ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 3, No. 1, 2018

MICROPROCESSOR WITH TAGGED REGISTERS REALIZING PARALLELISM

Volodymyr Dobrovolskyi

Independent CPU Architect, Kyiv, Ukraine
Author's e-mail: *vol.dobrovol@gmail.com*

Submitted on 07.12.2018

***Abstract*: A RISC microprocessor architecture that realizes a specific method of parallelism including the instruction level parallelism has been considered. The processor has been provided for 4-bit data type tag in each register of the register file. There are 14 data type tag values. The zero data type tag indicates that the register is free, otherwise it is busy. The destination register inherits the data type tag from the first source register. After an operation the data type tags in the source registers may be either zeroed, or may remain unchanged for further usage. All machine operations are classified into computational operations (about 40), and auxiliary operations (about 35-45). The computational operations include integer, unsigned, floating point, logical, string, and conversion operations. The processor has specific instruction formats in which there are 6-bit fields both for the operation code and the computational code. A single primary computational instruction having zero in the operation code field, and a meaningful code in the computational code field is enough to express all computational operations. A compiler generates groups of instructions to perform in parallel, the reordering of instructions may take place. There are several clones of the primary computational instruction with operation codes differing from zero. A clone computational instruction with a certain operation code is placed as a header instruction for the instruction group pointing out a certain number of instructions in the group to issue in parallel. The primary instructions may be placed inside the groups. The concept of flux is introduced as a composite of stream of instructions and a flow of processed data maintained by the flux hardware. Fluxes improve the usage of multiple functional units, and may be used for further parallelization.**

***Index Terms*: microprocessor, RISC, instruction set architecture, instruction level parallelism, flux, register tags.**

I. INTRODUCTION

Parallelization is the mainstream in the contemporary microprocessor architecture, and is implementing both within a single processor (uniprocessor or core), and in multicore processors in which several or several dozens of uniprocessors are arranged on one die and are connected with very fast interconnects.

Other trends in parallelization are systems of massively-parallel computing. First of all, these are Graphical Processing Units (GPUs), containing hundreds and thousands of relatively simple processors (cores) on one die, also they are called manycore processors. They perform not only graphical, but a wide spectre of other computations. Also there are spatially distributed systems of computers, which perform deeply parallelized problems (cloud computing). Supercomputers are systems of thousands, or tens of thousands both usual CPUs, and GPUs with very fast interconnects. Therefore, the supercomputer is a massively parallel system. Servers in data centers consist of several tens of poweful microprocessors.

There are many monographs and manuals on these vast topics, e. g. [1, 2, 3, 4, 5, 6, 7].

The principal ways of hardware parallelization in the traditional microprocessors are the following:

1. Multiple functional deployment units (multiunit architecture) on one die. Availability of multiple functional units is a mandatory condition for parallelization, otherwise the parallelization is not possible, though the data exchange between registers of CPU and the main computer memory may be performed in parallel with a computational operation.

2. Superscalar architecture, i. e hardware-based parallelization of instruction stream onto multiple functional units. Superscalar parallelization emerged in the mid of sixties or earlier. It is effective if the processor has at least two functional units. The superscalar approach does not demand any special efforts from the programmer to take into account, whereas the programming for multicore processors is more complicated and depends on the number of processor cores.

3. Speculative execution is used for instructions performing branch and conditional transfer of control, thus, enabling to economize one or more machine cycles.

4. Out-of-order execution is a transposition of instructions performed by hardware in a buffer which accumulates a score or more of instructions. The experience shows that such reordering of instructions substantially increases the efficacy of computations.

5. Simultaneous multithreading (SMT) is the main contemporary concept of parallelization for uniprocessors, invented in the nineties of the past century [8]. The SMT actually absorbs superscalarness, however the SMT processor may be called the superscalar one. The threads are named logical processors. They are formed by the hardware means, and this hardware is rather complex. It is better for transistors for the SMT hardware to be used for an additional functional unit. Really, the

most important characteristic of the SMT uniprocessor is the number of functional units. A significant feature of the SMT is the fact that each thread has its hardware context: register file, program counter, stack and register with the stack address, and the program status word or record (PSR) with various bit flags. The overwhelming majority of the contemporary SMT microprocessors has two threads, but each core of the newest IBM Power9 microprocessor has 8 threads.

