MULTICLET®
www.multiclet.com

Multicellular processors

(conception, architecture)

The overall volume of information technologies market amounts to trillions of US dollars. It expressly or by implication depends upon the processor architecture.

Since the first computer, this market has been absolutely dominated by the Von Neumann architecture. But according to experts this architecture’s age is coming to the end. The market is dynamically developing and we need a qualitatively new (post-Von Neumann) architecture that will be able to determine further development of microprocessors.

Introducing this architecture really favours leading positions.

The processor architectures research held by the “UralArchLab” Ltd shows that this is the only post-Von Neumann direction, it produces context-sensitive program processors. Only such processors are able to solve both existing problems and long-term tasks of the computer industry.

With the assistance from Foundation “Innovation technologies” we have developed the first context-sensitive program multicellular processor MCp0402100100 and this makes our chosen direction correct and perspective.

The proposed architecture is multi-faceted and qualitatively compiled, which permits us to position this architecture as a principally new, high-efficiency in post-Neumann direction to allow for further development of microprocessor standards.

CONTENTS

INTRODUCTION

The Von Neumann era in computer industry is coming to the end. This processor model has been dominating for more than 60 years but now as it is stated by the International Technology Roadmap for Semiconductors (ITRS) that this pivital development is fading away [htpp://itrs.net/Links/2005ITRS/Sys Drivers2005.pdf].

A retrospective analysis demonstrates that every new effort to improveVon Neumann architecture had demanded more and more efforts and generated less benefit.

Taking the role of architecture in consideration, we can conclude that we are facing another bifurcation point in the development of computer industry. The solution is development of qualitatively new, post-Von Neumann direction for processor architecture.

At the moment leading processor manufactures offer multi nuclear systems as the main architectural direction. But this decision can not be regarded as the beginning of a new post-Von Neumann era. This direction is extensive and temporary solution for the same old Von Neumann problem. Multi nuclear approach is not a fundamentally new step in the computer industry and it no longer solves its existing problems.

From the first calculators, the main direction in the von-Neumann architecture development was increasing the parallelism level when a command flow was executed. It was linked with attempts to weaken and avoid the key principle of the Von Neumann architecture, that is, the ordered and sequential command placement in the program and their execution according to this placement. For example, the conveyor is a partial combination of several commands in time. The superscalar processor organization and the VLIM-processor concept is a combination of several commands not only in time but also in space.

The demand for ordered placement and command execution is a necessary condition of realization of meditate form of information connections between commands that is used in the Von Neumann architecture. That is, the result of execution of any command is *alienated*, i.e. recorded in the openly accessible machine memory (registers, memory units) and only after this it becomes available (visible) to the programmer and may be used as an operand for the following commands.

The most known efforts to depart from mediate form and, by doing so, provide "natural" realization of parallelism are are data flow and reduction machines. They use not meditate but obvious information connections between commands.

So, in the data flow machine the command address is set directly in the command word of the command-source of this result. After the command-source is executed, the result is directly recorded in the operand field of the command-consumer and becomes its part.

In the reduction machine the address of the command-source is set in the command word of the command-consumer that also ensures direct transmitting and using the result.

Informational connections determine the sequence of execution in both of these machines. As a result, this execution falls out oforder – "when ready" or "when demanded". This circumstance conflicts with the computing model which is used in the most commonly spread imperative high-level languages.

As it is known, the computing model is the execution of an ordered sequence of operators in the imperative high-level languages. Every operator represents an indivisible and integral language structure which describes the data translation process. Each operator is the sequence, the operations are executed is set by ranging and parenthesis ordering, i.e. by indicating informational connections between operations. The intermediate computing results are not alienated inside the operator and not visible to the programmer. Only the result of the operator execution is alienated and visible. Consequently, the language operator serves as a command for the abstract machine directly realizing some high-level language.

The initial operation set of any algorithmic language is primordially fixed and finite. The set of operators that can be theoretically constructed using these operations is potentially infinite and correspondingly the machine where the architecture directly realizes the high-level language has no fixed command system.

This architecture is fundamental for a principally new direction in constructing the processors, i.e. the multicellular processors that have qualitative and quantitative characteristics of a new post-Von Neumann generation and that were approved by the first multicellular processor MCp0411100101.

1 CONCEPTION

1.1 Triad algorithm description

Any formula, for example, mentioned in the figure 1.1 (a), can be represented as a multilevel parallel form, as it is shown in the figure 1.1 (b).

```
g=e*(a+b)+(a-c)/f
```

$$\begin{tikzcd}
a \arrow[d] \arrow[drr] & b \arrow[dl] & c \arrow[d] & \\
+ \arrow[d] & e \arrow[dl] & - \arrow[d] & f \arrow[dl] \\
* \arrow[d] & & / \arrow[dll] & \\
+ \arrow[d] & & & \\
g & & &
\end{tikzcd}$$

(a) (b)

Figure 1.1 – Multilevel parallel representation of formula: formula (a); multilevel parallel form (b).

Let’s sequentially, for example, from left to right and from top to bottom, number every node of the multilevel parallel form mentioned in the figure 1.1(b). The identifiers of variables will be regarded as the read operations of these variables, if their values are used by other operations. If the identifier shows that the operation is executed, then it will be regarded as the write operation of this operation. Let’s write the operations being executed in the order of numbers. As the operands of the operation being executed we indicate the numbers of the operations that are executed and this execution results are its arguments.

