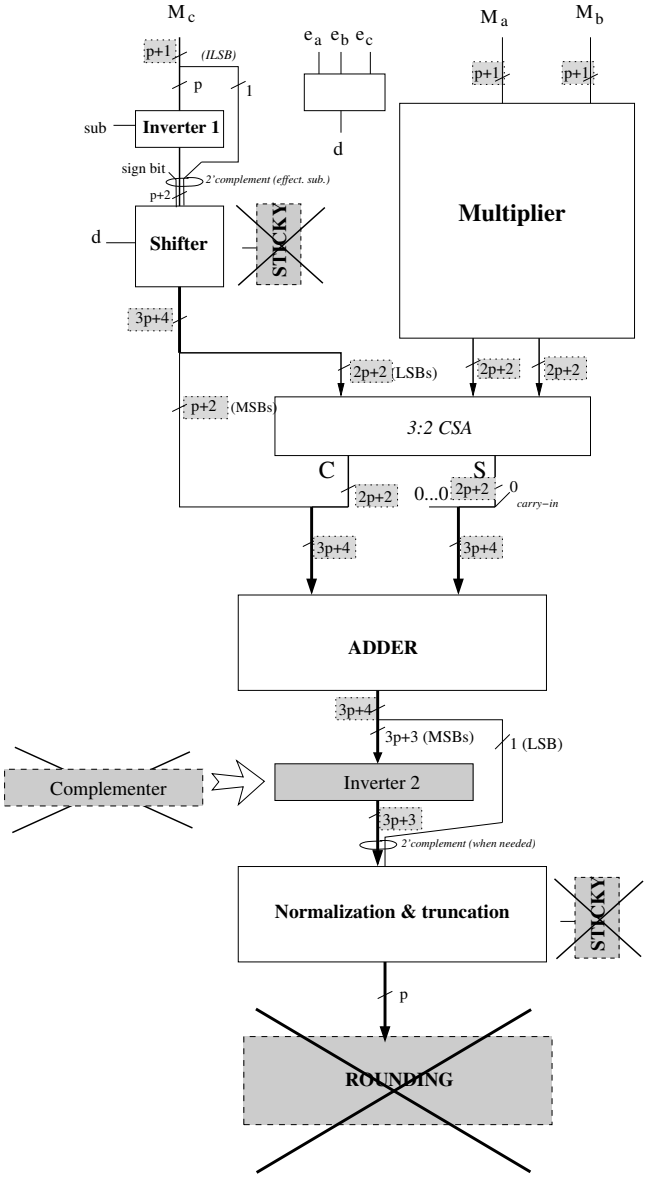


Fig. 4. FMA HUB architecture

In this section we explain the proposed design and compare it with the conventional architecture of Figure 2. In Figure 4 we have also pointed out in shadow the main hardware

elements that have been removed/modified from the classic design. The elements that have been removed are the sticky bit calculation, the rounding module and the complementer (the latter is substituted by an inverter). The elements that have been modified are related to the increase in the size of some buses (normally increased by one bit due to the ILSB). Let us note that to operate with a HUB number, generally, the ILSB is explicitly included which turns the HUB number into a conventional one [4]. The handling of the exponent is not drawn in the figure since it is similar to that of the conventional representation.

The multiplier of Figure 4 is similar to that of the conventional in Figure 2. However, in this case, the input operands have $p+1$ bits (the extra bit is necessary to accommodate the ILSB). Thus, for the same precision, this multiplier has one extra bit per input operand when compared with the conventional one. Nevertheless, the extra hardware required when designing the multiplier is somehow mitigated since both extra bits are always 1 (and thus the LSB of the product is always 1).

When an effective subtraction is required, the two's complement of the subtrahend (that is, the addend c) is performed simply by bit-wise inversion of M'_c, and the inverter 1 of Figure 4 is in charge of doing it. Another consequence of having HUB numbers is that the carry-in of the adder in Figure 4 is 0 since the two's complement operation is fully completed after the inverter (in an effective subtraction).

The shifter is in charge of the alignment of the addend. The sign bit and the ILSB are explicitly accommodated after the inverter, producing a $p+2$ bit input to the shifter, as shown in Figure 4.