Pipelining in functional units may be considered a specific parallelization in which instructions are issued with a shift on one machine cycle. It is a well mastered technology.

The present author proposed also a method of parallelization on the instruction level parallelism (ILP) for RISC processors via a special non-pipelined parallelizing instruction [9].

A separate case of uniprocessor architecture is a very long instruction word (VLIW) architecture. The VLIW architecture has been successful for the specialized processors for image and graphical data processing etc., but failed as the general purpose processor.

The proposed method of parallelizing, stated in the paper, is based, firstly, on the notion of the instruction group to issue in parallel, and secondly, on the idea of the tagged registers of the register file [10, 11]. The tagging permits to form a specific instruction format, so that the first instruction of an instruction group (a header instruction) points out the number of instructions in the group simultaneously. The instruction groups are created by the smart compiler. The proposed approach secures much more complete extraction of the ILP for ordinary programs, though, due to the usage of smart compilers. Also, the hardware is substantially simpler. Distantly, the proposed approach resembles a peculiar VLIW processor with variable number, one to four, of operative fields [12].

The subsequent material is stated on the exemplary 32-register file with 64-bit registers. The data of the floating point type may embrace 1, or 2 64-bit registers. The data of the bit, byte, and double byte types may embrace 1, 2, or 4 64-bit registers. Groups of instructions are assumed to contain maximally up to 10–15 instructions to issue in parallel. The instructions are 32-bit.

II. DATA TYPE REGISTER TAGS

The notion of the data tag was used as a data identification prefix in some computer architectures of the past. An American inventor J. K. Iliffe pioneered the usage of tags to mark data in the main computer memory, specifically to mark machine words [13]. The stack computers of the Burroughs Corporation in the early sixties had 3-bit tag to mark the data type of the 48-bit machine word [14]. The Soviet Elbrus-1 and Elbrus-2 stack computers (multiprocessor computing complexes) in the seventies were provided with the 8-bit tag that pointed the data type and the data access rules for the 64-bit machine word [7]. The tags were considered as some extension of machine words in the main computer memory. These computers had register stacks, not the register files. In both mentioned computers after the data (with tags) have been loaded into a stack, an executable instruction explores the tag to decide what further action to perform if data types were not adequate to the instruction.

History of computer science has shown that tagging data in the main memory was a fallacy, this idea revealed itself nonproductive. Some improvement in reliability due to tagging demanded large overhead for additional memory and did not justify tagging in the main computer memory at all. In the sixties (Burroughs) and in seventies (Elbrus) the main computer memory was very expensive, and to spend 6.25 and 12.5 percent of it was impractical.

The proposed microprocessor architecture has 4-bit data type tag in each register of the register file. The code in a register tag defines the data type. There are 14 values of the data type tag. The zero data type tag indicates that the register is free, otherwise it is busy. In the most of machine operations the destination register inherits the data type tag from the first source register. After an operation the data type tags in the source registers may be either zeroed, or may remain unchanged for the usage in other operations. On the micro-architectural level the tags may be placed separately from the register file. The idea of the data type tag as an extension of register considerably reduces the number of machine operations, simultaneously increasing their multiplicity [10, 11].

Table 1

Tag table with data type tags

Data type code	Short notation	Data type description
'0000'b = 0		Register is free for writing
'0001'b = 1	i8, or i	64-bit integer
'0010'b = 2	u8, or u	64-bit unsigned
'0011'b = 3	a8, or a	64-bit unsigned for addresses in the main computer memory
'0100'b = 4	f8, or f	64-bit floating point
'0101'b = 5	f16	128-bit floating point in 2 registers
'0110'b = 6	t1, or t	Bit string in 1 register
'0111'b = 7	b1, or b	Byte string in 1 register
'1000'b = 8	d1, or d	Double-byte string in 1 register
'1001'b = 9	t2, or t	Bit string in 2 registers
'1010'b = 10	b2, or b	Byte string in 2 registers
'1011'b = 11	d2, or d	Double-byte string in 2 registers
'1100'b = 12	t4, or t	Bit string in 4 registers
'1101'b = 13	b4, or b	Byte string in 4 registers
'1110'b = 14	d4, or d	Double-byte string in 4 registers
'1111'b = 15		Writing operation in register failed

The hardware tag table sets the correspondence between the tag and the register contents (Table 1). On the micro-architectural level the tag table may also contain the information about the kind of the functional unit to use. The tagged registers are well suited for the RISC processors, and are hardly suitable for the CISC processors. Actually, the data type tag may be considered as a continuation of the operation code at register to inform the executable instruction which the data type register contains.