Number	oc	op1	op2
0	RD	a	
1	RD	b	
2	RD	c	
3	+	0	1
4	RD	e	
5	-	0	2
6	RD	f	
7	*	3	4
8	/	5	6
9	+	7	8
10	WR	9	g

Figure 1.2 – Triad multilevel parallel form description

Shown in the figure 1.2, this description is analogous to the intermediate representation of the triad program; this representation is used in the compilation of the programs written in the high-level languages. This representation differs from its classical form by using read and write operations as well as references to these operations instead of direct using the identifiers (references to the identifiers' table). Thus, it can be regarded as a machine-adapted form of the primary program written in the high-level language.

In this form the program is recorded as a numbered sequence of triads. This sequence is divided into sections. Every section corresponds to one program operator and contains the subset of triads that realizes this operator. The sequence the sections are recorded corresponds to the sequence the operators are recorded in the program. Every triad describes how some operation is executed on the triads set by identifiers or references. The reference is the triad's number and its execution result is used as an operand, i.e. the reference clearly sets informational connection between operations. At this, the triad results being transmitted by the reference are not alienated.

Since information exchange is mediated between the operators and fulfilled through alienating the results, then, the subset of triads realizing the operator is closed. It has no references to the triads of other operators and the triads of other operators do not refer to the triads of this subset. It should be noted that no triads make sense, when outside their subset, and they are no longer a command system, i.e. an informational message that determines the executive unit actions and has integrity and indivisibility. The triad has its indivisibility but there is no integrity.

Any triad is meaningful and can be executed only in some certain context. That is, only when the triads are executed, it uses their results and those triads use its results, i.e. only inside its subset. So, the program generated by the triads' sequence is context-sensitive.

For comparison, the way this or that command is executed in the data flow processor or the Von Neumann one does not depend upon any context. The program of these processors is context-free. Every command of these processors has its indivisibility and integrity. Those commands that realize the operator have their integrity but there is no indivisibility.

In general case the operator's triads can be placed on the section at random. Unlike the Von Neumann architecture, their placement determines no sequence of execution. The sequence they are executed is determined by informational connections

Thus, if in the Von Neumann processor the program unambiguously determines "what" and "how" should be done, here the triad program unambiguously determines only "what" should be done. "How" should be done is determined by the processor. And this circumstance favours qualitatively new as well as improved quantitative characteristics of the processor.

1.2 Principles of processor construction

It is evident that the architecture of the processor capable of executing a program text in the triad language should principally differ from the known Von Neumann and non-Von Neumann (data flow and reduction) architectures.

First of all, the method the informational connections between the operations are described is different and consequently there will be a different method of their realization. If we take the Von Neumann model, there informational connections between commands (operations) are not clearly described and indirectly realized through the memory (general-purpose registers, MU (memory unit)), then in the triad language they are clearly set by indicating informational connections between commands. At this, unlike the non-Von Neumann models, the representation has a selective character rather than an address one. The result of a command is not sent to a certain consumer (data flow processors) and there is no concrete command to have a result (reduction processors) but consumers themselves should select necessary results out of a common flow of results that is formed imperatively by selecting and executing every command of the linear section rather than by references. Consequently, the processor architecture should have a mechanism to identify received results as well as intellectual commutation environment ensuring not only result broadcasting but also result selecting for concrete operations.

Secondly, here we have a different process in itself. If data flow and reduction processors have unordered selection and command execution, when data are ready or result is demanded, then the triad language presupposes ordered selection of the command of the linear section and their execution not only when data are ready but also when consumers of the results are ready. That is, the selected command can not be executed until all the operands are received (data readiness) and until all the commands using its result are selected (consumer readiness). Such approach to execute sequentially selected commands is connected with their disorder.

There is no fixed time span from command selection to command execution, this presupposes mechanisms of command buffering; these mechanisms ensure selected command storage, completing it with operands selected from a result flow and its issue to be executed after all the consumers of its result are selected.

2 ARCHITECTURE

2.1 Multicellular processor scheme

Let’s consider the parallel system shown in the figure 2.1. It consists of N processor units PU_0, PU_1, …, PU_n-1 connected by unidirectional “each with each” commutator (SB) that has N informational inputs and 2N informational outputs as well as 2N address inputs.

