New Flexible Multiple-Precision Multiply-Accumulate Unit for Deep Neural Network Training and Inference

Hao Zhang, *Student Member, IEEE*, Dongdong Chen, *Member, IEEE*, and Seok-Bum Ko, *Senior Member, IEEE*

Abstract—In this paper, a new flexible multiple-precision multiply-accumulate (MAC) unit is proposed for deep neural network training and inference. The proposed MAC unit supports both fixed-point operations and floating-point operations. For floating-point format, the proposed unit supports one 16-bit MAC operation or sum of two 8-bit multiplications plus a 16-bit addend. To make the proposed MAC unit more versatile, the bit-width of exponent and mantissa can be flexibly exchanged. By setting the bit-width of exponent to zero, the proposed MAC unit also supports fixed-point operations. For fixed-point format, the proposed unit supports one 16-bit MAC or sum of two 8-bit multiplications plus a 16-bit addend. Moreover, the proposed unit can be further divided to support sum of four 4-bit multiplications plus a 16-bit addend. At the lowest precision, the proposed MAC unit supports accumulating of eight 1-bit logic AND operations to enable the support of binary neural networks. Compared to the standard 16-bit half-precision MAC unit, the proposed MAC unit provides more flexibility with only 21.8 percent area overhead. Compared to a standard 32-bit single-precision MAC unit, the proposed MAC unit requires much less hardware cost but still provides 8-bit exponent in the numerical format to maintain large dynamic range for deep learning computing.

Index Terms—Multiply-accumulate unit, multiple-precision arithmetic, flexible precision arithmetic, deep neural network computing, computer arithmetic

1 Introduction

Deep learning [1] has achieved great success in recent years. In many applications, deep learning can achieve a performance that is near to or even better than human level. Although deep learning is powerful, the cost to implement a deep learning model is very expensive [2], especially in terms of computational intensity. In recent years, many research works have been done on efficient implementation of deep learning algorithms on hardware. Among these works, the numerical format required by deep learning training and inference are extensively investigated.

Currently, most of the deep learning training jobs are done using graphic processing units (GPUs) with 32-bit single-precision floating-point [3] operations. However, the data-path of single-precision floating-point units is complex and the hardware cost of implementing single-precision units are expensive. These lead to a high energy consumption and a large latency when implementing deep neural networks in customized hardware. In order to reduce the hardware cost, in recent years, many research works [4], [5], [6], [7] are focused on reducing the numerical precision required by deep neural network training. In [4], the authors proposed to train deep neural network with 16-bit half-precision number format. The 12-bit floating-point format is utilized in training in [5]. In [6] and [7], 8-bit floating-point format is used. The standard single-precision, double-precision, and quadruple-precision are well supported in many arithmetic unit designs in the literature [8], [9], [10], [11] and in many commercial products [12], [13]. However, there are not many works discussing the support of 16-bit or even lower precision floating-point operations. Due to the small bit-width of 16-bit or even lower precision floating-point formats, the implementation of arithmetic units based on half-precision and even lower precision floating-point formats is much more efficient than other floating-point formats [14]. As the interest of using low precision floating-point formats in deep neural networks rises, the support of low precision in arithmetic units is required to be investigated.

Using low precision floating-point during the deep neural network training is expected to reduce the energy consumption compared to using the standard single-precision floating-point [15]. However, further energy reduction can be achieved when each operation step can use its minimum required precision instead of being forced to use a uniform precision for all steps. This idea is feasible for deep neural network implementation since the minimum required numerical precision to maintain accuracy is different for

- *H. Zhang and S.-B. Ko are with the Department of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, Saskatchewan S7N 5A2, Canada. E-mail: {hao.zhang, seokbum.ko}@usask.ca.*
- *D. Chen is with Intel Corporation, San Jose, CA 95134. E-mail: doc220@mail.usask.ca.*

Manuscript received 8 Nov. 2018; revised 9 July 2019; accepted 14 Aug. 2019. Date of publication 5 Sept. 2019; date of current version 19 Dec. 2019. (Corresponding author: Seok-Bum Ko.) Recommended for acceptance by V. Piuri. Digital Object Identifier no. 10.1109/TC.2019.2936192

different deep neural networks [7], [16], [17]. Furthermore, even within the same deep neural network, different layers have different tolerance to the reduced numerical precision [7], [17]. Therefore, a flexible precision arithmetic unit is desired to further reduce energy consumption and to improve the performance of deep neural network operation.

For deep neural network computing, the dynamic range of a floating-point format (bit-width of exponent, BW_e) is more important than the precision (bit-width of mantissa, BW_m) [16]. With enough BW_e, the neural network can achieve satisfying accuracy and the accuracy of network does not have significant difference with various BW_m. The BFloat16 format introduced in Tensorflow [18] directly truncates the mantissa of single-precision numbers from 23-bit to 7-bit while reserves the 8-bit exponent. However, if the BW_e is not enough, the accuracy of the neural network will have a significant degradation. According to this feature, a flexible precision format can be achieved by a method where the total bit-width of a number is a constant but BW_e and BW_m can be mutually exchanged. In this method, the requirement of the representation range of the numerical format is first met and the remaining bits are allocated to mantissa. When the required data range is large, BW_e can be increased, so that the flexible format can still represent the data properly for deep neural network applications. When data range is small, BW_e can be reduced and thus the flexible format can represent the data more precisely with larger BW_m.

In addition to floating-point numbers, fixed-point numbers are often used for deep neural network inference. The works in [6] and [19] show that inference can be accomplished with 8-bit fixed-point numbers. The 16-bit fixed-point format is used by some deep neural network accelerators [20], [21]. Moreover, for more efficiency, binarized neural network [22] is proposed where neural network parameters are constrained to ± 1. For a versatile deep neural network processor, these fixed-point operations and binary operations are required to be supported.

In the literature, the Flexpoint [23], a software controlled flexible dynamic fixed-point format is proposed. In this format, the shared exponent can be dynamically changed to meet the dynamic range requirement of the deep learning computing. In addition, the flexible floating-point precision method has been applied in a recent work [24]. In [24], a tunable precision floating-point multiplier which supports 5 to 8-bit exponent and 4 to 24-bit mantissa is proposed. Their results show significant improvement in energy consumption. However, for deep learning applications, some improvements can be applied. First, as discussed in [7], [16], [17], deep neural networks might not need large BW_m when floating-point format is used. Second, between floating-point format and fixed-point format, the latter one is desired when performing inference. Although recent works [25], [26] show good inference results with logarithmic number system, conventional arithmetic unit is still more popular in many different hardware designs. Third, parallel operations can be supported to compensate for throughput degradation in lower precision operations of the conventional arithmetic unit. Last but not the least, fused operation units, such as multiply-accumulate (MAC) [27], [28] and dot-product (DOT), are preferred compared to separate multiplier and adder due to smaller area and better accuracy.

With all these requirements in consideration, in this paper, a new flexible multiple-precision multiply-accumulate unit is proposed. The proposed MAC unit supports both floating-point operations (for deep neural network training) and fixed-point operations (for deep neural network inference). For floating-point format, the proposed unit supports one 16-bit MAC operation (FLP16-MAC) or sum of two 8-bit multiplications plus a 16-bit addend (FLP8-DOT2). The bit-width of exponent and mantissa can be mutually exchanged to realize the flexible precision support. For 16-bit floating-point format, up to 8-bit exponents are supported, as 8-bit exponent, the exponent bit-width of standard single-precision format, can already represent nearly all neural network parameters. In 8-bit mode, the dot-product operation $A_1 \times B_1 + A_2 \times B_2 + C$ is supported where products of two parallel 8-bit multiplications can be added together and then accumulated to a 16-bit floating-point addend C. The BW_e and BW_m of the 16-bit addend can also be flexibly defined. For fixed-point format, the proposed unit supports one 16-bit MAC operation (FIX16-MAC), or sum of two 8-bit multiplications plus a 16-bit addend (FIX8-DOT2), or sum of four 4-bit multiplications plus a 16-bit addend (FIX4-DOT4). For fixed-point format, the location of the radix-point can be flexibly defined. In other word, the bit-width of integer part and fractional part can be exchanged. Furthermore, binary neural network operations are supported. The major contributions of this paper are summarized as follows:

- Propose the architecture of flexible multiple-precision multiply-accumulate unit.
  - Propose the method to correctly extract each component of all supported precisions.
  - Propose the method to correctly align the operands under all supported precisions for addition.
  - Propose the method to correctly round the results of all supported precisions, where roundTiesToEven [3] is supported.
  - Propose the method to support subnormal numbers of all supported precisions.
- A comprehensive analysis of the implementation results is performed. Compared to the standard floating-point unit, the proposed unit supports many more functions with only minor resource overhead.
- A case study of a simplified deep neural network application is performed with the proposed multiply-accumulate unit to show the power efficiency of the proposed unit.
- The proposed multiply-accumulate unit can be used in deep learning processors in datacenters or used as an neural network intellectual property (IP) core for FPGA devices.