Table 2

List of computational operations

Computational Operation Code	Computational Operation Description	Computational Operation Code	Computational Operation Description
'000000'b = 0	No operation is supported	'011000'b = 24	Logical multiplication AND
'000001'b = 1	Addition	'011001'b = 25	Logical exclusive OR (XOR)
'000010'b = 2	Subtraction	'011010'b = 26	Logical inversion LINV
'000011'b = 3	Multiplication	'011011'b = 27	Logical shift right LSR
'000100'b = 4	Division	'010111'b = 23	Logical addition OR
'000101'b = 5	Integer division with remainder	'011000'b = 24	Logical multiplication AND
'000110'b = 6	Combined “multiply” and “add”	'011001'b = 25	Logical exclusive OR (XOR)
'000111'b = 7	Addition of the constant to the register	'011010'b = 26	Logical inversion LINV
'001000'b = 8	Subtraction of the constant from the register	'011011'b = 27	Logical shift right LSR
'001001'b = 9	Subtraction of the register from the constant	'011100'b = 28	Logical shift left LSL
'001010'b = 10	Multiplication of the constant by the register	'011101'b = 29	Logical rotate right LRR
'001011'b = 11	Constant is assigned to the register with the same sign	'011110'b = 30	Logical rotate left LRL
'001100'b = 12	Constant is assigned to the register with the opposite sign	'011111'b = 31	Substitution of the part of string by sub-string
'001101'b = 13	Two sequential arithmetic operations r4d = (r3s + r2s) * r1s	'100000'b = 32	Search sub-string inside string in the forward direction
'001110'b = 14	Two sequential arithmetic operations r4d = (r3s – r2s) * r1s	'100001'b = 33	Search sub-string inside string in the backward direction
'001111'b = 15	Two sequential arithmetic operations r4d = r3s * r2s + r1s	'100010'b = 34	Copy a part of string
'010000'b = 16	Two sequential arithmetic operations r4d = r3s * r2s – r1s	'100011'b = 35	Delete a part of string
'010001'b = 17	Two sequential arithmetic operations r4d = r3s + r2s + r1s	'100100'b = 36	Relocate a part of the string in the same string
'010010'b = 18	Two sequential arithmetic operations r4d = r3s + r2s – r1s	'100101'b = 37	Concatenation of two strings
'010011'b = 19	Two sequential arithmetic operations r4d = r3s – r2s + r1s	'100110'b = 38	Test for coincidence of strings
'010100'b = 20	Two sequential arithmetic operations r4d = r3s – r2s – r1s	'101111'b = 39	Calculation of address index for array element
'010101'b = 21	Two sequential arithmetic operations r4d = r3s * r2s * r1s	40 … 63	Reserve
'010110'b = 22	Conversion		

Normally, all data type tags in registers should be non-zeroes. The hardware checks the correspondence between the data type tags in source registers with their adequacy to the prescribed data type fixed in the destination register. If these conditions are not complied, the “tag” error occurs, the data type tag gets value '1111'b, and the current value of the program counter is copied in the destination register for further analysis.

III. SET OF THE COMPUTATIONAL OPERATIONS

All machine operations are classified into two general classes: (1) computational machine operations; and (2) auxiliary machine operations (they are described in the next section). The computational operations include integer, unsigned, floating point, logical, string, and conversion operations, i. e. those that perform processing of the loaded data making useful work. There is a set of about 40 computational operations, and a set of about 35–45 auxiliary operations. This classification is conditional, to some extent. The list of computational operations is shown in Table 2

IV. INSTRUCTION FORMATS

Taking into account the classification of the machine operations into the computational and auxiliary one, the instructions are classified into (1) computational instructions including a single primary computational instruction with a collection of its clone computational instructions; and (2) auxiliary instructions.

