UDC 004.7
DOI: 10.15587/2706-5448.2023.290127

Georgi Luckij,
Oleksandr Dolholenko

DEVELOPMENT OF FLOATING POINT OPERATING DEVICES

The paper shows a well-known approach to the construction of cores in multi-core microprocessors, which is based on the application of a data flow graph-driven calculation model. The architecture of such kernels is based on the application of the reduced instruction set level data flow model proposed by Yale Patt. The object of research is a model of calculations based on data flow management in a multi-core microprocessor.

The results of the floating-point multiplier development that can be dynamically reconfigured to handle five different formats of floating-point operands and an approach to the construction of an operating device for addition-subtraction of a sequence of floating-point numbers are presented, for which the law of associativity is fulfilled without additional programming complications. On the basis of the developed circuit of the floating-point multiplier, it is possible to implement various variants of the high-speed multiplier with both fixed and floating points, which may find commercial application. By adding memory elements to each of the multiplier segments, it is possible to get options for building very fast pipeline multipliers. The multiplier scheme has a limitation: the exponent is not evaluated for denormalized operands, but the standard for floating-point arithmetic does not require that denormalized operands be handled. In such cases, the multiplier packs infinity as the result.

The implementation of an inter-core operating device of a floating-point adder-subtractor can be considered as a new approach to the practical solution of dynamic planning tasks when performing addition-subtraction operations within the framework of a multi-core microprocessor. The limitations of its implementation are related to the large amount of hardware costs required for implementation. To assess this complexity, an assessment of the value of the bits of its main blocks for various formats of representing floating-point numbers, in accordance with the floating-point standard, was carried out.

Keywords: *floating-point multiplier, superscalar processor, associativity law, Baugh-Wooley algorithm, CISC-RISC.*

Received date: 06.09.2023
Accepted date: 30.10.2023
Published date: 31.10.2023

How to cite
Luckij, G., Dolholenko, O. (2023). Development of floating point operating devices. Technology Audit and Production Reserves, 5 (2 (73)), 11–17. doi: https://doi.org/10.15587/2706-5448.2023.290127

1. Introduction

When building the cores of most modern microprocessors with the *x86-64* architecture, the *OoOE (Out-of-Order Execution)* technology is used, which is based on the implementation of restricted data flow architecture (*Restricted Data Flow* (*RDF*)) [1, 2]. Such microprocessors are called superscalar microprocessors [3], or microprocessors with *CISC-RISC* architecture («*CISC-outside RISC-inside*»). As *CISC – Complex Instruction Set Computing*, let's consider a set of instructions in the x86-64 architecture. The term *RISC – Reduced instruction set computing* means a shortened set of commands implemented by a set of microprocessor operating devices. Instead of *RISC*, microprocessor developers use the names: *uop*, *micro-ops*, *μops*, or similar terms.

During the operation of the cores of such microprocessors, a certain number of *CISC* commands of the currently active command flow are simultaneously decoded into a set of *RISC* operations. The planning of execution of *RISC* operations is carried out in accordance with the *RDF* architecture, based on the readiness for execution of operands of *RISC* operations. Before the *RISC* operations become ready for execution, they are placed in the cells of the reservation station [4–7]. A *RISC* operation that has reached the ready state can be transferred from the backup station cell to a free operating device that can execute it. Thus, dynamic parallelism at the level of *RISC* operations is organized in modern microprocessors.

When forming streams of *CISC* commands operating with floating-point operands, both programmers and developers of optimizing compilers have to take into account the peculiarities of the implementation of floating-point arithmetic [8]. As a result of these features, for example, standard mathematical laws, such as commutative and associative [9], are not fulfilled for floating-point arithmetic, and it is not difficult to conduct them so poorly that the calculated answers consist almost entirely of «noise» [10].

So, for example, the operations of multiplication and division do not greatly increase the relative error, but the subtraction of almost equal quantities can significantly increase it. One of the consequences of the possible unreliability of the floating-point addition-subtraction operation is the violation of the associativity law: $(u+v)+w \neq u+(v+w)$ for some u, v, w. The law of distributivity connecting operations $\times$ and $+$ may also be violated: $u\times(v+w) \neq (u\times v)+(u\times w)$. Even memos for programmers containing recommendations for organizing calculations to reduce errors have been developed [11]. So, for example, if it is necessary to add-subtract a long

ISSN 2664-9969

sequence of numbers [11], it is possible to sort them and perform operations starting with the smallest numbers.

