(54) **METHOD AND SYSTEM FOR CALCULATING DOT PRODUCTS**

(57) A method of performing dot product of an array of '2k' floating point numbers comprising two sets of k floating-point numbers a_i and b_i is disclosed. The method includes receiving both sets of 'k' floating point numbers and multiplying each floating point number a_i with a floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits. The method further comprises creating a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits. Further the method includes identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i) of each pair of floating point numbers a_i and b_i, aligning the magnitude bits of the numbers (y_i) based on the maximum exponent sum (e_{max}) and adding the set of 'k' numbers concurrently to obtain the dot product.
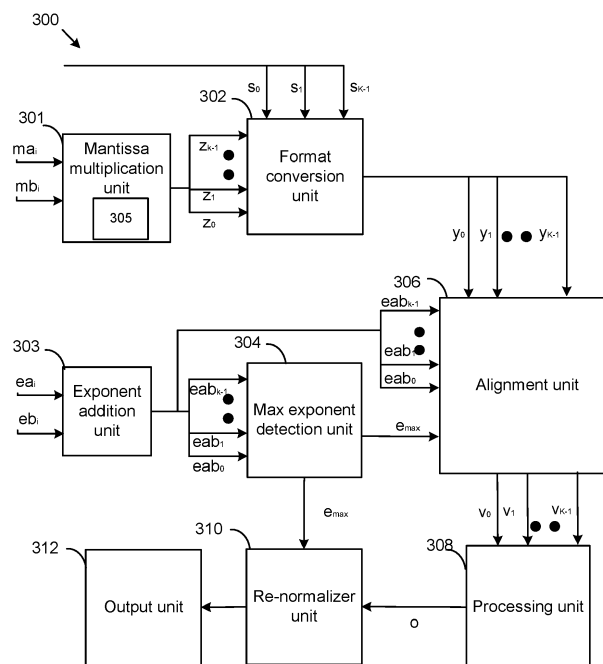
FIG. 3

Description

CROSS-REFERENCE TO RELATED APPLICATIONS

5   **[0001]**   This application claims priority from UK patent applications 2202126.5 and 2202128.1, each filed on 17 February 2022, and which are herein incorporated by reference in their entirety.

BACKGROUND

10   **[0002]**   Most computing systems use number formats, typically in binary notation or base 2, for performing various computations. These number formats include fixed-point or floating-point number formats. Fixed point number formats can provide additional precision but is used to represent only a limited range of values. Therefore, floating point number formats are used in most of the modern computing systems to provide a trade-off between range and precision.
[0003]   A floating-point number comprises a mantissa (m) having a bit length of 'b' bits, an exponent (e) having a bit

15   length of 'a' bits and optionally a sign bit (s) to represent a binary number. In some widely used formats the exponent is biased (i.e. offset) by a value (c) so as to represent numbers smaller than 1 and is used to encode exceptional values at its end points. For non-extremal values of e, the floating-point number x is said to be normalized and the number x is represented as $(-1)^s 2^{e-c}(1 + 2^{-b}m)$. Thus, floating point numbers can be used to represent very small or very large numbers precisely using scientific notation, in binary or in some other base. The use of floating-point numbers in arithmetic

20   computations provides varying degrees of precision depending on the bit length or type of floating-point format used.
[0004]   Computations involving convolution/dot products of large arrays of real valued numbers occur commonly in the solution of various numerical problems. The dot product of large arrays or two sets of numbers (a_0, a_1, a_2... a_{k-1}) and (b_0, b_1, b_2... b_{k-1}) is defined as

25

$$\mathrm{a.b} = \sum_{i=0}^{k-1} a_i b_i$$

[0005]   It is therefore advantageous to have hardware dedicated to performing dot products in high performance computing systems, graphic processing systems, neural network accelerators and the like. Conventionally, there are different

30   ways to achieve this, with different benefits and drawbacks.
[0006]   A known method in computing systems, to do dot product of two arrays/ sets of floating-point numbers, is by using separate floating point multiplication and floating point addition. A dot product unit 100 using this principle is shown in FIG.1. The dot product unit 100 comprises a set of floating point multiplication units 102a, 102b, 102c and 102d, and a set of floating point adder units 104a, 104b and 104c. The dot product unit 100 is implemented as a tree of floating

35   point multiplication units and addition units. The dot product unit 100 receives a first set of floating point numbers (a_1, a_2, a_3, and a_4) and a second set of floating point numbers (b_1, b_2, b_3, and b_4) as inputs. Consider that each number in the first set of floating point numbers a_i comprises a mantissa ma_i and an exponent ea_i. Similarly, each number in the second set of floating point numbers b_i comprises a mantissa mb_i and an exponent eb_i. Each floating point number a_i in the first set of floating-point numbers is provided as a first input to a respective one of floating point multiplication units

40   102a, 102b, 102c and 102d. Each floating point number b_i in a second set of floating-point numbers is provided as a second input to a respective one of floating point multiplication units 102a, 102b, 102c and 102d. Each floating point multiplication unit 102a, 102b, 102c and 102d performs the multiplication of floating point numbers a_i and b_i, to obtain a product C_i. Once the product c_i from each multiplication unit 102a, 102b, 102c and 102d is obtained, the results (floating point numbers) are accumulated by a series of adders in any dynamically or statically chosen order to obtain the output.

45   Thus, the adders can be arranged in any order. In one example, as depicted in the FIG.1 the product c_1 and c_2 from two consecutive floating point multiplication units 102a and 102b are provided to a first floating point adder unit 104a to add the products (c, and c_2). Similarly, the products c_3 and c_4 from the next two consecutive floating point multiplication units 102c and 102d are provided to a second floating point adder unit 104b to add the product (c_3 and c_4). Further, the accumulated values are further provided as an input to a third floating point adder 104c to be accumulated to obtain the

50   output y. The output from the multiplier or adder in each step is rounded, which causes a rounding error in the output generated by each step. The rounding error is a characteristic feature of floating-point calculations.
[0007]   Another known method in computing systems, to do dot product of two arrays/ sets of floating-point numbers, is by using fused multiplication and addition operations. A dot product unit 200 using this principle is shown in FIG. 2. The dot product unit 200 comprises a set of Fused Multiply and Add (FMA) units 202a, 202b, 202c and 202d. An FMA

55   unit performs the floating point multiplication and addition in a single step with a single rounding. Thus, FMA improves the speed and accuracy of computation of dot product involving accumulation of products. In FIG. 2, the results of one fused multiply add unit is provided as an input to another fused multiply add unit such that the product of two numbers is added to the product of the next two numbers. The dot product unit 200 receives both sets of floating point numbers

a_i and b_i as inputs. Each floating point number a_i in the first set of floating-point numbers is provided as a first input to a respective one of FMA units 202a, 202b, 202c and 202d. Each floating point number b_i in the second set of floating-point numbers is provided as a second input to a respective one of FMA units 202a, 202b, 202c and 202d. Each FMA 202a, 202b and 202c computes multiplication of numbers a_i and b_i and adds it with the results of a previous FMA with a single rounding (N.B. in the case of FMA 202d there is no 'previous' FMA, as FMA 202d is at the top of tree, and so the addition operation simply adds the result of the multiplication operation to zero). For example, as depicted in FIG.2 the FMA 202d receives the numbers a_4 and b_4 as the multiplicand inputs. Further a 0 input is provided as a third input. The FMA 202d multiplies a_4 and b_4 and adds the result with 0 to obtain the output d_4. Further, the FMA 202c receives the numbers a_3 and b_3 as the multiplicand inputs and d_4 as a third input. The FMA 202c multiplies a_3 and b_3 and adds the result with d_4 to obtain the output d_3. Similarly, FMA 202b obtains the output d_3 by multiplying a_2 and b_2 and adding the result with d_3. Further the FMA 202a obtains the final output $y=d_1$ by multiplying a_1 and b_1 and adding the result with d_2.

[0008] Thus, pairs of floating-point number a_i and b_i from both sets of floating point numbers are multiplied together and added to a previously calculated output to generate a new or cumulative output. In other words, the overall sum is performed as a sequence of multiplication and addition of numbers. A final output (y) generated after multiplying and adding all the floating-point numbers in the array is provided as the output.

[0009] Either of the above described methods can be implemented iteratively looping through the same unit(s) or concurrently by using a parallel or sequential composition of units. Whether using the first dot product unit 100 or 200, the precision of the output cannot be guaranteed as different ordering of inputs may yield different results, due to the intermediate rounding operations. Further, there is a high delay introduced due to the number of logic gates in the critical path.

[0010] Hence, existing methods and architectures for processing floating-point numbers have drawbacks.

SUMMARY

[0011] This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

[0012] A method of performing dot product of an array of '2k' floating point numbers comprising two sets of k floating-point numbers a_i and b_i is disclosed. The method includes receiving both sets of 'k' floating point numbers and multiplying each floating point number a_i with a floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits. The method further comprises creating a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits. Further the method includes identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i) of each pair of floating point numbers a_i and b_i, aligning the magnitude bits of the numbers (y_i) based on the maximum exponent sum (e_{max}) and adding the set of 'k' numbers concurrently to obtain the dot product.

[0013] According to a first aspect there is provided a method of performing dot product of an array of '2k' floating point numbers, $k \geq 3$, using a hardware implementation, the array comprising a first set of k floating-point numbers a_0, a,..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1...,b_{k-1}, wherein the method comprises: receiving both sets of 'k' floating point numbers; multiplying each floating point number a_i with a floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits; creating a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the bit length 'r' of the product numbers (z_i), wherein the 'n' bits comprises a number

$$r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x$$

of magnitude bits, wherein 'n' is bits, where x is an integer, and $x \geq 1$; identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i; aligning the magnitude bits of the numbers (y_i) based on the maximum exponent sum (e_{max}); and adding the set of 'k' numbers concurrently.

[0014] Optionally, each number in the first set of k floating-point numbers a_0, a_1..., a_{k-1} comprises a mantissa (ma_i) and an exponent (ea_i) and each number in the second set of k floating-point numbers b_0, b_1..., b_{k-1} comprises a mantissa (mb_i) and an exponent (eb_i), where each mantissa (ma_i) is having a bit length of 'p' bits and each mantissa (mb_i) is having a bit length of 'q' bits.

[0015] Optionally, multiplying each floating point number ai with the corresponding floating point number bi comprises multiplying mantissa (ma_i) and mantissa (mb_i) to obtain an intermediate mantissa product (mab_i).

[0016] Optionally, the method of performing a dot product emulates the precision obtained using separate multiplication and addition units, for performing dot product having an output mantissa bit length of P bits, by setting the mantissa bit length of 'r' bits as 'r=P+2' bits.

[0017] Optionally, the method emulates the precision obtained using fused multiplication and addition units, for performing dot product having an output mantissa bit length of Q bits, by setting the mantissa bit length of 'r' bits as 'r= max (Q+2, p+q+3)' bits.

[0018] Optionally, generating k product numbers (z_i) having the mantissa bit length of 'r' bits comprises: rounding, the bits of the intermediate mantissa product (mab_i) to r bits, if p+q+2>r bits; or padding, extra least-significant bits to the bit length of the intermediate mantissa product (mab_i) to generate r bits, if p+q+2 < r bits.

[0019] Optionally, identifying a maximum exponent sum (e_{max}) includes identifying the maximum value among k exponent sums (eab_i) where k exponent sums (eab_i) is obtained by summing exponent (ea_i) and exponent (eb_i).

[0020] Optionally, adding extra most-significant bits to the bit length 'r' of the product numbers (z_i) comprises adding at least $\lceil \log_2(k) \rceil$ number of the most-significant bits.

[0021] Optionally, adding extra least-significant bits to the bit length 'r' of the product numbers (z_i) comprises adding at least $\lceil \log_2(k - 1) \rceil + 1$ number of the least-significant bits.

[0022] Optionally, the method further comprises: calculating an output value by adding 'k' numbers (y_i); renormalizing the output value; and rounding the output value to represent the output value as a floating-point number.

[0023] Optionally, aligning the magnitude bits of the numbers (y_i) to be based on the maximum exponent (e_{max}) comprises the steps of, for each pair floating-point number (i): calculating the difference (ed) between the maximum exponent sum (e_{max}) and each exponent sum (eab_i); and shifting the magnitude bits of the corresponding number (y_i), to the LSB side, based on the calculated difference (e_d).

[0024] Optionally, further to shifting the magnitude bits of the numbers, the method further comprises performing rounding or truncating the bits of the numbers that are shifted outside the bit-length of the number.

[0025] Optionally, the method further comprises determining a two's complement of the magnitude bits of the numbers, based on a sign bit (s_i) of each corresponding number, if the set of 'k' floating point numbers comprises signed floating-point numbers.

[0026] Optionally, the set of 'k' floating point numbers in a first format comprises: unsigned floating point numbers which explicitly includes a leading bit; or unsigned floating point numbers which implicitly includes a leading bit or signed floating point numbers which explicitly includes a leading bit; or signed floating point numbers which implicitly includes a leading bit.

[0027] According to a second aspect there is provided a hardware implementation for performing dot product of an array of '2k' floating point numbers, $k \geq 3$, the array comprising a first set of k floating-point numbers a_0, a,..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1..., b_{k-1}, wherein the hardware implementation comprises a multiplication unit, a format conversion unit, a maximum exponent detection unit, an alignment unit, and a processing unit. The multiplication unit comprising a plurality of multiplier configured to: receive both sets of 'k' floating point numbers; and multiply each floating point number a_i with a corresponding floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits. The format conversion unit configured to create a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the bit length 'r' of the product numbers (z_i), wherein the 'n' bits comprises a number of magnitude bits, wherein 'n' is $r + \lceil \log_2(k) \rceil + \lceil \log_2(k - 1) \rceil + x$ bits, where x is an integer, and $x \geq 2$. The maximum exponent detection unit configured to identify a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number ai and the floating point number bi. The alignment unit configured to align the magnitude bits of the numbers based on the maximum exponent sum (e_{max}). The processing unit configured to add the set of 'k' numbers concurrently to generate an output value.

[0028] Optionally, the hardware implementation further comprises a renormalizing unit configured to: renormalize the output value; and round the output value to represent the output value as a floating-point number.

[0029] Optionally, each number in the first set of k floating-point numbers a_0, a_1..., a_{k-1} comprises a mantissa (ma_i) and an exponent (ea_i) and each number in the second set of k floating-point numbers b_0, b_1..., b_{k-1} comprises a mantissa (mb_i) and an exponent (eb_i), where each mantissa (ma_i) is having a bit length of 'p' bits and each mantissa (mb_i) is having a bit length of 'q' bits.

[0030] Optionally, the multiplication unit comprises a plurality of multiplier units configured to multiply concurrently each mantissa (ma_i) with corresponding mantissa (mb_i) to obtain an mantissa product (mab_i).

[0031] Optionally, the hardware implementation for performing a dot product operation emulates the precision obtained using separate multiplication and addition units, for performing dot product having an output mantissa bit length of P bits, by setting the mantissa bit length of 'r' bits as 'r= P+2' bits.

[0032] Optionally, the hardware implementation for performing a dot product operation emulates the precision obtained using fused multiplication and addition units for performing dot product having an output mantissa bit length of Q bits, by setting the mantissa bit length of 'r' bits as 'r= max (Q+2, p+q+3)' bits.

[0033] Optionally, the multiplication unit is configured to generate k product number (z_i) having the mantissa bit length of 'r' bits by: rounding, the bits of the intermediate mantissa product (mab_i) to r bits, if p+q+2>r bits; or padding, extra least-significant bits to the bit length of the intermediate mantissa product (mab_i) to generate r bits, if p+q+2 < r bits.

[0034] Optionally, the maximum exponent detection unit is configured to identify a maximum exponent sum (e_{max}) among k exponent sums (eab_i), where k exponent sums (eab_i) is obtained by summing exponent (ea_i) and exponent (eb_i).

[0035] Optionally, the alignment unit is configured to align the magnitude bits of the numbers to be based on the maximum exponent (e_{max}), wherein the alignment unit comprises a plurality of subtraction units and a plurality of shifter units. Each subtraction unit is configured to calculate the difference (e_d) between the maximum exponent sum (e_{max}) and exponent sum (eab_i). Each shifter unit configured to shift the magnitude bits of the corresponding number, to the LSB side, based on the calculated difference (e_d).

[0036] Optionally, the alignment unit is configured to further truncate the bits of the numbers that are shifted outside of the bit length of the numbers.