The rest of the paper is organized as follows: Section 2 presents the background information including the numerical formats that are supported in the proposed MAC unit and the architecture of conventional floating-point MAC unit. The design details of the proposed MAC unit is presented in Section 3. The synthesis results and their comparison with several standard arithmetic units are presented in Section 4. Section 5 presents two case studies to show the advantage of

TABLE 1
Supported Formats of the Proposed MAC Unit

Operations	MAC Operands				
	A and B			C	
	total	exponent	parallelism	total	exponent
FLP8-DOT2 ($\sum_{i=1}^{2} A_iB_i + C$)	8-bit	1∼6-bit	2	16-bit	1∼8-bit
FLP16-MAC ($AB + C$)	16-bit	1∼8-bit	1	16-bit	1∼8-bit
	total	fraction	parallelism	total	fraction
FIX4-DOT4 ($\sum_{i=1}^{4} A_iB_i + C$)	4-bit	0∼4-bit	4	16-bit	0∼15-bit
FIX8-DOT2 ($\sum_{i=1}^{2} A_iB_i + C$)	8-bit	0∼7-bit	2	16-bit	0∼15-bit
FIX16-MAC ($AB + C$)	16-bit	0∼15-bit	1	16-bit	0∼15-bit

the proposed MAC unit in deep neural network applications. Finally, Section 6 concludes the whole paper.

2 Background

2.1 Numerical Format

Standard floating-point formats, including half-precision, single-precision, double-precision, and quadruple-precision, are defined in IEEE754-2008 standard [3]. These formats are composed of 1-bit sign (S), m-bit exponent (E), and n-bit mantissa (M). For half, single, double, and quadruple precisions, m (n) equals to 5 (10), 8 (23), 11 (52), 15 (112), respectively. There is always an implicit bit im for the mantissa part. For normal numbers, $im = 1$. For zero and subnormal numbers, $im = 0$. The numerical value the IEEE 754 format represents is:

$$f_p = (-1)^S \times (im + 2^{-n} \times M) \times 2^{E-bias}, \quad (1)$$

where $bias = 2^{m-1} - 1$.

In the proposed MAC design, although the bit-width of exponent and mantissa are flexible, all the supported floating-point formats follow the IEEE 754 rule where there is always an implicit bit and the exponent part is biased.

When the exponent of a normalized number (with implicit bit equal to 1) is smaller than the minimum exponent of the corresponding format, the mantissa needs to be right shifted to bring the exponent back to the allowed range. If the difference between the exponent and the minimum exponent is smaller than the bit-width of the mantissa, this number is still representable and is called subnormal number. Otherwise, when using the IEEE 754-2008 default roundTiesToEven rounding mode, if the number is too small to be represented after rounding, it will be flushed to zero.

Fixed-point format is less complicated compared to the floating-point format. In the fixed-point format, numbers are encoded in two's complement format. They have p-bit integer and q-bit fraction. In the proposed design, p and q can be flexibly exchanged for different choices.

The numerical formats of each operand supported by the proposed multiply-accumulate unit in each operational mode are summarized in Table 1.

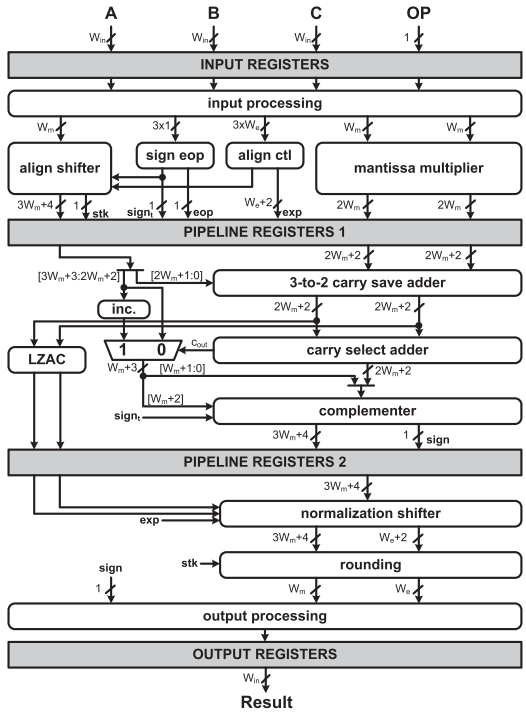

Fig. 1. Datapath of conventional multiply-accumulate unit based on standard floating-point format.

2.2 Floating-Point MAC Unit

The datapath of conventional multiply-accumulate unit based on standard floating-point format is shown in Fig. 1, where W_{in} represents the total bit-width of one operand, W_m represents the bit-width of mantissa including the implicit bit, and W_e represents the bit-width of exponent. It can perform the operation of $A \times B \pm C$ where addition or subtraction is defined by OP.

The whole datapath is usually divided into three pipeline stages to improve the throughput. The first pipeline stage contains the input processing module where each component of the floating-point format is extracted and evaluated. Then the mantissa multiplier is implemented to multiply the mantissa of A and B. In parallel to the mantissa multiplier, the alignment of C with $A \times B$ is performed with an alignment shifter. In the second pipeline stage, the aligned C is combined with the carry save format product of $A \times B$ through one-level of (3, 2) carry save adder. The resulting vectors are added with carry propagate adder. In parallel to the final adder, the leading zero anticipator and counting (LZAC) unit is implemented to predict the normalization shifting amount. In the last pipeline stage, normalization shifting and then rounding are performed. Then the exceptional cases handling of output is performed and the final result is generated.

In order to evaluate the resource overhead introduced by the flexible precision support in the proposed architecture, a standard half-precision MAC unit and a standard single-precision MAC unit are designed and implemented based on the architecture shown in Fig. 1. The standard half-precision MAC unit is used in the comparison because the proposed flexible MAC unit is also designed based on 16-bit format. The standard 32-bit single-precision MAC unit is compared in terms of functionality. Both the proposed design and the

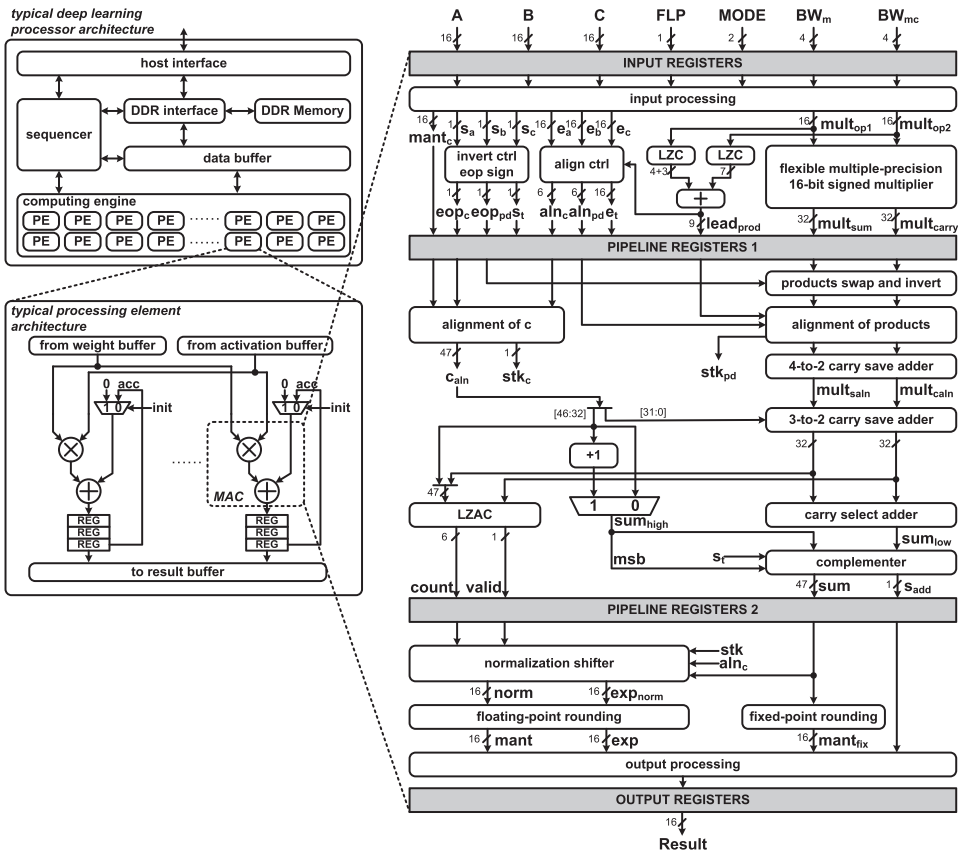

Fig. 2. Datapath of the proposed flexible multiple-precision multiply-accumulate unit and its usage in deep learning processor.

standard single-precision design can provide 8-bit exponent to meet the dynamic range requirement of deep learning applications. The total resource consumption of these two units can be compared. The details of results comparison and analysis are present in Section 4.