The analysis of such a note shows that it is often difficult for a programmer to follow such rules, for example, due to ignorance of the possible values of variables before the start of the program, due to the need to pre-sort numbers by size, due to the need for routine conversion of formulas, etc. The execution of such rules is not done by a programmer, and the compiler at the stages of preparing calculations for execution are also complicated, due to the same reasons. In addition, the enforcement of these rules, for example, changing the order of submission of operands, creates difficulties for the implementation of parallel calculations.

To reduce the error of adding-subtracting a long sequence of floating-point numbers, without performing their preliminary sorting, the Kahan algorithm, also known as compensatory summation, is used [12]. Error reduction is achieved by introducing an additional variable to store the increasing amount of errors. With compensatory summation, the worst-case error does not depend on the number of operands, so a large number of operand values can be summed with an error that depends only on the precision of the floating-point number representation format. But according to this algorithm, each operation of addition-subtraction of the next operand of the sequence to the intermediate sum is transformed into 4 operations of the type of addition-subtraction and 4 operations of assignment.

The aim of research is to develop a high-speed operating device of a multiplier for each microprocessor core and an inter-core adder-subtractor with a floating point, which can be used for dynamic branching of work, at the level of *RISC* operations, both universal and specialized multi-core superscalar microprocessors and at the same time, without additional software complications, will ensure the fulfillment of commutativity and associativity laws.

2. Materials and Methods

The object of research is a model of calculations based on data flow management in a multi-core microprocessor.

Fixed-point arithmetic is poorly suited for many engineering and scientific calculations. To provide a dynamic range of representation of real numbers (without the need to scale operands), a representation of a number in the form of a floating point is used.

The number of integers is infinite, but it is always possible to choose such a number of bits to represent any integer that arises when solving a particular problem. The set of real numbers is not only infinite, but also continuous, so no matter how many bits it takes, let's inevitably encounter numbers that do not have an exact representation. Floating-point numbers are one of the possible ways of representing real numbers, which is a compromise between precision and the range of accepted values.

Ensuring the fulfillment of the associativity law for the addition-subtraction operation on a sequence of floating-point numbers without additional programming complications is possible if the calculation of all intermediate results is performed without losing significant bits.

Below are described the functional circuits of the floating-point multiplier and adder-subtractor for various formats of the representation of floating-point numbers that do not require microprogram control elements in their operation.

3. Results and Discussion

3.1. Floating point multiplier. In work [13], a configuration library of nodes of the scheme from Fig. 1 when building them on the basis of PLD (programmable logic device).

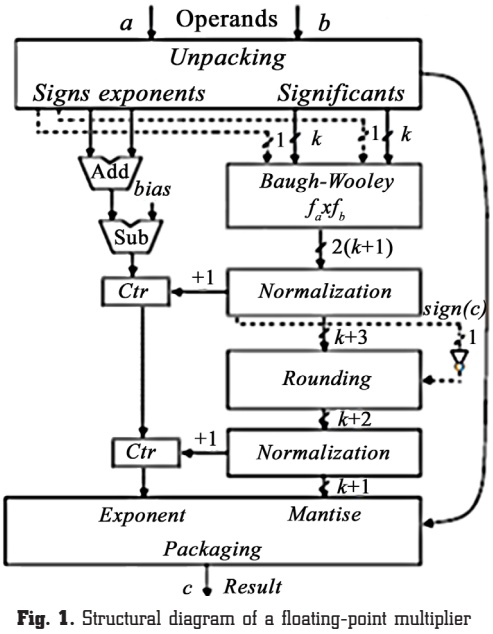

Fig. 1. Structural diagram of a floating-point multiplier

Fig. 1 shows the structural diagram of a floating-point multiplier capable of processing all formats of floating-point numbers provided by the standard [8]. There are five such formats: half precision (*SF*) – 16 bits, single precision (*F*) – 32 bits, double precision (*DF*) – 64 bits, double extended precision (*DEF*) – 80 bits and quadruple precision (*QF*) (128 digits).

Multiplication over floating-point operands $c=axb$ is reduced to the following operations:

$$(\pm f_a x 2^{(ea+bias)}) x (\pm f_b x 2^{(eb+bias)}) = \pm f_a x f_b x 2^{(ea+eb+bias)} = \pm f_c x 2^{(ec+bias)}.$$

Care must be taken when constructing a floating-point multiplier to ensure correctness and prevent undue loss of precision. In addition, handling of possible exceptions should be implemented.