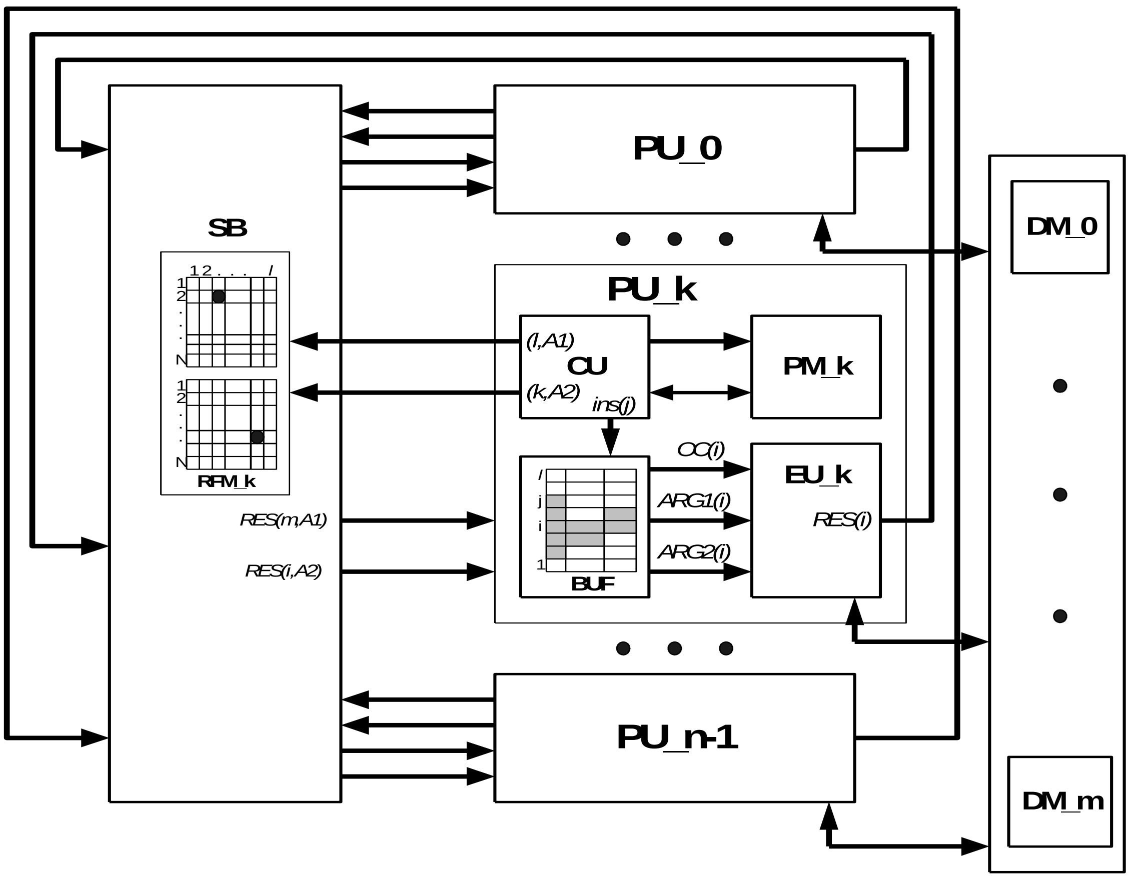


Figure 2.1 — Conceptual processor scheme

Let’s suppose that the system contains four processor units (cells). Let’s place the command sequence being considered into the PM processor units beginning from a zero PU as it is shown in the figure 2.2. Let's confront an individual tag to every triad (tag addresses and values are given for the zero PU).

Address	Tag	PM_0	PM_1	PM_2	PM_3
A+0	t+0	RD a	RD b	RD c	+ 0,1
A+4	t+4	RD e	- 0,2	RD f	* 3,4
A+8	t+8	/ 5,6	+ 7,8	WR 9,a	

Figure 2.2 – Program placement in the program memory of processor units

To execute this program simultaneously with the help of the processor functional units, the processor's structure is mentioned in the figure 2.1, we need to:

have a coordinated (coherent) selection of commands in one line;

organize a dynamic formation of the values of tags in order to take into account quantity of functional units that realize command selection and their relative numbers.

2.2 Program execution

The command word selection is initialized by transmitting all the control to the linear section. It begins from the first command of the linear section from the program memory (PM) and goes up to the last command. At that, the following actions are executed.

The tag value is dynamically formed for every selected command. It is equal to the sum of the last used tag value when selecting commands and functional units. The tag reference value is ascertained as the sum of the initial tag value in this linear section and the reference number. The tag value varies cyclically when selecting commands. Its maximum value is ascertained by buffer capacity.

The command is recorded into the free line of the buffer. If the command contains the argument value directly in the command word, for example, the address of the variable that is in the data memory (DM), then this value is also rerecorded into some corresponding field of this buffer line. After the “half-finished” command is recorded into the buffer line, the control unit (CU) starts to select the following command.

The process stops after the selection (specially marked) of the last command of the linear section; this last command is selected and executed according to standard procedure. It is necessary to note that any command can be the last one but not only the command that forms the initial address of a new linear section. The command forming the initial address of the following section can take any place in the section being executed. The formed address is sent to every functional unit which determines its address of control transmitting to the following section and realizes this transmission (selection is renewed) after we have the signal of the last command selection of the current linear section. The selection also finishes, when the buffer is filled and it is renewed, when the buffer is free.

In the buffer lines the argument fields are organized as two arrays with associative addressing. The associative address is a tag of a result being demanded. By this address we realize the value record of input result when it is equal to the tag result.

The command is in the buffer until all the necessary results come into the buffer and until all the commands using its result are recorded into the buffers. If these conditions are met at least for one command, the command execution is initialized. At this, if there are few of ready commands to be executed, then the command selected first of all is transmitted to be executed.

The executive unit (EU) executes a command and brings its result into the buffer with the tag equal to the tag of the executed command. This process finishes or stops, when the buffer has no commands ready to be executed.

It should be noted that while executing commands the cells act with no coordination and work independently. The cell executes commands but it is unknown who this result consumes. There is no determined sequence of executing commands. It is determined by data and command flows.

2.3 Architecture features

1. The multicellular architecture differs from the von-Neumann model by direct indication of informational connections between operations and consequently any requirement for ordered arrangement of operation description in the program is annulled.