[0037] Optionally, the alignment unit further comprises a plurality of complementing units configured to determine two's complement of the magnitude bits of each number, based on a sign bit (s_i) of the corresponding number, if the set of 'k' floating point numbers comprises signed floating point numbers.

[0038] According to a third aspect there is provided a method of performing dot product of an array of '2k' floating point numbers, $k \geq 3$, using a hardware implementation, the array comprising a first set of k floating-point numbers a_0, a_1..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1...,b_{k-1}, wherein the method comprises: receiving both sets of 'k' floating point numbers; multiplying each floating point number a_i with a floating point number b_i , each multiplication generating a first intermediate product number (z_i') and a second intermediate product numbers (z_i''), thereby generating 2k product numbers comprising k first intermediate product numbers (z_i') and k second intermediate product numbers (z_i''), each having a bit length of 'r+1' bits; creating a set of '2k' numbers comprising k first numbers (y_i') and k second numbers (y_i''), based on the 2k product numbers, each having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the bit length of the product numbers (z_i and z_i''), wherein the 'n' bits

$$r + 1 + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x$$

comprises a number of magnitude bits, wherein 'n' is   bits, where x is an integer, and $x \geq 1$;identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number ai and the floating point number bi; aligning the magnitude bits of the numbers (y_i' and y_i'') based on the maximum exponent sum (e_{max}); and adding the set of '2k' numbers concurrently.

[0039] According to a fourth aspect there is provided a hardware implementation for performing dot product of an array of '2k' floating point numbers, $k \geq 3$, the array comprising a first set of k floating-point numbers a_0, a_1..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1...,b_{k-1}, wherein the hardware implementation comprises a multiplication unit, a format conversion unit, a maximum exponent detection unit, an alignment unit and a processing unit. The multiplication unit comprising a plurality of multiplier configured to: receive both sets of 'k' floating point numbers; and multiply each floating point number ai with a corresponding floating point number bi to generate a first intermediate product number (z_i') and a second intermediate product numbers (z_i''), thereby generating 2k product numbers comprising k first intermediate product numbers (z_i') and k second intermediate product numbers (z_i''), each having a bit length of 'r+1' bits. The format conversion unit configured to create a set of '2k' numbers comprising k first numbers (y_i') and k second numbers (y_i''), based on the 2k product numbers, each having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the bit length of the product numbers (z_i and z_i''), wherein the 'n' bits

$$r + 1 + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x$$

comprises a number of magnitude bits, wherein 'n' is   bits, where x is an integer, and $x \geq 1$. The maximum exponent detection unit configured to identify a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i. The alignment unit configured to align the magnitude bits of the numbers (y_i' and y_i'') based on the maximum exponent sum (e_{max}). The processing unit configured to add the set of '2k' numbers concurrently to generate an output value.

[0040] The hardware implementation hardware implementation for performing dot product according to the first aspect discussed above may be embodied in hardware on an integrated circuit. There may be provided a method of manufacturing, at an integrated circuit manufacturing system, a hardware implementation for performing dot product. There may

be provided an integrated circuit definition dataset that, when processed in an integrated circuit manufacturing system, configures the system to manufacture a hardware implementation for performing dot product. There may be provided a non-transitory computer readable storage medium having stored thereon a computer readable description of a hardware implementation for performing dot product that, when processed in an integrated circuit manufacturing system, causes the integrated circuit manufacturing system to manufacture an integrated circuit embodying a hardware implementation for performing dot product.

[0041]   There may be provided an integrated circuit manufacturing system comprising: a non-transitory computer readable storage medium having stored thereon a computer readable description of the a hardware implementation for performing dot product according to the first aspect discussed above.; a layout processing system configured to process the computer readable description so as to generate a circuit layout description of an integrated circuit embodying the hardware implementation for performing dot product; and an integrated circuit generation system configured to manufacture the hardware implementation for performing dot product according to the circuit layout description.

[0042]   There may be provided computer program code for performing any of the methods described herein. There may be provided non-transitory computer readable storage medium having stored thereon computer readable instructions that, when executed at a computer system, cause the computer system to perform any of the methods described herein.

[0043]   The above features may be combined as appropriate, as would be apparent to a skilled person, and may be combined with any of the aspects of the examples described herein.

BRIEF DESCRIPTION OF THE DRAWINGS

[0044]   Examples will now be described in detail with reference to the accompanying drawings in which:

FIG. 1 is a block diagram illustrating a conventional floating point dot product calculator having separate multiplication and addition units;

FIG. 2 is a schematic block diagram illustrating another conventional floating point dot product calculator having fused multiplication and addition units;

FIG. 3 is a block diagram illustrating an example of a hardware implementation for performing dot product operation;

FIG. 4a is a block diagram illustrating a mantissa for a floating-point number in an incoming format representation;

FIG. 4b is a block diagram illustrating a product number in a first intermediate format representation;

FIG. 4c is a block diagram illustrating a second intermediate format representation of a signed number;

FIG. 5 is a block diagram illustrating the different units in the hardware implementation for performing dot product explained in FIG. 3;

FIG. 6a to FIG. 6d illustrate an example of a floating-point number converted from a first format representation to a second format representation;

FIG. 7 is a flowchart illustrating a method of performing dot product of two sets of k floating-point numbers;

FIG 8 is a graph illustrating a comparison of the implementation of the architecture 300 with other standard architectures for processing a set of floating-point numbers;

FIG. 9 shows a computer system in which a dot product calculator is implemented;

FIG. 10 shows an integrated circuit manufacturing system for generating an integrated circuit embodying a dot product calculator; and

FIG. 11 illustrates the architecture 300 implementing a carry-save multiplication.

[0045]   The accompanying drawings illustrate various examples. The skilled person will appreciate that the illustrated element boundaries (e.g., boxes, groups of boxes, or other shapes) in the drawings represent one example of the boundaries. It may be that in some examples, one element may be designed as multiple elements or that multiple elements may be designed as one element. Common reference numerals are used throughout the figures, where

appropriate, to indicate similar features.

DETAILED DESCRIPTION

[0046] The following description is presented by way of example to enable a person skilled in the art to make and use the invention. The present invention is not limited to the embodiments described herein and various modifications to the disclosed embodiments will be apparent to those skilled in the art.

[0047] Embodiments will now be described by way of example only.

[0048] As explained above, the conventional hardware for performing a dot product of two sets of numbers includes hardware implementing separate multiplication and addition units or one implementing fused multiplication and addition units. The multiplicands a_i and b_i from both sets of floating point numbers may be represented in an incoming format F comprising a mantissa and an exponent. The mantissa ma_i and mb_i each comprise a bit length of p bits when represented in the incoming format F. The output of the multiplication unit may have a floating point number format F'. The format F' need not be the same as format F, and may have a mantissa width large enough to hold the exact multiplication output - for example the mantissa product (c_i) may be twice as large as the mantissa bit length of the multiplicands a_i and b_i. In some cases, the multiplicands a_i and b_i may be represented in different formats such that the mantissas of a_i and b_i may have different bit lengths. In such cases, the multiplication output may have a floating point format F' where the mantissa product is the sum of the mantissa bitlength of a_i and the mantissa bit length of b_i.

[0049] Irrespective of whether the input sets have the same mantissa lengths, when the mantissa of the product c_i has a bit length at least as long as the sum of the bit lengths of the mantissas of the multiplicands a_i and b_i, the dot product unit 100 using separate multiplication and addition described in FIG.1 has the same precision as the dot product unit 200 using fused multiplication and addition units, assuming no overflow or underflow, because there is no rounding required. However, if the second format F' is not wide enough to hold the exact multiplication output then performing dot product using the dot product unit 200 implementing fused multiplication and addition unit is more precise.

[0050] Whether using fused multiplication and addition or separately performing multiplication and addition, different orderings of the input pairs as multiplicands may yield different results. This is because of the effect of certain phenomenon such as catastrophic cancellation occurring when accumulating values in floating-point numbers.

[0051] Some arrangements of floating point adders providing faster computation include arrangements for performing parallel summation. These can also be used to reduce the latency of the network. For example, Fig. 1 shows a particular implementation of a network of floating point adders taking the outputs of the multiplication units 102a, 102b, 102c and 102d. The illustrated network of floating point adders in Fig. 1 is a balanced tree (or a balanced tree adder), for performing parallel summation and aimed at reducing latency. With such a configuration, addition of 'n' floating-point numbers (i.e. in Fig 1, the multiplication outputs c_i) can be realized using $\lceil \log_2(n) \rceil$ stages.

[0052] Further, in generalized examples the tree adder need not be a balanced structure. The tree adder can instead add the floating-point numbers using a single adder at every different stage. For example, any two floating point numbers are added in a first stage to generate a first sum value. Further, the first sum value is added to another floating-point number in a second stage using a second adder to generate a second sum value and so on. The latency of the arrangement increases in this example compared to a balanced tree adder.

[0053] Consider the mantissa products L, -L, M and N are provided as an input to the adder unit either as the outputs of the multiplication units when using the dot product unit 100 using separate multiplication and addition units described in FIG.1 or as intermediate multiplication values when using the dot product unit 200 implementing fused multiplication and addition units. When performing the addition, in all these examples, in each stage the output values of the accumulation are rounded or truncated in order to fit the output value into its finite representation. Multiple rounding may result in catastrophic cancellations. For different orders in which each input is provided to a tree adder, different outputs are generated. A catastrophic cancellation can occur when very small numbers are added to very large ones, and may cause significant bits of the result to be lost due to rounding. E.g., when summing a large positive number, L, the corresponding negative number, -L, and two small positive numbers M and N, the precise value of the sum is (M+N). An arrangement of floating point additions summing L and -L in a first primary adder and M and N in a second primary adder ought to give the final result (M+N). However, if the inputs are ordered differently and the primary adders perform the sums (L+M) and (-L+N), and L is much larger than M and N, then the outputs of the primary adders could be rounded to L and -L, giving an overall output of 0. Similar effects can be observed when using a network of fused multiplication and addition units.

[0054] As discussed above, the existing methods of processing floating point numbers, such as performing dot product using separate multiplication and addition units as described in FIG.1 or using fused multiplication and addition units as described in FIG.2 generate output values of varying precision based on the order in which the numbers are provided as the input. That is, a certain order of providing the inputs generates the best result which is the closest approximation to the actual dot product of the numbers. Other orders of providing the inputs may generate results which are not so

close to the actual dot product of the numbers. Thus, based on the order in which the inputs are provided, there could be a range of results obtained, around the actual dot product of the floating-point numbers.

[0055] The varying precision of the results obtained is due to reasons such as truncation errors or rounding errors, and catastrophic cancellation as discussed earlier. Also, the delay in obtaining the dot product for a large array of numbers is drastic, as the multiplications and additions happen over several sequential steps. Even though the method of performing multiplication and addition using separate multiplication and addition enables the multiplications to occur in parallel, and then some of the additions to be performed in parallel, the overall addition still needs to be performed in various stages to generate a final output value. Further, re-normalizing and rounding is performed in each stage, which increases the delay in generating the output value. Therefore, there is a need for a method of processing a set of floating-point numbers more precisely and with less delay.

[0056] Described herein is a hardware implementation and method of processing a set of k floating-point numbers concurrently. The method includes receiving the inputs in their incoming format, generating the output of the multiplication unit in a first (intermediate) format, and then converting the first format to a number in a second (intermediate) format for performing addition (the output of which may be in a further format that may or may not be the same as any of the previously mentioned formats). More precisely the method includes receiving a floating-point number from each set in an incoming format, generating a product number having a first format by performing mantissa multiplication and exponent summation concurrently while emulating the precision of a chosen conventional multiplier. Further, the method includes processing the numbers in the second format concurrently (e.g. obtaining a sum by performing a single sum over all the numbers in the set, in contrast to performing multiple sums across the set) to generate an output value.

[0057] FIG. 3 is a block diagram illustrating an example of an implementation of an architecture for processing a set of k floating-point numbers. The architecture 300 is a dot product unit for performing multiplication and addition of a large array of 2k floating point numbers to generate an output value. The large array of 2k floating point numbers comprise a first set of k floating-point numbers (a_0, a_1, a_2... a_{k-1}) and a second set of k floating-point numbers (b_0, b_1, b_2... b_{k-1}). The architecture is particularly suitable for calculating the dot product of large arrays of numbers but can be used to calculate the dot product of two, or more than two, numbers as required. The architecture 300 comprises a mantissa multiplication unit 301, a format conversion unit 302, an exponent addition unit 303, a maximum exponent detection unit 304, an alignment unit 306, a processing unit 308, and a re-normalizing unit 310. Each number in the first set of 'k' floating-point numbers comprises a mantissa ma_i and an exponent ea_i. Each number in the second set of 'k' floating-point numbers comprises a mantissa mb_i and an exponent eb_i. Each number in the first set of 'k' floating-point numbers is having a mantissa bit length of 'p' bits and each number in the second set of 'k' floating-point numbers is having a mantissa bit length of 'q' bits. Both sets of 'k' floating-point numbers may be signed numbers or unsigned numbers. In the case of signed numbers, the numbers would also each comprise a sign bit (sa_i or sb_i) as well as the mantissa and exponent. However, some floating-point formats (e.g. unsigned formats) may not include a sign bit.

[0058] The first set of k floating-point numbers (a_0, a_1, a_2... a_{k-1}) and the second set of k floating-point numbers (b_0, b_1, b_2... b_{k-1}) can be received in an input unit (not shown in the figure). The input unit can be a storage or a memory unit that can store the received inputs. Both sets of 'k' floating-point numbers are stored in an incoming format. Both sets of 'k' floating-point numbers may be of the same format or of different incoming formats for example if $p \neq q$.

[0059] The bit-length of the mantissa and exponent of the numbers in the incoming format is identified based on the type of the floating-point number format. The incoming format may be a predefined format that the architecture 300 is designed to receive, or may be identified on a task-by-task basis (e.g. by a controller, not shown). Examples of various types of the floating-point number formats include but are not limited to IEEE formats including half precision floating-point numbers (16 bit float), single precision floating-point numbers (float) and double precision floating-point numbers (double), or other formats such as a brain floating-point numbers (bfloat16). In one example, for the explanation of the method, we consider both the number a_i in the first set and number b_i in the second set to have the same incoming format as IEEE single precision floating-point format having a mantissa (ma_i or mb_i) with a bit length of 23 bits and an exponent (ea_i or eb_i) with a bit length of 8 bits. In another example the informing formats of both the number a_i in the first set and number b_i in the second set may be different. For example, we consider IEEE single precision floating-point format as the incoming format of the number a_i in the first set having a mantissa ma_i with a bit length of 23 bits and an exponent ea_i with a bit length of 8 bits, and we consider brain floating-point format as the incoming format of the number b_i in the first set having a mantissa mb_i with a bit length of 7 bits and an exponent eb_i with a bit length of 8 bits. However, it is understood that the invention is not limited to these (combinations of) formats and a person skilled in the art would understand that the architecture 300 could be implemented to use numbers in any type of the floating-point number format to perform the method described herein. The mantissa ma_i of each number in the first set of 'k' floating-point numbers and the mantissa mb_i of each number in the second set of 'k' floating-point numbers stored in an incoming format is provided to a mantissa multiplication unit 301. Prior to providing the input the fractional part of the mantissa ma_i having the bit length of 'p' bits may be extended by the implicit leading bit to obtain a normalised mantissa of p+1 bits. Similarly, the fractional part of the mantissa mb_i having the bit length of 'q' bits may be extended by the implicit leading bit to obtain a normalised mantissa of q+1 bits.

[0060] The mantissa multiplication unit 301 comprises a plurality of multiplier units configured to generate 'k' product numbers (z_0, z_1, z_2... z_{k-1}) in a different, first, format, having a bit length of 'r' bits (where 'r' is an integer) as described in more detail below. Each multiplier unit is configured to perform a mantissa multiplication of corresponding mantissas from the first set and second set of k floating point numbers to obtain an intermediate mantissa product:

$$mab_i = ma_i \times mb_i$$

[0061] The bit length of the full precision result obtained when performing the mantissa multiplication may be larger/smaller compared to r bits. Thus, mantissa multiplication unit 301 fits the output of the plurality of multipliers into a bit length of 'r' bits thereby generating product number z_i in the first format. Thus, the output of each multiplier unit, mantissa product mab_i is either rounded to r bits or padded with extra (zero) bits to fit the mantissa product into r bits. The value of bitlength 'r' is set based on the required precision of the dot product unit 300. In particular, 'r' can be (broadly) considered as accounting for the number of bits required to emulate the precision of the multiplication aspect of a conventional dot product unit. It will consist of a number of explicit leading bits and a number of fractional bits. As the input floating point numbers a_i and b_i are normalized before multiplication, the product number having bit length of r bits in the first format comprises two explicit leading bits (as the multiplication of two numbers between 1.0 inclusive and 2.0 exclusive may generate a number between 1.0 inclusive and 4.0 exclusive). Hence there is a need to increase the bitlength by one to account for the explicit leading '1' bit position during summation.

