BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 66, No. 5, 2018
DOI: 10.24425/124273

Novel architecture for floating point accumulator with cancelation error detection

E. JAMRO[1]*, A. DĄBROWSKA-BORUCH[1], P. RUSSEK[1], M. WIELGOSZ[1], and K. WIATR[1,2]

[1]AGH University of Science and Technology, Department of Electronics, Al. Mickiewicza 30, 30-059 Kraków, Poland
[2]CYFRONET Academic Computer Centre, University of Science and Technology, ul. Nawojki 11, 30-950 Kraków, Poland

Abstract. A floating point accumulator cannot be obtained straightforwardly due to its pipeline architecture and feedback loop. Therefore, an essential part of the proposed floating point accumulator is a critical accumulation loop which is limited to an integer adder and 16-bit shifter only. The proposed accumulator detects a catastrophic cancellation which occurs e.g. when two similar numbers are subtracted. Additionally, modules with reduced hardware resources for rough error evaluation are proposed. The proposed architecture does not comply with the IEEE-754 floating point standard but it guarantees that a correct result, with an arbitrarily defined number of significant bits, is obtained. The proposed calculation philosophy focuses on the desired result error rather than on calculation precision as such.

Key words: floating point arithmetic, computing error, approximate computing.

1. Introduction

In this paper, a novel architecture for the floating point accumulator is proposed. Floating point adders are widely presented in literature, e.g. in [1, 2]. They are now widely used to accelerate computing [3–6] and accumulation is commonly used in e.g. matrix multiplication [4], neural network [7], signal filtering or damage diagnosis [8].

Nevertheless, a floating point accumulator cannot be constructed straightforwardly from floating point adders due to their strong pipeline architecture. Several different solutions to this problem have been presented in the literature. One of them is to employ $(N-1)$ parallel adders [9], where N is the number of arguments to be accumulated. This solution, however, is limited only to a small N due to hardware resources limitations. Another solution is to perform several accumulations/additions at the same time, different for each pipeline stage. This solution can be employed for matrix multiplications, for example, as several vector dot products can be carried out in parallel there [10]. The interleave factor (the number of computations performed at the same time) should be at least equal to the adder pipeline latency. A different approach was proposed in [11, 12], where the adder latency problem was solved by employing a special input and output buffer. This requires additional buffers and control logic and increases the total accumulator latency, or even introduces stalling states [11, 12].

The time-critical feedback can be easily implemented in fixed-point accumulators as the arguments' exponent does not change, thus there is no shift operation in the loop. This approach was exploited in [13]. The drawback of this method is that a long fixed-point accumulator is required; for single precision numbers, the accumulator bitwidth is equal to $24 + 255 = 279$ bits; where 24 is for the standard mantissa bitwidth and 255 – for the exponent that is coded on 8-bits. However, in some applications the exponent range of the accumulation result can be reduced. It is determined *a priori*, using rough error analysis or software profiling. This results in a smaller and more accurate accumulator than the one based on a floating point adder [13]. The solution is adapted by Xilinx and implementation results are presented in Table 7. Nevertheless, there is a large number of applications for which the accumulator exponent range cannot be easily specified. Furthermore, the approach of customized fixed-point accumulator leads to different bitwidth configurations for different types of calculations, e.g. when multiplying three matrices.

A similar approach is proposed in the exact accumulation of products [14, 15], for which a full-width fixed-point accumulator is employed. This solution utilizes a lot of hardware resources as the fixed-point accumulator is 279-bit wide for the single precision floating point format and $53 + 2047 = 2100$-bit wide for the double precision format. The major advantage of this method is that the accumulation result does not depend on an accumulation order and that it is not sensitive to massive cancellation. Consequently, exact arithmetic is strongly supported for inclusion of interval arithmetic in the P1788 IEEE standard [16]. It will be argued further that the accumulator proposed in this paper benefits partially from exact arithmetic without utilizing a significant amount of hardware resources.

A hybrid floating-fixed point accumulator for single precision floating point numbers was presented in [17]. This accumulator divides an exponent section into three parts:

- ❑ EXP [4:0] – the low order exponent. It is used to convert floating point numbers to pseudo-fixed point numbers. Consequently, the mantissa is extended by 31 bits (5-bit exponent), which results in $24 + 31 = 55$ bitwidth. It should be

*e-mail: jamro@agh.edu.pl

Manuscript submitted 2017-06-12, revised 2017-11-22, initially accepted for publication 2017-11-24, published in October 2018.

noted that the conversion is performed outside the critical accumulation loop, therefore it does not influence the critical timing path of the accumulation feedback loop.

- EXP [5] – the decision bit. Two independent accumulators are employed for arguments with EXP [5] bit equal to zero or one. At the final stage of accumulation, outside the time-critical loop, these two intermediate results are added, with one being shifted by 32-bit with respect to the other.
- EXP [7:6] – the high-order bits. Lower high-order bit values of the input argument or intermediate accumulation are treated as zeros due to at least 64-bit misalignment. This misalignment is far from single-precision accuracy.

In the accumulator described above, the critical accumulation loop consists only of an addition operation. To further reduce propagation delay, carry-save adders were employed in the time-critical accumulation loop.

A similar approach was adopted in [18–21], but only a single accumulator is used nevertheless, i.e. the decision bit EXP [5] is merged to the high-order bits EXP [7:6]. This, in comparison with [17], requires an additional 32-bit right shifter for both the input argument and accumulator to be included inside the time-critical loop. These shifters are employed when EXP [7:5] of the input argument and the accumulator intermediate result differ by one. To avoid precision loss, the accumulator underflow scenario may be handled, i.e. when at least 32 MSBs of the accumulator are ones or zeros, the accumulator result must be shifted left by 32-bits. The accumulator underflow is thoroughly considered in our paper as the underflow proper handling results in a signaling massive cancellation, which leads to precision loss. It should be noted that the accumulator underflow scenario was not considered in [17]. In [18], it is only suggested not to implement the accumulator underflow module due to the large hardware requirements. A similar design algorithm can be adopted for double precision numbers yet the low order exponent should be 6-bit wide, i.e. the 53-bit mantissa should be extended by 63-bit, thus the total mantissa width is 116-bit [18]. A combination of the hybrid floating/fixed point accumulator from [18] and the additional buffer employed to hide accumulator latency [11, 12] was proposed in [22]. For this accumulator, time-critical loop latency is reduced only to 1–4 cycles. The additional buffer is used only for the time-critical accumulation loop, therefore the buffer size and overall accumulator latency are lower.