Now we analyze the maximum shift of the addend. On the one hand, when the dominant term is the addend, a left shift of the addend up to $p+2$ positions is possible as shown in Figure 5.a (in the conventional one this distance is $p+3$). Unlike the conventional representation, in HUB representation no gap is required between the addend and the product for the maximum left shift (two-bit gap in the conventional one, the rounding and guard bits, see Figure 3.a). The rounding bit is not needed anymore since in HUB representation the round-to-nearest mode is performed by truncation. In the case of maximum left shift ($p+2$ bits) and under subtraction operation, the final result is always normalized (unlike the conventional, a normalization of one bit to the left is not required in HUB). Next, we prove this. Figure 3.b shows the only case in which the MSB of the result is fractional in the conventional representation (which involves a further normalization). The counterpart situation for HUB is shown in Figure 5.b, where we can see that a carry propagation in subtraction is absorbed by the ILSB of the addend and the result holds its MSB as an integer (normalization is not required and thus the guard bit is not needed).

In Figure 5.a, we can also see that all bits of the product placed to the right of the ILSB position of the addend are condensed in a sticky bit, which is always 1 since at least the LSB of the product is always 1 (notice that the multiplication of two HUB numbers produces a result whose LSB is 1). Thus, another important consequence of working with HUB representation is that the logic for calculation of the sticky bit is not needed anymore, as shown in Figure 4.

On the other hand, when the dominant term is the product, a right-shift of the addend up to $2p+1$ positions is involved (maximum right shift, see Figure 5.c). The bits of the addend placed beyond the LSB of the product are condensed into a sticky bit, which is always 1 due to the ILSB of the addend. Thus, as in the case of the left shift, the calculation of the sticky bit is not needed.

As consequence, the shifter has to be able to shift the addend up to $3p+3$ positions, as summarized in Figure 5.d.

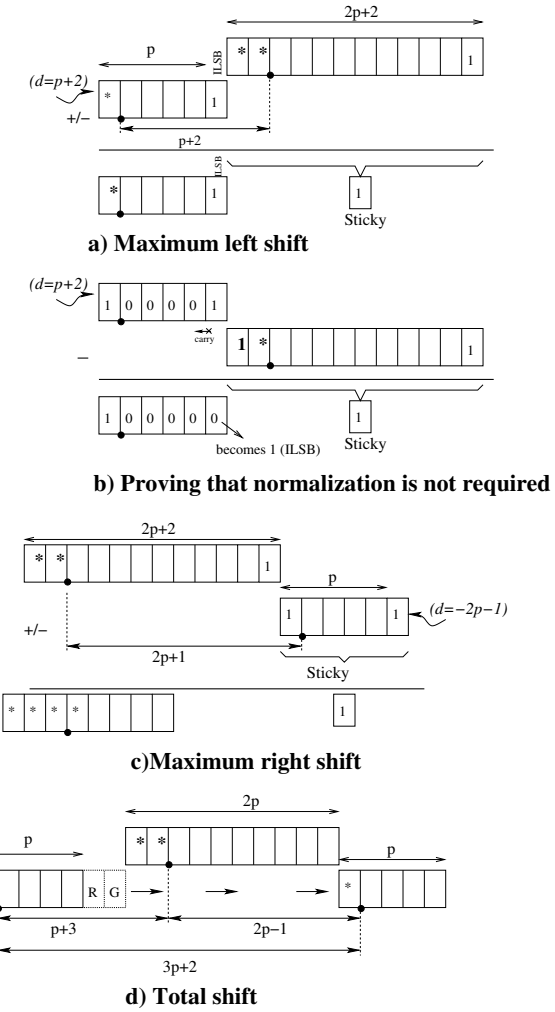


Fig. 5. Alignments under HUB representation

The 3-2 CSA and adder of the HUB architecture are wider than that of the conventional one due to the ILSB. In this case, it involves an increase in area and delay that is compensated

with the savings of area and delay of the removed elements, as proved in Section V.

Unlike in the conventional architecture, the conversion into sign-magnitude for a negative result at the output of the adder of Figure 4 is carried out simply by an inverter instead of a complementer. Moreover, the LSB of the adder is bypassed and connected to the LSB of the normalization module, as shown in Figure 4. Let us explain in detail this issue. First let us remind that the effective subtraction is always performed by adding the product, which is a positive number, plus the two's complement of the addend. For example, if ab is the product (ab is positive), and c is the addend (c is positive), and we want to calculate $R = c - ab$, the actual operation in the architecture is $R = -(ab - c)$ (the sign is managed in a separate logic). The value $(ab - c)$ is obtained at the output of the adder in Figure 4 (and also in the conventional one, see Figure 2). Since the result has to be delivered in sign-magnitude representation, the two's complement of the result R is required only if $c > ab$ since the actual operation in the adder is $(ab - c)$. Next example shows it (worst case):

```
Operation required R=c-ab
Adder actual operation: ab-c=ab+2'C(c)
ab= 0 1.110 1111
c = 0 1.111  -> 2'C(c) = 1 0.001
R = 1 0.000 0001 (sign-magnitude)
Operation:
                      s
       ab     ->     0 1.110 1111
       2'C(c) -> +   1 0.001
                    ---------------
Adder output          1 1.111 1111
Inverter input:       1 1.111 111
Inverter output:      0 0.000 000
Norm. module input: 0 0.000 0001
```