[0062] In a first case scenario, the hardware implementation of a dot product unit 300 may emulate the precision of P bits obtained with the dot product unit 100 using separate multiplication and addition units. In this context, a precision of P bit means that the dot product unit 300 achieves a precision not less than the worst-case precision achieved by the (emulated) dot product unit 100 generating a final output having a mantissa of P bits. In other words, P is the bit length of the mantissa output when performing multiplication and mantissa input when performing addition when implementing separate multiplication and addition units to perform dot product. However, for any given bit length P, the actual precision of the dot product unit 100 (as already discussed, due to the accumulation aspect) will depend on the order in which the inputs are processed. As such, in the present context, the dot product unit 300 is configured to be at least as precise as the worst-case precision that the emulated dot product unit 100 might achieve. To achieve this, when the hardware implementation of a dot product unit 300 emulates the precision of P bits obtained with the dot product unit 100, the bit length of 'r' bits is set as 'r=P+2' bits.

[0063] In a second case scenario, the hardware implementation of dot product unit 300 may emulate the precision of Q bits obtained with the dot product unit 200 using fused multiplication and addition units. Again, in this context, a precision of Q bits means that the dot product unit 300 achieves a precision not less than the worst-case precision achieved by the (emulated) dot product 200 generating a final output having a mantissa of Q bits. In other words, Q is the bit length of the mantissa output and accumulation mantissa input when performing multiplication and addition when implementing fused multiply and add units to perform dot product. However, for any given bit length Q, the actual precision of the dot product unit 200 (as already discussed, due to the accumulation aspect) will depend on the order in which the inputs are processed. As such, in the present context, the dot product unit 300 is configured to be at least as precise as the worst-case precision that the emulated dot product unit 200 might achieve. To achieve this, when the hardware implementation of dot product unit 300 emulates the precision of Q bits obtained with the dot product unit 100, the bit length of 'r' bits is set as 'r= max (Q+2, p+q+3)' bits.

[0064] As already mentioned, if the value of 'r' is less than the full precision bit length of mantissa multiplication i.e. p+q+2, then the mantissa product is faithfully rounded to obtain the desired bit length. Further, if the value of 'r' is greater than the full precision bit length of mantissa multiplication i.e. p+q+2, then the mantissa product is padded with zeros to obtain the desired bit length.

[0065] That is, if p+q+2>r bits, the mantissa product (mab_i) is faithfully rounded to r bits to obtain product number z_i. Rounding the mantissa product can be achieved in many ways. In one example, the plurality of multiplier units can be implemented using a truncated multiplier. When using the truncated multiplier to perform mantissa multiplication, the truncated multiplier directly computes r bits of the mantissa product mab_i by truncating the extra bits over 'r' bits thereby directly producing the product number z_i in the first format. In another example, the plurality of multiplier units can be implemented using a full multiplier. When using the full multiplier to perform mantissa multiplication, the multiplier computes an intermediate mantissa product mab_i, having a bit length of larger than 'r' bits which is further rounded to 'r' bits thereby generating the product number z_i in the first format.

[0066] Further if p+q+2 < r bits, the mantissa product (mab_i) is padded with extra least-significant bits to generate product numbers z_i having r bits. Thus, the product number z_i can be represented as a fixed point value $mab_i 2^{-r+2}$.

[0067] Similarly, in order to emulate the precision of dot product unit 200, the value of 'r' is greater than the full precision bit length of mantissa multiplication, i.e. p+q+2, by at least one position (if $Q+2 \leq p+q+3$) or more (if $Q+2 \geq p+q+3$). Hence the mantissa product is padded with zeros to obtain the desired bit length.

[0068] Concurrently, the exponent ea_i of each number in the first set of 'k' floating-point numbers and the exponent eb_i of each number in the second set of 'k' floating-point numbers stored in an incoming format is provided to an exponent addition unit 303. The exponent addition unit comprises a plurality of adder units, each adder unit configured to generate an exponent sum,

$$eab_i = ea_i + eb_i$$

[0069] The format conversion unit 302 receives 'k' product number numbers $(z_0, z_1, z_2... z_{k-1})$ from the mantissa multiplication unit 301. The format conversion unit 302 converts 'k' product numbers $(z_0, z_1, z_2... z_{k-1})$ in a first format to 'k' numbers in a different, second format $(y_0, y_1, y_2... y_{k-1})$, as described in more detail below.

[0070] The format conversion unit 302 converts the 'k' product numbers $(z_0, z_1, z_2... z_{k-1})$ in the first format to numbers in the second format. This comprises converting each product number z_i in the set of 'k' product numbers to a number 'y_i'. The format conversion unit 302 converts the product numbers z_i having a bit length of 'r' bits (in first format) to create the number 'y_i' with a bit length of 'n' bits (to represent a second format). The bit length of 'n' bits is obtained by adding both one or more extra most-significant bits (MSBs) and one or more extra least-significant bits (LSB) to the product number z_i (of bit length 'r' bits in the first format. Thus, the bit length 'n' is always greater than the bit length 'r' of the product numbers generated and hence greater than the original mantissa of the input floating point numbers a_i and b_i.

[0071] If the sets of 'k' floating-point numbers received are unsigned floating point numbers, then the representation of unsigned numbers created, with a bit length of 'n' bits includes n magnitude bits. If the sets of 'k' floating-point numbers received are signed floating point numbers, then the extra MSBs added to the mantissa of the first format can comprise a bit representing a sign bit. Thus, the representation of signed numbers created, with a bit length of 'n' bits includes a sign bit and (n-1) magnitude bits. The sign bits of the floating point numbers a_i and b_i are XORed to generate the sign bit of the corresponding number y_i.

[0072] The product numbers in the first format are converted to numbers in the second format based on the number of floating-point numbers (k) in the set. That is, the number of extra MSBs and LSBs added to the product numbers (z_i) of the first format is determined based on the number 'k'. The bit length of the product numbers (z_i) is extended to the MSB side by at least a logarithmic amount of the number 'k' $(\lceil \log_2(k) \rceil)$ bits and to the LSB side by at least a logarithmic amount of the number 'k-1' $(\lceil \log_2(k-1) \rceil)$ bits to obtain the number 'y_i'. An extra bit is added to the MSB to represent a sign bit if the input floating point numbers are signed floating point numbers. Therefore $\lceil \log_2(k) \rceil + 1$ extra MSBs and $\lceil \log_2(k-1) \rceil + 1$ LSBs are added on either side of the bit length 'r' of bit length of the product numbers (z_i). That is, one additional bit (other than the $\lceil \log_2(k) \rceil$ bits) in the extra MSBs added is assigned for the sign bit s_i. The sign bit is obtained by XORing the sign bits of the corresponding input floating point numbers a_i and b_i. The additional bit (other than the $\lceil \log_2(k-1) \rceil$ bits) in the extra LSBs added is a precision bit, for obtaining extra precision. The extra MSBs and LSBs added prevent overflow or underflow of bits, while processing the set of 'k' numbers which is explained in detail later. The numbers of extra MSBs and LSBs added to each side could be the same or different in different examples. In general, the bit length 'n' of the number 'y_i' in the second format can be obtained as

$$n = r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x \text{ bits}$$

where x is an integer and preferably $x \geq 1$, and where the value of x depends on the number of extra bits added to represent leading bit, sign bit and precision bits, if any. For example, x may be as small as 1 when there are no sign bits in the original received numbers, or may be as small as 2 when the original received numbers do have a sign bit. In both cases, x may be larger to provide greater precision.

[0073] In a second different implementation, the product number z_i in the first format may comprise r + log (k-1) + 1 bits, instead of 'r' bits described in the above paragraphs, so that when performing the multiplication stage rounding (if p+q+2>r bits) as many bits as possible of the mantissa product are retained for addition. In such a case, in order to emulate the precision of P bits obtained with the dot product unit 100 using separate multiplication and addition units by the hardware implementation of a dot product unit 300, the bit length of 'r' bits is set as 'r=P+1-log(k-1)'.

[0074] In such a case, the format conversion unit 302 converts the product numbers z_i having a bit length of 'r+ log

(k-1) +1' bits (in first format) to create the number 'y$_i$' with a bit length of 'n' bits (to represent a second format). The bit length of 'n' bits is obtained by adding one or more extra most-significant bits (MSBs) to the product number z$_i$ of bit length 'r' bits in the first format. The bit length of the product numbers (z$_i$) is extended to the MSB side by at least a logarithmic amount of the number 'k' $(\lceil \log_2(k) \rceil)$ bits. Also a number of additional bits can be added to the LSBs as precision bits, for obtaining extra precision. Thus, the bit length 'n' of the number 'y$_i$' in the second format can be obtained as

$$n = r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x \text{ bits}$$

[0075] FIG. 4a illustrates the representation of a mantissa (either mantissa ma$_i$ or mb$_i$) in an incoming format in an example in which both sets of incoming numbers share a common format and FIG. 4b illustrates the representation of product numbers (z$_i$) in a first (intermediate) format. FIG. 4c illustrates the representation of signed numbers (y$_i$) in a second (intermediate) format. In FIG. 4a, the incoming format is shown as a brain floating point number with a mantissa bit length (p) of 7 bits.

[0076] FIG.4b shows the product numbers (z$_i$) represented in a first format having a bit length of 'r' bits. As discussed above, the bit length of 'r' bits is set as 'r=P+2' bits or as 'r= max (Q+2, p+q+3)' bits based on the required precision. The output obtained by multiplying the mantissas (ma$_i$) having bit length of p bits and (mb$_i$), having bit length of q bits is truncated/ rounded to fit to r bits, if p+q+2>r. However, if p+q+2 < r bits, the output obtained by multiplying the mantissas (ma$_i$) and (mb$_i$) is padded with extra least-significant bits to generate product numbers z$_i$ having r bits. Suppose the value of r bits is 16 bits. The representation of the product number in the first format comprises two explicit leading bits (LB) as a part of the r bits.

[0077] FIG. 4c illustrates the representation of signed numbers (y$_i$) in a third format. The representation in the FIG. 4c illustrates signed number 'y$_i$' with a bit length of $n = r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 2$ bits. This is obtained by adding log$_2$(k) + 1 extra MSBs and log$_2$(k - 1) + 1 extra LSBs to the bit-length 'r' of the mantissa. Thus, in an example when a set of 8 floating point numbers, for which the bit length r is set to 17 bits (e.g. extended from an initial bit length of 7 as shown in FIG 4a), are added, i.e. k=8, then the signed number in the second format will have a bit length of n=25 bits.

[0078] It is clear from the example in FIG. 4c, that the number 'y$_i$' represented in a second format, as defined herein, comprises r bits which includes two bits assigned for representing the leading bits (LBs), a bit assigned for representing a sign bit (s$_i$), and a further $\lceil \log_2(k) \rceil$ bits as extra MSBs, and $\lceil \log_2(k-1) \rceil + 1$ extra LSBs. As such, the number 'y$_i$' of the second format is a signed number and comprises both a sign bit and magnitude bits (i.e. the bits indicating the absolute magnitude of the represented value). The sign bit is assigned as a '0' or '1' bit based on whether the number is a positive or negative number.

[0079] In the example shown in FIG. 4c, the number 'y$_i$' comprises $\lceil \log_2(k-1) \rceil + 1$ extra LSBs added to the product numbers (z$_i$). The number 'y$_i$' could comprise $\lceil \log_2(k-1) \rceil + u$ extra LSBs added to the mantissa (m$_i$), where u is any integer, u≥1. Preferably, the number 'y$_i$' comprises $\lceil \log_2(k-1) \rceil + 1$ extra LSBs added to the product numbers (z$_i$). The extra LSBs added to the product numbers (z$_i$) increase the precision of the result obtained and reduce underflow of bits of the product number while aligning the number 'y$_i$' which is explained in detail below.

[0080] Thus, in the example described in the above paragraphs where the incoming format is a signed floating point number and the first format is a number that has two explicit leading bits, at least $\lceil \log_2(k) \rceil + 1$ extra MSBs and $\lceil \log_2(k-1) \rceil + 1$ LSBs are added on either side of the bit length 'r' of product numbers z$_i$ to create the number 'y$_i$', thus making the number of additional bits x ≥ 2.

[0081] Similarly, in another example case, assuming the incoming format is an unsigned floating point number and the first format is a number that has two explicit leading bits, $\lceil \log_2(k) \rceil$ extra MSBs and at least $\lceil \log_2(k-1) \rceil + 1$ extra LSBs are added on either side of the bit length 'r' of product numbers (z$_i$) to create the number 'y$_i$', thus making the number of the additional bits x ≥ 1.

[0082] Thus, in a generalized example, x ≥ 1 and x ≤ 'r' bits, making the maximum bit length of number 'y$_i$',

$$n = 2r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil$$

[0083] The extra MSBs and LSBs added to the product numbers (z_i) other than the sign bit are initially assigned '0' bits in the second format. The sign bit is assigned as a '0' or '1' bit based on whether the number is a positive or negative number.

[0084] Further, the exponent sum 'eab$_i$' (eab$_0$, eab$_1$, eab$_2$, eab$_3$...eab$_{k-1}$) of each pair of floating point numbers ea$_i$ and eb$_i$ in the first incoming format is provided as an input to the maximum exponent detection unit 304. The input 'eab$_i$' to the maximum detection unit is provided from the exponent addition unit 303 as shown in FIG. 3. In some other arrangements, the exponent sums may be passed through the format conversion unit 302 to the maximum exponent detection unit 304.

[0085] The maximum exponent detection unit 304 identifies the maximum exponent sum (e_{max}) from the k exponent sums (eab$_0$, eab$_1$, eab$_2$, eab$_3$...eab$_{k-1}$). The maximum exponent detection unit 304 detects the maximum exponent sum using various methods or functions. An example of a method of identifying the maximum exponent sum is using a binary tree structure. A method of identifying the maximum exponent sum (e_{max}) is described in detail, below, with reference to FIG. 5. However, which option is preferable may depend on the available resources (e.g. parallel processing may be faster overall, but more computationally intensive).

[0086] In addition to being provided to the maximum exponent detection unit 304, the exponent sum values eab$_i$ are provided, from the exponent addition unit 303, as input to the alignment unit 306. The alignment unit 306 receives the exponent sums 'eab$_i$' of each pair of floating point numbers a$_i$ and b$_i$ as a first input. The alignment unit 306 further receives the maximum exponent sum (e_{max}) from the maximum exponent detection unit 304 as a second input and the number 'y$_i$' from the format conversion unit 302 as the third input. In one implementation, the alignment unit 306 may comprise the format conversion unit implemented as a part of the alignment unit rather than a separate unit. In such case the alignment unit 306 receives the product number 'z$_i$' as an input and converts the product number 'z$_i$' into the number 'y$_i$' before shifting. The alignment unit 306 aligns the magnitude bits of each number 'y$_i$', thereby converting the number 'y$_i$' to a different number (or integer v$_i$) with a bit-length of n bits based on the maximum exponent. The method of aligning the number 'y$_i$' is explained in detail with reference to FIG.5, but in summary the numbers in the second format are adjusted to be based on the maximum exponent, and the adjusted numbers (v$_0$, v$_1$, v$_2$...v$_{k-1}$) are treated for convenience as integers for the subsequent processing in the processing unit 308.

[0087] Thereafter, the k integers (v$_0$, v$_1$, v$_2$...v$_{k-1}$) thus generated are provided to the processing unit 308. The processing unit 308 is an adder unit. The processing unit 308 processes the k integers (i.e. the k aligned numbers) concurrently. That is the processing unit performs a process on all the integers in the set at the same time rather than, for example, processing elements of the set sequentially. The processing unit 308 performs addition of the k integers to generate an output value o. It is noted that addition of a negative number to a positive number is equivalent to performing a subtraction, and so the term processing is used herein to cover the acts of both addition and subtraction, alone or in combination.