2. Proposed accumulator

The proposed accumulator is similar to the one presented in [18], but more sophisticated shift scheduling is implemented outside the time-critical accumulation loop. A block diagram of the proposed accumulator for the single precision input is presented in Fig. 1. In it, the following parameters are introduced:

E – input exponent (EXP) width
M – input mantissa (MAN) width (excluding the leading one)
G – number of accumulator guard bits.

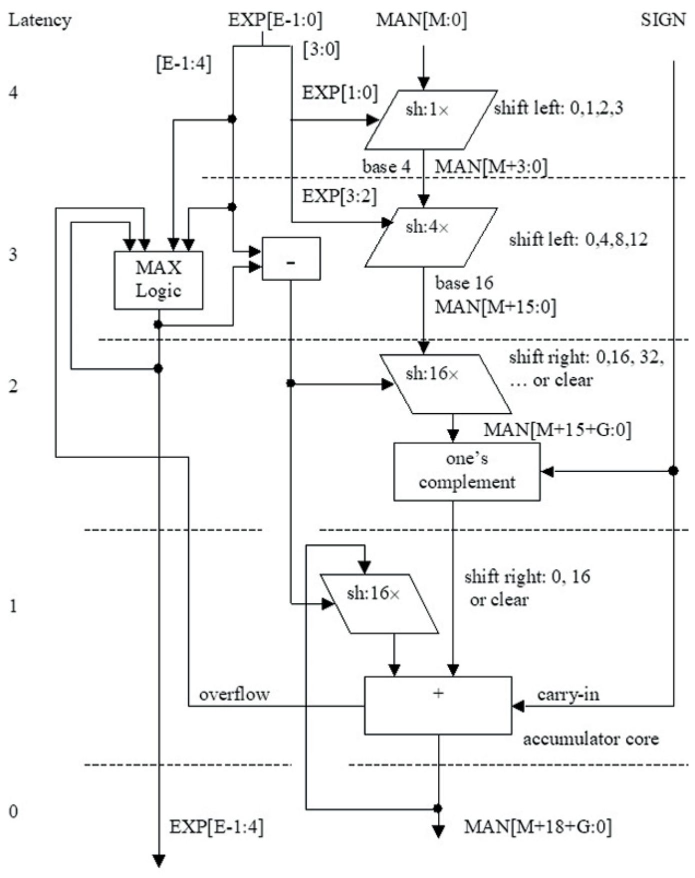

Fig. 1. Block diagram of the proposed accumulator for single precision

Figure 1 does not include the conversion from pseudo-fixed to floating point format that has to follow the accumulation process; this conversion module is further denoted as *acc2float* and is described in Section 3.1. In contrast with the solution given in [18], the mantissa is extended only by 15 bits in the proposed accumulator, i.e. a 16-bit base is introduced, and the [3:0] LSBs of the exponent are used to convert the number to a pseudo-fixed point format. This conversion is achieved in the first two clock cycles. Meanwhile, the [E-1:4] MSBs of the exponent remain unchanged. In Fig. 1, the pipeline register is instantiated whenever data cross the dash line.

In spite of using only the 16-bit exponent base, the accumulator can be employed for any floating point representation, e.g. for single or double precision. Furthermore, the time-critical loop consists only of an adder and a 16-bit right shifter (see the stage denoted as 1 in Fig. 1), which fits into a single 5-input LUT per bit (or 4-input LUT and a flip-flop with synchronous reset), commonly incorporated in FPGAs. Consequently, the propagation delay within the time-critical path is limited only to the smallest possible delay.

The key issue presented in our paper is that the right shift of the intermediate accumulation result by 32 bits can be achieved by two 16-bit shifts performed in consecutive clock cycles. This becomes possible because the successive exponent values can be calculated ahead by several clock cycles due to the pipeline registers being inserted only on the mantissa path (see the stages denoted as 3 and 4 in Fig. 1). The exponent path does not require any operation at the time. Summing up, the successive accumulator exponent value can be anticipated employing the following: current accumulator exponent value, accumulator overflow logic and maximum exponent value for the different pipeline stages of input data. Knowledge of the intermediate accumulator exponent value before the accumulation process starts allows us to implement greater mantissa shifts as consecutive 16-bit shifts.

For double precision numbers, the required accumulator intermediate right shift can be 16, 32, 48 and 64 bits. Thus, up to four 16-bit consecutive shifts are required. This requires two additional dummy pipeline stages (additional stages 5 and 6, not marked in Fig. 1) at the accumulator input and some more logic to calculate maximum exponent value for the input data at different pipeline stages. An alternative solution is to double the accumulator base from 16 bit to 32 bits, and employ 32-bit shifts instead. However, this would increase the accumulator width by 16 bits, which would result in additional hardware resources and propagation delay, or even extra pipeline latency, required during floating point to pseudo-fixed point format conversion. Therefore, the authors have decided to add extra pipeline stages rather than increase the base size. It should be noted that extra registers in the pipeline may increase the maximum frequency by reducing the routing propagation time – the FPGA place and route program can place logic elements with less restraints by employing register balancing.