Unpacking includes extracting the sign of the significant, as well as restoring the hidden *MSB* bit for each of the operands. During unpacking, the formats of the operands are also converted to the internal format of the computing module (for example, to the quadruple precision format). The unpacked operands are tested for the presence of exceptions among them: 0, *NaN*, ±∞. If there are exceptions, the result of operations is formed in the appropriate form [8] and sent to the package, bypassing multiplication.

The previous exponent of the product is calculated by adding the two shifted exponents of the operands and subtracting from the resulting shift sum:

$$(e_a+bias)+(e_b+bias)-bias=(e_a+e_b)+bias=e_c+bias.$$

The most difficult to implement in this scheme is the matrix multiplier of the significants represented by the

additional code. The *Baugh-Wooley* algorithm [14] was used for its construction.

The *Baugh-Wooley* algorithm is currently the best of the known algorithms because it increases the regularity of the multiplier structure and is well suited for implementing a matrix multiplier based on it [13].

Let's consider in detail the principle of operation of the *Baugh-Wooley* algorithm using the example of the implementation of the 5×5 multiplier (Fig. 2) [14].

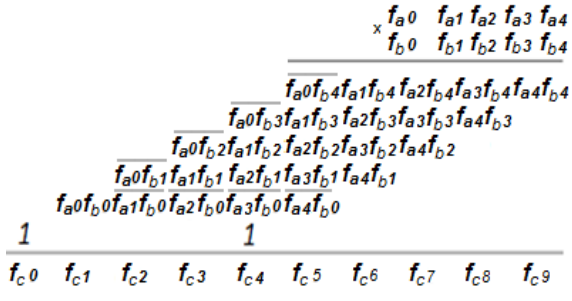

Fig. 2. An example of the *Baugh-Wooley* algorithm when implementing a 5×5 multiplier

The first 4 lines are partial products of the multiplier a by the significant bits of the multiplier b_j (j=0, 1, 2, 3) and are formed by four two-input *AND* elements (from one significant bit of a and bit b_j) and one two-input *NAND* element (from the sign bit a and bit b_j). The fifth line is a partial product of the multiplier a on the sign bit of the multiplier b_4 and is formed by one *AND* (from the sign bit a and bit b_4) and four NANDs (from one significant bit a and bit b_4).

The *Baugh-Wooley* algorithm needs correction of the result. For this, when multiplying n-bit operands, 1 should be added to the digits of the formed product with weights: 2^{2n-1} and 2^n.

According to the processed format, the significant multiplication unit has the ability to process significants (along with the sign and hidden bits) of sizes: 12, 25, 54, 66 and 114 bits. The functional scheme of such a significant multiplier is shown in Fig. 3.

Let's consider the operation of the scheme using the example of the significant multiplication operation:

$$f_c = f_a \times f_b.$$

Reconfiguration of the significant multiplier block to handle the desired operand format is done by padding the lower digits of the significants f_a and f_b with zeros to the size of the significant in the *QF* format. Thus, a 114×114 significant multiplication is always performed in the significant multiplier block. The significants f_a and f_b supplemented to the *QF* format must be fed to the inputs of the significant multiplier block as shown in the diagram of Fig. 3.

The significant multiplier scheme consists of 114×114 two-input & elements, each of which calculates one digit of intermediate products (some of these elements, according to the *Baugh-Wooley* algorithm, are changed to *NOT* & elements in Fig. 3). The digits of intermediate products calculated in this way are summed using single-bit full adders (*FA-full adder* in Fig. 3).

The product calculated by the Baugh-Wooley algorithm needs correction [13]. In our case, the correction of the product consists in the fact that «1» should be added to the 0th and 113th digits of the product, as shown in Fig. 3.

At the final stage of multiplication of significants, the spread of transfers is carried out in the upper half of the calculated product. For this, *carry-skip adders* [15] are used, which occupy an intermediate place between carry-skip adders and carry-skip adders, both in terms of speed and cost.

After multiplying two normalized significants, each of which is in the range [1, 2), the significant of the result can be in the range [1, 4). In this regard, its normalization may require a shift by one digit to the right with an increase of 1 from the previous value of the exponent of the product. After the first normalization, the 2(n+1) digit significant of the product is truncated to n+3 digits.

When rounding the significant of the product f_c, its normalization may again be lost. At the same time $|f_c|$ may be equal to 2 and its normalization may again need to be shifted one bit to the right with an increase of e_c+*bias* by 1.