Figure 6 shows the snapshot of this example in the architecture of Figure 4. In this example $c > ab$ and the adder output $(ab - c)$ has a negative number (1 1.111 1111). The LSB of this amount is directly connected to the LSB of the input of the normalization module, and the rest of the bits are inverted and connected to the normalization module.

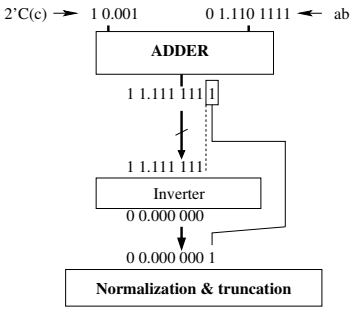


Fig. 6. Snapshot of the two's complement at the inverter 2 of Figure 4 for ab=01.110 1111 & c=01.111 (2'C(c)=10.001)

The normalization process is similar to the conventional one: locate the position of the most significant 1, update the exponent accordingly, and shift the value at the input to the right until reaching the first 1 (a leading zero anticipator could also be used).

Therefore, the result of the operation $r = a \cdot b + c$ is a HUB number such that

$$r = (-1)^{s_r} M_r 2^{e_r} \tag{9}$$

where the sign s_r depends on the MSB of the output of the adder and the sign of the three input operands e_a, e_b, e_c, the exponent e_r is obtained from the maximum of $\{e_a + e_b, e_c\}$ plus the correction of the normalization, and M_r is composed by the truncation of the p MSBs of the result after normalization plus the ILSB (see equation (2)).

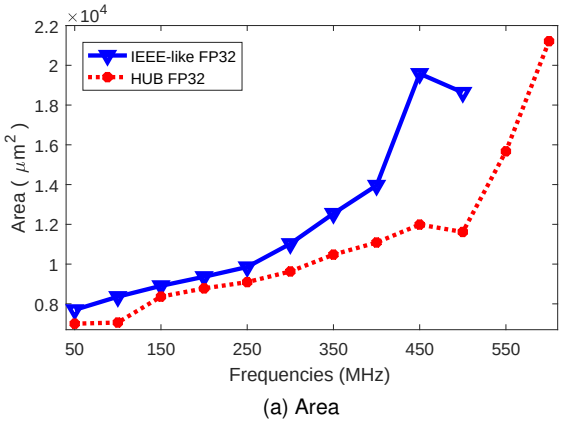
This result is already rounded since the nearest HUB number to a number is obtained by truncation of the p MSBs (for p-bit precision). Consequently, as shown in Figure 4, no special hardware is required for rounding (unlike in the conventional representation where a specific module is needed for rounding, see Figure 2). This is one of the most important advantages of working with HUB representation.

V. Implementation results and comparison

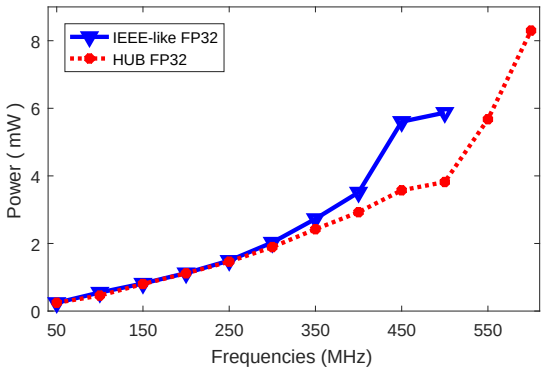
Both the proposed HUB floating-point FMA architecture, along with an equivalent architecture for conventional floating-point representation have been implemented using Verilog and Synopsys DesignWare building blocks version H-2013.03-DWBB_201303.2. These architectures only implement the basic FMA operation, i.e., they do not consider special cases (except zero) or exceptions, and denormals are flush to zero. Including these features would affect both implementations similarly, and thus, they do not contribute to the comparison. Moreover, recent architectures tailored for deep learning, such as in [1] and [2], avoid their implementation to reduce hardware cost. On the other hand, we should also note that the conventional architecture only implements rounding-to-nearest tie-to-even to provide a fair comparison with the HUB version.