The accumulator presented in Fig. 1 goes through a number of steps to perform its operation. At pipeline stage 4 (direct input – in Fig. 1 the pipeline stage number decreases with every clock cycle delay) a leading one is inserted at the left most position of the mantissa in the case when the input exponent is different from zero, i.e. the input number is neither equal to zero nor denormalized. For the input exponent equal to zero, EXP [0] (the LSB) is set to 1 in order to properly handle denormalized numbers. Abnormal states, i.e. NaN (Not a Number) or infinity, are also detected and propagated to the accumulator output at this stage. Then at stages 4 and 3, the mantissa is shifted left by 0 to 15 bits according to EXP [3:0] to obtain a base-16 representation. At pipeline stage 2 and for the single precision format, the input mantissa is shifted right by 0, 16, 32 bits or cleared according to the difference between the input exponent EXP [E-1:4] and the predicted accumulator exponent. For the double precision format, the input mantissa should be shifted right by 0, 16, 32, 48 or 64 bits. At this stage, calculation accuracy may be improved by extending the mantissa (on the LSBs side) by additional G guard bits. These G bits significantly reduce the round-off error while improving calculation accuracy.

After the right shift, the input mantissa is converted to one's complement format (every individual bit is inverted), provided that the sign bit is equal to one – the sign-magnitude floating point mantissa format is converted to two's complement one. The actual accumulation process is performed at stage 1. The accumulator is extended by 3 bits at the MSB side: one bit due to operating in two's complement format; two bits are the overflow protection bits. These two overflow protection bits are employed as the accumulator is strongly pipelined, thus one overflow bit does not protect it from catastrophic overflow in the case where two consecutive maximum value input data are added. Extending the adder by two overflow bits gives a margin of three-clock cycle latency to react to the overflow conditions.

3. Conversion to floating point format

3.1. *Acc2float* module. The intermediate result of accumulation, which is produced by the presented accumulator, is converted to the IEEE-754 floating point representation in the module denoted as *acc2float.* A block diagram of this module is presented in Fig. 2. The *acc2float* module operates as follows: at pipeline stage 0, input mxantissa is converted from the two's complement to sign-magnitude format; then the sign-magnitude mantissa is normalized, i.e. the most significant bit equal to one is found and the mantissa is shifted accordingly. The shift

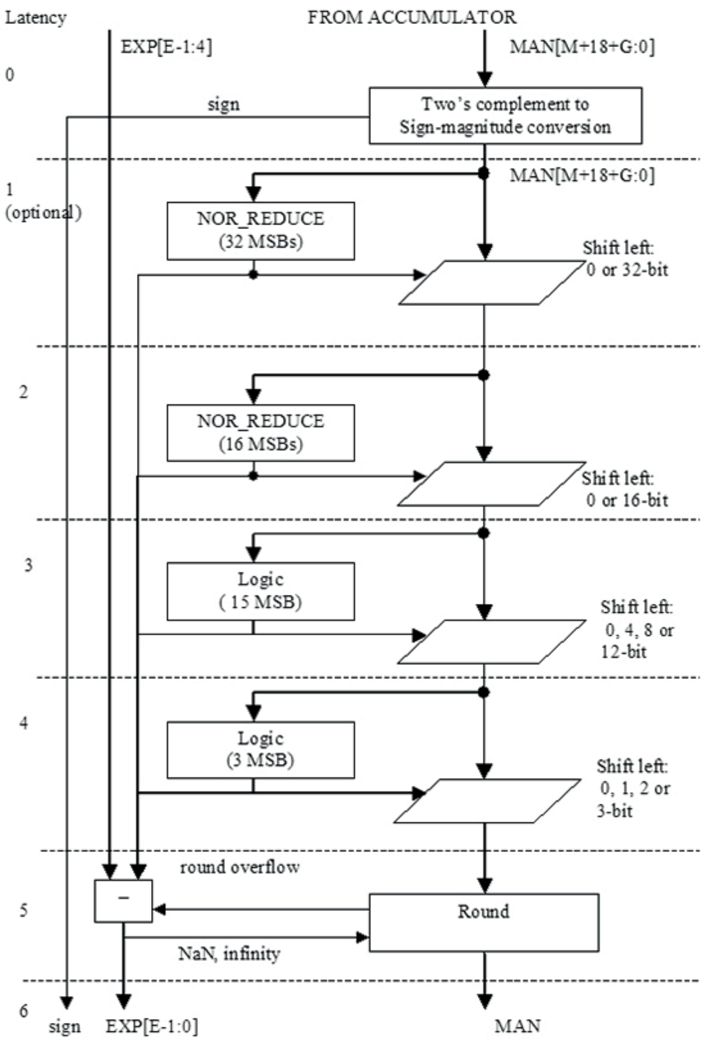

Fig. 2. Block diagram of the *acc2float* module

operation of unrestrained value is performed iteratively in the four pipelined steps that are organized as the 32, 16, (8-and-4) and (2-and-1) bit shifts.

Min_valid_bit is an additional parameter of the *acc2float* module. This parameter defines the minimum number of valid bits of the input mantissa. Consequently, if the value of the mantissa is smaller than $2^{min\_valid\_bit-1}$, a catastrophic cancellation event is initiated. A circuit for detection of a catastrophic cancellation is small as it contains only an OR gate, which tests whether at least one bit on the left of MAN[*min_valid_bit* – 2] is equal to one. An alternative solution is to check if the aggregated shift performed in the *acc2float* module is greater than *din_man_width* – *min_valid_bit*, where *din_man_width* is the *acc2float* input mantissa bitwidth. In FPGA implementation, the latter solution requires one 6-input LUT and one pipeline register per bit only.

It should be noted that the 32-bit shifter is optional, and it is implemented if: $din\_man\_width > 31 + min\_valid\_bit$. A shift by at a maximum of 31-bits can be obtained at pipeline stages 2 to 4, therefore larger shifts require an additional 32-bit shifter at pipeline stage 1.

3.2. Parallel addition / Accumulations. The accumulator presented herein can be used in various computing data paths, and it generates more efficient processing architectures. For instance, two or more input data can be accumulated simultaneously in a single clock cycle when additional floating point adder(s) are employed. An example of two-input parallel accumulation is given in Fig. 3.