This disorder makes all the methods (superscalararity, broad command word, super-pipeline, forecasting of transitions, etc.) unnecessary; these methods ensure operation speed but at the same time dramatically complicate the processor as well as development software (compilers, debuggers) design and increase their cost.

2. It is different from the well known non-von Neumann architectures by means of the sequential fetching which realizes imperative programming languages as well as by dynamically generated tags but not command addresses of indicating informational connections. Any command is executed at the “data readiness” and “its output users’ readiness”.

3. The cell instruction set is based upon some intermediate presentation of a compiled program after the syntax analysis (triads) and actually it is a sort of hardware realization of input programming language. It minimizes the labour costs to create compilers due to the fact that blocks of machine-oriented optimization and paralleling disappear as well as command generating block dramatically decreases. The notion "assembler programming" disappears as the processor language is not visible and thus it is “not programmable”. The software becomes really hardware-independent.

4. If necessary, the disordered triads ensure an individual object code for every processor after every compilation. This fact as well as the closure of the triad subsets make it dramatically impossible to unauthorizedly, secretly and from without interfere into the system software.

5. The system code is individual and unprivileged users use only the high-level language for programming which permits to create a new and effective toolkit against viruses.

6. The triads make it possible to read and execute several commands simultaneously and without analysis of their execution consequences or informational connecting, i.e. they ensure “natural” realization of parallelism. It is initially conditioned by mechanisms of command execution and execution type. In the multicellular processor there is no hardware identifying informational connections between chosen operations (commands) and distributing them by functional devices, i.e. there is no dynamic paralleling. There is also no static paralleling because the triad-type program describes informational connections, includes its linear structure but includes no indications of what and how it is possible to do in parallel.

7. The fully connected intellectual commutation environment functioning in “broadcasting” mode ensures effective realization of any type of tasks since it makes no topological restrictions as to intercellular data exchange.

All these architectural features ensure both universal character and effective scaling of the processor; the process of scaling practically increases performance directly proportionally to the number of cells.

8. The compiled program can be executed at any number of cells. At that it is possible to see dynamic change of their number that ensures the gradual degradation methodology of the processor to be realized at the failure of its cells. The processor can re-arrange itself and be functional when we have the commutation environment and at least one cell is operable.

This code independence upon used resources ensures the permanent self-adapting of the processor to a task flow and, when including new resources, ensures its self-repairing after failures.

9. The asynchronous and decentralized organization of the multicellular processor both at the system level – between the cells (when paralleling is realized) and

at the intracellular level – between the cell’s blocks (when commands are realized) additionally guarantees:

- minimum number of design objects and decrease of their complexity;
- decrease of crystal area because the device volume is less at the decentralized control than that at the centralized one;
- increase of performance and decrease of power consumption several times (see features MCP-1.1xx) because it realizes effective computing process.
- use of the individual synchronization system for every cell, when, in perspective, realizing on the one chip of tens and hundreds of cells.

As a result, we have a well-structured modular system that permits us to considerably simplify the processor as well as consequently decrease labor costs and improve project quality.

CONCLUSION

The proposed architecture is multi-faceted and qualitatively compiled which permits to position it as a principally new and high-efficient post-Neumann direction in development of the microprocessor hardware.

Appendix 1

PROGRAMMABLE SYSTEMS TAXONOMY

1. Processor architectures construction and taxonomy

1.1. Task formulation

It is known that the base of absolutely any taxonomy system is abstraction. It permits to create a model for the system being taxonomized and take only the most necessary, i.e. taxonomy parameters, without any secondary details. We choose these parameters relying on our taxonomy purposes and requirements. So, in our case it is obligatory to abstract from any realization features. Among the known computing systems' taxonomies [3] there is Flynn's taxonomy [4] that partially satisfies our obligation. It regards the computing system at the most possible, contensive and conceptual level of abstraction. At this level the system consists of command units (control units), executive units and memory units connected by command and data flows. The taxonomy parameters are the command and data flows' quantitative assessments.

There are such flows that connect the systemic units as a whole, take place in absolutely all the known systems and reflect the computing systems' internal nature. The others are created by realization, i.e. by those technical decisions made to achieve certain goals when creating the concrete systems.

The first certainly include the command flows that are formed by the control units and enter into the executive units. If we take some data flows circulating in the differently organized systems, the result flows being formed by the executive units are common for all the systems. The data flows being used in Flynn's taxonomy and "called" when executing commands, i.e. the flows outgoing from the memory to be processed, take place only in some of the systems.

For example, the data flow machine has no data flows coming from the memory and being used to execute some other command. All the data necessary to execute command go together with the command as a component of the command word.

It is evident that at the heart of the system's models abstracted from the realization features as well as at the heart of their taxonomy using the command and data flows' assessment as the taxonomy parameters should be only those flows that are present in absolutely all the computing systems.

This above-mentioned conceptual difference between the data flow machine architecture and Von Neumann's traditional architecture is connected with the fact that no called data flows are present. The executed commands' results are immediately recorded into the argument fields of those commands which use them. The commands are executed when all the necessary arguments are received. Thus, the data flow formed by the executive unit directly determines what consequence these commands are executed in.