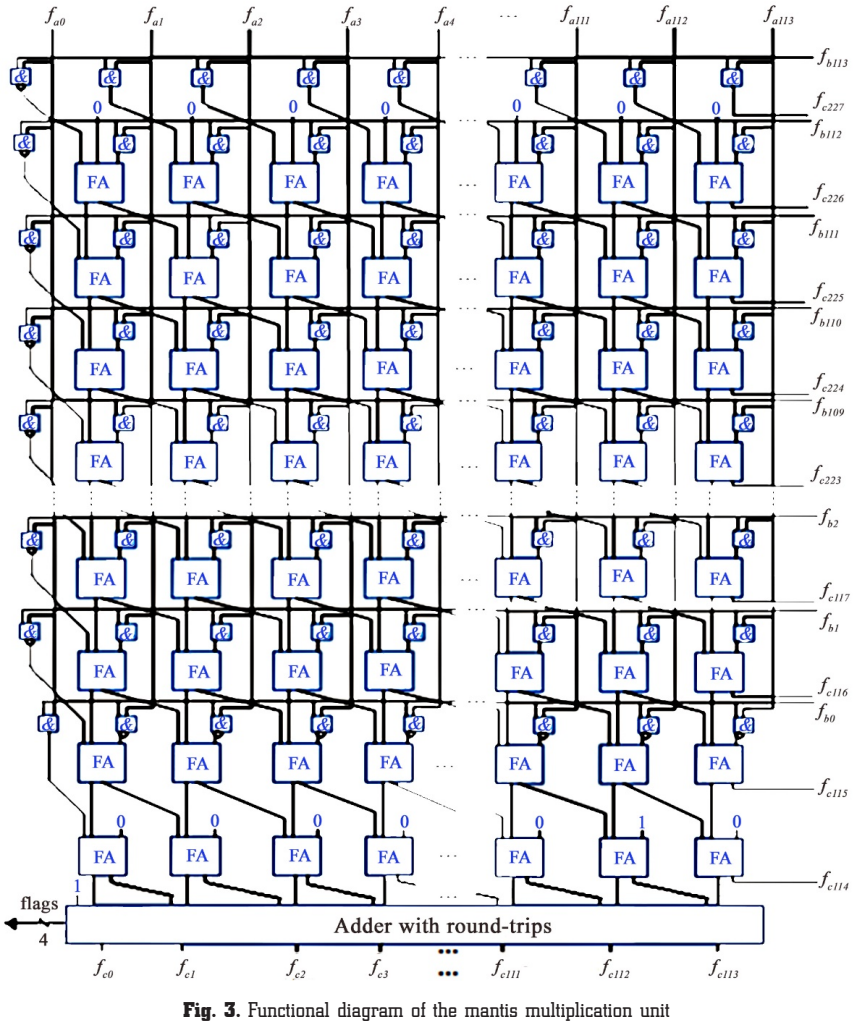

Fig. 3. Functional diagram of the mantis multiplication unit

ISSN 2664-9969

To increase speed, it is possible to precalculate the value of e_c+*bias* increased by 1 at each normalization step and choose its correct value after it is known whether an offset is required after rounding. Due to the fact that the multiplication of significants is the most complicated part of floating-point multiplication, there is enough time for such calculations. Also, rounding should not be a separate step at the end of the operation. It can be combined with the mantis reproduction apparatus.

3.2. An inter-core floating-point adder-subtractor operating device. Let's consider the possibility of building an inter-core operational device (IOD) adding-subtracting a sequence of floating-point numbers that performs calculations of all intermediate results without losing significant bits. For this, at each step of the calculation, it performs the following operation:

$$o = o \pm x,$$

where x is the next operand of the sequence, which, at each step of calculations, can be accepted at the IOD inputs for processing; o is an intermediate result of addition-subtraction of a sequence of numbers. To increase the accuracy, o in IOD will be calculated with an accuracy limited only by the range of the exponent change and the accuracy of the representation of the significant of the processed format [8]. At the beginning of the calculation of a new sequence of numbers, o will be set to zero. Simultaneously with the calculation of the new value of o, its previous value will be converted to the processed number format with rounding to the nearest [8] and output to the IOD outputs.