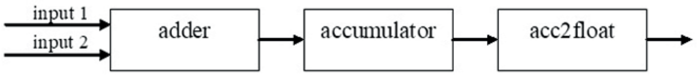

Fig. 3. Two-input accumulation in a clock cycle

In this scheme, a simplified floating point adder should be employed, as a standard floating point adder normalizes the result according to the floating point standard, which may cause a loss of the catastrophic cancellation information. Consequently, the simplified floating point adder is similar to a standard floating point adder, such as the one proposed in [23], but final normalization is skipped. The normalization process is performed later by the *acc2float* module. The lack of the normalization process reduces the required hardware resources of the adder, and this is the main feature for which the simplified adder was previously used. In this paper, the simplified adder is also used for detection of catastrophic cancellations. The lack of normalization causes the adder's result exponent to become equal to the greater of the input exponents. The mantissa of the adder's result is in the range of –4 to 4 (it is from –2 to 2 in a standard floating point adder). As normalization is skipped, the introduction of three additional mantissa bits, i.e. i_2, i_1 and i_0, is necessary to properly handle both overflow and underflow events. The simplified adder output F_{add} is represented as follows:

$$F_{add} = (i_2 i_1 i_o . f_1 f_2 \ldots f_k) \cdot 2^e, \tag{1}$$

where: i_2 i_1 i_0 – integer mantissa part bits in two's complement format,

$f_1, f_2, \ldots f_k$ – fractional mantissa part bits (standard mantissa bits),

e – exponent (maximum of the input exponents).

For adding 4 parallel inputs, 3 similar adders should be employed. The final adder's result is in a range from –8 to 8, thus four additional integer bits are required. The mantissa of the adder result F_{add} is represented in two's complement rather than the sign-magnitude format, thus mantissa format conversion is not required in the adder. However, the sign-magnitude format would simplify the normalization process.

4. Error estimation

4.1. Detection of catastrophic cancellation. The main feature of the proposed architecture is that the event of the catastrophic cancellation is automatically detected on the output of the hybrid floating-fixed point accumulator. Thus, whenever the mantissa value is equal to zero or close to zero, and the exponent is different from zero on the accumulator output, a catastrophic cancellation occurs. It should be noted that the catastrophic cancellation can be cancelled, i.e. in some cases, it does not influence the final result, unlike the NaN. For example, $(1.00000001 - 1.00000000) + 0.99999999$ will generate a proper result equal to 1.00000000, even though the intermediate result $(1.00000001 - 1.00000000)$ may generate the catastrophic cancellation event.

A similar solution was proposed for the quadrupled precision sticky accumulator [24]. However, the exact hardware architecture for the sticky accumulator was not presented. In our accumulator, detection of the catastrophic calculation is a by-product, which significantly simplifies the accumulator architecture. In [17, 18], a similar hybrid floating-fixed point accumulator was proposed yet detection of catastrophic cancellation was not considered.

To reduce the catastrophic cancellation occurrence, the number of accumulator guard bits g (see Fig. 1) can be increased. The increasing of g significantly improves calculation accuracy, especially when a large number of values are accumulated. For example, adding more than 2^{24} single precision numbers may result in a large error, but increasing g may significantly reduce this error. When g is large enough, our architecture can become an alternative to higher precision accumulators, e.g. the proposed single precision accumulator can replace the double precision or exact accumulation [14]. Replacement of the exact accumulator with our accumulator significantly reduces hardware resources. In the exact accumulator, either the LSBs influences the final result in an insignificant manner or MSBs are not used (zero). The proposed accumulator with large g offers advantages of both the floating point data format and the exact accumulator. Unlike in the case of exact accumulation, the proposed accumulator does not guarantee that the final result is independent of the accumulation order. Nevertheless, the calculation order has an insignificant effect on large g (effect similar to that for the exact accumulator). Besides, the proposed accumulator detects a catastrophic cancellation which is closely related to the cases when a calculation order might significantly influence the final result.

The exact accumulator guarantees the correct result provided that the input values carry no errors, e.g. round-off errors. Let's consider the following operation for the following single precision numbers: $\left(2^{64}+1\right)-2^{64}$. The proposed accumulator results in the catastrophic cancellation event. The exact accumulator gives a *proper* result, i.e. one. Nevertheless, the rounding error for the number, 2^{64}, is much greater than one. Therefore, to obtain a correct result, the input mantissa width should be at least 64-bit, which is not the case for single precision numbers. Summing up, the exact accumulator still generates erroneous and random results.

4.2. Cancellation-effect input error magnification. The accumulator described herein can signal when the input error (e.g. round-off error) significantly disturbs the final result, i.e. when the catastrophic cancellation magnifies the input error. This error is further denoted as cancellation-effect input error magnification (CIEM). As it will be proven, detection of CIEM can be achieved by monitoring the maximum value of inputs' exponents. According to Fig. 1, the input exponent, after truncating four LSBs, is compared to the accumulator exponent, and the greater of the values is selected. This operation is similar to the finding of the maximum exponent value, thus our accumulator exponent roughly represents the maximum input exponent. However, as the input's exponent is deprived of the four LSBs, this results in a rather inaccurate detection of the maximum input exponent. Besides, the accumulator overflow causes the accumulator exponent to become enlarged. Summing up, the accumulator described herein detects the CIEM effect with limited accuracy.

In the case where the number of accumulated inputs N is relatively small, it is possible to build a tree of adders to add many input elements in parallel without employing the proposed accumulator, as has been described in Section 1. The exponent field is fully represented in the adders tree (unlike the base-16 format employed in the proposed accumulator), thus the maximum exponent value is straightforwardly obtained. This allows for more accurate catastrophic cancellation detection as compared with the accumulator. Therefore it may be used as an alternative solution for parallel arithmetic, presented in e.g. [25, 26], which requires significant hardware overheads in comparison with straightforward arithmetic.

Much better results can be obtained when the maximum value of input exponents is monitored by dedicated logic. When the difference between the maximum input and the final accumulator exponent value is greater than the arbitrarily selected threshold t, the CIEM event is generated. The monitoring logic is very simple and requires insignificant hardware overheads.

The mathematical background for the above consideration is as presented below. Let us consider accumulation of N input numbers $x_1, x_2, \ldots, x_N$:

$$y=\sum_{i=1}^{N} x_i . \tag{2}$$

Each number x_i can be represented as:

$$x_i=\left(m_i \pm \Delta_i\right) \cdot 2^{e_i} \tag{3}$$

where: m_i is the mantissa, e_i is the exponent and Δ_i is the mantissa error.