[0088] The output value o from the processing unit 308 and the maximum exponent sum from the maximum exponent detection unit 304 is further provided into the re-normalizing unit 310. The renormalizing unit 310 converts the output value from the processing unit to a floating-point number with a mantissa mi and exponent 'ei'. The format of the output value can be selected depending on the desired precision (e.g. depending on whether the aim is to emulate the precision of an arrangement such as that of Fig. 1 or Fig. 2). The output unit 312 stores the converted output value (i.e. the output floating point number).

[0089] FIG. 5 is a block diagram illustrating the different units in the implementation of the architecture 300 in FIG. 3. Consider a scenario where the set of k floating point numbers in the first format comprises three numbers, i.e. k=3. The input unit (not shown in the figure) may receive a first set of three floating point numbers ai (a0, a1, a2) and a second set of three floating point numbers bi (b0, b1, b2) as input.

[0090] The mantissa multiplication unit 301 receives the mantissas (ma$_0$, ma$_1$, ma$_2$) of the first set of three floating point numbers (a$_0$, a$_1$, a$_2$) and the (mb$_0$, mb$_1$, mb$_2$) of the second set of three floating point numbers b$_i$ (b$_0$, b$_1$, b$_2$) as input from the input unit. The mantissa multiplication unit 301 comprises a plurality of multiplier units 501a, 501b, and 501c. Each multiplier unit is configured to generate a product number z$_i$ having a bit length of 'r' bits. Each multiplier unit is configured to perform a mantissa multiplication of corresponding mantissas from the first set and second set of k floating point numbers to obtain a mantissa product:

$$mab_i = ma_i \times mb_i$$

[0091] The multiplier unit 501a multiplies the mantissas mao and mbo of the floating point numbers a$_0$ and b$_0$ respectively to generate a product number z$_0$. Similarly, the multiplier unit 501b and 501c generate the product numbers z$_1$ and z$_2$ respectively.

[0092] As discussed earlier, the value of bitlength 'r' is set based on the required precision of the dot product unit 300. In a first case scenario, where the hardware implementation of dot product unit 300 emulates the precision of P bits obtained when performing dot product using separate multiplication and addition, the bit length of 'r' bits is set as 'r=P+2' bits. In a second case scenario, where the hardware implementation of dot product unit 300 emulates the precision of Q bits obtained when performing dot product using fused multiplication and addition, the bit length of 'r' bits is set as 'r= max (Q+2, 2p+3)' bits when both sets of floating point number have same incoming format (p=q) or 'r= max (P+2, p+q+3)' bits when both sets of floating point number have different incoming format (i.e. $p \neq q$).

[0093] The plurality of multiplier units 501a, 501b and 501c may, in one example, be implemented using a truncated multiplier. When using the truncated multiplier to perform multiplication of the mantissas, the truncated multiplier directly computes r bits of the output by truncating the extra bits over 'r' bits thereby directly producing the product number zi in a first format. In another example, the plurality of multiplier units 501a, 501b and 501c may be implemented using a full multiplier. When using the full multiplier to perform mantissa multiplication, the multiplier computes an intermediate mantissa product mabi, having a bit length of larger than 'r' bits which is further rounded to 'r' bits thereby generating the product number zi in the first format.

[0094] If p+q+2>r bits, the mantissa product (mabi) is faithfully rounded to r bits. Further if p+q+2 < r bits, the mantissa product (mabi) is padded with extra least-significant bits to generate product numbers zi having r bits.

[0095] Concurrently, the exponent addition unit 303 receives the exponents (ea_0, ea_1, ea_2) of the first set of three floating point numbers (a_0, a_1, a_2) and the (eb_0, eb_1, eb_2) of the second set of three floating point numbers (b_0, b_1, b_2) as input from the input unit. The exponent addition unit 303 comprises a plurality of adder units 503a, 503b and 503c. Each adder unit is configured to calculate the sum of the exponent ea_i and the exponent eb_i of floating point numbers in each set to generate an exponent sum eab_i corresponding to each product number z_i

$$eab_i = ea_i + eb_i$$

[0096] The adder unit 503a is configured to add the exponent eao of the floating point number a_0 and the exponent ebo of the floating point number b_0 to generate an exponent sum eab_0. Similarly, the adder units 503b and 503c generates exponent sums eab_1 and eab_2 respectively.

[0097] In examples where the first set of three floating point numbers a_i (a_0, a_1, a_2) and a second set of three floating point numbers b_i (b_0, b_1, b_2) are signed floating point numbers, the sign bits of the corresponding floating point numbers ai and bi are XORed to obtain the value of a sign bit s_i (s_0, s_1, s_2) corresponding to the product numbers z_i (z_0, z_1, z_2).

[0098] The output of the mantissa multiplication unit 301 is further provided to the format conversion unit 302. In other words the plurality of multiplier units 501a, 501b and 501c in the mantissa multiplication unit 301 provides the product numbers z_0, z_1 and z_2 in the first format having a bit length of r bits to the format conversion unit 302. Further, the sign bits s_i (s_0, s_1, s_2) if any are also provided to the format conversion unit 302. The format conversion unit 302 converts the set of three numbers in the first format to three numbers y_i in the second format as described with reference to FIG. 3. In this example, consider both sets of 3 floating point numbers in the incoming format are signed numbers and the set of k product numbers in the first format comprises two explicit leading bits, holding the integer part of the mantissa. Therefore, the set of three floating point numbers are converted to three signed numbers having a bit-length of n bits including a sign bit s_i and (n-1) magnitude bits f_i including two explicit leading bits. In other example cases, the both sets of three floating point numbers in the incoming format could be unsigned numbers.

[0099] Further, the exponent sums 'eab$_i$' (eab_0, eab_1, eab_2, and eab_3) from the exponent addition unit 303 are provided to the maximum exponent detection unit 304.

[0100] The maximum exponent detection unit 304 in FIG. 5 may comprise two maximum function logics to identify the maximum exponent sum. This is by way of example only, and other implementations may have a different structure for finding the maximum exponent, or similar structures but with a different number of logics to account for a different number of inputs.

[0101] In the example of using two maximum function logics, the first maximum function logic may receive the exponent sums eab_0 and eab_i. The first maximum function logic identifies the maximum exponent value among eab_0 and eab_1. Further, the output of the first maximum function logic and the exponent eab_2 are provided to the second maximum function logic. The second maximum function logic identifies the maximum exponent value among the output of the first maximum function logic and t exponent eab_2 to detect the e_{max} i.e. the maximum exponent sum among the input exponent sums eab_0, eab_1, and eab_2.

[0102] As mentioned above, the maximum detection unit 304 can be implemented in different other ways. For example, maximum detection unit 304 can be implemented using a binary search tree.

[0103] Returning to the depicted example, the maximum exponent sum identified by the maximum detection unit 304 is provided as input to the alignment unit 306. Further, the exponent sums eab_0, eab_1, and eab_2 are provided as input to the alignment unit 306. Further the three signed numbers 'y$_i$' in the second format are provided as input to the alignment

unit 306. The alignment unit 306 aligns the magnitude bits f_i of each signed number 'y_i', based on the maximum exponent sum and the respective exponent sum of the exponents of the numbers generating the product number in the first format corresponding to the signed number. In other words, the magnitude bits of the signed numbers for which the corresponding product number did not already have the sum of exponents as the maximum exponent are shifted to account for the difference in exponent sum for the pair of numbers in both sets compared to the maximum exponent sum (effectively adding zeros before the first (or, at least, the first non-zero) magnitude bit, and removing trailing bits as required, to re-align the magnitude bits as appropriate). The alignment unit 306 thus converts each signed number ('y_i') to another integer (v_i) that is output by the alignment unit 306. The integer v_i is considered as a fixed point number format. Similarly, in case of unsigned numbers the alignment unit shifts the magnitude bits of the unsigned numbers based on the maximum exponent sum and the respective exponent sum of the numbers generating the product number in the first format corresponding to the unsigned number.

[0104] The conversion of the signed number 'y_i' to an integer v_i is illustrated with examples shown in FIG.6a- 6d. It will be appreciated, in describing this series of figures, that the starting point is the signed number 'y_i' in a second format number, as output by the format conversion unit 302, and the end point is the converted signed number v_i (integer v_i) mentioned above. However, for ease of reference, the intermediary stages may also be referred to as signed numbers in the description below.

[0105] Consider an example of two sets of three floating point numbers, each floating-point number in an incoming format with an implicit leading bit and a sign bit separate to the mantissa. Each number has a mantissa m_i having a bit length of 7-bits in the first format (such as bfloat 16). Suppose the bit length product number z_i generated by multiplying the mantissa of two floating point numbers one from each set is set as r=17 bits. Thus, in this example, each number in a set of three numbers, when converted to the second format comprises a signed number y_i having a bit length 'n' (including a sign bit s_i), where

$$n = r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 2 = 17 + \lceil \log_2(3) \rceil + \lceil \log_2(3-1) \rceil + 2$$

$$= 17 + 2 + 1 + 2 = 22$$

[0106] FIG. 6a illustrates the signed number 'y_i' in the second format with a bit length of n bits. In an example, consider that the number shown FIG. 6a represents a signed number y_0 provided as input to the alignment unit 306. The alignment unit 306 comprises a plurality of subtraction modules 505. The alignment unit 306 further comprises a plurality of shifter units 506 and a plurality of complementing units 507.

[0107] The alignment unit 306 receives an exponent sum eab_i as a first input from exponent addition unit 303, the maximum exponent sum e_{max} as a second input from the maximum exponent detection unit 304 and a signed number 'y_i' from the format conversion unit 302 as a third input. As explained in FIG.3, the format conversion unit 302 can, in one example, be implemented as a part of the alignment unit 306. The exponent addition unit 303 provides the exponent sum 'eab_i' of the three numbers to each subtraction module. Each of the subtraction modules receives an exponent sum eab_i as a first input from exponent addition unit 303, the maximum exponent sum e_{max} as a second input from the maximum exponent detection unit 304. Each subtraction module calculates a difference 'e_{di}' between the maximum exponent sum 'e_{max}' and the exponent sum 'eab_i' of two numbers number. In FIG. 5, a first subtraction module receives an exponent eab_0 of two first numbers from exponent addition unit 303 and the maximum exponent sum e_{max} from the maximum exponent detection unit 304. The first subtraction module calculates a first difference referred to as e_{d0} in FIG.5. Similarly, the remaining subtracting modules calculate the differences e_{d1} and e_{d2}, as shown in FIG.5. As shown, the plurality of subtraction modules calculates the differences e_{di} for each number in parallel, but other arrangements are possible - e.g. a single subtraction module performing each subtraction in series. Returning to the example as depicted, each calculated difference e_{d0}, e_{d1}, and e_{d2} from the plurality of subtraction modules 505 is further provided to a corresponding shifter unit among the plurality of the shifter units 506.

[0108] Each shifter unit among the plurality of shifter units 506 receives the calculated difference e_{di} corresponding to a product number as a first input and the magnitude bits f_i of the corresponding signed number 'y_i' as the second input. Further, each shifter unit among the plurality of shifter units 506 shifts the magnitude bits f_i, of the signed number 'y_i' based on the corresponding calculated exponent difference. The magnitude bits f_i (except the sign bit) are shifted to the least significant bit side (i.e. the right in the depicted format) by a number of positions equal to the calculated exponent difference. FIG. 6a illustrates that the magnitude bits f_i of signed number 'y_0' of the first input include the 'r bits' corresponding to the bits of the product number in the first format, comprising two explicit leading bits. The remaining extra bits of the signed number 'y_0' are padded with '0' bits. The original 'r bits' as well as the explicit leading bits are shifted by the shifter unit. Further, in FIG. 6a, the sign bit of the signed number y_0 is assigned with a '1' bit indicating that the signed number y_0 is a negative number. The sign bit is not shifted by the shifter unit.

[0109] In the example, a first shifter among the plurality of shifters 506 receives the first input (magnitude bits f_0 of the

signed number yo), from the format conversion unit 302. Further, the first shifter receives the calculated difference e_{d0} (in an example, for the first number having a mantissa (fo) shown in FIG. 6a, consider that the calculated difference e_{d0} (difference e_{max} and eabo) is equal to 4) from the first subtraction module as the second input. Therefore, the first shifter unit shifts the magnitude bits f_0 of signed number y_0 by 4 positions to the right. FIG.6b illustrates the shifted number. The calculated difference e_d is never a negative number. Thus, the magnitude bits f_i are always shifted to the least significant bit side (i.e. to the right in the example) based on the calculated difference e_d for each number, possibly by zero positions.

[0110]   Similarly, the other shifter units among the plurality of shifter units 506 shift the magnitude bits (f_1, f_2 and f_3) of the remaining three numbers based on the corresponding calculated differences e_{d1}, e_{d2}, and e_{d3}. Thus, all the shifter units in the plurality of shifter units 506 perform the shifting of magnitude bits f_i in parallel whereas in most of the existing architectures for processing floating point numbers, the shifter shifts or aligns the mantissa in sequence as and when required which increases the delay to a considerable extent. Since, in the disclosed architecture, as the shifting or aligning of all the numbers occurs in parallel, the delay in processing could be significantly reduced as the number of floating-point numbers to be processed increases. In another implementation, it is possible that the plurality of shifter units 506 perform the shifting of magnitude bits f_i in series despite the fact that delay is increased due to limitation on the available resources (e.g. parallel processing may be faster overall, but more computationally intensive).

[0111]   It can be seen in FIG. 6b that the shift of the magnitude bits results in 4 bits being shifted out of the bit-width of the signed number (and thus out of the stored representation of the number). The shifter units truncate the bits of the signed number that are shifted outside the bit length of n bits. The bits corresponding to the mantissa of the original number in the first format are shifted out of the bit length of 'n' bits when the calculated exponent difference for the corresponding number is greater than the number of extra LSBs added to the mantissa m_i of the number in the first format when converting to the second format. When the calculated difference is greater than the number of extra LSBs, it causes underflow of bits from the original (first format) mantissa. When the calculated difference is less than the number of extra LSBs, it only causes underflow of 'zero' bits added by the format conversion unit 302. As mentioned above, FIG. 6b shows the 4 bits that are moved out of the bit length of n bits, when the bits were shifted by 4 bits. FIG. 6c illustrates the signed number after performing truncation. It is evident from the figure that though the bits are shifted by 4 bits, only 1 bits of the actual number (i.e. 1 bit of the product number in the first format) is lost in this case, due to the extra LSBs that were added during conversion to the second format. Thus, the extra LSBs act to reduce the loss of precision that would occur if, for example, all numbers were shifted to use the same, maximum, exponent in the first format.

[0112]   The output from the each of shifter unit 506 is further provided to a corresponding complementing unit among the plurality of complementing units 507. The complementing units receive the aligned magnitude bits from the shifter units as a first input and the sign bit of the signed number 'y_i' as a second input. However, in other arrangements, the function of the complementing unit could be performed before the function of the shifting unit or as a part of adder unit (processing unit 308). In any case, the complementing unit performs the two's complement of the magnitude bits f_i for those numbers having a sign bit indicating a negative number. In this case, the shifted positive signed numbers in the set are provided to the processing unit 308 (adder 508 in FIG. 5) without complementing. Further, the two's complements of the negative numbers in the set are provided to the processing unit. The processing unit 308 receives the output from the plurality of complementing units 507 and processes the aligned signed numbers v_i concurrently to generate the output. The output obtained from each complementing unit is an aligned number v_i. FIG. 6d represents the number po obtained by complementing the mantissa as shown in FIG 6c.

[0113]   Thus, the alignment unit 306 aligns the magnitude bits of the number 'y_i' to generate a set of numbers (or integers) v_i by performing the steps of shifting and truncating the magnitude bits f_i of the number 'y_i'. The alignment unit also converts to two's complement representation any numbers with a sign bit indicating the number is negative. In case of unsigned numbers 'y_i', the alignment unit performing the steps of shifting and truncating the magnitude bits f_i of the number 'y_i'. The only difference is that there is no need of performing the step of complementing in case of unsigned number. The alignment unit 306 is capable of processing each number in parallel for the steps of shifting, truncating and complementing the bits of the mantissa. The number v_i obtained after conversion is an integer. The number v_i is computed as

$$v_i = \lfloor y_i \times 2^{e_{max} - e_i + \lceil \log_2(k-1) \rceil + 1} \rfloor$$

[0114]   In a different implementation, the alignment unit 306 may perform a step of rounding up after shifting the magnitude bits f_i of the number 'y_i' rather than truncating. In such a case, the integer v_i can be obtained by rounding up the magnitude bits f_i of the number 'y_i'. Thus, the number v_i is computed as

$$v_i = \lceil y_i \times 2^{e_{max} - e_i + \lceil \log_2(k-1) \rceil + 1} \rceil$$

[0115] It would be evident for a person skilled in the art that the step of rounding the number 'y$_i$' can be performed by implementing any rounding up or rounding down methods. As shown in FIG. 5, the converted numbers i.e. the integers v$_0$, v$_1$, v$_2$ and v$_3$ are further provided to the adder 508 which is the processing unit 308. The signed integer v$_0$, thus generated by performing shifting, truncating, and complementing the magnitude bits f$_0$ of the signed number y$_0$ in the example is illustrated in FIG. 6d. In an example, the adder 508 is a carry save adder capable of adding 3 'n' bit integers.