The mentioned determination plays a key role to separately single out the kind of the data flow machines. If we only use the command and data flows' quantitative assessments without regard to the existing dependencies between the flows, it may scarcely decide the task of systematizing these (non-traditional) architectures.

What is more, the approach that forms this assessment and is used in Flynn’s taxonomy does not permit to clearly taxonomize even the systems constructed on the basis of Von Neumann’s traditional decisions (for example, vector pipeline machines).

So, any quantitative assessment may be correct when using a measuring instrument independent of this or that measuring object and when measurement conditions are equal. We can formulate these two requirements in the following way:

- at the accepted abstraction level it is necessary to unambiguously define the terms “single” and “multiple” actual with respect to any flows (both command and data) and any architectures;
- all the computing systems being taxonomized should be regarded at the same abstraction level.

Neither the first nor the second requirement is fulfilled in Flynn’s taxonomy. No quantitative indexes are determined. Their values are ascertained post factum when comparing the systems being taxonomized and reference patterns for every class.

As a result, to describe these two radically different phenomena, they use the same characteristic “multiple data flow” both in the MIMD (multiprocessor system) reference patterns and the SIMD (matrix processor) ones. There are independent data flow multitudes in the multiprocessor system and vector (multicomponent) data flows in the matrix processor.

It is also necessary to note that the SIMD reference pattern does not correspond to the conceptual level. The matrix processor functioning model providing for a simultaneous command execution with respect to the vector data flow reflects one of the possible ways to realize these vector commands. This model is not applicable for any other commands, for example, the scalar ones. It is consequently inapplicable at the conceptual level where the command is generally considered.

Thus, if we want to construct this multitude of the computing systems’ models abstracted from the realization features and taxonomize this multitude, we need to:

1. regard the result flows (command execution results) that are formed by the executive units as the data flows;
2. introduce some functional dependence between the flows as the taxonomy parameter;
3. unambiguously define the terms “single” and “multiple” actual with respect to any flows (both command and data) and any architectures.

When the mentioned requirements are fulfilled, it leads to a new taxonomy system where these processor architectures are considered at the conceptual level.

1.2. Basic notions and definitions

At the accepted level of abstraction any processor will consist of the control units and executive units integrally connected by the command and data flows.

The control unit (CU) means a command flow source. Every command is an indivisible and integral informational message that unambiguously determines some indivisible and integral action sequence of the receiver (receivers). No command may be given and fulfilled partially.

The executive unit (EU) means an intermediate (unfixed) data and final (fixed) result source. The EU completely or partially accepts this or that command flow and executes that. When executing the EU changes the processor's state and/or generates the data flow (informational messages). When the command is executed, we can see the changed processor (fixed change) state and/or the changed environment.

The command execution sequence is the processor state's sequential change; it is usually accompanied with some sequential environmental change. In which case, the command execution process is analogous to the transient. The parameter is truly ascertained (processor state) only when the transient is finished.

All the sources are independent, i.e. their simultaneous processes are connected in no way. But this independence excludes no coordinated actions. The processes themselves are independent but their actions (for example, their beginning and ending) may be coordinated.

Any source forms only one (single) flow that can enter into several receivers.

Two and more sources of the same type (CU or EU) form a multitude of sources. These sources may form a multitude of flows that may be used as a group of single flows or as some integral multiple flow. Like the single one, it can also enter into several receivers.

The "flow" is a key notion. The flow means a direct informational connection between the flow source and its receiver; this connection unites them into one whole. The flow elements exist only when transmitting. When in the receiver, they are either used by that or disappear. If we need to retain and further use the flow element, then it should be alienated, i.e. taken from the flow and put outside the source and receiver into the environment (in this case, into the memory). It is obvious that this alienation breaks the immediate connection (flow) between the source and the receiver. This connection becomes indirect.

It is also to note that the connection is informational. This connection exists only when the content of the ingoing flow's elements is considerably important for the receiver.

With regard to these notions we can give the following definitions concerning the processor architecture and the computing system.

The processor architecture is an abstract structure that reflects processor organization and it is presented as an oriented graph consisting of many tops and including at least one CU and at least one EU as well as many arcs created by the command flows coming from the CU and by the data flows coming from the EU, and in this case:

- an appropriate non-oriented graph is connected;
- every outgoing (coming from) command flow is an ingoing (entering) one for at least one EU;
- every EU has at least one ingoing command flow.

The computing system architecture is an abstract structure that reflects systemic organization and it is created by a multitude of unconnected oriented graphs; each of the graphs represents the processor architecture.

Any data exchange between the processors in the computing system may be consequently made only by the data flow elements' alienation.

Let's take Von Neumann's traditional processor. Its architecture is graphically presented in the figure 1. The architecture consists of the two tops (CU and EU) united by the command flow.

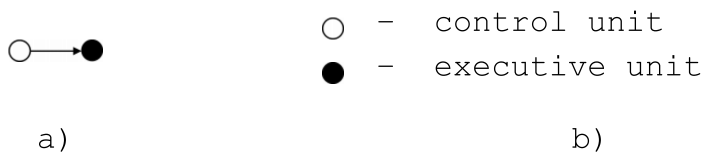


Fig. 1 Von Neumann's traditional processor (graphically):
a) graphic presentation; b) conventional signs

It is necessary to note that any other indirect informational connection except the connection concerning the command flow is radically impossible in Von Neumann's processor model where the description and execution are made only as some ordered sequential operations changing the environment state (memory). This method initially implies the alienation of every received result and correspondingly the data flows are absent in this architecture.