To measure the benefit of using HUB floating-point format instead of the conventional one, both combinational architectures have been synthesized, and the results gathered for comparison. For synthesizing, Synopsys Design Compiler H-2013.03-SP2 and the TSMC 65nm library have been used. We set "typical-case" operating conditions in which the temperature is 25ºC and voltage Vdd = 1.0V. Both architectures have been synthesized for the same target clock frequencies, and the area and power consumption have been compared. Two different word-lengths have been analyzed: IEEE single-precision (FP32) and the new BFloat16 format (BF16) for both conventional and HUB formats. In the figures, IEEE-like FP32 and BF16 refer to conventional formats whereas HUB FP32 and HUB BF16 refer to their HUB counterparts.

Figure 7a and Figure 7b show the area and power results, respectively, for single-precision. First, we can see that the conventional architecture only reaches 500MHz, whereas the

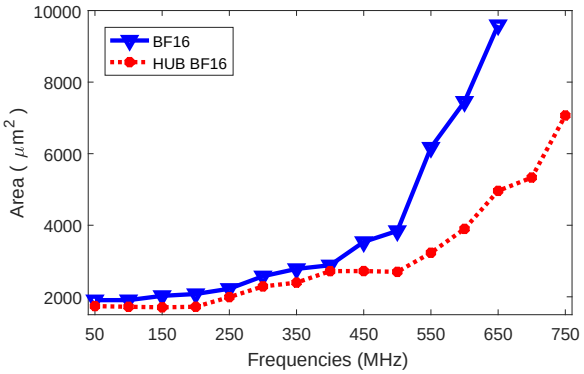


(a) Area

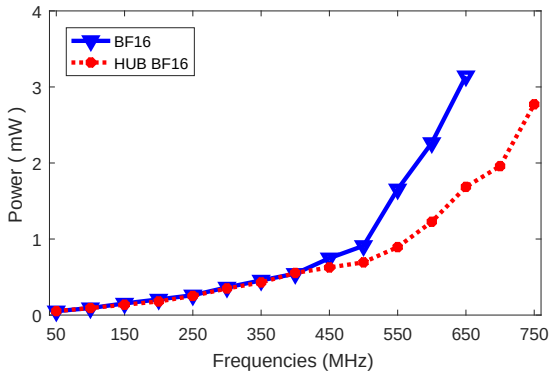


(b) Power consumption

Fig. 7. HUB and conventional FMA results for single-precision.



(a) Area



(b) Power consumption

Fig. 8. HUB and conventional FMA results for Bfloat16.

HUB one reaches 600MHz. That means a 20% of speedup. This improvement is mostly because the HUB approach does not require the rounding logic, which is in the critical path. In Figure 7a, it is also observed that HUB architecture always requires less area. This fact is much more evident for clock frequencies above 350MHz. Regarding the power consumption, the reduction achieved by the HUB approach is only clearly seen for clock frequencies above 350MHz. For lower frequencies, the power reduction varies ranging from 0% to 16% (see Figure 9b).

On the other hand, Figure 8a and Figure 8b show the area and power results, respectively, corresponding to BFloat16 for both conventional and HUB architectures. The advantage of using the HUB approach for BFloat16 format is even greater than for single-precision. Again, the HUB architecture reaches higher clock frequencies and a significant area and power reduction, especially for high frequencies. Even for 750MHz the area and power of HUB architecture are significantly lower than the one for conventional numbers for 650MHz. Specifically, HUB architecture reaches a 15% higher clock frequency and, even then, the HUB architecture uses 26% less area, and 12% less power, than the conventional one. That means that the HUB approach for BFloat16 reduces simultaneously area, power, and delay.

Although the benefits of the HUB format can be observed from Figure 7 and Figure 8, Figure 9 shows more clearly which percentage of area and power reduction is achieved by the HUB approach for both sizes. For BFloat16, the area reduction when using the HUB format varies around 5%-15% for frequencies lower than 400MHz, whereas it goes up when the clock frequency increases reaching almost a 50% of reduction. Similarly, the power reduction varies around 0%-5% for frequencies below 400MHz, and it reaches almost 50% above 500MHz. For single-precision, the area reduction is around 10% for frequencies below 300MHz but for higher frequencies goes up to 38% of reduction. Regarding power reduction for single-precision is only noticeable for high frequencies, achieving up to 35%. As mentioned before these reductions are mainly due to the fact that no rounding logic is required for the HUB approach. As the rounding logic is in the critical path in conventional architectures, the higher the frequencies, the more hardware needed to reach such frequencies and hence the greater reduction percentage compared to HUB architectures.