The computational instructions should have the obligatory destination register. These instructions, both primary and clone ones, have two 6-bit operand fields, the first for the operation code (OC), and the second for the computational code (CC). The operation code for the primary computational instruction is zero, as all computational machine operations are ensured by the computational code. One primary instruction induces a collection of clone instructions; each is intended to embrace the defined number of instructions in the instruction group. Details about usage of clone computational instructions for parallelization are in the next section.

The auxiliary instructions contain operation code in the corresponding 6-bit field in the range 1 to 30–40. It is enough to express all auxiliary operations. They include instructions for various settings, the transfer of control, load/store operations, copy register to register, move register into other register, swap of two registers, push register into memory, and pull register from memory, and comparison of two magnitudes. The auxiliary instructions perform some necessary ancillary work to secure the processing of data.

Table 3 gives a representation of the operation and computational codes (OC and CC) in different instruction formats (the quantity n is the necessary number of clones).

Table 3

Table of computational operations

Operation	Operation Code	Computational Code
Primary computational	0	0 … 63
Clone computational	64 - n … 63	
Auxiliary	1 … 63 - n	0

The structures of the load/store and the computational instructions are shown in Fig. 1 and Fig. 2 respectively (digits in the second rows are the lengths of the operand fields in bits). The formats for the other auxiliary instructions are not considered.

OC	Q	DTT	ST	IR	BR	DR
6	2	4	5	5	5	5

Fig. 1. Format of the load/store instructions

OC	CC	SR1	SR2	SR3	DR
6	6	5	5	5	5

Fig. 2. Format of the computational instruction

Denotations for operands for the load and computational instructions are the following: (1) operand OC (Operation Code) is the 6-bit operation code; (2) Q is the 2-bit Qualifier that may detail the machine operation having the same operation code, e. g. to point out whether or not to zero register data tags after instruction is performed; (3) operand DTT (Data Type Tag) is the 4-bit data type tag that is fixed by the programmer, and then assigned to the hardware register tag, and in the case of the store instruction the DTT is compared with the hardware register tag for strong verification of the store operation; (4) operand ST contains the increment/decrement step (register, or an integer constant) in machine words that is added to the index register IR after the load/store operation; (5) operand IR is the index register which is fixed by the programmer before a series of the load/store operations, then it is incremented/decremented with the ST operand upon completion of each load/store operation; (6) operand BR is the base register containing the base address (data type is unsigned for the addresses in the main computer memory) for multiple usage of the load/store operations, and the BR is relatively permanent; (7) operand DR is the destination register being loaded, the data type register tag is taken from the DTT operand.

And these are specifically designations for operands for the computational instructions: (8) operand CC is the 6-bit computational code; (9) operands SR1, SR2, and SR3 contain the source data to process. Also the source operands may contain an immediate constant, or constants, in such a case some source operand fields are merged.

The load instruction loads a single or concatenated registers, and also the proper register data type tag is filled in how the programmer specifies it. The store instruction is analogous to the load instruction, the operand DTT is used to check the correctness of the data storing. The computational instructions process data in the registers. An example of designation of the computational instruction is as following

*madd sr*2,*sr*3,*dr*

The instruction realizes the formula dr = dr + sr2 * sr3. The denotation *madd* means the multiply-and-add machine operation; registers *sr*2 and *sr*3 contain the source operands; register *dr* contains the destination operand. This instruction does not use the register field *sr*1.

V. PARALLELIZATION BY USAGE OF CLONE INSTRUCTION

The set of computational instructions makes up a collection consisting of a single primary computational instruction and several clone computational instructions. Each clone instruction is intended to embrace the defined number of instructions in the instruction group. The clone instructions coincide with a primary one inherently, but have other operation codes. The operation codes for the clone instructions are placed in the upper part of the 6-bit operand field for the operation code. For instance, a collection of 9 clone instructions has operation codes '111111'b to '110111'b (63 to 55). The clone computational instruction is used in the capacity of the header instruction in the instruction group to issue it in parallel. Groups consisting of a lone instruction are possible (one-instruction groups). Any other primary computational instruction does not take part in control of the parallel group, it is a usual member of an instruction group together with auxiliary instructions.