Let each floating-point number x have the form at the IOD input, in accordance with the standard [8]:

$$x = f_x \times 2^{ex},$$

where f_x – the n-digit normalized ($1 \le |f_x| < 2$, at $x \ne 0$) fractional part of the number x (significant); e_x – the exponent of the number (a non-negative integer from the interval $[e_{max}, 0]$); f_x and e_x are represented in direct binary. A floating-point number has two signs: the sign of the number (*sign*) is displayed by a separate bit; the sign of the exponent is reflected by the exponent bias (*bias*) [8].

The schematic solution for building IOD floating-point addition/subtraction with increased execution accuracy is shown in Fig. 4. Its work is carried out as follows.

Submission of terms is carried out one by one per cycle of operation of the IOD. So, on the i-th cycle of operation, another term (number x) is applied to the IOD inputs. At the same time, at control node 1, it is converted from the processed format of floating-point numbers [8] to the internal range of number processing r, where $r = e_{max}+1+n$ and is the number of binary digits in the representation of a fixed-point number. Namely, the normalized significant term is applied to the input f_x, the exponent of the term is applied to the input e_x, and the sign of the term is applied to the input $sign_x$. With the arrival of the leading edge of the *clock* pulse at the clock input, data is written, respectively, in the significant *RG* f_x, exponent *RG* e_x and control trigger *Tg* $sign_x$ registers. At the same time, if the number x is the starting operand of a new sequence of numbers, the value of *RG* f_0 is reset to zero using the *reset* signal.

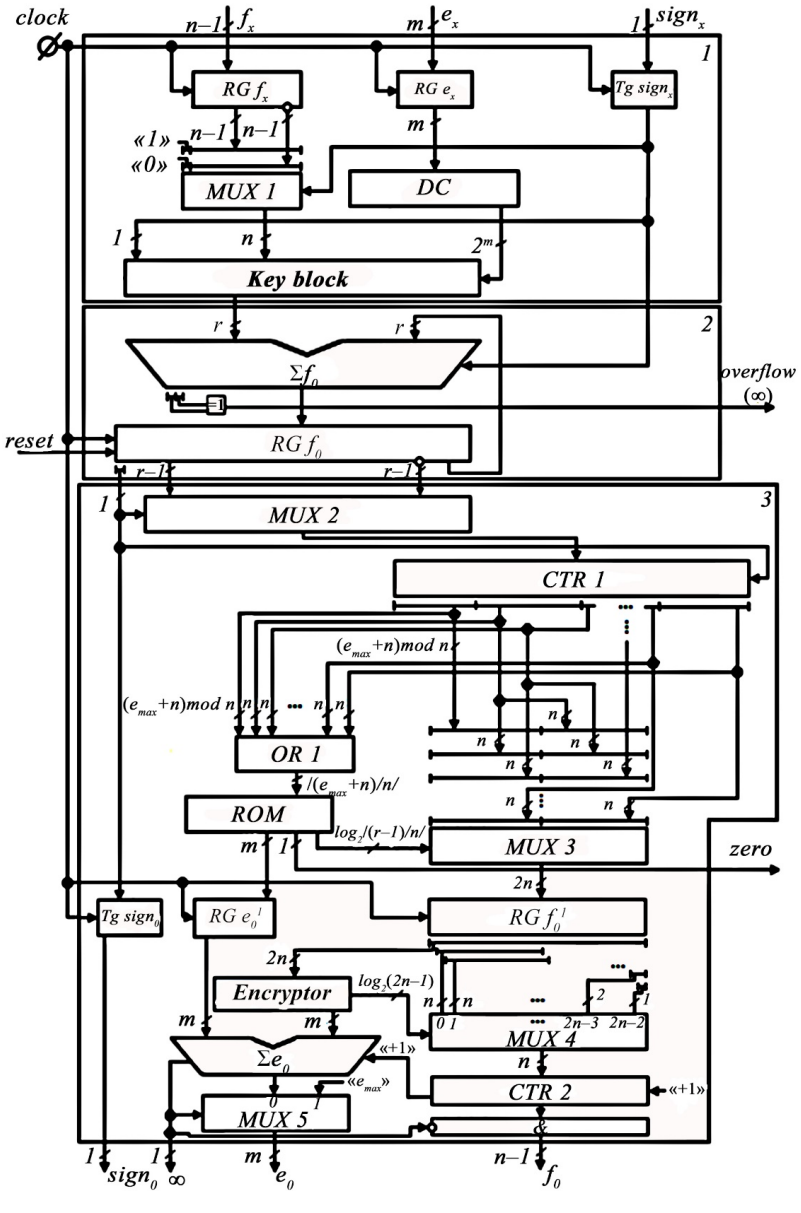

Fig. 4. Functional diagram of a floating-point addition-subtraction inter-core operating device (IOD)