The value of the integer v$_i$ ranges between $-2^{r+\lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 1}$ and $2^{r+\lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 1} - 1$. The carry save adder performs the addition of 3 signed integers vi to generate a sum value o (output).

$$o = \sum_{i=0}^{k-1} v_i$$

[0116] The magnitude of the summands (i.e. the 3 integers v$_i$) is less than $2^{r+\lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 1}$ and hence the sum value will be less than $2^{r+\lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + 1}$ and does not overflow the 'n' bits. That is, the largest possible value integer y$_i$ will have a 1 exactly one position from the MSB end (accounting for the sign bit), resulting in v$_i$ having a 1 at least $\lceil \log_2(k) \rceil + 1$ bits from the MSB end (again accounting for the sign bit), and the sum of *k* numbers of that value (i.e. considering the extreme case where all the numbers have the maximum exponent sum) cannot overflow the additional $\lceil \log_2(k) \rceil$ bits provided at the MSB end after the sign bit. The adder 508 processes the set of 'k' floating point numbers to generate the same output value irrespective of the order in which the set of 'k' floating point numbers are provided as inputs.

[0117] The sum value o is further provided to the re-normalizing unit 310. It will be noted that in this example the value o, like the values v$_i$, will be a signed integer in two's complement format. The normalizing unit 310 comprises a shifter 510a and a subtractor 510b. The shifter 510a shifts the bits of the sum value o to generate a normalized value (in general format). The shifter 510a represents the sum value o in a normalized format by counting the number of leading '0' bits or '1' bits (represented as 'd') occurring continuously in the MSB's (i.e. including the sign bit). The number of leading '0' bits are counted when the sum value obtained is a non-negative number and the number of leading '1' bits are counted when the sum value obtained is a negative number. The shifter shifts the bits of the number to generate a normalized number (n$_k$) in a normalized format. The number (n$_k$) is further rounded to represent the normalized number (n$_k$) with a desired bit-length -The normalized number (n$_k$) is represented as (assuming $\lceil \log_2(k) \rceil + 1$ MSBs were added when converting to the second format):

$$n_k = o \times 2^{\lceil \log_2(k) \rceil + 1 - d}$$

[0118] The subtractor 510b receives the maximum exponent sum as the first input and d (the number of leading '0's or '1's) and the number of extra LSBs added to the mantissa of the first format) as the input. Further, the subtractor calculates the exponent of the normalized number based on the inputs and represents the exponent over a bitlength equal to the maximum of the bit length of the exponent of the floating point number a$_i$ or the bit length of the exponent of the floating point number b$_i$, with an additional bit. i.e. the exponent of the normalized number can be represented over a bitlength of 'max (bit length of ea$_i$, bit length of eb$_i$)+1' bits. The exponent of the final output is calculated as (again, assuming $\lceil \log_2(k) \rceil + 1$ MSBs were added when converting to the second format)

$$e_k = e_{max} + \lceil \log_2(k) \rceil + 2 - d$$

[0119] This is an example, and it is not limited to a person skilled in the art that in other examples, different other known methods can be used to calculate the exponent e$_k$. The final output or the sum value obtained is thus represented with a normalized mantissa (n$_k$) and the exponent (e$_k$).

[0120] The architecture 300 of the adder can be used to add any number of floating point numbers. The example shown in FIG. 5 is a specific example of the dot product unit 300 for performing dot product of a set of 3 floating point

numbers. Further, additional number of elements can be added to each unit in the adder 500 in a similar manner, thereby expanding it to perform dot product of any number of floating-point numbers (for example 20 floating-point numbers or 50 floating-point numbers) concurrently.

[0121] FIG. 7 is a flowchart illustrating a method of processing two sets of 'k' floating point numbers. The method includes performing dot product, using a hardware implementation of an architecture 300 for performing dot product multiplication. The method includes performing multiplication and addition operations on a large array of 2k floating point numbers to generate an output value. The large array of 2k floating point numbers comprises a first set of k floating-point numbers (a_0, a_1, a_2... a_{k-1}) and a second set of k floating-point numbers (b_0, b_1, b_2... b_{k-1}).

[0122] In step 701, the method includes receiving both sets of 'k' floating point numbers each in an incoming format. Each number in the first set of 'k' floating-point numbers comprises a mantissa ma_i and an exponent ea_i. Each number in the second set of 'k' floating-point numbers comprises a mantissa mb_i and an exponent eb_i. The mantissa ma_i of the number a_i is having a bit length of 'p' bits and the mantissa mb_i of the number b_i is having a with a bit length of 'q' bits. Both sets of 'k' floating-point numbers may be signed numbers or unsigned numbers. The bit-length of the mantissa and the bit-length of the exponent (e_i) in the incoming format is identified based on the type of the floating-point number format. Further the floating-point numbers could be signed or unsigned number with an implicit or explicit leading bit. For example, a single precision (32 bit) floating point number as an incoming format, may typically be a signed number with an implicit leading bit that comprises a mantissa having a bit-length of 23 bits without including the leading bit, an exponent having bit length of 8 bits and an extra sign bit (s_i). In other examples a single precision (32 bit) floating point number in a first format may be a signed number with an explicit leading bit, and then the mantissa has a bit length of 23 bits including the explicit leading bit.

[0123] When the single precision (32 bit) floating point number in the incoming format is an unsigned number with an implicit leading bit there would not be any extra sign bit and the mantissa could be represented by a bit length of 24 bits (without including the leading bit). Further when the single precision (32 bit) floating point number in the incoming format is an unsigned number with an explicit leading bit, the bit length of 24 bits of mantissa includes an explicit leading bit. Both sets of 'k' floating-point numbers may be of the same incoming format or of different incoming format such as if $p \neq q$.

[0124] On receiving the set of 'k' floating point numbers in the first format, at step 702, the method includes generating 'k' product number numbers (z_0, z_1, z_2... z_{k-1}) in a different, first, format, having a bit length of 'r' bits. Prior to providing the input the fractional part of the mantissa ma_i having the bit length of 'p' bits may be extended by the implicit leading bit to obtain a normalised mantissa of p+1 bits. Similarly, the fractional part of the mantissa mb_i having the bit length of 'q' bits may be extended by the implicit leading bit to obtain a normalised mantissa of q+1 bits. The k product numbers are generated by performing a mantissa multiplication of corresponding mantissas ma_i and mb_i from the first set and second set of k floating point numbers and fitting the output of each mantissa multiplication into a bit length of 'r' bits.

[0125] The value of bitlength 'r' is set based on the required precision of the dot product unit 300 as explained earlier. In order to emulate the precision obtained with the dot product unit 100 using separate multiplication and addition, the bit length of 'r' bits is set as 'r=P+2' bits. Further to emulate the precision obtained with the dot product unit 200 using fused multiplication and addition, the bit length of 'r' bits is set as as 'r= max (Q+2, p+q+3)' bits.

[0126] That is, if p+q+2>r bits, the mantissa product (mab_i) is faithfully rounded to r bits to obtain product number z_i. Further if p+q+2 < r bits, the mantissa product (mab_i) is padded with extra least-significant bits to generate product numbers z_i having r bits.

[0127] At step 703, the method includes generating a sum eab_i of the exponent ea_i of a number in the first set of 'k' floating-point numbers and the exponent eb_i of a corresponding number in the second set of 'k' floating-point numbers. This step could be performed before or after the step 702 or could be even performed in parallel to the step 702.

[0128] Further, at step 704, the method includes converting the 'k' product numbers (z_0, z_1, z_2... z_{k-1}) in a first format to 'k' numbers (y_0, y_1, y_2... y_{k-1}) into a different, second format. The numbers (y_i) are obtained by adding both extra MSBs and extra LSBs to the bit-length 'r' of the product number z_i in the first format. The bit length of r bits is extended based on the number 'k' (the number of floating-point numbers in the set). In an example with the sets of 'k' floating point numbers in the incoming format as signed numbers adding extra MBS and LSBs comprises adding preferably $\lceil \log_2(k) \rceil + 1$ number of the most-significant bits and $\lceil \log_2(k-1) \rceil + 1$ number of least-significant bits. The number of extra MSBs and extra LSBs added to the bit-length b of the mantissa could be the same or different. The extra MSBs added include, in this example, a bit representing a sign bit. Thus, the signed number is represented with a bit-length of 'n' bits including the sign bit s_i. The bit length 'n' is represented as

$$n = r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x \text{ bits}$$

where x is an integer and preferably $x \geq 2$.

[0129] Further, the method at step 706 comprises identifying a maximum exponent sum (e_{max}) among the exponent sums (eab_i) of the set of 'k' floating point numbers. The maximum exponent sum (e_{max}) is identified by a maximum exponent detection unit 304. The maximum exponent detection unit 304 implements an algorithm such as a maximum function for identifying a maximum value among a set of values (exponent sums eab_i). Step 706 could be performed before or after the step 704 or could be even performed in parallel to the step 706.

[0130] The method further comprises, at step 708, aligning the magnitude bits of the numbers 'y_i' to be based on the maximum exponent sum (e_{max}). The number 'y_i' is an integer represented as a fixed-point number having a bit length of n-bits. The method of aligning the magnitude bits of the numbers is discussed with respect to FIG.7b. Aligning the magnitude bits of the numbers based on the maximum exponent sum is performed by an alignment unit 306. The alignment unit 306 thus generates an aligned number which is an integer v_i.

[0131] The method further comprises, at step 710, processing the set of 'k' aligned numbers v_i concurrently to generate an output value o. The processing of the integers v_i includes performing addition of the k numbers. It is noted that addition of a negative number to a positive number is equivalent to performing a subtraction, and so the term processing is used herein to cover the acts of both addition and subtraction, alone or in combination. The processing of the k numbers is performed concurrently. That is the processing unit performs a process on all the integers in the set at the same time rather than, for example, processing elements of the set sequentially or processing the elements of the set in pairs. The processing unit 308 performs addition of the k integers to generate an output value.

[0132] Further, at step 712, the method includes renormalizing and rounding the output value o to represent the output value as a floating-point number in any format with a normalized mantissa n_k and an exponent e_k. The method includes renormalizing the output value to represent the output value o as a standard normalized number. Further, the method performs rounding the normalized number n_k to represent the number with a mantissa having a particular bit-length. For example, the normalized number is rounded to a bit length depending on the desired precision (e.g. depending on whether the aim is to emulate the precision of an arrangement such as that of Fig. 1 or Fig. 2). The normalizing is performed by initially counting the number of recurring '0' bits or '1' bits on the MSB side. The recurring '0' bits are counted when the output value 'o' is a positive number. The recurring '1' bits are counted when the output value 'o' is a negative number. Further, the normalizing is performed by shifting the bits of the output value o to the LSB side around the radix point to represent the signed number as a standard normalized number. Further, the method calculates an exponent value based on the maximum exponent and the counted number of recurring bits. Thus, the output 'o' is normalized to be represented as a floating-point number in the first format.

[0133] Further the architecture 300 can also be implemented as a dot product unit as shown in FIG. 11, for multiplying the two sets of floating-point numbers, in an optimised manner. The dot product unit 1100 comprises an multiplication unit 1101 comprising a plurality of multiplier units 1101_a, 1101_b,... 1101_{k-1}, an alignment unit comprising a plurality of shifter units 1106_a, 1106_b,... 1106_{k-1}, an accumulator unit 1108 and a normalizer unit 1110

[0134] The dot product unit 1100 receives a large array of floating point numbers comprising a first set of k floating-point numbers (a_0, a_1, a_2... a_{k-1}) and the second set of k floating-point numbers (b_0, b_1, b_2... b_{k-1}). The plurality of multiplier units 1101_a, 1101_b, ..1101_{k-1}, performs multiplication of the mantissas ma_i and mb_i as explained with respect to FIGs 3 and 5. However, each multiplier unit 1101_i generates two intermediate mantissa products, a first intermediate mantissa product m_i' and a second intermediate mantissa products m_i'' such that the sum of m_i' and m_i'' generates the full precision mantissa product mab_i. This feature exploits the fact that hardware multipliers typically operate based on shifting and adding, such that the final calculation step is normally an addition of two numbers. In the present example, that final addition can be omitted, because the multiplication is followed by an addition anyway, so two inputs can be sent to the subsequent addition, rather than one (i.e. the multiplication result is in carry-save form). This increases the number of values to be summed in the next stage, but reduces the size of the multiplication units required to implement the dot product unit which may be desirable to reduce latency or area of the implementation. As will be apparent from consideration of FIGs 3 and 5, this will result in 2k product numbers comprising k product numbers z_i' and k product numbers z_i'' being output from the mantissa multiplication unit, which are generated by rounding or padding the intermediate mantissa products m_i' and m_i''. To ensure the same precision as using a fully resolved multiplication output (i.e. the full mantissa product mab_i), the carry-save outputs (i.e. the intermediate mantissa products m_i and m_i') are output with a bit length r for product numbers z_i' and z_i'' that is extended by one precision bit compared to that detailed above with respect to using fully resolved multiplication outputs.

[0135] Further, the product numbers (z_i' and z_i'') from each multiplier unit 1101_i are provided to the shifter unit 1106_i in the alignment unit. Each shifter unit 1106 may comprise two shifters for shifting the product number z_i' and z_i''. In another example the shifter unit may comprise only one shifter unit and the product number z_i' and z_i'' may be provided sequentially to be shifted by the shifting unit.

[0136] The alignment unit comprising the plurality of shifter units 1106_a, 1106_b,... 1106_{k-1} converts each product number z_i' and z_i'' to generate 2k numbers y_i having a second format. The 2k numbers y_i may be represented for simplicity as k numbers y_i' and k numbers y_i'' generated based on the product number z_i' and z_i''. Each shifter unit aligns the numbers y_i' and y_i'' having the second format based on the exponent sum and the max exponent sum as explained in detail in

FIG. 3 and 5, to generate integers v_i' and v_i''. Further the numbers v_i' and v_i'' in the second format are provided to the accumulator 1108 which i 'is a processing unit as shown in Fig 3' or 'is an adder as shown in Fig 5.

[0137]   The aligned number v_i' and v_i'' in the second format are further added by the processing unit to obtain the output o'. The output is further normalized by the normalizing unit 1110 to generate a normalized floating point number as the final output based on the exponent sum and the max exponent sum as explained in detail in FIG. 3 and 5.

[0138]   The architecture eliminates a final step of generating a fully resolved multiplication output by the multiplication unit. Instead an intermediate mantissa product which is a carry save representation is converted and added together. The architecture 1110 reduces the carry propagate adder delay and area. However, the architecture 1110 requires twice as many shifting operations, and thus twice as many shifters (to avoid additional latency), as required for the architecture 300 or 500.

[0139]   In another embodiment, each multiplier unit 1101_i among the plurality of multiplier units 1101_a, 1101_b, ..1101_{k-1}, in the dot product unit shown in FIG. 11, performs multiplication of the mantissas ma_i and mb_i to generate two product numbers (z_i' and z_i'') each having a bit length $r+ \log(k-1)+2$ bits compared to the $r+1$ bits as explained in the above paragraph with reference to FIG.11. As discussed above in connection with the fully-resolved multiplier example, this maintains additional precision from the multiplication into the accumulation stage of the dot product unit, by extending the multiplication output LSBs as far as the minimum additional LSB extension that would otherwise be performed at the accumulation stage. Further, the product numbers (z_i' and z_i'') from each multiplier unit 1101_i are provided to the shifter unit 1106_i in the alignment unit. The alignment unit further perform the steps as explained with respect to FIG. 3, FIG. 5 and FIG. 11 to generate integers v_i' and v_i''. Further the numbers v_i' and v_i'' in the second format are further processed to obtain the output of the dot product.