3 The Proposed Flexible Multiple Precision Multiply Accumulate Unit

A typical deep learning processor architecture [29], [30], [31] contains host interface, DDR memory, on-chip buffer, sequencer, and processing elements (PE), as shown in Fig. 2. The host interface is used to communicate with the host processors. It receives instructions and input data from host and sends results back to host. The sequencer receives the instructions and coordinates the operations of all other components. The DDR memory is used to save the data received from host and the processing results from PEs. The data to be processed in the current operation is buffered in on-chip buffer. All the computations are performed in PEs.

Each PE unit contains many multiply-accumulate units and registers to perform dot-product operations or matrix multiplication operations for convolution layer and fully-connected layer computations. Two multiplier inputs of the MAC unit are from activation buffer and weight buffer, respectively. The addend input is from the accumulation registers. At the initial cycle of a new operation, the addend is set to zero.

The core of the PE unit is the MAC unit. The functionality of the MAC unit can determine the functionality of the PE unit. On one hand, if the MAC unit only supports one precision format, then all the computations are forced to use the same precision format. Although the operations of other precisions can be achieved by software implementation, however, multiple iterations are usually required for that kind of operations and thus the performance will be degraded. On the other hand, if the MAC unit is versatile, then operations of various precisions can be supported in hardware which will benefit the performance. In addition, each application can choose to use their minimum required precisions so that the energy consumption will be reduced compared to the case of using one precision for all applications. In some cases, parallel operations can help improve the throughput and power efficiency. Based on these considerations, this paper aims to propose a MAC architecture with more functionality. It is designed to support various precisions at runtime. To make the proposed MAC architecture efficient, resource sharing among different precisions are extensively investigated to maintain the area overhead as small as possible compared to a single mode MAC unit.

The datapath of the proposed MAC unit is shown in Fig. 2. The whole design is divided into three pipeline stages. The first pipeline stage contains the input processing module, where each component of the operands are extracted, the flexible mantissa multiplier, and the alignment and invert control modules. The second pipeline stage contains the alignment shifter for input C and products and then the carry select adder and in parallel the leading zero anticipator and counting (LZAC) logic. The third pipeline stage contains normalization shifter and rounding modules.

The pipeline allocation of the proposed design is different from the typical MAC or fused multiply-add (FMA)

TABLE 2
Synthesis Results of Each Pipeline Stage of the Proposed Design

Pipeline Stage	Delay (ns)	Area (μm^2)	Power (mW)
Multiply and Control	0.21	1177	0.69
Align and Add	0.23	1356	0.60
Normalize and Round	0.19	319	0.18

design, as shown in Fig. 1, where the alignment of C is usually running in parallel with the mantissa multiplier in the same pipeline stage. In the proposed design, due to the support of flexible precision, the alignment control module will take larger delay than that of a standard design. In this case, if alignment shifter of C still runs in the same pipeline stage, although the delay of an alignment shifter is small, the critical path delay of the first pipeline stage will be significantly larger than the following two pipeline stages. This can lead to an imbalanced pipeline allocation which may increase the total latency of a single operation. Through our preliminary experiment, we found that the second pipeline stage has smaller delay than the other two. In addition, the alignment shifter for the product has to be put in the second pipeline stage (products are not available until the multiplication finishes). Therefore, we move the alignment shifter of C to the second pipeline stage. This makes the three pipeline stages balanced. This can be verified through the synthesis results shown in Table 2.

There are seven input signals to the proposed MAC unit. Three operands, A, B, and C, are of 16-bit for each. The 1-bit FLP is used to control weather the proposed MAC unit works in floating-point mode ($FLP = 1$) or fixed-point mode ($FLP = 0$). The 2-bit $MODE$ signal controls the precision mode of the proposed unit, where $MODE = 00$ represents binary mode, $MODE = 01$ represents 4-bit mode, $MODE = 10$ represents 8-bit mode, and $MODE = 11$ represents 16-bit mode. The 4-bit BW_m represents the bit-width of mantissa (in floating-point mode) or fraction (in fixed-point mode) in the operands A or B. The other 4-bit signal BW_{mc} has similar functionality but is used for operand C.

For the following subsections, the design details of each of the modules will be discussed. Emphasis will be put on design issues of flexible precision arithmetic unit, which include (1) operands extraction; (2) flexible multiplier; (3) flexible alignment control; (4) flexible rounding scheme; and (5) flexible subnormal handling.

3.1 Input Processing

For fixed-point operands, although the bit-width of integer part and fraction part might vary, these two parts can be used as a whole and directly sent to the multiplier. For floating-point operands, however, the exponent part and mantissa part must be correctly divided. In addition, the implicit bit must be correctly prefixed to the mantissa.

To extract mantissa, a mask signal mm (mantissa mask) is generated from the input BW_m and applied to the input operands. For 8-bit floating-point, the bit-width of mantissa may vary from 1 to 6. Therefore, the least significant 3-bit of BW_m are used to represent the number of mantissa for two sets of 8-bit operands. Two sets of 8-bit masks, mm_hi and mm_lo will be generated for two sets of 8-bit operands.

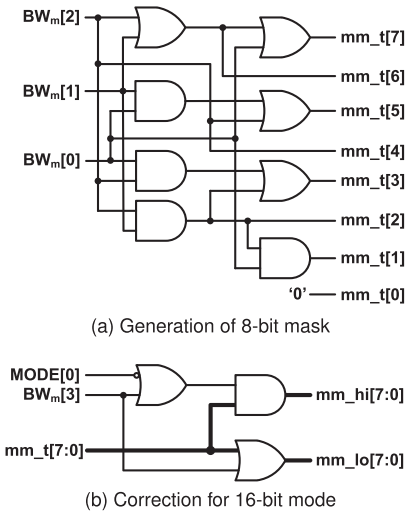

Fig. 3. Circuit to generate the mantissa mask mm_hi and mm_lo.

In 16-bit floating-point mode, the bit-width of mantissa will be between 7 and 14, the whole 4-bit BW_m are required to represent the number of mantissa in operands. The two masks, mm_hi and mm_lo are reused in 16-bit mode and they are combined to represent the mantissa mask for 16-bit operands. Still, the lower 3-bit of BW_m are used to generate mm_hi and mm_lo, however, for 16-bit operations, one more step is required. On one hand, if $BW_m[3] = 0$, then mantissa bits are all in the least significant 8-bit of the operand, and thus mm_hi is set to all zeros while mm_lo is not changed. On the other hand, if $BW_m[3] = 1$, then the bit-width of the mantissa is no less than 8-bit, and thus mm_lo is set to all ones, representing the whole least significant 8-bit contains mantissa bits, while mm_hi is not changed.

The circuit diagram of this process is shown in Fig. 3. mm_t is the 8-bit mask generated using the least significant 3-bit of BW_m. This mm_t is further processed by the circuit shown in Fig. 3b to generate the mantissa masks mm_hi and mm_lo for 8-bit mode and 16-bit mode. The mantissa mask is only used in floating-point mode. To distinguish 8-bit and 16-bit floating-point mode, the least significant bit (LSB) of $MODE$ can be used. In 8-bit floating-point mode, $MODE[0] = 0$ and $BW_m[3] = 0$, therefore, mm_hi and mm_lo are the same as mm_t. In 16-bit floating-point mode, $MODE[0] = 1$. If $BW_m[3] = 0$, mm_lo will be the same as mm_t and mm_hi will be set to all zeros. If $BW_m[3] = 1$, mm_lo will be set to all ones and mm_hi will be the same as mm_t.

The exponent mask em can be generated by inverting the mantissa mask and then set the most significant bit (MSB) of the generated vector to zero because the MSB is always sign bit. For 8-bit mode, bit 7 which is the sign bit of lower 8-bit number is also set to zero.

These two masks em and mm will be applied to input operands by performing an AND operation to extract exponent and mantissa. To obtain the exponent value, the extracted exponent needs to be right shifted with an amount of BW_m. The extracted exponent will be used to determine the implicit bit. To add implicit bit to the mantissa, the mm will be used again. By adding 1 to mm (generating mm_pone), there will be a 1 generated in the position of implicit bit and leaving all other position as zeros. Then if

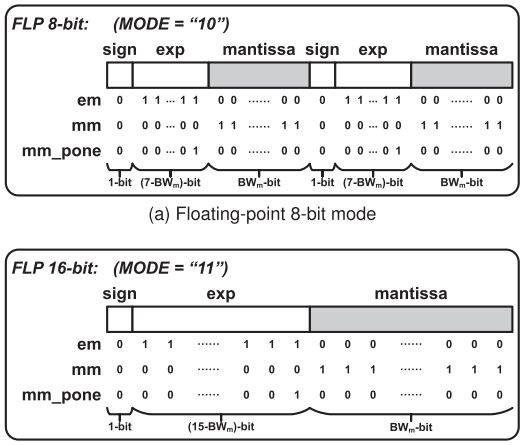

Fig. 4. Format of mantissa mask mm, exponent mask em, and implicit bit mask mm_pone.