Having increased the notation suggested by M. Flynn and using capital letters to describe the sources and small ones to indicate the flows, this architecture may be described as SISD(si). After the sources described, in brackets they indicate the ingoing flows, i.e. the flows that determine the source functioning. Thus, Von Neumann's traditional processor has the architecture with one command source (SI) and one data source (SD); its functioning is determined by the single command flow SD(si).

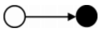
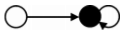
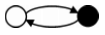
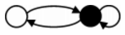
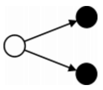
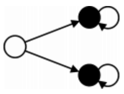
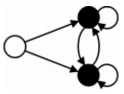
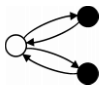
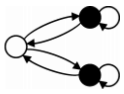
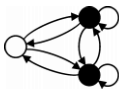
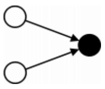
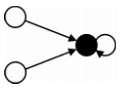
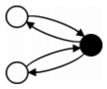
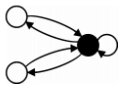
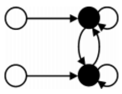
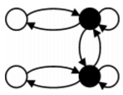
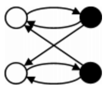
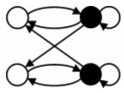
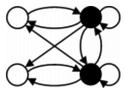
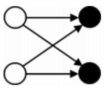
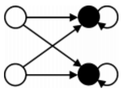
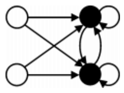
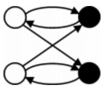
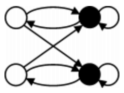
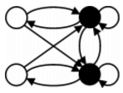
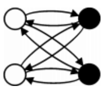
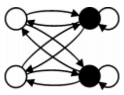
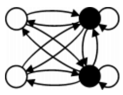
1.3. Processor architectures taxonomy

Using the "single" and "multiple" sources and flows to make the quantitative assessment limits any other possible variants of dependencies. The two-component description of the architecture implies its homogeneity and symmetry. That is, when some sources are multiple, the existing dependence covers the whole multitude (homogeneity), and when the command and data flows are multiple, their dimensions are regarded as equal.

Considering Von Neumann's architecture as a starting point we can, relying on the developmental principle, successively complicating the dependencies between the flows and excluding any unconnected structures, construct the whole set of possible processor architectures. This set is divided into two major classes:

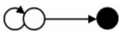
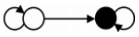
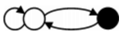
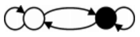
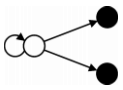
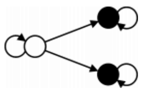
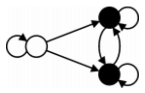
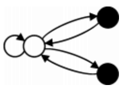
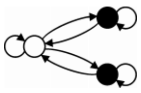
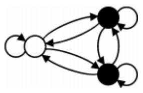
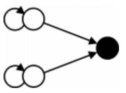
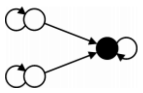
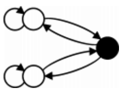
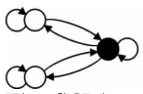
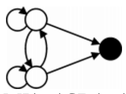
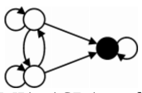
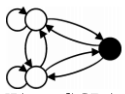
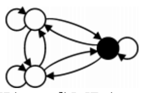
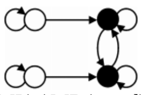
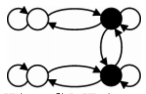
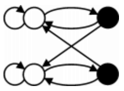
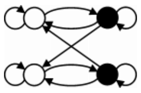
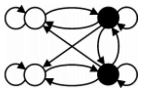
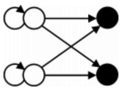
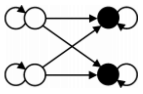
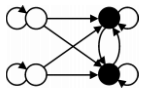
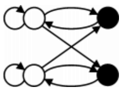
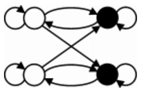
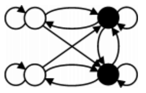
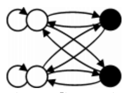
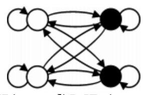
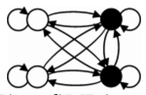
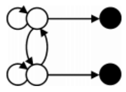
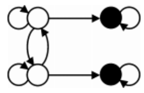
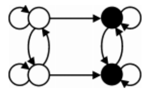
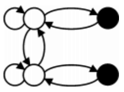
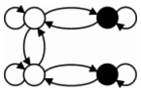
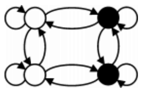
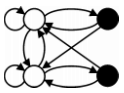
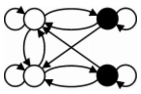
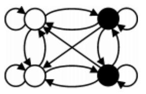
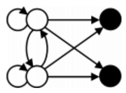
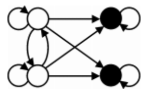
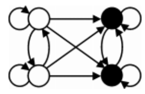
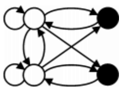
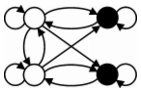
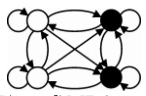
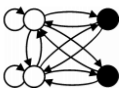
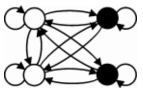
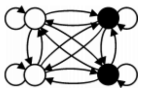
- the architectures having a stored program (context-free program) presented in the table 1;
- the architectures having a stored algorithm (context-sensitive program) presented in the table 2.