If Δ_i is limited only by the round-off error, then $\Delta_i \leq 0.5$ LSB $=2^{-M-1}$; where M is the number of mantissa bits. Consequently, the maximum accumulation error Δ_{MAX} can be upper-bounded by:

$$\Delta_{\mathrm{MAX}} \leq \sum_{i=1}^{N} 2^{-M-1} \cdot 2^{e_i} \leq N \cdot 2^{-M-1} \cdot 2^{e_{\max}} \tag{4}$$

where: e_{max} is the maximum exponent value of e_i.

The main drawback of the above method is that the maximum error Δ_{MAX} may be significantly overestimated. This proves true especially in the case of large N and an average exponent value significantly lower than e_{max}. The advantage of this method lies in the insignificant hardware overheads, i.e. the calculation of maximum value $e_{max}=\mathrm{MAX}(e_i)$ requires insignificant hardware.

4.3. Exponent-based maximum error estimator (EMEE). The maximum error can be more accurately estimated by the following formula:

$$\Delta_{\mathrm{MAX}} \leq \sum_{i=1}^{N} 2^{-M-1} \cdot 2^{e_i}=2^{-M-1} \cdot \sum_{i=1}^{N} 2^{e_i} . \tag{5}$$

The hardware structure for calculations given by (5) is further denoted as an exponent-based maximum error estimator (EMEE). The core of the EMEE is an evaluation of $\sum_{i=1}^{N} 2^{e_i}$. At first approach, the EMEE can be seen as an accumulator of floating point numbers for which the mantissa value is hardwired to 1.0 $\left(\text{or } 2^{-M-1}\right)$. Therefore only one bit of mantissa is non-zero. The hardware requirements for the floating point accumulation are relatively high and, consequently, several simplifications were introduced to the EMEE. The most significant is the internal number representation, as the floating point representation suffers from the time-critical loop. Therefore, similarly to the proposed floating point accumulator, the EMEE also introduces the hybrid floating-fixed point data format. Nevertheless, instead of a base-16, a much smaller base-4 is selected, thus 2 LSBs of the intermediate exponent are hardwired to zero. Besides, the EMEE mantissa bitwidth can be significantly smaller, e.g. 4 to 16 bits, as the accumulation error needs not be calculated as accurately as the accumulation value.

When adding (accumulating) two floating point numbers, the maximum exponent e_{max} of these two input numbers is first evaluated and then the mantissa associated with the smaller exponent e_i is shifted right by $(e_{max} - e_i)$ bits. Then these two mantissas are added. Similar calculations are performed for the EMEE module presented in Fig. 4. The maximum of input e_i and current exponent e_{max} is evaluated at pipeline stage 1. At pipeline stage 2, mantissa shifting and accumulation take place. The input number mantissa is hardwired to 1.0, therefore instead of the shifter, the binary to one-hot transcoder is used to form input mantissa m_i. Then mantissa m_i is accumulated with the accumulated mantissa m_{sum}. Accumulator m_{sum} is the mantissa part of the hybrid floating-fix point representation of the result.

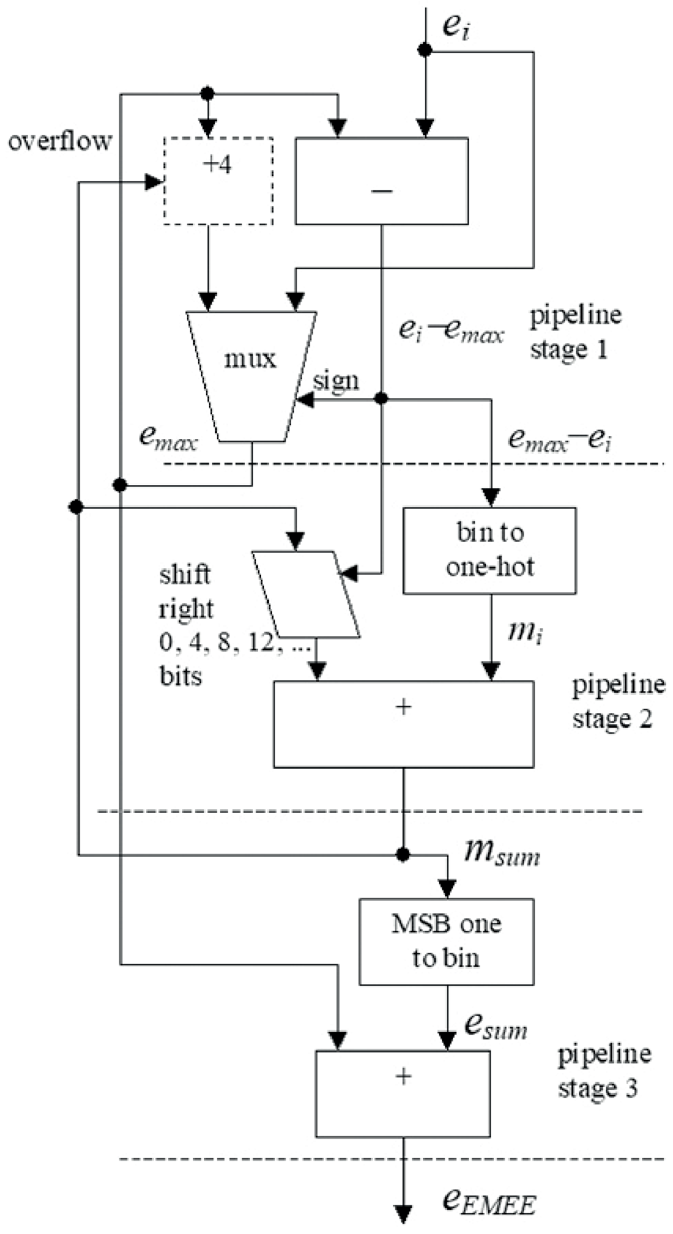

Fig. 4. Block diagram of exponent–based maximum error estimator (EMEE)

The width of the m_{sum} can be parameterized by a designer (parameter *width* in Table 5 and Table 6) and its bitwidth is usually set between 4 and 16, plus 3 bits due to base-4 representation, plus 1 bit due to overflow protection. Therefore, the total bitwidth of the m_{sum} is 4 bits wider than the parameter *width*. In the case where the accumulator overflows or the input exponent e_i is greater than the maximum exponent e_{max}, the accumulation result m_{sum} is shifted right. This path is time-critical, therefore base-4 was introduced; thus, the shifter shifts only by the multitude of 4 bits, e.g. 0, 4, 8, 12, … bits.