Using the *MUX*1 multiplexer and depending on the value of $sign_x$, the inverted significant code f_x is transferred from the *RG* f_x to the input of the key block with the restoration of the hidden bit *MSB* [8] and $sign_x$. Depending on the value of the exponent e_x, a signal is generated at one of the outputs of the *DC* decoder, which ensures the transmission of the input of the key block to the first input of the adder Σf_o. As a result of this transfer,

an arithmetic left shift of the inverted code f_x is carried out together with the *MSB* and $sign_x$ on e_x digits, and all other lower digits of the first input of the adder $\sum f_o$ are filled with the value of $sign_x$, which together with the supply of $sign_x$ also to the input of the transfer of the lower digit $\sum f_o$ ensures the formation of the additional code x reduced to the range r.

On the $(i+1)$-th cycle of IOD operation, the next term of the calculated sequence can be accepted at its inputs. At the same time, the result of the summation of the previous term will be written into the *RG* f_o register at node 2 of the summation from the outputs $\sum f_o$, and the sign, exponent and significant of the partially normalized of the previous value of o (from the summation of the term that could be accepted at the IOD inputs in the $(i-1)$-th cycle of its operation).

Thus, the MOP is a conveyor converter of information consisting of three segments. At each cycle of IOD operation, the next value of the number x of the performed operation $o=o+x$ can be accepted at its inputs, in general, from different microprocessor cores (from the core that captured the cycle of IOD operation). If one of the cores needs to perform the operation $o=o-x$, then the $sign_x$ value applied to the IOD inputs must be changed to the opposite.

Simultaneously with the summation on $\sum f_o$ of the number x with the accumulated sum f_o, which flows from the *RG* f_o register to the second input $\sum f_o$ through the feedback circuit, an overflow signal is formed (as a sum modulo 2 of the two higher digits $\sum f_o$), which indicates about the output of a new value f_o from the range r.

Simultaneously with the formation of a new value of f_o, at node 3 of the formation of the result, the previous accumulated amount of f_o is transferred into an intermediate representation in the form of a floating point with a $2n$-bit binary code of the significant of the result f_{oI} in direct code and an m-bit value of the exponent corresponding to the significant of f_{oI}. To do this, first, the value f_o from *RG* f_o is converted into a direct code using the *MUX*2 multiplexer and the *CTR*1 counter, and $sign_o$ is removed from its representation, which is written to *Tg* $sign_o$ in the next IOD operation cycle. Then, with the help of group *OR*1 consisting of $[(e_{max}+n)/n]n$ input elements «*OR*» (the oldest element in the group has $(e_{max}+n)modn$ inputs) the input address to the permanent *ROM* memory is formed (actually turns out to be the largest non-zero group of binary bits in f_o). At this address, the value of exponent e_o^I is read from the first *ROM* outputs, which corresponds to finding the most significant digit f_o at the position of the least significant digit of the most significant non-zero group of binary digits in f_o. This read value of the exponent of e_o^I is written to *RG* e_o^I on the next cycle of IOD operation. A zero signal is read from the second ROM output at address zero, which indicates that $f_o=0$. At the same time, a zero value of exponent e_o^I is read from the first *ROM* outputs [8]. From the third outputs of the *ROM* at the address formed in *OR*1, the control information is read by the multiplexer *MUX*3. This information provides passage from the *CTR*1 outputs through *MUX*3 to the *RG* f_o^I inputs of only the most senior non-zero group of binary digits f_o and the group of digits following it. On the next cycle of IOD operation, these two groups of digits are written to *RG* f_o^I.

In the next IOD operation, the significant of the f_o result is normalized with the removal of the hidden bit, its rounding to the nearest, and the corresponding correction of the e_o value is carried out. For this, control information is generated by the *MUX*4 multiplexer using the encoder, depending on the number of zeros to the first significant bit in f_o. This information provides the passage from the outputs of *RG* f_o^I through *MUX*4 to the inputs of the counter *CTR*2 n most significant bits of f_o, shifted to the left by k bits, where k is the number of zeros to the first significant bit in f_o. At the same time, the first significant bit in f_o is not transmitted to the inputs of the *CTR*2 counter, which ensures the removal of the hidden bit. At the same time, a correction of the exponent e_o equal to $(e_{max}+n)modn-k$ is formed on the other outputs of the encoder, if the first significant bit in the direct code f_o was detected in the older group of bits, or $n-k$ – in all other groups of bits.