[0140]   FIG 8 is a graph illustrating a comparison of the implementation of the architecture 300 with other standard architectures for processing a set of floating-point numbers. This is particularly relevant to the accumulation aspect of a dot product unit. However, for clarity, it is noted that FIG. 8 is discussed more generally than the context of a dot product unit.

[0141]   In the FIG. 8 a graphical representation of results of a first experiment comparing the implementation of the architecture 300 explained in FIG. 3 with other standard architectures is shown. The first experiment includes comparing area versus delay trade-offs of the different architectures. As shown in FIG.8, in the first experiment, three architectures (arch 1, arch 2 and arch 3) are used for comparison. The arch 1 is an architecture of a balanced tree of floating-point adder implementation (with pairwise round to nearest and tie to even implementation). The results obtained for arch 1 are represented by cross (+) symbols on a first curve in FIG. 8. The arch 2 is another architecture of a balanced tree of floating-point adder implementation (with pairwise faithful rounding implementation). The results obtained for arch 2 are represented by circle (o) symbols on a second curve shown in FIG. 8. The arch 3 is the implementation of the architecture 300 disclosed in this document (with faithful rounding implementation). The results obtained for arch 3 are represented by square symbols on a third curve in FIG. 8. In the first experiment, the three architectures are implemented in software using VHDL.

[0142]   In the first experiment, a set of floating-point numbers in a first format of single precision (32-bit) floating point number was used as input. Each floating-point number comprised a mantissa m_i having bitlength 'r' of 24 bits (r=24 bits), exponent 'e_i' having a bitlength 't' 8 bits (t=8 bits) and a sign bit. The first experiment included synthesising the three architectures for various timing targets so as to observe area versus delay trade-offs. From the graph in FIG. 8 it is observed that the arch 3 (architecture 300) has the least delay and least area. In particular, the fastest circuit synthesized from architecture 300 uses less than 50% the area and has less than 50% the delay of the fastest circuits synthesized from the other architectures under consideration.

[0143]   Further, the complexity of the hardware implementation of the different architectures are compared. The complexity of the hardware implementation such as the critical path is expressed using the Big O Notation. For the architecture 300, the maximum exponent detection unit 304 is implemented with $O(\log(k) \log(t))$ gates on the critical path (where k is the number of values being summed by the adder). Further, the alignment unit 306 is implemented with $O(\log(r))$ gates on the critical path. The processing unit (308) i.e. the adder 508 is implemented with $O(\log(k) + \log(r))$ gates on the critical path. The normalizing unit 310 is implemented with $O(\log(t))$ gates on the critical path. Thus, the total hardware implementation could be implemented with $O(\log(k) \log(t) + \log(r))$ logic gates on its critical path. For increasing array size k and mantissa width r, the critical path is asymptotically shorter than architectures of a balanced tree of floating-point adders.

[0144]   For the same input of the set of k floating point numbers in the first format, a straightforward implementation of a multiple-input adder consisting of a binary tree of floating-point adders (with a fixed rounding mode, for instance rounding towards zero) is explained below. By construction, the implementation produces a pairwise faithfully rounded sum. The critical path in a balanced tree of floating-point adders goes through $O(\log(k))$ adders, each adder featuring $O(\log(rt))$ gates on its critical path. In total, architectures of a balanced tree of floating-point adders thus have $O(\log(k) \log(rt))$ logic gates on their critical path.

[0145]   Further, the implementation of the architecture 300 generates output having a precision not worse than the

worst case of the pairwise addition performed while using the architecture 100 of binary tree adders and multipliers or the architecture 200 of fused multiplication and addition units with faithful rounding. A mathematical proof for the precision of the architecture 300 is provided later. It is shown that the accuracy of the floating-point summation result is not lower than the worst-case pairwise floating-point addition with a faithful rounding scheme. This means that for any given array to be summed, performing pairwise addition by iteratively replacing two terms in the array by their sum rounded to the nearest larger or smaller representable value can always yield a result less precise or equal to the result generated by addition as part of the architecture 300 disclosed. An imprecise choice of ordering the inputs and for performing the step of rounding, in known architectures, is to add numbers in increasing magnitude to the largest one and always round in the same direction. As the precision of intermediate multiplication results is also not less than in architectures 100 or 200, the precision of output obtained by the architecture 300 is not less precise than that the result obtained by making these choices.

[0146] The delay and area performance of the architecture 300 dramatically improves compared to a tree of floating-point adders by removing intermediate normalisation steps and replacing intermediate carry propagation steps with a single carry-save addition as shown in FIG.8. The empirical precision of the architecture 300 as described above is shown to significantly outperform trees of floating-point adders as measured on Gaussian distributed inputs centred around zero.

[0147] Finally, the architecture 300 is commutative for addition, such that any order of input pairs (a_i, b_i) yields the same output. This leads to better reproducibility of result, as the order in which floating point numbers in two sets are bound to the inputs to the architecture 300 does not influence the result.

[0148] A mathematical proof for the precision of the architecture 300 is provided below. In the section below it is demonstrated that the precision of our algorithm is not less than the worst-case iterated pairwise addition with faithful rounding.

[0149] Firstly, some basic property of faithful rounding schemes are defined and proved. Let $F1$, $F2 \subseteq R \cup \{ \pm \infty \}$ be two number formats and $r \in R \cup \{ \pm \infty \}$ be a number. We say that q is a faithful rounding of r in format F, written $q \approx_F r$, when q is the least upper bound or the greatest lower bound of r in F.

[0150] We say that F1 is finer than F2 in the neighbourhood of r when the least upper bound and greatest lower bound of r in F2 belong to F1. The following proposition follows straightforwardly.

[0151] **Proposition 1:** If F1 is finer than F2 in the neighbourhood of r then for all values q1, q2 such that $q1 \approx_{F1} r$ and $q2 \approx_{F2} q1$ we have $q2 \approx_{F2} r$.

[0152] Now let H be a floating-point format with 't' exponent bits, 'r' mantissa bits including R fractional bits (and r-R explicit leading bits) and an exponent bias 'c' used at the input and output of our computation. We assume that the mantissa is normalised and its precision reduced to hold at most R+1 non-zero consecutive bits. In other words, numbers in format H have a mantissa such that at least one of the r-R leading bits is '1' and when i MSBs are '0', for some $0 \leq i \leq r - R - 1$, then $r - R - i$ LSBs are '0'. Further, a set of 'k' floating point numbers $x_0,...,x_{k-1}$ in format H is given as input. The algorithm proceeds by conversion to a fixed-point format G aligned on the largest exponent e_{max} in the array.

[0153] Numbers in format G are given as a signed integer v over $r + \lceil \log(k) \rceil + \lceil \log(k - 1) \rceil + 2$ bits, taking its value in the range $-2^{r+\lceil \log(k) \rceil + \lceil \log(k-1) \rceil + 1}, ..., 2^{r+\lceil \log(k) \rceil + \lceil \log(k-1) \rceil + 1} - 1$ and representing the real number $2^{e_{max}-R-c-\lceil \log(k-1) \rceil - 1} v$. Each input floating point number x_i is converted to a fixed point value $2^{e_{max}-R-c-\lceil \log(k-1) \rceil - 1} v_i \approx_G x_i$, where the choice of rounding is left to the implementation.

[0154] The fixed point values are then added together and their sum converted back to the original format hence producing the result $y \approx_H 2^{e_{max}-R-c-\lceil \log(k-1) \rceil - 1} \sum_{i=0}^{k-1} v_i$, where the choice of rounding is again left to the implementation.

[0155] For the purpose of this analysis numbers in the input array are classified in two categories: small numbers, whose absolute value is less than $2^{e_{max}-\lceil \log(k-1) \rceil - 1}$, and large numbers, whose exponent is at least $2^{e_{max}-\lceil \log(k-1) \rceil - 1}$. The input array is partitioned into an array of small numbers $x'_0, ..., x'_{k'-1}$ and an array of large numbers $x'_{k'}, ..., x'_{k-2}, x'_{k-1}$ such that the exponent of x'_{k-1} is e_{max}. The count k' of small numbers verifies $0 \leq k' \leq k - 1$. For all $i = 0,...,k - 1$, n'_i and e'_i denotes the mantissa and exponent of x'_i respectively, and v'_i the conversion

of x'_i to the fixed-point format G, such that $2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_i \approx_G x'_i$. Remark that while small numbers may incur a rounding error, large numbers are represented exactly. This is because mantissas are normalised, hence any bit of weight less than $2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}$ in a large number is guaranteed to be '0'.

[0156] A sequence w_i is constructed by letting $w_0 = 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{k-1}$, $w_i = w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}$ for i = 0,...,k - 2, and w_{k-1} = y. Recall that $y \approx_H 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}\sum_{i=0}^{k-1}v_i$, hence $w_{k-1} \approx_H w_{k-2} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{k-2}$. Two lemmas regarding the magnitude of intermediate sums is proved in this decomposition.

[0157] First, it is demonstrated that no underflow can happen in format G when adding small numbers to the one with largest exponent.

[0158] **Lemma** 1: If x_l is normalised, then

$$\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq 2^{e_{max}-c-1}$$

for all i = 1, ..., k' - 1.

[0159] **Proof:** It is shown by induction that $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq 2^{e_{max}-c} - i2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$ for all i = 1, ..., k' - 1. Let i be an integer between 1 and k' - 1. Firstly, $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq |w_{i-1}| - 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}|v'_{i-1}|$ by triangular inequality. Then, as $i \leq k'$ - 1 we have $|x'_{i-1}| < 2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$ by hypothesis. Moreover, $\pm 2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$ are representable in G, hence after rounding we have $\left|2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{k-1}\right| \leq 2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$.

> If i = 1, then $w_0 = 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{k-1} = x_l$ as x_l is representable in G, and in turn $|x_l| \geq 2^{emax-c}$ as x_l is normalised.

> If i > 1, then $w_{i-1} = 2^{e_{max}-c} - (i-1)2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$ by induction hypothesis.

[0160] In either case, $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq 2^{e_{max}-c-1} - i2^{e_{max}-c-\lceil\log(k-1)\rceil-1}$ is obtained, which concludes the induction. As an immediate consequence we have $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq 2^{e_{max}-c-1}\left(2 - i2^{-\lceil\log(k-1)\rceil}\right) \geq 2^{e_l-c-1}\left(2 - \frac{i}{k-1}\right)$, so that $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \geq 2^{e_{max}-c-1}$ for all i = 1, ..., k' - 1 as $k' \leq k - 1$.

[0161] Next, we show that no overflow can happen in format G when adding all other numbers to the one with largest exponent.

[0162] **Lemma 2:** If x_l is normalised, then $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{i-1}\right| \leq 2^{e_{max}-c+\lceil\log(k)\rceil+r-R} - 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}$ for all i = 1, ... , k - 1.

[0163] **Proof:** Remark that $|x_i| \leq 2^{emax-c+r-R} - 2^{emax-c-R}$ for all i = 0, ..., k - 1 as e_{max} is the largest input exponent, and in turn $2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}|v'_i| \leq 2^{e_{max}-c+r-R} - 2^{e_{max}-c-R}$ as the value $\pm(2^{emax-c+r-R} - 2^{emax-c-R})$ is also

representable in G. It follows that $\left|w_{i-1} + 2^{e_{max}-R-c-\lceil \log(k-1)\rceil-1}v'_{i-1}\right| \le i(2^{e_{max}-c+r-R} - 2^{e_{max}-c-R})$ for all $i = 1, \dots, k - 1$ by straightforward induction on i. In turn

$$\left|w_{i-1} + 2^{e_{max}-R-c-\lceil \log(k-1)\rceil-1}v'_{i-1}\right| \le k(2^{e_{max}-c+r-R} -$$

$$2^{e_{max}-c-R}) \le 2^{e_{max}-c+\lceil \log(k)\rceil+r-R} - 2^{e_{max}-R-c-\lceil \log(k-1)\rceil-1}.$$

[0164] The worst-case precision of the summation of architecture 300 is stated and proved in the following theorem.

[0165] **Theorem** 1: For any array $x_0, \dots, x_{k-1}$ there exist a pairwise faithfully rounded sum z of $x_0, \dots, x_{k-1}$ such that applying to architecture 300 a set of multiplication results $x_0, \dots, x_{k-1}$ yields an output y such that

$$\left|y - \sum_{i=0}^{k-1} x_i\right| \le \left|z - \sum_{i=0}^{k-1} x_i\right|.$$

[0166] **Proof:** Consider the sequence $l_0, \dots, l_{k-1}$ such that $l_0 = x'_{k-1}$ and l_i is the greatest lower bound of $l_{i-1} + x'_{i-1}$ in H for all $i = 1, \dots, k - 1$, and the sequence $u_0, \dots, u_{k-1}$ such that $u_0 = x_l$ and u_i is the least upper bound of $u_{i-1} + x'_{i-1}$ in H for all $i = 1, \dots, k - 1$. These sequences define the pairwise faithfully rounded sums l_{k-1} and u_{k-1}, obtained by systematically rounding intermediate sums in the same direction. It is clear that $l_{k-1} \le \sum_{i=0}^{k-1} x_i \le u_{k-1}$. We claim that $l_i \le w_i \le u_i$ for all $i = 0, \dots, k - 1$, so that $l_{k-1} \le y \le u_{k-1}$. This immediately gives us $\left|y - \sum_{i=0}^{k-1} x_i\right| \le \left|z - \sum_{i=0}^{k-1} x_i\right|$ for at least one of $z = l_{k-1}$ or $z = u_{k-1}$.

[0167] We now show by induction that $l_i \le w_i \le u_i$ for all $i = 0, \dots, k - 1$.

> $i = 0$: we have $l_0 = u_0 = x_l$ by definition and $w_0 = x_l$ as $w_0 \approx_G x'_{k-1}$ and x'_{k-1} is representable in G.

> $i = 1, \dots, k' - 1$: by induction hypothesis we have $l_{i-1} \le w_{i-1} \le u_{i-1}$, and in turn $l_{i-1} + x'_{i-1} \le w_{i-1} + x'_{i-1} \le u_{i-1} + x'_{i-1}$. Moreover by Lemmas 1 and 2 we have $2^{e_{max}-c-1} \le |w_i| \le 2^{e_{max}-c+\lceil \log(k)\rceil+r-R} - 2^{e_{max}-R-c-\lceil \log(k-1)\rceil-1}$ so that G is finer than F in the neighbourhood of $w_{i-1} + x'_{i-1}$ as for this exponent range G accommodates at least the same number of mantissa bits as H. Since G is a fixed-point format and the sum does not overflow or underflow we also have that rounding x'_i and then adding the result to w_{i-1} is equivalent to adding x'_{i-1} to w_{i-1} and then rounding the result. We have that l_i is the greatest lower bound of $l_{i-1} + x'_{i-1}$ in H, and less or equal to the greatest lower bound of $l_{i-1} + x'_{i-1}$ in G. Similarly u_i is the least upper bound of $u_{i-1} + x'_{i-1}$ in H, and greater or equal to the least upper bound of $u_{i-1} + x'_{i-1}$ in G. Hence by definition of a faithful rounding we have $l_i \le w_i \le u_i$.

> $i = k', k' + 1, \dots, k - 2$: by induction hypothesis we have $l_{i-1} \le w_{i-1} \le u_{i-1}$, and in turn $l_{i-1} + x'_{i-1} \le w_{i-1} + x'_{i-1} \le u_{i-1} + x'_{i-1}$. Since x'_i is a large number, it is representable in G and following Lemma 2 we have $w_i = w_{i-1} + x'_{i-1}$. As $l_i \le l_{i-1} + x'_{i-1}$ and $u_i \ge u_{i-1} + x'_{i-1}$ we obtain $l_i \le w_i \le u_i$.

> $i = k - 1$: It follows from the induction hypothesis that $l_{k-2} + x'_{k-2} \le w_{k-2} + x'_{k-2} \le u_{k-2} + x'_{k-2}$. Using a similar argument as previously we have either $w_{k-2} + x'_i = w_{k-2} + 2^{e_{max}-R-c-\lceil \log(k-1)\rceil-1}v'_{k-2}$ or G

is finer that H in the neighbourhood of w_{k-2} + x'_{k-2} and

$$w_{k-2} + x'_{k-2} \approx_G w_{k-2} + 2^{e_{max}-R-c-\lceil\log(k-1)\rceil-1}v'_{k-2}$$. Then from Proposition 1 we obtain

$w_{k-1} \approx_H w_{k-2} + x'_{k-2}$ and in turn $l_{k-1} \leq w_{k-1} \leq u_{k-1}$ by definition of a faithful rounding.