The EMEE final result should be coded similarly to its input, i.e. the output mantissa should be hardwired to 1.0. Consequently, at the end of the accumulation process (pipeline stage 3), the most significant *one* bit of m_{sum} is found and transcoded from the one-hot to binary code to form exponent e_{sum}. This exponent e_{sum} is then added to the maximum exponent e_{max} to form the output value e_{EMEE}.

The difference between the EMEE output, e_{EMEE}, and the floating point accumulator exponent, e_{acc}, represents how many ULP bits of the final accumulator are incorrect due to the cancellation.

At first glance, the hardware requirements of the EMEE are high – similarly to the accumulator. Nevertheless, the input and output data bitwidth (exponent) and the intermediate bitwidth are significantly smaller. The exponent width is usually only 8-bit (single) or 11-bit (double precision). Similarly, the internal EMEE accumulator (signal m_{sum}) width can be 4–16 bits. For the input exponent, e_i, much lower than maximum exponent e_{max}, the mantissa is incremented at the LSB – always rounding up. Therefore, shortening the EMEE accumulator width causes the EMEE module to overestimate the calculation error more. Nevertheless, this overestimation is insignificant: e.g. for the number of accumulated inputs equal to 2^{16}, and the 16-bit accumulator, the worst case possible error overestimation is doubling the error.

In this section the typical round-off error was considered mainly, i.e. the maximum error was assumed to be 0.5 ULP. However, it is also possible to assume another value of the error. The only constraint is that the error must be upper-bounded by the value of ULP that is a power of two. In the case where the maximum error (expressed in ULP) for every input is the same, the final result of the EMEE can be proportionally increased, i.e. the parameter *min_valid_bit* can be increased accordingly. Otherwise, every input should be represented by two values: the standard floating point and the error expressed in the exponent value which is processed by the EMEE module. This requires only additional input data width, and no extra hardware is required to compute the maximum error.

The approach of two-values-representation of inputs can be employed to calculate approximated error of complex operations. For example, multiplication of two matrices $(\boldsymbol{A} \cdot \boldsymbol{B})$ can generate a catastrophic cancellation, which can be roughly tracked by the proposed accumulator alone. Nevertheless, multiplication of three matrices $(\boldsymbol{A} \cdot \boldsymbol{B} \cdot \boldsymbol{C})$ requires two cascaded matrix multiplication operations: $\boldsymbol{P} = (\boldsymbol{A} \cdot \boldsymbol{B})$ and then $\boldsymbol{P} \cdot \boldsymbol{C}$. These cascaded operations may generate two consecutive cancellations which aggregate onto one another. Therefore the second (final) EMEE result (*EMEE2*) is fed by the result of the first EMEE (*EMEE1*) module plus an exponent part of an appropriate element $ec_{i,j}$ of the matrix $\boldsymbol{C}$.

$$EMEE2_{i,j} = \sum_{k=1}^{N} \left(EMEE1_{k,j} + ec_{j,k} + 1 \right) \tag{6}$$

where

i, j, k – index of matrix element

N – matrix dimension.

This holds as the product $p \cdot c$ can be approximated by:

$$\begin{aligned} p \cdot c &= \left(mp \cdot 2^{ep} \pm 2^{EMEE1} \right) \cdot \left(mc \pm \Delta c \right) \cdot 2^{ec} \approx \\ &\approx \left(mp \cdot mc \cdot 2^{ep+ec} \right) \pm mp \cdot 2^{EMEE1+ec} \end{aligned} \tag{7}$$

where:

mp, mc – mantissa, (upper-bounded by 2)

ep, ec – exponent

2^{EMEE1}, Δc – error of input p, c.

5. Implementation results

The implementation results are given for the Xilinx Virtex-6 xc6vlx75t-ff484–3 FPGA, ISE 12.2 development tools, and a differing number of guard bits g. The implementation results are expressed in the number of look-up tables (LUTs), registers (flip flop – FF), and the minimum clock period T. As it can be seen in Tables 1, 3 and 4, the resources applied for the proposed accumulators and the floating point adders, all from Xilinx's CORE Generator, are similar. In comparison to Table 1, Table 2 gives only the implementation results of the proposed accumulator, and the *acc2float* module is not included. *The acc2float* conversion module occupies a significant amount of FPGA resources. However, for matrix multiplications, for example where several accumulators are usually used in parallel, many simplified accumulators and only the one *acc2float* module can be used [13]. This holds as every accumulator utilizes a large number of input values in order to produce a single output value. An example of such a system is given in Fig. 5.

Table 1
Proposed accumulator (with the *acc2float* module) for the single precision floating point format and a different number of the accumulator guard bits g

g	**0**	**8**	**16**	**32**
LUT	408	468	528	581
FF	349	426	495	566
T [ns]	4.40	4.18	3.62	4.13

Table 2
Implementation results for the proposed accumulator (without *acc2float* module) for the single precision floating point format

g	**0**	**8**	**16**	**32**
LUT	233	249	275	341
FF	184	200	259	313
T [ns]	3.891	3.56	4.16	5.04

Table 3
Implementation results for the proposed accumulator (with *acc2float* module) for the double precision floating point format

g	**0**	**8**	**16**	**32**
LUT	761	887	887	960
FF	733	842	729	777
T [ns]	5.33	5.50	5.23	4.84