In assembler language the denotation for the primary computational instruction, e. g. for multiplication, might have the appearance *mul 0*, *sr*2, *sr*3, *dr* where zero means the operation code. This zero may be omitted for shortening: *mul sr*2, *sr*3, *dr*. The denotation for the clone computational instruction should contain nonzero operation code with digits 2 to, e. g. 7, pointing out the number of instructions in the group including the header instruction. Thus, a clone computational instruction for the group of 7 instructions has the demotation *mul 7*, *sr*2, *sr*3, *dr*.

Instructions	Explanation
Previous group	
mul 6, r1, r2, r3	multiplication (header for 6 instructions)
load f8, r4, r5, r6, r7	load floating point number
load t1, r8, r9,r10,r11	load bit string
sub r12, r13, r14	subtraction of two numbers
madd r15, r16, r17	computation on formula r17 = r17 + r15 * r16
cmp r18,r19	comparison of two magnitudes
jump -250	jump on 250 bytes in backward direction
Next group	

Fig. 3. Example of instruction group to issue in parallel

The instruction groups are generated by the smart compiler, and for better efficacy the reordering of instructions should take place. The smart compiler is able to investigate large fragments of the source code, even the whole procedure or function, and extract all possible static parallelism. The smart text source editor shows instruction groups due to the feedback between the compiler and the editor enclosing the formed instruction groups in parentheses. An example of group of six instructions to issue in parallel is shown in Fig. 3. The proposed parallelization is maintained by the hardware. The actual efficacy of parallelization depends upon the availability of multiple functional units. Thus, in the case of shortage of functional units the dispatch of a parallel group may be done on two or more machine cycles.

VI. CONCEPT OF FLUX

The flux is defined as a composite that includes the software and hardware components within the scope of uniprocessor. From the program point of view, the flux is a stream of instructions and the corresponding flow of processed data. These streams and flows are maintained by the flux hardware that includes register file, program counter, stack and register with the stack address, the program status word or record (PSR) with various states and interrupts bit flags, and other control information fields. Also, the flux contains the special flux instruction buffer (described below). The main computer memory and pipelined functional units are reckoned as a common resource for all fluxes, and are used on request. Fluxes may have either individual L1 instruction caches, or a common L1 instruction cache. The same is true for the data caches. A flux looks like a partial processor. Also, a flux may be looked upon as a channel, window, or medium in which a program executes, using its register file and PSR, and borrowing the required functional units from their totality or pool.

For effective work the uniprocessor should be multi-flux one, i. e. should have at least two fluxes. The maximal number of fluxes in uniprocessor is 2 to 4 – the effective width of data paths between a flux and functional units is a limiting factor. The hardware discerns fluxes through their n-bit flux distinctive labels, e. g. for the four-flux processor it is the 2-bit labels. The functional units remember from which flux the data to process are received, and to which flux the results should be returned. A flux maintains either a single program process or a number of processes in the time-sharing mode. Also, the parts of a properly designed program may execute in different fluxes in parallel, interacting between each other. This is projected by the programmer, and is maintained by the operation system. The certain instruction set architecture must contain an instruction which informs the programmer the number of fluxes the concrete microprocessor contains.

Each flux is provided with the flux instruction buffer as a small and very fast intermediate storage where the instruction groups are accumulated to process them further in parallel. The buffer has at least two buffer sections (two-section buffer), each section accumulates

an instruction group. The accepted number of instructions in the section predetermines the maximal number of the computational clone instructions plus one supported by the concrete microprocessor. It is better to have several buffers to avoid latencies and increase performance.

The flux control unit fills in the first section, and simultaneously passes the instruction group from another section for the further processing and, eventually onto the functional units. The empty section is free to be filled with the next instruction group. Instructions came into the flux buffer in a sequential stream as some input groups, but issue of instructions is performed in the form of compiled groups. The hardware controls the completeness of formed instruction groups. In the contemporal RISC microprocessors up to 8 instructions may be fetched simultaneously during one machine cycle, and they are loaded in a separate primary instruction buffer which should be located before the flux buffer, and both buffers make up the unified buffer.