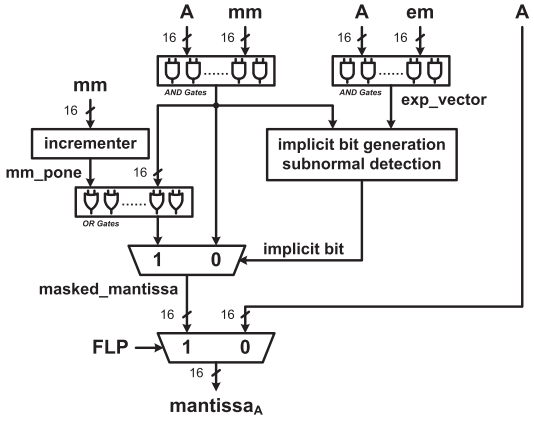

Fig. 5. Diagram of input processing for A operand.

the implicit bit is 1, mm_pone and generated mantissa vector can be ORed together to add implicit bit into mantissa. The format of these three masks are graphically shown in Fig. 4. For operand C, the signal BW_{mc} is used to perform exponent and mantissa extraction and the process is the same as A and B. The overall diagram of input processing for operand A is shown in Fig. 5. The same circuit can be applied to operands B and C.

3.2 Flexible Multiple-Precision Multiplier

For fixed-point operands, a signed multiplier is required. For floating-point operands, an unsigned multiplier is required. However, as the bit-width of floating-point mantissa is always smaller than the bit-width of fixed-point number (there is always at least 1-bit exponent for floating-point format), the mantissa can always be sign extended with zeros and converted to a signed positive number. Therefore, in a uniformed multiplier design, a signed multiplier is implemented.

In the proposed design, in order to reduce the cost of 16-bit multiplication, the radix-4 modified Booth multiplier [32] is applied. The multiplicand, multiplier, and the generated partial product (pp) array of the proposed flexible multiplier are shown in Fig. 6. The partial products are generated with the method proposed in [33]. For each precision mode, the generated partial product array is shown in Fig. 7. In Fig. 7, S represents the sign of the corresponding partial product and E is the extended bit to the partial product. According to [33], $E = 1$ when the multiplicand and the partial product have the same sign or partial product is $+0$. Otherwise, when the multiplicand and the partial product have opposite signs or partial product is -0, $E = 0$.

In 16-bit mode, the whole multiplicand and the multiplier are used to generate partial products. According to the radix-4 Booth multiplier algorithm, the multiplier is always padded with 1-bit zero after the LSB. Then the resulting 17-bit vector is divided into eight 3-bit groups as shown in Fig. 6. Each bit group is used to generate one partial product. The generated partial product array in 16-bit mode is shown in Fig. 7a

In order to support parallel multiplications in 8-bit and 4-bit mode, the partial product array is divided into multiple regions as shown in Fig. 6. In 8-bit mode, two parallel multiplications, MUL8-1 and MUL8-2, are supported. The partial products of MUL8-1 are in row_r1 and row_r2 in row direction, and col_r1 and col_r2 in column direction, as shown in Fig. 7b. Similarly, the partial products of MUL8-2 are in row_r3 and row_r4, and col_r3 and col_r4. To generate the four partial products in row_r1 and row_r2, the most significant 8-bit of the multiplicand are set to zeros and only the least significant 8-bit are used. When generating partial products in row_r3 and row_r4, the least significant 8-bit of the multiplicand are set to zeros and only the most significant 8-bit are used. In addition, as MUL8-2 is an independent multiplication, when generating $pp5$, the LSB of the corresponding multiplier group is set to 0 instead of using the 8th bit of the multiplier. Similarly, in 4-bit mode, as

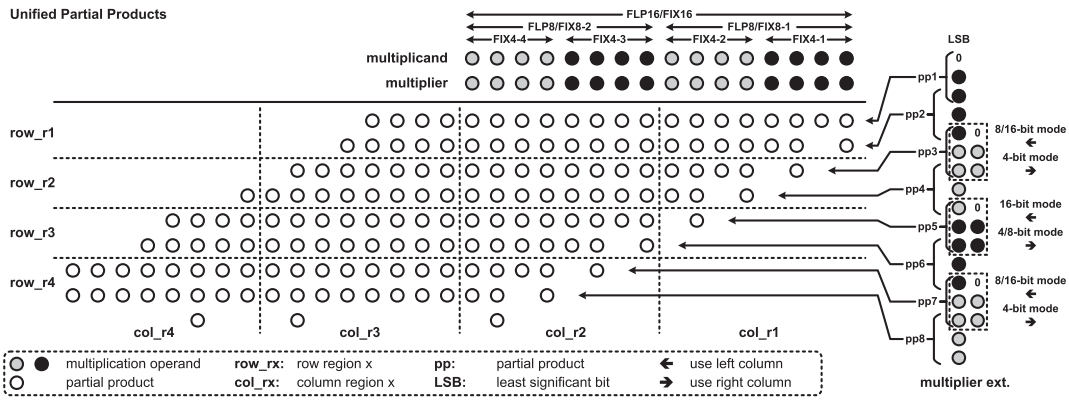

Fig. 6. Partial product generation and partial product array of the proposed flexible multiple-precision multiplier.

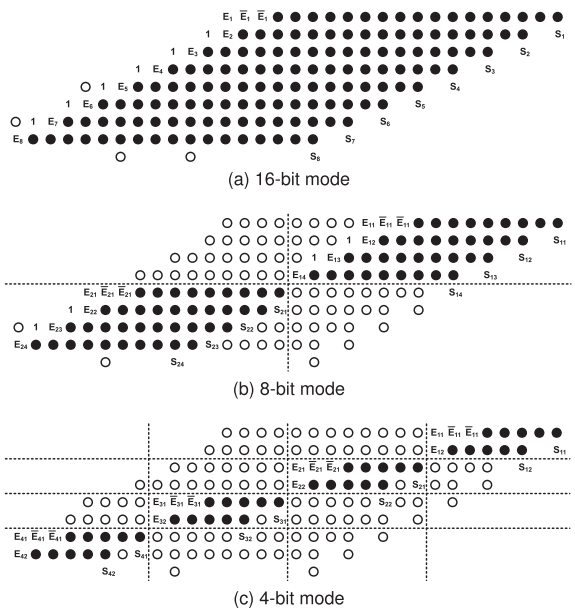

Fig. 7. Partial product array of each precision mode (white dots represent the bits not used; black dots represent partial products).

shown in Fig. 7c, only $FIX4-1$, $FIX4-2$, $FIX4-3$, or $FIX4-4$ is used to generate the partial products in row_r1, row_r2, row_r3, or row_r4, respectively. In addition, when generating $pp3$, $pp5$, and $pp7$, the LSBs of the corresponding multiplier groups are set to zero.

To ensure correct multiplication results for low precision modes, carry propagation during partial products accumulation is managed. In 8-bit mode, carry is not allowed to be propagated through the second vertical dash line in Fig. 6 (or the vertical dash line in Fig. 7b), and in 4-bit mode, carry is not allowed to be propagated through the three vertical dash lines in Fig. 6 or Fig. 7c.

3.3 Alignment Control and Shifter

In 16-bit floating-point mode, the general alignment shifting method used in many previous FMA designs [34] is applied. The initial position of the processed mantissa of C operand and product $A \times B$ is shown in Fig. 8a. The mantissa of C is placed 2-bit to the left of the carry save format product. In 16-bit floating-point mode, BW_m can be at most 14-bit. Therefore, the 2-bit zeros gap are already included in $mult_{sum}$ and $mult_{carry}$. The mantissa of C is then right shifted according to the difference of exponents and the bit-width of mantissa. The shifting amount can be determined by:

$$shift_c = 32 + BW_{mc} - 2 \times BW_m - d, \tag{2}$$

where $d = e_c - (e_a + e_b)$ is the exponent difference among the three input operands and e_a, e_b, and e_c are the exponent value of operand A, B, and C. The maximum effective shifting amount, $shift_{cmax} = 48$-bit, occurs when all bits of the C mantissa are shifted to the right of the product, which is shown in Fig. 8b.