With the help of the adder $\sum e_o$, this correction is added to the value of e_o^I, which comes to the other inputs of the adder with *RG* e_o^I and is summed up with the value of the carryover input of the lower digit of the adder, which comes from the counter *CTR*2 and indicates the overflow of the significant f_o as a result of its rounding to the nearest [8]. The significant value f_o thus obtained, as without the *CTR*2 overflow signal, will be normalized, rounded, and with the hidden bit removed, and fed from the outputs of *CTR*2 through a group of $(n-1)$ 2-input *AND* elements to the output f_o of the IOD. At the same time, from the outputs $\sum e_o$ through input 0 of the *MUX*5 multiplexer, the value of the exponent of the intermediate result is output to the output e_o of the IOD. When the transfer signal from the higher exponent $\sum e_o$ occurs, which indicates an overflow of e_o, i. e. $e_o > e_{max}$ according to [8], a zero value is given to the output f_o of the IOD, and the value e_{max} is given to the output e_o of the IOD via input 1 of the *MUX*5 multiplexer.

3.3. Discussion. *Practical significance*. The results obtained in the course of the study can help business entities implement a multiplying operating device that can be included in the core of a superscalar microprocessor. Modern microprocessor cores, after becoming superscalar, have become so complex that only a small number of specialists, except, for example, employees of the Intel research center located in the city of Haifa (Israel) and engaged in the development of new microarchitectures, understand the details of their work. Thus, most of the world's universities study the peculiarities of the operation of modern microprocessor cores according to the work [3], which has already passed its sixth edition and has the telling title A Quantitative Approach. Features of the construction and operation of the hardware, which are part of the modern microprocessor core, are a commercial secret of the companies that produce them. In this regard, it is not easy to find in the open information space publications explaining some features of the operation of such hardware. The results obtained during the research are primarily aimed at teaching students to understand how their parallel program will be executed in a modern microprocessor core with a superscalar architecture, which they will most likely deal with in their professional activities.

ISSN 2664-9969

Limitations of research. The proposed multiplier scheme has limitations: the product is not calculated for denormalized operands, but the standard [8] does not require mandatory processing of denormalized operands. In such cases, the multiplier packs infinity as the result.

The limitations of the implementation of the inter-core operating device of the floating-point adder-subtractor are associated with a large amount of hardware costs required for its implementation. To assess this complexity in the Table 1 summarizes the bit values of its main blocks for various formats of floating-point number representation, according to [8].

Table 1

Bit values of the main IOD blocks for different formats of number representation

IOD block	Half (SF)	Single (F)	Double (DF)	Double Extended (DEF)	Quadrupled (QF)
RG f_x	10	23	52	64	112
RG e_x, RG e_{0i}^I $\sum e_{0i}$, MUX5	5	8	11	15	15
MUX1, MUX4, CTR2	11	24	53	65	113
RG f_{0i}, $\sum f_0$	43	280	2101	32833	32881
MUX2, CTR1	42	279	2100	32832	32880
OR1	4	12	40	506	291
ROM	16×8	4K×13	T×18	2^{506}×25	2^{291}×25
MUX3, RG f_0^I	22	48	106	130	226

The impact of martial law. Modern microprocessor cores with a superscalar architecture include ten operating devices that are focused on performing various scalar and vector operations with both fixed and floating point and work in parallel. To implement their microprocessor cores, *Intel* and *AMD* use technologies for the production of large integrated circuits with topological norms of 10^{-5} nm. Currently, Ukraine does not have such production technology. The full-scale invasion of the territory of Ukraine had a negative impact on their development and implementation, since a large amount of material resources of enterprises are directed to support the army, and not to the development, purchase and implementation of such technologies.

Prospects for further research. On the basis of the developed circuit of the floating-point multiplier, it is possible to implement various variants of the high-speed multiplier with both fixed and floating points, which may find commercial application. By adding memory elements to each of the multiplier segments, it is possible to get options for building very fast pipeline multipliers.

4. Conclusions

A structural diagram of a floating-point multiplier has been developed, which can be dynamically reconfigured to handle five different formats of operands, which are regulated by the standard for floating-point arithmetic [8]. The advantages of the developed circuit are that it is made as a set of combinational circuits, without the use of memory elements and microprogram control.