Each instruction occupies a “slot” in the section. There are 1-bit hardware labels that mark slots: when a slot is occupied it is marked with '1'b, otherwise with '0'b. When all labels in a section have the '0'b value then the section is considered free, and is ready to be filled in with the next instruction group. The information about the number of instructions in the group is important, and is being passed further.

The concept of the flux only remotely resembles the concept of SMT, and is characterized by the following features: 1. Structurally, the binary code in the proposed architecture consists of the instruction groups, and transfer of control is made to the header instruction of the group, whereas instructions in the SMT are not connected with each other, and are not grouped. 2. In the scope of one flux the groups of instructions, formed by the compiler, are dispatched for execution from the special flux buffer, whereas in the scope of one SMT thread a single instruction is only executed. 3. Program counter changes its value on the length of the instruction group (in bytes, or in the number of instructions), not on the length of the separate instruction. 4. The flux instruction buffer has quite different and simple hardware compared with the hardware that forms the SMT threads and provides for superscalarness.

VII. THE WORK OF MICROPROCESSOR REALIZING PARALLELIZATION

There are four modes of interactions between the hardware units and data paths for different kinds of instructions, maintainig the formation and passing further parallel groups of instructions. These modes are for:

(1) load/store instructions; (2) computational instructions; (3) comparison and branch instruction; (4) jump instruction. The modes for load/store and computational instructions are shown in Fig. 4 and Fig. 5 as examples.

The handling of instructions is fulfilled on stages which are maintained by a certain hardware. For the proposed microprocessor architecture this hardware includes: program counter with its controller; fetch unit with combined instruction buffer; decoder unit; dispatch unit; tag analyzer; integer-and-logical, floating point, and other functional units. The sequentiality of actions the microprocessor with the described architecture fulfills on different stages, which are as follows:

1. The fetch unit reads the address of next instruction group from the program counter. The unit partly decodes operation code of the header instruction of the instruction group to ascertain the number of instructions in the group. Then, the unit fetches the rest of instructions and fills in a section in the flux instruction buffer with the fetched group. Until a group is fetched the program counter is not permitted to change its value.