In the initial alignment position, the 2-bit zeros gap between the product and the addend is to ensure the alignment shifting of the addend is a single direction shifting so that the shifter design can be simplified. In the proposed design, when the mantissa bit-width BW_m becomes smaller,

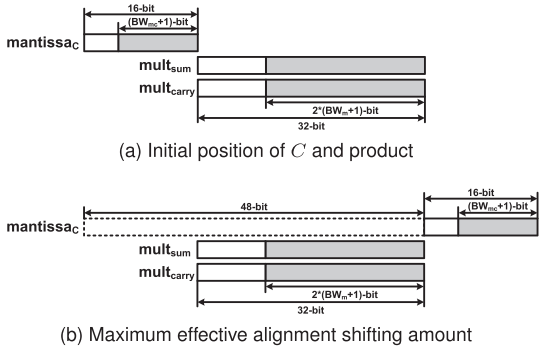

Fig. 8. Alignment shifting of C with product in FLP16 mode.

the gap between the product and the addend becomes larger. However, in those cases, the alignment shifter is still a single direction shifter. Therefore, in the proposed design, in order to simplify the shifter design, we do not force the gap to be 2-bit. Instead, a unified initial position of the addend and the product is used, as shown in Fig. 8a. The alignment shifting amount can always be calculated with Equation (2).

For 8-bit floating-point mode, in addition to the alignment of C, the two products generated also need to be aligned. To simplify the shifting circuit, in the proposed design, the two products are compared first to determine the product with larger exponent. The product with larger exponent is used as the anchored product. Both C and the other product will be aligned to the larger product. The initial position of the larger product $mult\_one_{sum}$ and $mult\_one_{carry}$, the smaller product $mult\_two_{sum}$ and $mult\_two_{carry}$, and mantissa of C is shown in Fig. 9. The smaller product is placed to the right of the LSB of the addition range. This is because in subnormal cases, it needs to be left renormalized first and then right aligned. In the proposed design, the renormalization amount and the alignment amount will be considered at the same time, where the effective renormalization amount will be generated and the smaller product only needs a left shift. This can simplify the shifter design. The addition range shown in Fig. 9 is 47-bit. The MSB position of the mantissa of C before alignment shifting is not included in the adder. This is because in floating-point mode, the mantissa of C could only occupy at most 15-bit. In fixed-point mode, 47-bit adder is large enough for 32-bit product accumulation.

In order to support subnormal numbers, the maximum alignment shifting happens when the MSB of the addend is shifted 2-bit to the right of the LSB of the product [35]. By extending the analysis in [35], when the multiplier operands and the addend have different mantissa bit-width, the gap to the right of the LSB of the product can be calculated with $BW_{mc} - BW_m + 2$. For 16-bit floating-point mode, when both the addend and the multiplier operands have 14-bit mantissa, as the exponent is only 1-bit, the addend C can never reach the maximum alignment shifting position as shown in Fig. 8b. For other mantissa bit-width, as the range of the addend is usually chosen to be equal to or larger than the range of the multiplier operands, the bit-width of the addend is smaller than the multiplier operand. Therefore, at most 2-bit gap to the right of the product LSB is enough to handle subnormal numbers. In this case, as the mantissa

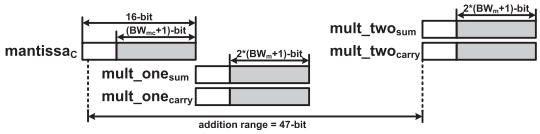

Fig. 9. Alignment shift of C and two products in 8-bit floating-point mode.

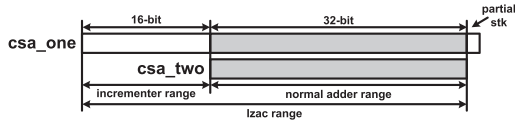

Fig. 10. Addition arrangement and the leading zero anticipator and counting (LZAC) range of the proposed unit.

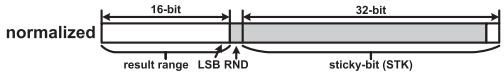

Fig. 11. Rounding method for floating-point modes.

bit-width is smaller, there is more than 1-bit zero appear in the 16-bit addend. There is no need to add extra zero bit gap. For 8-bit floating-point mode, BW_{mc} and BW_m can be at most 14-bit and 6-bit, respectively. Therefore, at most 10-bit gap is required. As shown in Fig. 9, at least 15-bit gap is available. Considering all these cases, there is no need to add extra zero gap to the right of the product LSB.

For 16-bit fixed-point mode, although the bit-width of fraction part can be flexibly changed, within a specific mode, the alignment shifting amount is a constant value. Therefore, for 16-bit mode, the alignment shifting can reuse the alignment shifter designed for floating-point mode by setting the shifting amount to a constant. The constant shifting amount for C, $shift_{const\_c}$, can be determined by:

$$shift_{const\_c} = 32 + BW_{mc} - 2 \times BW_m, \tag{3}$$

where the BW_m and BW_{mc} represent the bit-width of fraction in fixed-point mode. For 8-bit and 4-bit fixed-point mode, there is no need to used alignment shifter. In these two modes, BW_{mc} can be set to be equal to $2 \times BW_m$ (If $BW_{mc} > 2 \times BW_m$, the extra fraction bits can never be used. If $BW_{mc} < 2 \times BW_m$, some data bits will be lost.). Before performing addition, the LSB of the addend can be directly put to the LSB position of the product.

3.4 Addition

The aligned C and products will be accumulated with carry save adders. The generated carry-save format vectors will be added using a carry propagate adder. For the proposed design, a total of 47-bit addition is provided. The lower 32-bit addition will be implemented with a carry select adder and the higher 15-bit will be implemented using an incrementer, as shown in Fig. 10. For the higher 15-bit addition, both the results of $carry\_in = 0$ and $carry\_in = 1$ are generated. The output carry from the lower order adder will be used to select these two results. As in 4-bit mode or 8-bit mode, dot-product operation is performed. Therefore, for the adder there is no need to generate parallel separate results, instead only one single addition result is generated.

3.5 Leading Zero Anticipator and Counting

In parallel to the adder, the leading zero anticipator and counting (LZAC) logic is implemented. As both positive result and negative result can be generated, both leading zeros and leading ones can occur. Therefore, the general case indicator presented in [36] is used.

The counting result of a normal LZA might have 1-bit error. This error can be easily corrected in the normalization shifter for a standard precision design. However, as the proposed unit supports flexible precision, it is complex to find the MSB position of the result and then detect the MSB value to determine whether a correction should be performed. Therefore, in order to reduce the cost of later normalization stage, the exact LZA unit in [37] is used in the proposed design.

As shown in Fig. 10, the LZAC is applied to whole 48-bit range of the adder. When BW_{mc} is small, there might exist some leading zeros that are not occupied by the mantissa. Therefore this bit count should be subtracted from the LZAC count when determining the normalization shifting amount. This can be achieved by:

$$norm\_shift = lzac\_count - (15 - BW_{mc}). \tag{4}$$

3.6 Normalization Shifting

Normalization shifting is performed with a 6-level dynamic shifter to shift the 48-bit vector. As the LZA can generate the exact leading zero count, there is no need to add one more stage to compensate the LZA error. The normalization shifter will bring the MSB of the addition result back to the left of the radix point position, as shown in Fig. 11.

After alignment shifting, the base exponent becomes $e_{base} = e_c + shift_c$. If $e_{base} - norm\_shift > e_{min}$, then result is still a normal number and the normalization shifting amount is $norm\_shift$ and the result exponent is set to $e_{base} - norm\_shift$. Otherwise, the result will become a subnormal number. In this case, normalization shift amount is $e_{base} - e_{min}$ and the result exponent is set to e_{min}.

3.7 Rounding

For floating-point mode, roundTiesToEven, which is the default rounding mode in IEEE 754-2008 [3], is implemented. After normalization, the result will be brought back to a position shown in Fig. 11. Therefore, the rounding position for floating-point is fixed, which is the LSB position of the C operand. Therefore, the bits required to perform rounding, the LSB, the rounding bit, and the sticky bit, can be easily generated.

For 16-bit fixed-point operations, the product has larger fractional bit-width than the addend. Therefore, rounding is also required. Unlike the floating-point modes where the rounding position is fixed, in fixed-point case, the rounding position depends on the number of fraction bits in the input operands. To find the correct rounding position and generate rounding bits, the masks, mm and mm_pone, generated for input processing will be used again. The process of using mm and mm_pone to find three rounding bits, the LSB, the round bit (RND), and the sticky bit (STK), is shown in Fig. 12. Both mm and mm_pone are extended with zeros to make 16-bit mm_ext and mm_pone_ext. Then these two extended vectors are right shifted by 1-bit. By ANDing

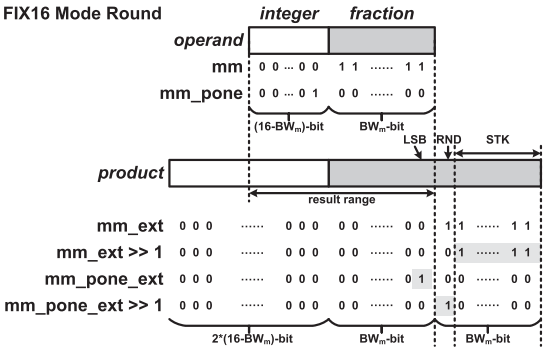

Fig. 12. Rounding method for 16-bit fixed-point mode.