The entire operand multiplication operation starts and ends in one machine cycle. The inputs of the floating-point multiplier must be supplied with operands from the block of general-purpose registers, and the processed format signal must come from the reservation station, which stores the *RISC* operation of the executed *CISC* multiplication instruction. The result of the performed operation must be entered in the result reordering buffer.

To reconfigure the multiplier to process operands of the desired format, no additional actions are required, except for applying a signal of the processed format to its control inputs.

This paper also considers the possibility of building an inter-core operating device for adding-subtracting a sequence of floating-point numbers with accuracy limited only by the range of the exponent change and the accuracy of the representation of the significant of the processed format.

This approach to building an operating device for adding-subtracting a sequence of floating-point numbers looks very promising due to the simplification of the calculation process from the programmer's point of view, because it ensures the implementation of the law of associativity without additional programming complications.

The implementation of such a device can be considered as a new approach to the practical solution of dynamic planning tasks when performing addition-subtraction operations within the framework of a multi-core microprocessor.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this study, including financial, personal, authorship, or any other, that could affect the study and its results presented in this article.

Financing

The study was conducted without financial support.

Data availability

The manuscript has no associated data.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies when creating the presented work.

References

1. Patt, Y., Hwu, W. et al. (1986). Experiments with HPS, a Restricted Data Flow Micro architecture for High Performance Computers. *COMPCON 86,* 254–258.
2. Simone, M., Essen, A., Ike, A., Krishnamoorthy, A., Maruyama, T., Patkar, N. et al. (1995). Implementation trade-offs in using a restricted data flow architecture in a high performance RISC microprocessor. *ACM SIGARCH Computer Architecture News, 23 (2),* 151–162. doi: https://doi.org/10.1145/225830.224411
3. Hennessy, J. L., Patterson, D. A. (2019). *Computer Architecture: A Cuantitative Approach.* Morgan Kaufmann, 1527.
4. Kanter, D. (2012). *Intel's Haswell CPU Microarchitecture.* Available at: http://www.realworldtech.com/haswell-cpu/
5. Shen, J., Lipasti, M. (2013). *Modern Processor Design: Fundamentals of Superscalar Processors.* Waveland Press, 642.
6. Lutskyi, H. M., Dolholenko, O. M., Aksonenko, S. V., Storozhuk, V. O. (2014). Modeliuvannia obmezhenoi realizatsii arkhitektury potoku danykh v strukturi superskaliarnoho protsesora. *Visnyk NTUU «KPI». Informatyka, upravlinnia ta obchysliuvalna tekhnika, 60,* 83–94.

7. Dolholenko, A. O., Yatsun, V. O. (2016). Realizatsiia operatsiinoho prystroiu sumatora/vidnimacha z plavaiuchoiu krapkoiu dlia yadra superskaliarnoho protsesora. *Visnyk NTUU «KPI». Informatyka, upravlinnia ta obchysliuvalna tekhnika, 64,* 106–116.
8. *IEEE 754: Standard for Binary Floating-Point Arithmetic* (2019). Available at: https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/
9. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Available at: https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/02Numerics/Double/paper.pdf
10. Knut, D. (1977). *Iskusstvo programmirovaniia dlia EVM*. Vol. 2. Moscow: Mir, 724.
11. Mak-Kraken, D., Dorn, U. (1977). *Chislennye metody i programmirovanie na FORTRANE*. Moscow: Mir, 584.
12. *Strictly, there exist other variants of compensated summation as well: see Higham, Nicholas* (2002). Accuracy and Stability of Numerical Algorithms. SIAM, 110–123.
13. Lutskyi, H. M. et al. (2016*). Metody ta zasoby pidvyshchennia efektyvnosti rishennia zdach na osnovi perestroiuvanykh obchysliuvalnykh zasobiv na PLIS* – Zakl. zvit po NDR No. DR 0216U007635. Kyiv, 244.
14. Baugh, C. R., Wooley, B. A. (1973). A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Transactions on Computers, C–22 (12),* 1045–1047. doi: https://doi.org/10.1109/t-c.1973.223648
15. Parhami, B. (2000). *Computer Arithmetic. Algorithms and Hardware Designs*. New York: Oxford University Press, 491.

Georgi Luckij, *Doctor of Technical Science, Professor, Department of Informatics, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-3155-8301*

✉***Oleksandr Dolholenko,*** *PhD, Associate Professor, Senior Researcher, Department of Informatics, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: aleks.dolgolenko@gmail.com, ORCID: https://orcid.org/0000-0003-3375-7117*

✉*Corresponding author*