Table 1. Basic (homogeneous) models of the processor architectures with stored program

	Von Neumann architectures	**Non Von Neumann architectures**				
SISD	 SISD(si)	 SISD(si,sd)		 SI(sd)SD(si)	 SI(sd)SD(si,sd)	
SIMD	 SIMD(si)	 SIMD(si,sd)	 SIMD(si,md)	 SI(md)MD(si)	 SI(md)MD(si,sd)	 SI(md)MD(si,md)
MISD	 MISD(mi)	 MISD(mi,sd)		 MI(sd)SD(mi)	 MI(sd)SD(mi,sd)	
MIMD			 MIMD(si,md)			 MI(sd)MD(si,md)
				 MI(md)MD(si)	 MI(md)MD(si, sd)	 MI(md)MD(si,md)
	 MIMD(mi)	 MIMD(mi,sd)	 MIMD(mi,md)	 MI(sd)MD(mi)	 MI(sd)MD(mi,sd)	 MI(sd)MD(mi,md)
				 MI(md)MD(mi)	 MI(md)MD(mi,sd)	 MI(md)MD(mi,md)

These two classes radically differ from each other because the command flows are dependent. In the architectures having a stored program the command flow may only depend upon the data flow. In the architectures having a stored algorithm it always depends upon itself or upon itself and the data flow.

Table 2. Basic (homogeneous) models of the processor architectures with stored algorithm

	Non Von Neumann architectures					
SISD	 SI(si)SD(si)	 SI(si)SD(si,sd)		 SI(si,sd)SD(si)	 SI(si,sd)SD(si,sd)	
SIMD	 SI(si)MD(si)	 SI(si)MD(si,sd)	 SI(si)MD(si,md)	 SI(si,md)MD(si)	 SI(si,md)MD(si,sd)	 SI(si,md)MD(si,md)
MISD	 MI(si)SD(mi)	 MI(si)SD(mi,sd)		 MI(si,sd)SD(mi)	 MI(si,sd)SD(mi,sd)	
	 MI(mi)SD(mi)	 MI(mi)SD(mi,sd)		 MI(mi,sd)SD(mi)	 MI(mi,sd)MD(mi,sd)	
MIMD			 MI(si)MD(si,md)			 MI(si,sd)MD(si,md)
				 MI(si,md)MD(si)	 MI(si,md)MD(si,sd)	 MI(si,md)MD(si,md)
	 MI(si)MD(mi)	 MI(si)MD(mi,sd)	 MI(si)MD(mi,md)	 MI(si,sd)MD(mi)	 MI(si,sd)MD(mi,sd)	 MI(si,sd)MD(mi,md)
				 MI(si,md)MD(mi)	 MI(si,md)MD(mi,sd)	 MI(si,md)MD(mi,md)
	 MI(mi)MD(si)	 MI(mi)MD(si,sd)	 MI(mi)MD(si,md)	 MI(mi,sd)MD(si)	 MI(mi,sd)MD(si,sd)	 MI(mi,sd)MD(si,md)
				 MI(mi,md)MD(si)	 MI(mi,md)MD(si,sd)	 MI(mi,md)MD(si,md)
	 MI(mi)MD(mi)	 MI(mi)MD(mi,sd)	 MI(mi)MD(mi,md)	 MI(mi,sd)MD(mi)	 MI(mi,sd)MD(mi,sd)	 MI(mi,sd)MD(mi,md)
				 MI(mi,md)MD(mi)	 MI(mi,md)MD(mi,sd)	 MI(mi,md)MD(mi,md)

The two tables have some empty fields. They reflect the systems' existence, i.e. unconnected structures; the systems were excluded when constructing the set.

Let's take the most known processors in this taxonomy.

When the data flow is considered as derived from the command flow, this unites the processors' architecture of such machines as, for example, vector pipeline and matrix, into one type with Von Neumann's traditional machine. Their further taxonomy should be made inside the type and relied on the realization features. That is, according to the operations (scalar, vector). Among the vector machines it is according to the vector operations' realization (vector pipeline, matrix).

One of the first steps in Von Neumann model's development included coprocessors. Actually they used an additional executive unit to, for example, execute the floating point operations in parallel. The processor's architecture is described in the following way – SIMD(si).

The multicore processor having several command sources and, for example, one executive unit has the MISD(mi) architecture. If there are several executive units and each of them can execute the commands coming from any source, then it will be the MIMD(mi) architecture.

The other models, for example, the data flow processor has the SISD(si, sd) architecture. i.e. one command flow source and one data flow source; the latter's flow depends upon the single ingoing command flow and upon its formed data flow. When the data flow depends upon itself, this is characteristic to execute commands "when ready". For example, the synputer [5-7] has the MIMD(si,md) formula.

It is to note that regardless this great variety of the existing processors, they for the most part do not conceptually differ from each other. And if the first table includes the realized types, for example, the four Von Neumann architectures, the data flow architectures, the second does not.