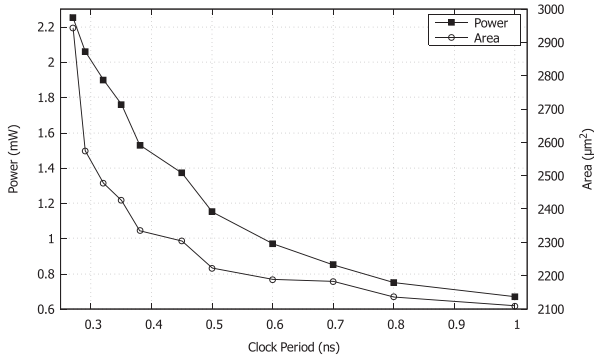

Fig. 13. Power-Delay curve and Area-Delay curve of the proposed design under STM-28nm.

mm_pone_ext with the result, the LSB can be extracted. Similarly, by using the shifted mm_ext and shifted mm_pone_ext, the STK and the RND can be generated as shown in Fig. 12. By using these three bits, the fixed-point rounding can be performed. The floating-point mode and the fixed-point mode can share the incrementer to perform rounding plus one operation. For the rounding in integer part, a simple truncation can be performed.

For other lower precision fixed-point operations, as the bit-width of C is always larger than operands in A and B, C can have the same or more fraction bits than the product. Therefore, there is no need to perform rounding for the fraction. Truncation can be performed for the integer.

3.8 Output Processing

After processing the sign, exponent, and mantissa separately, they need to be combined again to generate the final results. For fixed-point, a simple truncation can be used to generate the final 16-bit result.

For floating-point, basically two operations are required. As the mantissa is already right aligned to the LSB position, for mantissa part, we only need to remove the implicit bit. To do so, the mantissa mask mm will be used. The mm contains all ones at the position of mantissa (except implicit bit). By ANDing the mm with the result vector, the implicit bit can be removed.

The second operation for floating-point is to shift the exponent back to the correct position. In order to obtain the actual value of exponent to generate the shifting amount, exponent is right shifted with the amount of BW_m or BW_{mc} in alignment control module. At the output processing module, the exponent is shifted back with the amount of BW_{mc}.

Finally, the exponent vector and mantissa vector are ORed together and sign bit is added into the MSB position to form the final floating-point result.

4 Results and Analysis

The model of the proposed MAC architecture is implemented with Verilog HDL. Since the proposed design supports non-standard floating-point format, there is no simulation tool that can generate test vectors for all supported precisions. In order to simulate and verify the proposed design, a customized software model written in C language is built. With the designed software model, extensive testing vectors for each of the supported precisions can be generated. These testing vectors are used to simulate the proposed design in Modelsim. The proposed design is verified to work properly in all supported modes.

The proposed design is then synthesized in Synopsys Design Compiler using STM-28nm technology with typical case parameters ($1.00V$ and 25°C). The timing and area metrics are generated. For power consumption measurement, the Verilog netlist generated by synthesis is simulated with the testing vectors again to obtain a signal activity file. Then the synthesized netlist and the signal activity file are imported to Synopsys PrimeTime PX for an accurate power estimation.

Each pipeline stage of the proposed design is synthesized first. The delay of each pipeline stage is analyzed and the result is shown in Table 2. In the preliminary experiment, the second pipeline stage (only contains carry save adder, carry propagate adder, and LZA) has a delay of 0.14 ns which is smaller than the other two stages. The worst case delay 0.3 ns appears in the first pipeline stage (contains multiplication and alignment shifter). This pipeline allocation is unbalanced. By moving the alignment shifter to the second pipeline stage, as the results shown in Table 2, the pipeline stage allocation is more balanced. The second pipeline stage consumes more area because it contains two alignment shifter in addition to the adder and LZA circuit. Overall, the multiplier still consumes the largest area and power consumption.

The whole design is then synthesized. The synthesis results show that after adding pipeline registers, the worst case delay of the pipelined design is 0.27 ns. In this timing constraint, the proposed design consumes an area of 2943 $\mu\mathrm{m}^2$ and an average power consumption of 2.25 mW. The proposed design is also synthesized under different timing constraints to find the trade-off between timing and area, and timing and power. The area-delay curve and power-delay curve can be found in Fig. 13. To obtain the best performance, the design point with the smallest worst case delay (0.27 ns) is chosen and the corresponding design metrics are compared with the standard arithmetic unit. In addition, this design point is used to analyze the power consumption of the proposed unit under different operational modes.

In different operational modes, the proposed design has different power value. In floating-point mode, by using different number of mantissa, the power number is different. This comes from the trade-off between multiplier and shifter. With larger bit-width for mantissa, more logic of the multiplier will be enabled to perform multiplication.

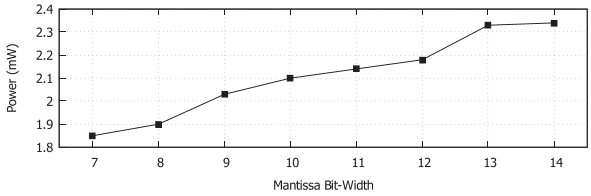

Fig. 14. Power consumption of the proposed design in 16-bit floating-point mode (with timing constraint 0.27 ns).

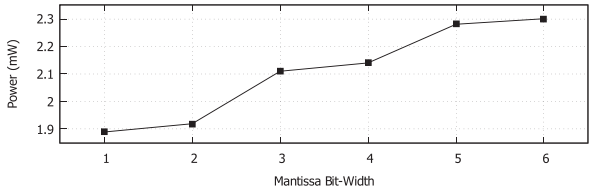

Fig. 15. Power consumption of the proposed design in 8-bit floating-point mode (with timing constraint 0.27 ns).

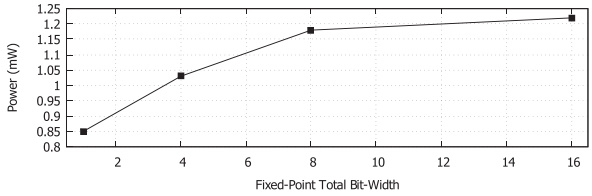

Fig. 16. Power consumption of the proposed design in various fixed-point mode (with timing constraint 0.27 ns).

However, as the exponent bit-width in this case is small, fewer levels of shifter can be enough. The power figure of the proposed design in 16-bit floating-point mode is shown in Fig. 14. As the mantissa bit-width increases, the power consumption of the proposed design also increases.

The power figure of the proposed design in 8-bit floating-point mode is shown in Fig. 15. It has similar trend as the 16-bit floating-point mode where the power increases with the number of mantissa bit-width increases. The power consumption of a single 8-bit floating-point operation should be less than that of a 16-bit floating-point operation. However, as the proposed MAC unit supports two parallel 8-bit floating-point operations, the total power consumption is similar to a 16-bit floating-point operation as they actually share the same hardware resources.

In fixed-point mode, the power consumption of the proposed unit is much smaller than that of floating-point mode. The power figure of the proposed design under different fixed-point mode is shown in Fig. 16. In fixed-point mode, all the shifting and alignment can be done with constant shifter. The alignment control and LZAC used in floating-point mode are disabled. In addition, there is no need to handle exponent for fixed-point modes. Moreover, the rounding unit for fixed-point mode is less complex compared to the floating-point rounding unit. The subnormal handling logics are also not required in fixed-point mode. Within fixed-point modes, when using lower precision, as shown in Fig. 7, only part of the multiplier array are used. In addition, in lower precision mode, the higher order part of the adder is not used. All these lead to the power reduction.

To the best of our knowledge, there is no such flexible precision arithmetic unit design appearing in the literature. In order to show the advantage of the proposed design, the proposed design is compared with several standard arithmetic units, including a standard 16-bit fixed-point MAC (FIX16-MAC), a standard 16-bit floating-point MAC (FLP16-MAC), a standard MAC unit designed for BFloat16 format [18] (BFL16-MAC), and a standard 32-bit floating-point MAC (FLP32-MAC). Moreover, in order to measure the resource overhead introduced by the flexible precision support, a multiple-precision MAC unit (MP-MAC) is also designed. This MP-MAC unit does not have the flexible precision feature, but supports the same operation mode as the proposed design. The comparison results of these designs are shown in Table 3. Compared to the standard 16-bit fixed-point MAC, the proposed design required larger area and power, and has larger delay. However, in addition to 16-bit fixed-point MAC operations, the proposed design also supports many other arithmetic operations including floating-point operations.