[0168] Consider a floating-point format H' finer than H, obtained by extending the bit length of H with further mantissa bits at the LSB side or allowing more than R+1 consecutive non-zero bits. Then the worst-case precision of the summation of architecture 300 configured with format H' considering arbitrary rounding directions is at least as high the worst-case precision of the summation of architecture 300 configured with format H considering arbitrary intermediate rounding directions. This is because any value rounded to format H can assume a range at least as wide as the same value rounded to format H'. Hence the range of possible values output of the summation under all possible rounding directions when using format H' are enclosed in the range of values that can be output when using format H.

[0169] In a conventional architecture 100 using separate multiplication and addition, each multiplication result is rounded/padded to include P fractional mantissa bits. Hence, after rounding/padding, the output mantissa of a multiplication in a conventional architecture 100 will be of the normalised form and have up to P+1 consecutive non-zero bits. When configured to emulate architecture 100, the multiplication unit 301 as part of architecture 300 produces at least r=P+2 mantissa bits including R=P fractional bits with possibly more than R+1=P+1 consecutive non-zero bits, resulting in a format H' possibly finer than H. Hence every input to the alignment unit 306 is enclosed between the possible rounded values a multiplier in a conventional architecture 100 can produce. It follows that the overall precision of the dot product implementation 300 is enclosed between the smallest and largest possible values considering arbitrary accumulation order and rounding direction in a conventional architecture 100. In other words, it guarantees a pairwise faithfully rounded accuracy.

[0170] In a conventional architecture 200 using fused multiplication and addition, each intermediate multiplication result includes the full p+q fractional mantissa bits. Hence the internal mantissa of a multiplication in a conventional architecture 200 will be of the normalised form and have up to p+q+2 consecutive non-zero bits. When configured to emulate architecture 200, the multiplication unit 301 as part of architecture 300 and produces r=max(Q+2,p+q+3) mantissa bits including R=max(Q,p+q) fractional bits with at most R+1=max(Q+1,p+q+2) consecutive non-zero bits, resulting in a format H. Hence every input to the alignment unit 306 holds the same value as the intermediate multiplication result in a conventional architecture 200. It follows that the overall precision of the dot product implementation 300 is enclosed between the smallest and largest possible values considering arbitrary accumulation order and rounding direction in a conventional architecture 200. In other words, it guarantees a triplet-wise faithfully rounded accuracy.

[0171] Figure 9 shows a computer system in which the graphics processing systems described herein may be implemented. The computer system comprises a CPU 902, a GPU 904, a memory 906 and other devices 914, such as a display 916, speakers 918 and a camera 908. A processing block 910 (corresponding to processing blocks 110) is implemented on the GPU 904. In other examples, the processing block 910 may be implemented on the CPU 902. The components of the computer system can communicate with each other via a communications bus 920. A store 912 (corresponding to store 112) is implemented as part of the memory 906.

[0172] While Fig. 9 illustrates the implementation of a graphics processing system, it will be understood that a similar block diagram could be drawn for an artificial intelligence accelerator system - for example, by replacing the GPU 904 with a Neural Network Accelerator (NNA), or adding the NNA as an additional unit. In such cases, the architecture 300 of the adder can be implemented in the NNA.

[0173] The adder described herein may be embodied in hardware on an integrated circuit. Generally, any of the functions, methods, techniques, or components described above can be implemented in software, firmware, hardware (e.g., fixed logic circuitry), or any combination thereof. The terms "module," "functionality," "component", "element", "unit", "block" and "logic" may be used herein to generally represent software, firmware, hardware, or any combination thereof. In the case of a software implementation, the module, functionality, component, element, unit, block, or logic represents program code that performs the specified tasks when executed on a processor. The algorithms and methods described herein could be performed by one or more processors executing code that causes the processor(s) to perform the algorithms/methods. Examples of a computer-readable storage medium include a random-access memory (RAM), read-only memory (ROM), an optical disc, flash memory, hard disk memory, and other memory devices that may use magnetic, optical, and other techniques to store instructions or other data and that can be accessed by a machine.

[0174] The terms computer program code and computer readable instructions as used herein refer to any kind of executable code for processors, including code expressed in a machine language, an interpreted language, or a scripting language. Executable code includes binary code, machine code, bytecode, code defining an integrated circuit (such as

a hardware description language or netlist), and code expressed in a programming language code such as C, Java or OpenCL. Executable code may be, for example, any kind of software, firmware, script, module or library which, when suitably executed, processed, interpreted, compiled, executed at a virtual machine or other software environment, cause a processor of the computer system at which the executable code is supported to perform the tasks specified by the code.

[0175] A processor, computer, or computer system may be any kind of device, machine or dedicated circuit, or collection or portion thereof, with processing capability such that it can execute instructions. A processor may be any kind of general purpose or dedicated processor, such as a CPU, GPU, NNA, System-on-chip, state machine, media processor, an application-specific integrated circuit (ASIC), a programmable logic array, a field-programmable gate array (FPGA), or the like. A computer or computer system may comprise one or more processors.

[0176] It is also intended to encompass software which defines a configuration of hardware as described herein, such as HDL (hardware description language) software, as is used for designing integrated circuits, or for configuring programmable chips, to carry out desired functions. That is, there may be provided a computer readable storage medium having encoded thereon computer readable program code in the form of an integrated circuit definition dataset (which may also be referred to as a hardware design) that when processed (i.e. run) in an integrated circuit manufacturing system configures the system to manufacture a computing device comprising any apparatus described herein. An integrated circuit definition dataset may be, for example, an integrated circuit description.

[0177] Therefore, there may be provided a method of manufacturing, at an integrated circuit manufacturing system, an architecture of adder as described herein. Furthermore, there may be provided an integrated circuit definition dataset that, when processed in an integrated circuit manufacturing system, causes the method of manufacturing an adder to be performed.

[0178] An integrated circuit definition dataset may be in the form of computer code, for example as a netlist, code for configuring a programmable chip, as a hardware description language defining hardware suitable for manufacture in an integrated circuit at any level, including as register transfer level (RTL) code, as high-level circuit representations such as Verilog or VHDL, and as low-level circuit representations such as OASIS (RTM) and GDSII. Higher level representations which logically define hardware suitable for manufacture in an integrated circuit (such as RTL) may be processed at a computer system configured for generating a manufacturing definition of an integrated circuit in the context of a software environment comprising definitions of circuit elements and rules for combining those elements in order to generate the manufacturing definition of an integrated circuit so defined by the representation. As is typically the case with software executing at a computer system so as to define a machine, one or more intermediate user steps (e.g. providing commands, variables etc.) may be required in order for a computer system configured for generating a manufacturing definition of an integrated circuit to execute code defining an integrated circuit so as to generate the manufacturing definition of that integrated circuit.

[0179] An example of processing an integrated circuit definition dataset (e.g. a hardware design) at an integrated circuit manufacturing system so as to configure the system to manufacture an adder will now be described with respect to Figure 10.

[0180] Figure 10 shows an example of an integrated circuit (IC) manufacturing system 1002 which is configured to manufacture an adder as described in any of the examples herein. In particular, the IC manufacturing system 1002 comprises a layout processing system 1004 and an integrated circuit generation system 1006. The IC manufacturing system 1002 is configured to receive an IC definition dataset/ hardware design (e.g. defining an adder as described in any of the examples herein), process the IC definition dataset, and generate an IC according to the IC definition dataset (e.g. which embodies an adder as described in any of the examples herein). The processing of the IC definition dataset configures the IC manufacturing system 1002 to manufacture an integrated circuit embodying an adder as described in any of the examples herein.

[0181] The layout processing system 1004 is configured to receive and process the IC definition dataset/ hardware design to determine a circuit layout. Methods of determining a circuit layout from an IC definition dataset are known in the art, and for example may involve synthesising RTL code to determine a gate level representation of a circuit to be generated, e.g. in terms of logical components (e.g. NAND, NOR, AND, OR, MUX and FLIP-FLOP components). A circuit layout can be determined from the gate level representation of the circuit by determining positional information for the logical components. This may be done automatically or with user involvement in order to optimise the circuit layout. When the layout processing system 1004 has determined the circuit layout it may output a circuit layout definition to the IC generation system 1006. A circuit layout definition may be, for example, a circuit layout description.

[0182] The IC generation system 1006 generates an IC according to the circuit layout definition, as is known in the art. For example, the IC generation system 1006 may implement a semiconductor device fabrication process to generate the IC, which may involve a multiple-step sequence of photo lithographic and chemical processing steps during which electronic circuits are gradually created on a wafer made of semiconducting material. The circuit layout definition may be in the form of a mask which can be used in a lithographic process for generating an IC according to the circuit definition. Alternatively, the circuit layout definition provided to the IC generation system 1006 may be in the form of computer-readable code which the IC generation system 1006 can use to form a suitable mask for use in generating an IC.

[0183] The different processes performed by the IC manufacturing system 1002 may be implemented all in one location, e.g. by one party. Alternatively, the IC manufacturing system 1002 may be a distributed system such that some of the processes may be performed at different locations, and may be performed by different parties. For example, some of the stages of: (i) synthesising RTL code representing the IC definition dataset to form a gate level representation of a circuit to be generated, (ii) generating a circuit layout based on the gate level representation, (iii) forming a mask in accordance with the circuit layout, and (iv) fabricating an integrated circuit using the mask, may be performed in different locations and/or by different parties.

[0184] In other examples, processing of the integrated circuit definition dataset at an integrated circuit manufacturing system may configure the system to manufacture an adder without the IC definition dataset being processed so as to determine a circuit layout. For instance, an integrated circuit definition dataset may define the configuration of a recon-figurable processor, such as an FPGA, and the processing of that dataset may configure an IC manufacturing system to generate a reconfigurable processor having that defined configuration (e.g. by loading configuration data to the FPGA).

[0185] In some embodiments, an integrated circuit manufacturing definition dataset/ hardware design, when processed in an integrated circuit manufacturing system, may cause an integrated circuit manufacturing system to generate a device as described herein. For example, the configuration of an integrated circuit manufacturing system in the manner described above with respect to Figure 10 by an integrated circuit manufacturing definition dataset may cause a device as described herein to be manufactured.

[0186] In some examples, an integrated circuit definition dataset could include software which runs on hardware defined at the dataset or in combination with hardware defined at the dataset. In the example shown in Figure 10, the IC generation system may further be configured by an integrated circuit definition dataset/ hardware design to, on manufacturing an integrated circuit, load firmware onto that integrated circuit in accordance with program code defined at the integrated circuit definition dataset or otherwise provide program code with the integrated circuit for use with the integrated circuit.

[0187] The implementation of concepts set forth in this application in devices, apparatus, modules, and/or systems (as well as in methods implemented herein) may give rise to performance improvements when compared with known implementations. The performance improvements may include one or more of increased computational performance, reduced latency, increased throughput, and/or reduced power consumption. During manufacture of such devices, ap-paratus, modules, and systems (e.g. in integrated circuits) performance improvements can be traded-off against the physical implementation, thereby improving the method of manufacture. For example, a performance improvement may be traded against layout area, thereby matching the performance of a known implementation but using less silicon. This may be done, for example, by reusing functional blocks in a serialised fashion or sharing functional blocks between elements of the devices, apparatus, modules and/or systems. Conversely, concepts set forth in this application that give rise to improvements in the physical implementation of the devices, apparatus, modules, and systems (such as reduced silicon area) may be traded for improved performance. This may be done, for example, by manufacturing multiple instances of a module within a predefined area budget.

[0188] The applicant hereby discloses in isolation each individual feature described herein and any combination of two or more such features, to the extent that such features or combinations are capable of being carried out based on the present specification as a whole in the light of the common general knowledge of a person skilled in the art, irrespective of whether such features or combinations of features solve any problems disclosed herein. In view of the foregoing description it will be evident to a person skilled in the art that various modifications may be made within the scope of the invention.

Claims

1. A method of performing dot product of an array of '2k' floating point numbers, $k \geq 3$, using a hardware implementation, the array comprising a first set of k floating-point numbers a_0, a_1..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1..., b_{k-1}, wherein the method comprises:

   receiving both sets of 'k' floating point numbers;
   multiplying each floating point number a_i with a floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits;
   creating a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the mantissa bit length 'r' of the product numbers (z_i), wherein the 'n' bits comprises a number of magnitude bits, wherein 'n' is

$$r + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil + x$$ bits, where x is an integer, and $x \geq 1$;

identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i;

aligning the magnitude bits of the numbers (y_i) based on the maximum exponent sum (e_{max}); and

adding the set of 'k' numbers concurrently.

2. The method as claimed in claim 1, wherein each number in the first set of k floating-point numbers a_0, a_1..., a_{k-1} comprises a mantissa (ma_i) and an exponent (ea_i) and each number in the second set of k floating-point numbers b_0, b_1..., b_{k-1} comprises a mantissa (mb_i) and an exponent (eb_i), where each mantissa (ma_i) is having a bit length of 'p' bits and each mantissa (mb_i) is having a bit length of 'q' bits.

3. The method as claimed in claim 2, wherein multiplying each floating point number a_i with the corresponding floating point number b_i comprises multiplying mantissa (ma_i) and mantissa (mb_i) to obtain an intermediate mantissa product (mab_i).

4. The method as claimed in claims 1 to 3, wherein the method of performing a dot product emulates:

the precision obtained using separate multiplication and addition units, for performing dot product having an output mantissa bit length of P bits, by setting the mantissa bit length of 'r' bits as 'r=P+2' bits; or

the precision obtained using fused multiplication and addition units, for performing dot product having an output mantissa bit length of Q bits, by setting the mantissa bit length of 'r' bits as 'r= max (Q+2, p+q+3)' bits.

5. The method as claimed in any preceding claim, wherein:

generating k product numbers (z_i) having the mantissa bit length of 'r' bits comprises:

rounding, the bits of the intermediate mantissa product (mab_i) to r bits, if p+q+2>r bits; or padding, extra least-significant bits to the bit length of the intermediate mantissa product (mab_i) to generate r bits, if p+q+2 < r bits; and/or

wherein identifying a maximum exponent sum (e_{max}) includes identifying the maximum value among k exponent sums (eab_i) where k exponent sums (eab_i) is obtained by summing exponent (ea_i) and exponent (eb_i).

6. The method as claimed in any preceding claim, wherein:

adding extra most-significant bits to the mantissa bit length 'r' of the product numbers (z_i) comprises adding at least $\lceil \log_2(k) \rceil$ number of the most-significant bits; and/or

adding extra least-significant bits to the mantissa bit length 'r' of the product numbers (z_i) comprises adding at least $\lceil \log_2(k-1) \rceil + 1$ number of the least-significant bits.

7. The method as claimed in any preceding claim, wherein the method further comprises:

calculating an output value by adding 'k' numbers (y_i);

renormalizing the output value; and

rounding the output value to represent the output value as a floating-point number.

8. The method as claimed in in any preceding claim, wherein aligning the magnitude bits of the numbers (y_i) to be based on the maximum exponent (e_{max}) comprises the steps of, for each floating-point number (i):

calculating the difference (e_d) between the maximum exponent sum (e_{max}) and each exponent sum (eab_i); and

shifting the magnitude bits of the corresponding number (y_i), to the LSB side, based on the calculated difference (e_d);

and optionally wherein the method further comprises either:

further to shifting the magnitude bits of the numbers, performing rounding or truncating the bits of the numbers that are shifted outside the bit-length of the number; or

determining a two's complement of the magnitude bits of the numbers, based on a sign bit (s_i) of each corresponding number, if the set of 'k' floating point numbers comprises signed floating-point numbers.

9. A hardware implementation for performing dot product of an array of '2k' floating point numbers, $k \geq 3$, the array comprising a first set of k floating-point numbers a_0, a,..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1..., b_{k-1}, wherein the hardware implementation comprises:

> a multiplication unit comprising a plurality of multiplier configured to:
>
>> receive both sets of 'k' floating point numbers;
>> multiply each floating point number a_i with a corresponding floating point number b_i to generate k product numbers (z_i), each product number (z_i) having a mantissa bit length of 'r' bits;
>
> a format conversion unit configured to:
> create a set of 'k' numbers (y_i) based on the k product numbers (z_i), the numbers (y_i) having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the mantissa bit length 'r' of the product numbers (z_i), wherein the 'n' bits comprises a number of magnitude bits, wherein 'n' is
>
> $$r + \lceil \log_2(k) \rceil + \lceil \log_2(k - 1) \rceil + x$$ bits, where x is an integer, and $x \geq 2$;
> a maximum exponent detection unit configured to identify a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i;
> an alignment unit configured to align the magnitude bits of the numbers based on the maximum exponent sum (e_{max}); and
> a processing unit configured to add the set of 'k' numbers concurrently to generate an output value.