Table 4
Implementation results for the floating point adders created by the Xilinx's CORE Generator tool [1]

width / optimization	**32 / high speed**	**32 / low latency**	**64 / high speed**	**64 / low latency**
LUT	404	507	730	977
FF	546	615	944	1,157
T [ns]	1.834	2.274	2.775	3.375

Table 5
EMEE implementation results for the 8-bit exponent (for the single precision floating point format)

width	**4**	**8**	**16**
LUT	75	96	130
FF	37	43	52
T [ns]	2.33	2.43	2.54

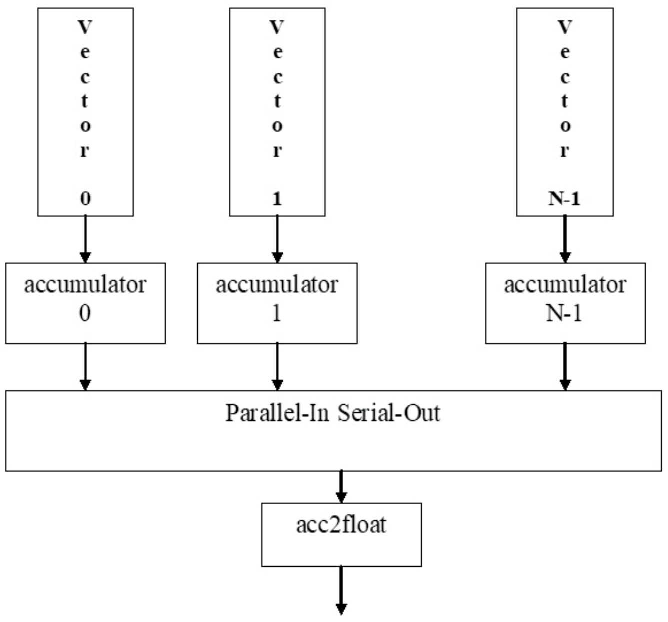

Fig. 5. Block diagram of the system employing several simplified accumulators and only one *acc2float* converter

Table 6
EMEE implementation results for the 11-bit exponent
(for the double precision floating point format)

width	4	8	16
LUT	100	122	155
FF	46	53	61
T [ns]	3.13	3.15	3.26

Table 7
Implementation results for the floating point full-range
and quarter-range accumulator provided by Xilinx [27] (Kintex-7)

	Single precision		Double precision	
	Full range	Quarter range	Full range	Quarter range
LUT	2980	921	31 142	6538
FF	3424	1183	24 340	7056
f [MHz]	436	465	338	421

Summing up, the proposed accumulator significantly simplifies system design in comparison to floating point addition, and therefore the accumulator loop-back signal is not time-critical although it usually is for floating point adders. Besides, the proposed accumulator may significantly reduce the hardware requirements in parallel calculations.

The given accumulator can also be compared with the full or limited range accumulator [13] described in Section 1 and provided by Xilinx as a full range or quarter range solution [27]. It can be seen from Table 1 and 3 (the proposed accumulator) vs. Table 7 (Xilinx design) that the proposed accumulator offers significant reduction of hardware resources.

It should be noted that the modules described herein are a compromise between clock frequency and occupied hardware resources. Nevertheless, they meet most system clock frequency requirements. Otherwise, extra pipeline stages can be added. It can be seen in Table 1 and Table 2 that the *acc2float* module significantly reduces the maximum clock frequency – this module is the source of the maximum signal propagation time. The *acc2float* module can be significantly sped-up by introducing a higher level of pipelining. The pipeline's stage 1 in Fig. 1 will be denoted as *facc_core* in further discussion. In the proposed design, the *facc_core* module contains the time-critical accumulation loop, but this does not influence the clock period. This is very fortunate as only this module cannot be easily sped-up by introduction of additional pipeline stages. The authors' experiments showed that FPGA hardware resources and maximum signal propagation time in *facc_core* are roughly the same as for the ripple-carry adder of the same bitwidth, i.e. roughly one LUT and 20 ps for an additional carry-chain bit. The total hardware resources for *facc_core* are 100 LUTs, and its minimum clock period is 3.62 ns for 100 bitwidth (29 guard bits for the double precision format).

When the critical accumulation loop significantly slows down the operation frequency, the carry-save adder, which is preferred in FPGAs, can be employed instead of standard ripple-carry adders [19]. It must be noted also that the minimum clock period reported here is a rough value only, and it can vary for different implementations.

The hardware requirements for the exponent-based maximum error estimator (EMEE) and the different internal accumulator widths are given in Table 5 and Table 6.

6. Conclusions

The proposed floating point accumulator presents a novel approach to reducing the delay of the critical accumulation loop. Only the adder and 16-bit shifter are employed inside the loop. Consequently, the proposed floating point accumulator can be efficiently employed in many applications as the hardware requirements for the accumulator and the corresponding adder are very similar or even lower for parallel version of the proposed accumulator. It should be noted that for fixed point arithmetic an accumulator is commonly used, and for DSPs multiply and accumulate (MAC) operations rather than multiply and add operations are commonly employed. Another aspect of the presented accumulator is a detection of the catastrophic cancellation, which is an extra advantage of the presented architecture. As a result, out-of-order (parallel) accumulation might be employed, as in the case where an addition order influences the final result, as indicated by the catastrophic cancellation event. The alternative solution to parallel arithmetic is given in [25], but it requires significant hardware overheads as compared to straightforward arithmetic.

It should be noted that addition is a critical operation due to the (catastrophic) cancellation. Floating point multiplication, division, etc. guarantee proper results within 0.5 ULP. Even more complicated mathematical operations, such as exponent calculations, guarantee maximum error of 1 ULP [28, 29]. Therefore, the error evaluation of the accumulation operation is crucial for obtaining proper results. Consequently, the idea behind the presented paper is to stop complying with the IEEE 754 standard, which does not guarantee proper results but only, in theory, the same (possible incorrect) results on different computing machines. In [30], a software application detects cancellation in order to balance the speed and accuracy of floating point operations. This paper presents a circuit for rough error evaluation, which might serve as a foundation for similar hardware approaches: limiting the bitwidth (precision) of floating point operations and roughly evaluating the calculation error. When a calculation error is unacceptable, increasing the floating point precision is required. A similar software approach was proposed in [31]. The calculation error might even be a threshold to generating sparse matrix operations to increase the calculation speed [4].