Compared to the standard 16-bit floating-point MAC unit, the proposed design has only 22 percent area overhead and 13 percent power consumption overhead. The proposed design is only 8 percent slower compared to the standard 16-bit floating-point unit. BFloat16 format has smaller BW_m than the standard half-precision format, and thus the area and power of the BFloat16 MAC become even smaller. Compared to the BFloat16 MAC unit, the proposed design has 37 percent area overhead and 28 percent power consumption overhead. However, in terms of functionality, the proposed

TABLE 3
Comparison of the Proposed MAC Unit with Standard Arithmetic Units

Design	Latency	Delay (ns)	Area (μm^2)	Power (mW)	Energy/op# ($\times 10^{-13}$J)					
					FLP32	FLP16	FLP8	FIX16	FIX8	FIX4
FIX16-MAC	3	0.16	1509	1.16	-	-	-	5.57	-	-
FLP16-MAC	3	0.25	2415	1.98	-	14.9	-	-	-	-
BFL16-MAC*	3	0.25	2140	1.75	-	13.1	-	-	-	-
FLP32-MAC	3	0.40	6689	5.65	67.8	-	-	-	-	-
MP-MAC†	3	0.26	2681	2.09‡	-	16.3	8.15	8.97	4.25	1.91
Proposed	**3**	**0.27**	**2943**	**2.25**‡	-	**18.2**	**9.11**	**9.88**	**4.78**	**2.08**

MAC unit designed for BFloat16 format (1-bit sign, 8-bit exponent, and 7-bit mantissa).
†*Multiple-Precision MAC unit supporting the same operation modes as the proposed design without flexible precision support.*
‡*Average power of all supported operation modes.*
#*energy/op = (latency × delay × power)/number_parallel_operations.*

design can support many other operations, such as flexible floating-point operations and fixed-point operations. With the proposed design, the applications will have more choices to select their most suitable numerical precisions.

The proposed MAC unit is also compared to a standard 32-bit floating-point unit. The standard 32-bit floating-point unit can provide 8-bit exponent for the application. In conventional processors, when the required bit-width of exponent is larger than 5, half-precision unit cannot perform the operations and one has to resort to the single-precision unit. However, in the proposed design, because the flexible precision support, 8-bit exponent can be provided in the same hardware. As discussed in the introduction, for neural network computing, the exponent part is more important than mantissa part and the bit-width of exponent should be satisfied first. The remaining bits are allocated to mantissa. Therefore, in terms of functionality, the proposed design can be compared with 32-bit single-precision MAC unit. As shown in Table 3, due to the large mantissa bit-width, the hardware costs of the single-precision unit is much higher than those of the proposed unit. Therefore, for deep learning applications, the proposed MAC unit can provide almost the same accuracy with much lower hardware cost.

Compared to the MP-MAC design, the proposed design has the flexible precision support. This flexible precision support introduces only 9.7 percent area overhead and 7.6 percent power overhead. The overhead of energy consumption under each operation mode is also small. The resource overhead mainly comes from the mask operation during input and output processing. With small resource overhead, the support of flexible precision will bring more flexibility for the applications, especially the machine learning training and inference.

Note that the standard high precision unit can also be used to perform low precision operations, for example FIX16-MAC can be used to calculate FIX8 MAC. However, as there is no parallel low operations support, only one operation can be performed at each clock cycle. Although the power consumption is slightly reduced due to a reduced toggle rate at higher order bit positions, the energy per operation is almost the same as the one when performing operations for the highest supported precision. On the other hand, for the proposed design, parallel low precision operations are supported and thus the energy per operation at low precision can be significantly improved.

5 Case Study

In order to show the power merits of the proposed MAC unit in actual deep neural network applications, a case study of deep neural network inference operations is performed. In this case study, the precision required by neural network inference are extracted from the results of [38]. For inference operations, only the fixed-point formats are used in this study.

To simplify the testing process and to put emphasis on arithmetic unit of the deep learning processor, we do not use a whole deep learning processor architecture to perform this case study. Instead, we simulate the deep neural network computing process. For example, when simulating LeNet, input image data are used as the test vectors in the testbench of the proposed MAC unit. Then, the generated results, which are the first layer's outputs, are used again as the second layer's input. This process is repeated until the final neural network layer is processed. For each test case, 100 images are randomly selected from the validation set of MNIST [39] (for LeNet) and ImageNet database [40]. The signal activity file dumped during this simulation process is used to measure the power consumption of using the proposed MAC unit in Synopsys PrimeTime PX. In addition, the inference results from the simulation process are collected to evaluate the neural network accuracy when using the proposed MAC unit as computing elements.

This case study focuses on the different precision requirements of different layers and of different neural network models. In this case study, six neural network models are included: LeNet, AlexNet, NiN, GoogLeNet, VGG-M, and VGG-19. Only convolutional layers in these neural networks are considered because most of the neural network computations are in convolution layers [2]. The test set used in this case study contains both small scale and large scale neural networks. In addition, according to the minimum precision requirements of these networks in [38], this test set contains both small precision operations and large precision operations. Specifically, the precision configuration is: LeNet (2-3), AlexNet (9-7-4-5-7), NiN (8-8-7-9-7-8-8-9-9-8-7-8), GoogLeNet (10-8-9-8-8-9-10-8-9-10-8), VGG-M (6-8-7-7-7), and VGG-19 (9-9-9-8-12-10-10-12-13-11-12-13-13-13-13-13), where the number represents the bit-width of activation and weight in a layer. Under this configuration, these neural networks can achieve 99 percent of the accuracy when using 32-bit floating-point numbers.

In addition to the proposed MAC unit, the standard 16-bit fixed-point MAC is also used in this case study. The 16-bit fixed-point MAC is used in two ways: (1) FIX16: all activation and weight of the six neural networks are quantized to 16-bit fixed-point numbers and then fed to the 16-bit fixed-point MAC unit; (2) FIX16F: all activation and weight use the flexible precision configuration (the same as the one used for the proposed unit) and they are then sign extended to fit 16-bit bit-width. The results of the proposed MAC unit are compared with these two units.

The average power consumption when implementing using the proposed unit is 1.18 mW while the FIX16 implementation can achieve 1.61 mW and the FIX16F can achieve 1.16 mW. The FIX16 implementation consumes more power because it uses higher precision format than the proposed unit and the FIX16F, and thus the signal toggle rate is higher. The proposed unit consumes higher power than FIX16F because of the extra logic to support other operation modes.

The power efficiency of these implementations is also estimated, as it is a common merit to evaluate the performance

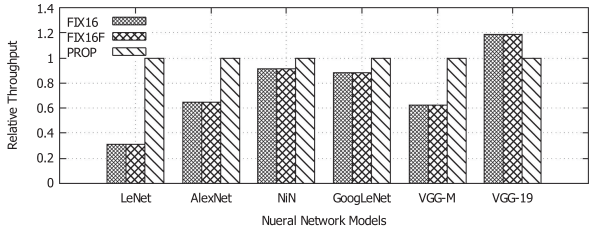

Fig. 17. Relative throughput of three implementations for six neural networks.

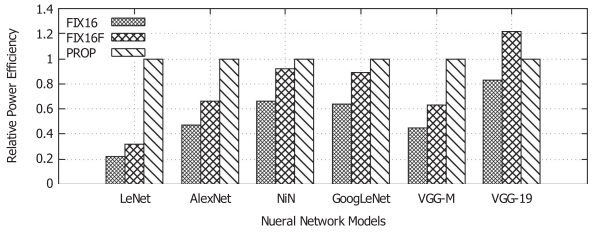

Fig. 18. Relative power efficiency of three implementations for six neural networks.

of a deep learning hardware design [41]. In this case study, power efficiency is measured in terms of image processed per second per watt (img/s/W). To calculate power efficiency, the throughput in terms of img/s is first calculated. As only a simplified neural network computing model is used in this case study, we cannot actually measure the exact processing time. Instead, the number of required MAC operations to process each image is used to represent the processing time. Therefore, the throughput can be computed with $1/(\text{mac\_operations} \times \text{delay\_of\_each\_cycle})$. The full results are shown in Fig. 17. Here the throughput of FIX16 and FIX16F relative to the proposed unit is shown. Although FIX16 and FIX16F have smaller delay compared to the proposed design, for some neural network models, as the proposed unit has the ability to perform parallel low precision operations, the throughput of the proposed unit can be higher than FIX16 and FIX16F. For VGG-19, except the fourth layer, all other layers require more than 8-bit precision. In this case, operations cannot be done in parallel which leads to the throughput degradation for the proposed unit.

The power efficiency is then evaluated in terms of img/s/W. The full results are shown in Fig. 18. Although the average power consumption of the proposed unit is higher than that of standard 16-bit fixed-point unit, as parallel low precision operations are supported in the proposed unit, the proposed design can achieve an improved power efficiency under most of the neural network implementations. VGG-19 is still an exception since it cannot be quantized to 8-bit or less. Therefore, when low precision operations are feasible for a neural network mode, the proposed MAC unit can process with a good power efficiency.