VI. Conclusions and Future Work

FMA operation is present in the instruction set architecture of most modern processors. In current demanding trending applications like Neural Network, this operation is intensively used and new representation number formats are being proposed to improve the performance. Following this trend, we

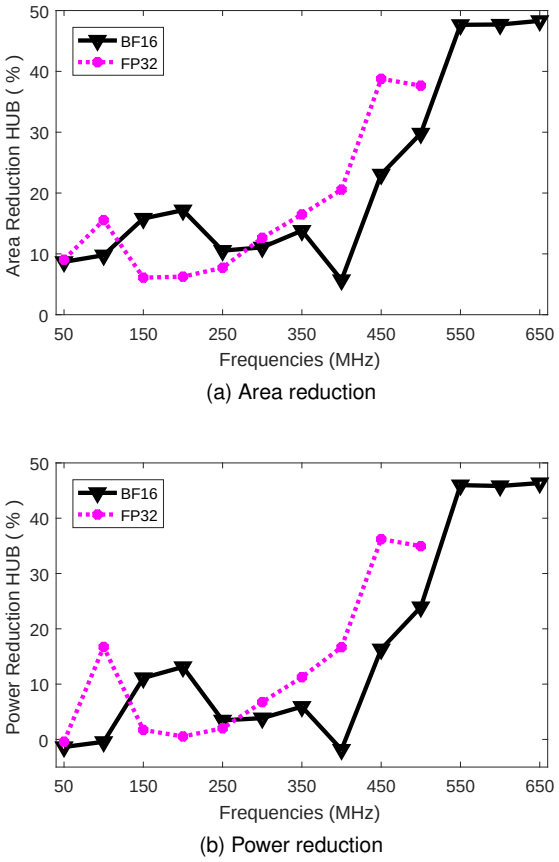


Fig. 9. Reduction results for Bfloat16 and FP32 formats.

propose an architecture to deal with the FMA operation under the HUB format. In this paper, we design a HUB FMA architecture and its conventional counterpart and compare them. The features of the HUB format lead to achieving important improvements in delay, area, and power for the proposed architecture when compared with the conventional one for the same precision. The implementation results show that for single-precision the HUB FMA reduces area and power up to 38% and 35%, respectively, and for Bfloat16 the saving is even greater being of up to 48% and 46% for area and power, respectively. In addition, for BFloat16, the HUB FMA increases the speed by a 15%, and even then, reduces the area and power by 26% and 12%, respectively.

In future works, we will study the use of HUB format in other FMA designs including FMA units that multiply two Bfloat numbers and accumulate a single-precision number, an operation that is present in many deep learning accelerators.

REFERENCES

[1] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, June 2019, pp. 88–91.

[2] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "Dlfloat: A 16-b floating point format designed for deep learning training and inference," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, June 2019, pp. 92–95.

[3] J. Hormigo and J. Villalba, "New Formats for Computing with Real-Numbers under Round-to-Nearest," *Computers, IEEE Transactions on*, vol. 65, no. 7, pp. 2158 – 2168, 2016.

[4] ——, "Measuring Improvement When Using HUB Formats to Implement Floating-Point Systems under Round-to-Nearest," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 6, pp. 2369–2377, 2016.

[5] J. Villalba-Moreno, "Digit Recurrence Floating-point Division under HUB Format," *23rd IEEE Symposium on Computer Arithmetic, Silicom Valley (California, USA)*, July 2016.

[6] J. Villalba-Moreno and J. Hormigo, "Floating Point Square Root under HUB Format," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 447–454.

[7] H. Zhang, D. Chen, and S. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035–1048, July 2019.

[8] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener, "P6 binary floating-point unit," in *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, 2007, pp. 77–86.

[9] M. Boersma, M. Kroner, C. Layer, P. Leber, S. M. Muller, and K. Schelm, "The power7 binary floating-point unit," in *2011 IEEE 20th Symposium on Computer Arithmetic*, July 2011, pp. 87–91.

[10] S. Boldo and J. Muller, "Exact and approximated error of the fma," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 157–164, Feb 2011.

[11] P. W. C. E. Hokenek, R. Montoye, "Second–Generation RISC Floating Point with Multiply–Add Fused," *IEEE J. Solid–State Circuits. Vol. 25*, no. 5, pp. 1207—-1213, 1990.

[12] R. K. Montoye, E. Hokonek, and S. L. Runyan, "Design of the floating-point execution unit of the IBM risc system/6000," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59–70, 1990.

[13] J. Muller, N. Brisebarre, F. Dinechin, C. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehele, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser, 2010.

[14] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, San Francisco, 2004.