10. The hardware implementation as claimed in claim 9, further comprising a renormalizing unit configured to:

> renormalize the output value; and
> round the output value to represent the output value as a floating-point number.

11. The hardware implementation as claimed in claim 9 or 10, wherein each number in the first set of k floating-point numbers a_0, a_1..., a_{k-1} comprises a mantissa (ma_i) having a bit length of 'p' bits and an exponent (ea_i) having a bit length of 'a' bits and each number in the second set of k floating-point numbers b_0, b_1..., b_{k-1} comprises a mantissa (mb_i) having a bit length of 'q' bits and an exponent (eb_i) having a bit length of 'b' bits.

12. A method of performing dot product of an array of '2k' floating point numbers, $k \geq 3$, using a hardware implementation, the array comprising a first set of k floating-point numbers a_0, a_1..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1..., b_{k-1}, wherein the method comprises:

> receiving both sets of 'k' floating point numbers;
> multiplying each floating point number a_i with a floating point number b_i, each multiplication generating a first intermediate product number (z_i') and a second intermediate product numbers (z_i''), thereby generating 2k product numbers comprising k first intermediate product numbers (z_i') and k second intermediate product numbers (z_i''), each having a mantissa bit length of 'r+1' bits;
> creating a set of '2k' numbers comprising k first numbers (y_i') and k second numbers (y_i''), based on the 2k product numbers, each having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the mantissa bit length of the product numbers (z_i and z_i''), wherein the 'n' bits
>
> comprises a number of magnitude bits, wherein 'n' is $r + 1 + \lceil \log_2(k) \rceil + \lceil \log_2(k - 1) \rceil + x$ bits, where x is an integer, and $x \geq 1$;
> identifying a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i;
> aligning the magnitude bits of the numbers (y_i' and y_i'') based on the maximum exponent sum (e_{max}); and
> adding the set of '2k' numbers concurrently.

13. A hardware implementation for performing dot product of an array of '2k' floating point numbers, $k \geq 3$, the array comprising a first set of k floating-point numbers a_0, a_1..., a_{k-1}, and a second set of k floating-point numbers b_0, b_1..., b_{k-1}, wherein the hardware implementation comprises:

a multiplication unit comprising a plurality of multiplier configured to:

receive both sets of 'k' floating point numbers; and
multiply each floating point number ai with a corresponding floating point number b_i to generate a first intermediate product number (z_i') and a second intermediate product numbers (z_i''), thereby generating 2k product numbers comprising k first intermediate product numbers (z_i') and k second intermediate product numbers (z_i''), each having a mantissa bit length of 'r+1' bits;

a format conversion unit configured to create a set of '2k' numbers comprising k first numbers (y_i') and k second numbers (y_i''), based on the 2k product numbers, each having a bit-length of 'n' bits obtained by adding both extra most-significant bits and extra least-significant bits to the mantissa bit length of the product numbers (z_i and z_i''), wherein the 'n' bits comprises a number of magnitude bits, wherein 'n' is

$$r + 1 + \lceil \log_2(k) \rceil + \lceil \log_2(k-1) \rceil +$$ x bits, where x is an integer, and x ≥ 1;

a maximum exponent detection unit configured to identify a maximum exponent sum (e_{max}) among k exponent sums (eab_i), each exponent sum is the sum of exponents of the floating point number a_i and the floating point number b_i;

an alignment unit configured to align the magnitude bits of the numbers (y_i' and y_i'') based on the maximum exponent sum (e_{max}); and

a processing unit configured to add the set of '2k' numbers concurrently to generate an output value.

14. An integrated circuit definition dataset that, when processed in an integrated circuit manufacturing system, configures the integrated circuit manufacturing system to manufacture the hardware implementation of any of claims 9-11 or 13.

15. A non-transitory computer readable storage medium having stored thereon a computer readable description of a hardware implementation as claimed in any of claims 9-11 or 13 that, when processed in an integrated circuit manufacturing system, causes the integrated circuit manufacturing system to manufacture an integrated circuit embodying the hardware implementation..
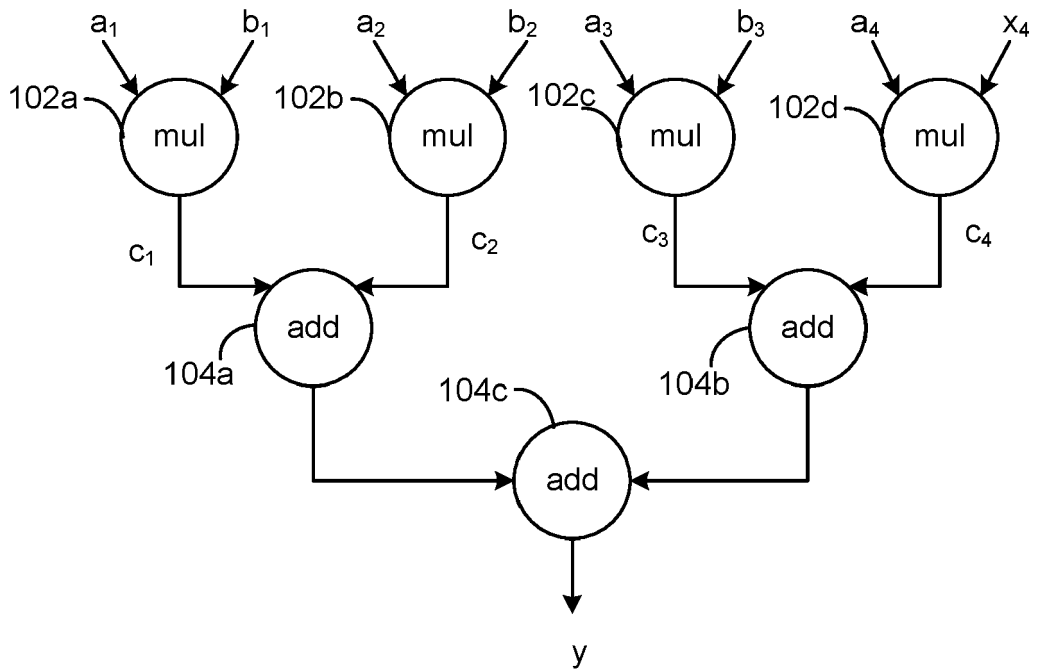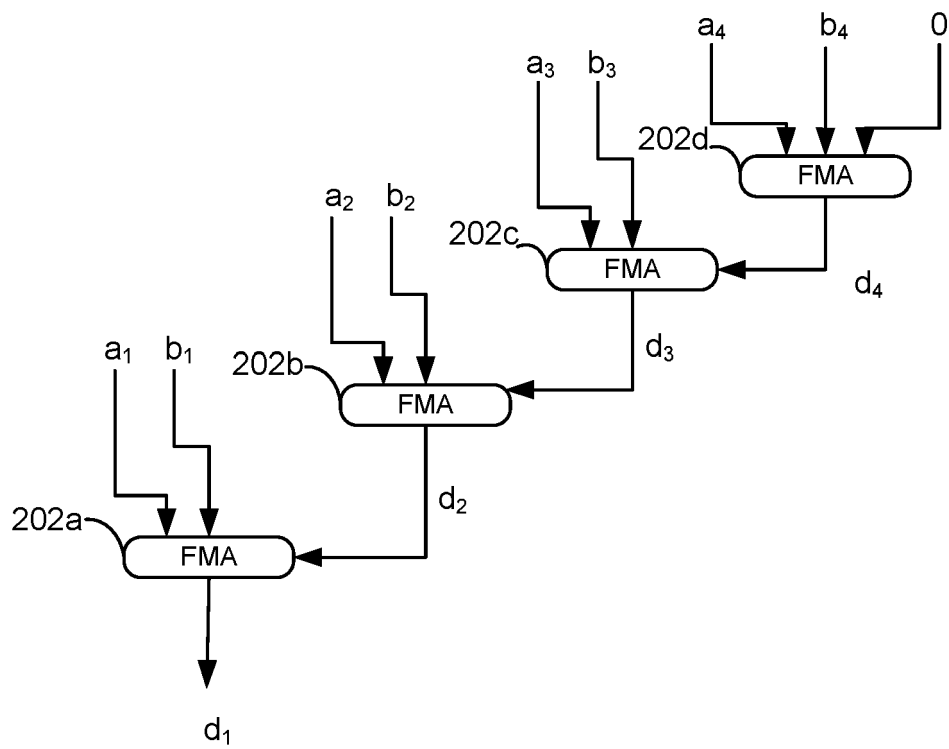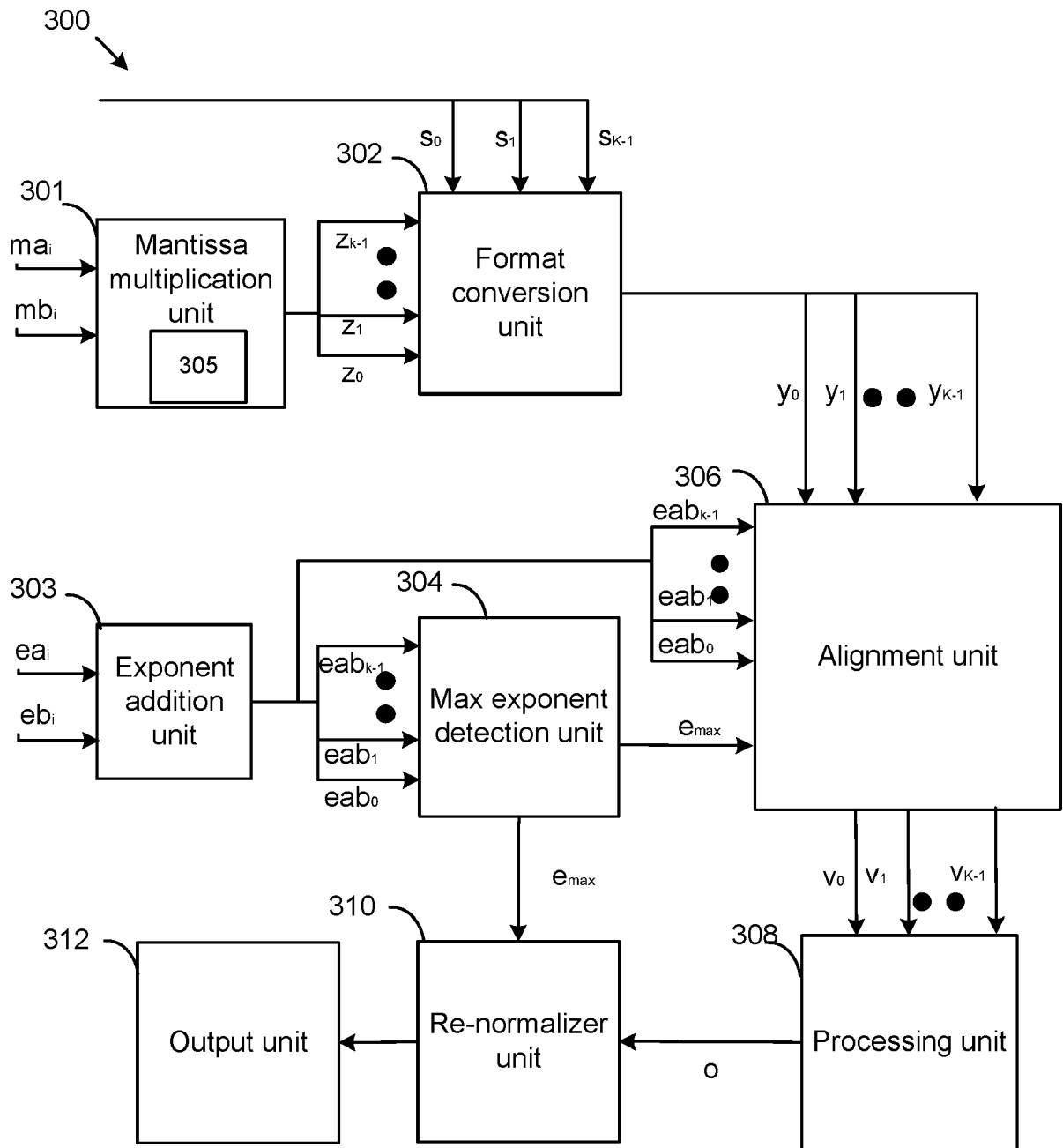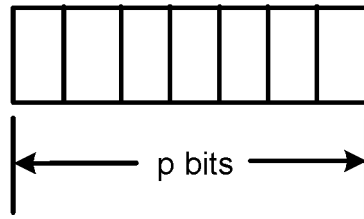
FIG. 1

FIG. 2

300

301

ma_i
mb_i

Mantissa
multiplication
unit

305

z_{k-1}
z_1
z_0

302

s_0 s_1 s_{K-1}

Format
conversion
unit

y_0 y_1 y_{K-1}

306

eab_{k-1}
eab_1
eab_0

303

ea_i
eb_i

Exponent
addition
unit

eab_{k-1}
eab_1
eab_0

304

Max exponent
detection unit

e_{max}

Alignment unit

v_0 v_1 v_{K-1}

e_{max}

312

Output unit

310

Re-normalizer
unit

o

308

Processing unit

FIG. 3

FIG. 4a

FIG. 4b

FIG. 4c

FIG. 5

1(s) log₂(k) 2(LB) log₂(k-1) 1

| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

← r bits →

FIG. 6a

1 log₂(k) 2 log₂(k-1)1

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 0 0 0

← r bits →

FIG. 6b

1 log₂(k) 2 log₂(k-1) 1

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

← r bits →

FIG. 6c

1 log₂(k) 2 log₂(k-1)1

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

← r bits →

FIG. 6d

Receive both sets of floating point numbers in an incoming format — 701

Generate k product numbers z_i in a first format having r bits — 702

Calculate the exponent sum of corresponding floating point numbers in the two sets of floating point numbers — 703

Create a set of 'k' signed numbers (v_i) based on k product numbers z_i — 704

Identify the maximum exponent sum value from exponent sums of floating point numbers — 706

Align the magnitude bits of the signed numbers based on the maximum exponent sum corresponding to the product number — 708

Process the set of 'k' signed numbers concurrently — 710

Renormalize output value to generate the mantissa and exponent of output floating point number — 712

FIG. 7

FIG. 8

902

904

914

CPU

GPU

910

Display 916

Speakers 918

Camera 908

920

906 Memory

912

FIG. 9

1002

1004

1006

IC definition
dataset

Layout
processing

Circuit layout
definition

Intergrated
circuit
generation

Integrated
circuit

FIG. 10

1100

a_0 b_0 a_1 b_1 a_{ki1} b_{k-1}

1101a

1101a $*$ $*$ •   •   •   • $*$ 1101k-1

1106a $>>$ 1106b $>>$ •   •   •   • $>>$ 1106k-1

1108   Processing unit

1110 $<<$

o

FIG. 11

Europäisches
Patentamt

European
Patent Office

Office européen
des brevets

EUROPEAN SEARCH REPORT

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
A	EP 3 719 639 A2 (INTEL CORP [US]) 7 October 2020 (2020-10-07) * paragraphs [0056] - [0062]; figure 9 * -----	1-15	INV. G06F7/483 G06F7/544
A	US 2018/300105 A1 (LANGHAMMER MARTIN [GB]) 18 October 2018 (2018-10-18) * paragraphs [0017] - [0020], [0066] - [0075]; figure 3 * -----	1-15	

TECHNICAL FIELDS
SEARCHED   (IPC)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
Munich	19 June 2023	Prins, Leendert

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT
ON EUROPEAN PATENT APPLICATION NO.

EP 23 15 5922

19-06-2023

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
EP 3719639	A2	07-10-2020	CN	111767516 A	13-10-2020
			EP	3719639 A2	07-10-2020
			US	2020310756 A1	01-10-2020
US 2018300105	A1	18-10-2018	CN	108694038 A	23-10-2018
			DE	102018108431 A1	18-10-2018
			US	2018300105 A1	18-10-2018
			US	2019196786 A1	27-06-2019

EPO FORM P0459

REFERENCES CITED IN THE DESCRIPTION

Patent documents cited in the description

- GB 2202126 A **[0001]**
- GB 2202128 A **[0001]**