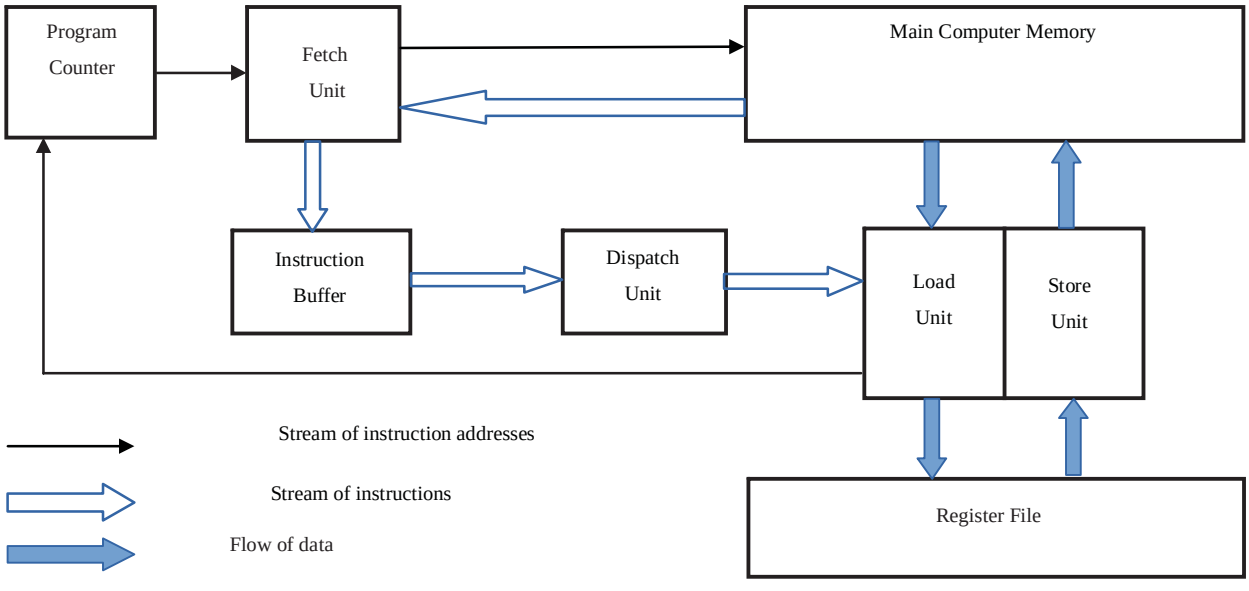

Fig. 4. Units and data paths for load/store instructions

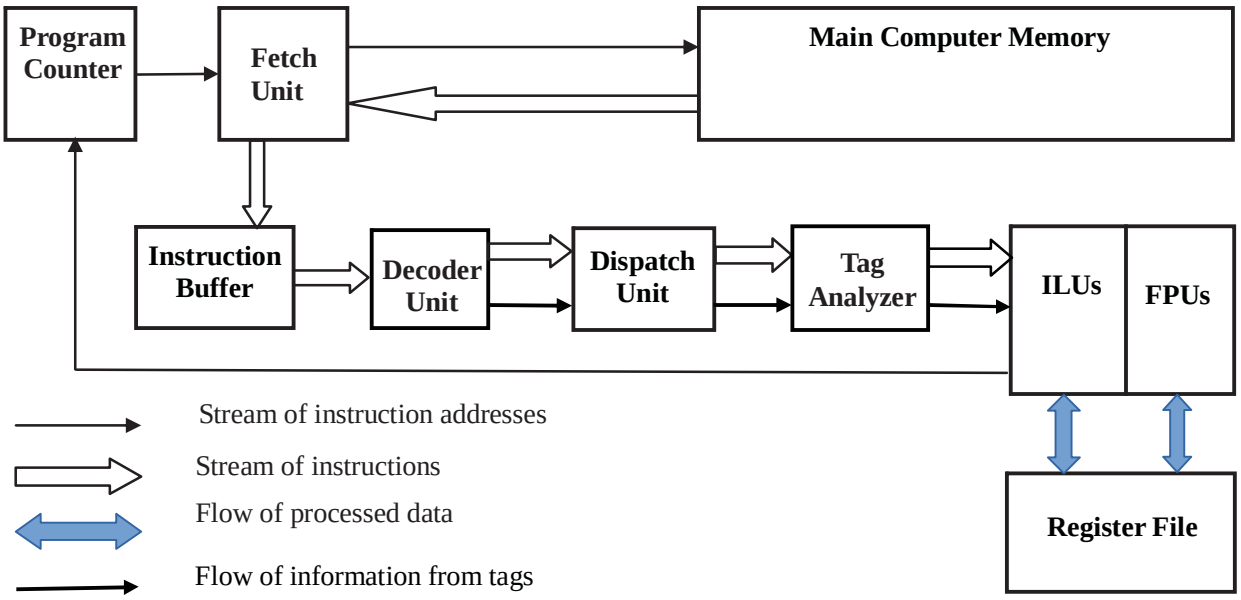

Fig. 5. Units and datapaths for the computational instructions (ILU is Integer-and-Logical Unit, FPU is Floating Point Unit)

2. The decoder unit makes the rest of the decoding work. It fully decodes all auxiliary instructions. The decoding of the computational instructions is fulfilled partly, as on this stage it is impossible to ascertain the required functional units.

3. The dispatch unit receives fully decoded instructions in the instruction groups, and issues the formed groups onto the required pipelined functional units, group by group on each machine cycle. In the case of the computational instruction the dispatch unit passes it to the tag analyser (unit) which explores the register tag of the first source register ascertaining the data type, and, therefore, the kind of the required functional unit. If in a concrete microprocessor the number of the functional units, or the number of specific kinds of the units is insufficient, the rest of instructions of the instruction group may be issued on the next machine cycles.

4. The multiple pipelined functional units work on traditional scheme. The loading/storing unit loads the data from the main computer memory to the registers with the help of the load instruction. The functional units execute the necessary operation, taking into account the data type tags in the source and destination registers, the result of the operation being written in the destination register together with the tag. At last, the loading/storing unit stores the data from the destination register to the main computer memory with the help of the store instruction. The necessary ancillary instructions are executed as well.

VIII. CONCLUSIONS

There are three main advantages of the proposed microprocessor architecture.