Note that one may use parallel standard low precision units to improve the power efficiency, however, in this case, higher precision operations will not be supported and thus some neural networks, for example VGG-19, cannot be implemented. The proposed unit, on the other hand, also supports high precision operations and even floating-point operations which will be very flexible for neural network computing.

6 Conclusions

In this paper, an efficient flexible multiple-precision multiply-accumulate (MAC) unit is designed for deep neural network training and inference. The proposed MAC unit is designed based on the requirements of deep neural network computing, for example, low-precision requirements, multiple-precision requirements, and flexible precision requirements for different operations and model. The proposed MAC unit supports both floating-point operations and fixed-point operations. For floating-point operations, the proposed MAC unit supports one 16-bit operations or two 8-bit operations. With flexible precision support, the bit-with of the exponent and mantissa can be mutually exchanged. The proposed unit also supports fixed-point operations for deep neural network inference. It supports one 16-bit operations, or two 8-bit operations, or four 4-bit operations. At the lowest precision, it also supports binary neural network operations. The proposed MAC unit can be used in deep learning processors to enable both training and inference in the same architecture and used together with reduced precision deep learning tools, such as Ristretto and Intel Distiller, to facilitate the deep neural network accelerator design towards more functionality and more energy efficiency. The proposed MAC architecture can also be used to design neural network IP cores for FPGA devices.

For future work, the support for asymmetric arithmetic operations can be explored since many deep neural network implementations use different precisions for activations and weights. In addition, other deep neural network computing features, such as weight sharing and sparsity aware method, can be supported.

Acknowledgments

The authors would like to thank the Natural Sciences and Engineering Research Council (NSERC) of Canada and the Department of Electrical and Computer Engineering at the University of Saskatchewan for their financial support for this project. This work was also partly supported by the R&D program of MOTIE/KEIT [No. 10077609, Developing Processor-Memory-Storage Integrated Architecture for Low Power, High Performance Big Data Servers].

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.
[2] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[3] *IEEE Standard for Floating-Point Arithmetic*," IEEE Std 754-2008, pp. 1–70, Aug. 2008.
[4] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, pp. 1–12, 2017. [Online]. Available: http://arxiv.org/abs/1710.03740
[5] M. Ortiz, A. Cristal, E. Ayguade, and M. Casas, "Low-precision floating-point schemes for neural network training," *CoRR*, vol. abs/1804.05267, pp. 1–16, 2018. [Online]. Available: http://arxiv.org/abs/1804.05267
[6] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 11, pp. 5784–5789, Nov. 2018.
[7] Z. Deng, C. Xu, Q. Cai, and P. Faraboschi, "Reduced-precision memory value approximation for deep learning," Hewlett Packard Labs, Palo Alto, CA, Tech. Rep. HPL-2015-100, Dec. 2015.
[8] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low-cost binary128 floating-point FMA unit design with SIMD support," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 745–751, May 2012.
[9] D. Tan, C. E. Lemonds, and M. J. Schulte, "Low-power multiple-precision iterative floating-point multiplier with SIMD support," *IEEE Trans. Comput.*, vol. 58, no. 2, pp. 175–187, Feb. 2009.
[10] H. Zhang, D. Chen, and S.-B. Ko, "Area- and power-efficient iterative single/double-precision merged floating-point multiplier on FPGA," *IET Comput. Digital Techn.*, vol. 11, no. 4, pp. 149–158, 2017.
[11] H. Zhang, D. Chen, and S. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Trans. Comput.*, vol. 68, no. 7, pp. 1035–1048, Jul. 2019.

[12] T. Trader, "How knights mill gets its deep learning flops," Jun. 2017. [Online]. Available: https://www.hpcwire.com/2017/06/22/knights-mill-gets-deep-learning-flops/
[13] *NVIDIA Tesla P100 Whitepaper*, WP-08019-001 ed., NVIDIA, 2016.
[14] P. Konsor, "Performance benefits of half precision floats," Aug. 2012. [Online]. Available: https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats
[15] Y. Lee, Y. Choi, S.-B. Ko, and M. Ho Lee, "Performance analysis of bit-width reduced floating-point arithmetic units in FPGAs: A case study of neural network-based face detector," *EURASIP J. Embedded Syst.*, vol. 2009, no. 1, pp. 4:1–4:11, Jul. 2009.
[16] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *CoRR*, vol. abs/1703.03073, pp. 1–10, 2017. [Online]. Available: http://arxiv.org/abs/1703.03073
[17] P. Judd, J. Albericio, T. H. Hetherington, T. M. Aamodt, N. D. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-precision strategies for bounded memory in deep neural nets," *CoRR*, vol. abs/1511.05236, pp. 1–12, 2015. [Online]. Available: http://arxiv.org/abs/1511.05236
[18] M. Abadi, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://tensorflow.org
[19] *Deep Learning with INT8 Optimization Xilinx Devices*, WP486 ed., San Jose, CA, USA: Xilinx, Apr. 2017.
[20] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Machines*, Apr. 2017, pp. 101–108.
[21] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 45–54.
[22] M. Courbariaux, and Y. Bengio, "BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, pp. 1–11, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830
[23] U. Koster, et al., "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," *CoRR*, vol. abs/1711.02213, pp. 1–14, 2017. [Online]. Available: http://arxiv.org/abs/1711.02213
[24] A. Nannarelli, "Tunable floating-point for energy efficient accelerators," in *Proc. IEEE 25th Symp. Comput. Arithmetic*, Jun. 2018, pp. 29–36.
[25] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient mitchells approximate log multipliers for convolutional neural networks," *IEEE Trans. Comput.*, vol. 68, no. 5, pp. 660–675, May 2019.
[26] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *CoRR*, vol. abs/1603.01025, pp. 1–10, 2016. [Online]. Available: http://arxiv.org/abs/1603.01025
[27] 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. 105:1–105:24, Mar. 2014.
[28] M. Imani, D. Peroni, and T. Rosing, "CFPU: Configurable floating point multiplier for energy-efficient computing," in *Proc. 54th ACM/EDAC/IEEE Des. Autom. Conf.*, Jun. 2017, pp. 1–6.
[29] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.
[30] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. F. Chang, "A reconfigurable streaming deep convolutional neural network accelerator for internet of things," *IEEE Trans. Circuits and Syst. I: Reg. Papers*, vol. 65, no. 1, pp. 198–208, Jan. 2018.
[31] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.
[32] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.
[33] G. W. Bewick, "Fast multiplication: Algorithms and implementation," Ph.D. dissertation, Dept. Electrical Eng., Stanford Univ., Stanford, CA, Feb. 1994.
[34] T. Lang and J. D. Bruguera, "Floating-point multiply-add-fused with reduced latency," *IEEE Trans. Comput.*, vol. 53, no. 8, pp. 988–1003, Aug. 2004.
[35] E. M. Schwarz, M. Schmookler, and S. D. Trong, "FPU Implementations with denormalized numbers," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 825–836, Jul. 2005.
[36] M. S. Schmookler and K. J. Nowka, "Leading zero anticipation and detection-a comparison of methods," in *Proc. 15th IEEE Symp. Comput. Arithmetic*, 2001, pp. 7–12.
[37] J. D. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction," *IEEE Trans. Comput.*, vol. 48, no. 10, pp. 1083–1097, Oct. 1999.
[38] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture.*, Oct. 2016, pp. 1–12.
[39] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[41] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl deep learning accelerator on arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 55–64.

Hao Zhang received the BEng degree in electronic engineering from Shandong University, Jinan, Shandong, China, in 2012, the coursebased MSc degree in electronic and information engineering from the City University of Hong Kong, Hong Kong SAR, in 2013, and the MSc degree in electrical and computer engineering from the University of Saskatchewan, Saskatoon, Canada, in 2015. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, Canada. His research interests include computer arithmetics, reconfigurable computing, and hardware design for deep learning. He is a student member of the IEEE.

Dongdong Chen received the BEng degree in electrical engineering from the Northwestern Polytechnical University, Xian China, in 2003, the MSc degree in system-on-chip from the Lund University, Lund, Sweden, in 2006, and the PhD degree in electrical and computer engineering from the University of Saskatchewan, Saskatoon, Canada, in 2011. Currently, he works as staff design engineer in Intel Corporation, San Jose, CA, United States. His research interests include the algorithms and architectures for computing binary and decimal transcendental functions, floating-point arithmetic, reconfigurable computing, and hardware design for deep learning. He is a member of the IEEE.

Seok-Bum Ko received the PhD degree in electrical and computer engineering from the University of Rhode Island, Kingston, Rhode Island, in 2002. He is currently professor with the Department of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, Canada. He worked as a member of technical staff for Korea Telecom Research and Development Group, Korea from 1993 to 1998. His research interests include computer arithmetic, computer architecture, deep learning processor architecture, efficient hardware implementation of compute-intensive applications, and biomedical engineering. He is a senior member of the IEEE Circuits and Systems Society.