Acknowledgements. This work has been supported by AGH University of Science and Technology Statutory Activity, grant No. 11.11.230.017, and by the National Science Centre (NCN) grant No. DEC-2011/01/B/ST6/03024.

References

[1] Xilinx Inc., "Floating point Operator v5.0", DS335 June 24, 2009, www.xilinx.com.

[2] Altera Ltd. "Floating-Point IP Cores User Guide", UG-01058, 2016.12.09, www.altera.com.

[3] A. Gramacki, M. Sawerwain, and J. Gramacki, "FPGA-based bandwidth selection for kernel density estimation using high level synthesis approach", *Bull. Pol. Ac.: Tech* 64(4), 821–829 (2016).

[4] E. Jamro, T. Pabiś, P. Russek, and K. Wiatr, "The algorithms for FPGA implementation of sparse matrices multiplication", Computing and Informatics 33(3), 667–684 (2014).

[5] M. Karwatowski, P. Russek, M. Wielgosz, S. Koryciak, and K. Wiatr, "Energy efficient calculations of text similarity measure on FPGA-accelerated computing platforms", *Parallel Processing and Applied Mathematics*, PPAM, Lecture Notes in Computer Science, Springer, Cham 9573, 31–40 (2015).

[6] E. Jamro, P. Russek, A. Dabrowska-Boruch, and et al., "The implementation of the customized, parallel architecture for a fast word-match program", *Computer Systems Science And Engineering* 26, 285–292 (2011).

[7] M. Wielgosz and M. Pietron, "Using spatial pooler of hierarchical temporal memory to classify noisy videos with predefined complexity", *Neurocomputing* 240, 84–97 (2017).

[8] A. Głowacz and Z. Głowacz, "Recognition of rotor damages in a DC motor using acoustic signals", *Bull. Pol. Ac.: Tech* 65(2), 187–194 (2017).

[9] A.R. Lopes and G.A. Constantinides, "A fused hybrid floating point dot-product for FPGAs", ARC'2010 Bangkok, Thailand, LNCS 5992, March 2010, 157–168.

[10] M. de Lorimier and A. De Hon, "Floating point sparse matrix- vector multiply for FPGAs", *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 75–78 (2005).

[11] L. Zhuo, G.R. Morris, and V.K. Prasanna, "Designing scalable FPGA-based reduction circuits using pipelined floating point cores", *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'05 (2005).

[12] X. Wang, S. Braganza, and M. Leeser, "Advanced components in the variable precision floating point library", 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (2006).

[13] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating point accumulation and sumof-products", ICECE Technology, FPT 2008 10 Dec, 33−40 (2008).

[14] U. Kulisch, "Very fast and exact accumulation of products", *ISC 2010*, Hamburg, (2010).

[15] Y. Uguen and F. de Dinechin, "Designs pace exploration for the Kulisch accumulator", 2017. <hal-01488916v2>.

[16] IFIP WG, "IEEE P1788 letter", dated Sep. 9 2009.

[17] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating point accumulations in configurable hardware with delayed addition techniques", *IEEE Transactions On Computers* 49 (3), 208–218 (2000).

[18] S.R. Vangal, Y.V. Hoskote, N.Y. Borkar, and A. Alvandpour, "A 6.2-GFlops Floating point Multiply-Accumulator with Conditional Normalization", *IEEE Journal of Solid-State Circuits* 41 (10), 2314–2323 (2006).

[19] A. Paidimarri, A. Cevrero, P. Brisk, and P. Ienne, "FPGA implementation of a single-precision floating point multiply-accumulator with single-cycle accumulation", *17th IEEE Symposium on Field Programmable Custom Computing Machines*, Napa, 267–270 (2009).

[20] S. Jain, V. Erraguntla, S.R. Vangal, Y. Hoskote, N. Borkar, T. Mandepudi, and Karthik VP, "A 90mW/GFlop 3.4GHz reconfigurable fused/continuous multiply-accumulator for floating point and integer operands in 65nm", VLSID, 23rd International Conference on VLSI Design, 2010, pp. 252–257.

[21] T.O. Bachir and J.-P. David, "Performing floating point accumulation on a modern FPGA in single and double precision", *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 105–108 (2010).

[22] K.K. Nagar and J.D. Bakos, "A High-Performance Double Precision Accumulator", *Proc. IEEE International Conference on Field-Programmable Technology (IC-FPT'09)*, Dec. 9–11, 2009, 500–503.

[23] J. Liang, R. Tessier, and O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs", Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03), 185–194.

[24] M. Daumas and D.W. Matula, "Validated Roundings of Dot Products by Sticky Accumulation", *IEEE Transactions on Computers* 46(5), 623–629 (1997).

[25] N. Kapre and A. DeHon, "Optimistic parallelization of floating point accumulation", 18th IEEE Symposium on Computer Arithmetic. Proceedings/Symposium on Computer Arithmetic. IEEE, Los Alamitos, CA, 2007, 205–216.

[26] J. Demmel and H.D. Nguyen, "Parallel reproducible summation", *IEEE Trans. on Computers* 64(7), 2060–2070 (2015).

[27] Xilinx Inc., "Performance and resource utilization for floating-point, v7.1", https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html.

[28] E. Jamro, K. Wiatr, and M. Wielgosz, "FPGA implementation of 64-bit exponential function for HPC", 2007 International conference on Field programmable logic and applications (FPL), Amsterdam, Netherlands, August 27–29, 2007, 718–721.

[29] J.M. Muller, *Elementary Functions, Algorithms and Implementation*, Brikhauser, Boston, 2005.

[30] M.O. Lam, J.K. Hollingsworth, and G.W. Stewart, "Dynamic Floating point Cancellation Detection", *Parallel Computing* 39(3), 146–155 (2013).

[31] J.R. Shewchuk, "Robust adaptive floating point geometric predicates", *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, 141−150 (1996).