Firstly, a method of parallelization has been considered. It is able to extract all possible parallelism covering the large source instruction window, though depending on the perfection of the compiler. Even in static mode the compiler is able to extract more parallelism than the two-threaded SMT hardware. Thus, the proposed microprocessor will show higher productivity than the processor with the SMT. The proposed method is based considerably on the idea of data type tags at registers of the register file, and demands small amount of hardware.

Secondly, a great advantage of the proposed architecture is a possibility to diversify operations due to the data type tags, and simultaneously at lesser number of instructions. Instructions are generalized, e. g. one instruction for multiplicaion is applicable to different data types defined by the data type tags. The architecture simplifies the processor design due to small number of instructions. The architecture improves the reliability of computations on the stages of development of the source code, compilation, and execution due to the data type tags.

Thirdly, the concept of the flux is introduced permitting to use functional units more efficiently, and to organize execution of parts of a program in parallel in different fluxes interacting between each other, further widening of parallelization may be realized by using multiple cores. In evaluation of the microprocessor productivity it does not matter what architecture is implemented, dynamic, or static, what matters is the number of issued instructions per machine cycle. The concept of flux supercedes the SMT architecture and may substitute it at less hardware.

REFERENCES

[1] David A. Patterson and John L. Hennessy. Computer Organization and Design. The Hardware / Software Interface, Fourth edition, Morgan Kaufmann Publishers, 2009, 940 p.

[2] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2014, 92 p.

[3] David A. Patterson, "Reduced Instruction Set Computers", in *Communications of the ACM*, volume 28, Number 1, January, 1985, pp. 8-21.

[4] Joseph D. Dumas II. Computer Architecture. Fundamentals and Principles of Computer Design (University of Tennessee at Chattanooga, Chattanooga, TN, USA), Taylor & Francis Group, LLC, 2017, 450 p.

[5] Dezső Sima, Terry J. Fountain, Péter Kacsuk. Advanced Computer Architectures: A Design Space Approach, Addison-Wesley, 1997, 766 p.

[6] Melnyk A. O. Architecture of Computer. Manual (Lviv Polytechnic National University), Lutsk regional printing, Ukraine, 2008, 470 p. (Мельник А. О. Архітектура комп'ютера. Підручник).

[7] Korolev, L. N. Architecture of electronic computers, Nauchny mir, Moscow, Russia, 2005, 272 p. (Королев Л. Н. Архитектура электронных вычислительных машин, М, Научный мир).

[8] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors", in *IEEE Micro*, September/October, 1997, pp. 12–19.

[9] V. K. Dobrovolskyi, "Microprocessor with Explisit Parallelism", in *Proceedings of VIth International Scientific Conference SIMULATION-2018 (МОДЕЛЮВАННЯ-2018)*, September 12–14, 2018, Kyiv, Ukraine, pp. 135–138, ISBN 978-966-02-8587-3.

[10] V. K. Dobrovolskyi, "Microprocessor with Tagged Registers", in *Proceedings of the Vth International Scientific Conference Simulation-2016*, May 25–27, 2016, Kiev, Ukraine, pp. 57–60, ISBN 978-966-02-7928-5 (file), ISBN 978-966-02-7927-8 (printed edition).

[11] Dobrovolskyi, Volodymyr. Microprocessor with Tagged Registers. Version 1.1, Kyiv, University Publishing House "Pulsary", 2017, 60 p., ISBN 978-617-615-073-2.

[12] Marc Tremblay, Jeffrey Chan, Shailender W. Conigliaro, Shing Sheung Tse, "The MAJC Architecture: A Synthesis of Parallelism and Scalability", in *IEEE MACRO*, November-December, 2000, pp. 12–25.

[13] Iliffe, J. K. Basic Machine Principles, London, MacDonald & Co., 1968, vii+86 p.

[14] Instruction operations for the B8501 Central Processing Module. Reference Manual, Burroughs The Corporation, 1966, 101

Dobrovolskyi Volodymyr (Dobrovolsky in some publications) graduated from Lviv Polytechnic Institute in Technology of Machine Building, received a PhD in Mathematical Economics from Institute of Economics of National Academy of Sciences of Ukraine. The research interests are the CPU and microprocessor architecture, and the mathematical modeling of economy.