1.4. Stored algorithm conception (context-sensitive program)

The computing model in the high-level imperative languages is the execution of the operators' ordered sequence. Every operator is an indivisible and integral language structure describing the way the data are changed. Inside the operator the operations are executed in order by their ranging and bracketing, i.e. by indicating informational connections between these operations. No intermediate results inside the operator are alienated. They alienate only the result of the operator's execution. The language operator is consequently considered to be a command for the abstract machine that directly realizes some high-level language.

Let's regard the triad program in the intermediate representation achieved after the first compiling phase (syntax analysis). This representation is a machine-adapted form of the primary code written with the high-level language.

In this form the program is recorded as a numbered sequence of triads. This sequence is divided into sections. Every section corresponds to one program operator and contains the subset of triads that realizes this operator. The sequence the sections are recorded corresponds to the sequence the operators are recorded in the program. Every triad describes some operation with respect to the operands multitude by identifiers or references. The reference is a triad number; when the triad is executed, its result is used as an operand, i.e. the reference clearly specifies the informational connections between the operations. In this case, no triad results given by the reference are alienated.

The subset of the triads that realize the operator is closed as the informational exchange is indirect between the operators and is made through alienating any results. The subset has no reference to the other operators' triads and the other operators' triads do not refer to the triads of this subset. It should be noted that no triads make sense, when outside their subset, and they are no longer a command system. They certainly have their indivisibility, but there is no integrity and consequently no commands.

Any triad is meaningful and may be executed only in some certain context. That is, only when the triads are executed, it uses their results and those triads use its results, i.e. only inside its subset. Thus, the whole subset of triads has its integrity and indivisibility in whole, but not separately.

For comparison, the way this or that command is executed in the data flow processor or the Von Neumann one does not depend upon any context. All the commands of these processors have their indivisibility and integrity. Those commands that realize the operator have their integrity but there is no indivisibility.

It is well-known that the operations' initial set of any algorithmic language is originally fixed and finite. The set of the operators that may be theoretically constructed using these operations is potentially endless. Since any program has its multitude of operators and correspondingly its command system, then it is possible to say that the machine directly realizing the high-level language has no fixed command system, but has its operation set.

In general case the operator's triads may be placed on the section at random. The sequence they are executed is determined by informational connection, i.e. by preceding (forming operands) and following (using results) triads. Since this or that command is unambiguously determined by its operations' sequence, then it is possible to say that the command being executed is directly formed in the processor's working and its direct image depends upon itself, that is, upon its internal organization. The same as the command flow correspondingly depends upon itself.

The fact that the command flow depends upon itself is not the only difference of the stored algorithm architectures (context-sensitive program) from the stored program ones (context-free program). There are two more fundamental differences between these architectures.

The first includes the fact that any context-free program architecture's programs are countable. The program multitude of any context-sensitive program architecture has power of continuum.

The second is the context-sensitive program architectures may not be generally presented by the Turing machine.

2. Multicellular processor

At the present time they're creating the MI(mi)MD(si,md) multicellular processor and it is the first parallel processor of the stored algorithm architectures. It is created by the multitude of the independently but interactively functioning cells connected by the commutation environment.

The processor machine language represents the developed triad language.

No text of the triad language program is connected with the cells' number. By this "resource" independence, command disorder inside their context sequence (line section) and broadcast distribution of the results to all the cells “naturally” realize the parallelism (without parallelization) as well as effectively scale up the processor [8].

The mentioned particularities also make any methods (superscalarity, VLIW, superpipeline, speculative and predicate execution etc.) useless. In spite of the fact that these methods made the Von Neumann model more rapid in action, they dramatically complicated its organization. Having refused the methods and decentralized organization, we can substantially simplify the multicellular processor and correspondingly reduce our labor costs and improve project quality.

At the same time, in comparison with the Von Neumann traditional decisions, we improve the processor characteristics. The first assessments show increase in performance 2 – 4 times and decline in power consumption 10 – 15 times.

The multicellular processor is hardware realization of the high-level language. This realization is based upon the essence of language expressions rather than some exterior form and therefore:

- they maintain the whole software created in the traditional imperative high-level languages and increase its effectiveness;
- the compilation process from the high-level language is actually limited by the initial machine-free (“front-end”) phase which permits to dramatically reduce costs when developing the compilers;
- there is no such term as “assembler programming” because the processor language is not visual and that is why practically “non-programmable”;
- as form of existence the initial text becomes more effective for the programs and correspondingly the programs become open.

It is necessary to note that neither of the formerly made processor architectures integrally decides those problems that are decided by the multicellular processor. This permits to speak about a qualitatively new direction in the microprocessors’ construction.

3. Conclusion

The suggested programmable systems taxonomy permits to systematize the known processor architectures and purposefully choose way of their development.

This new development represents the creation of the stored algorithm architectures (context-sensitive program). These architectures decide many tasks of the computer industry that may not be principally decided in the Von Neumann traditional model or other known models. In this case the processor is simplified and its project price is brought down as well as its technical characteristics are improved. Also the decisions include the increased processor language level, decreased compiler as wells software costs, “naturally” realized parallelism using traditional imperative languages, effectively and dynamically reconfigurated processor.

Literature

- Tsilker B.Y., Orlov S.A. Organization of the Computers and the Systems. Spb.: Piter, 2004. 668p.
- Wiener N. Cybernetics or Control and Communication in the Animal and the Human Being – Moscow: Sovetskoe radio, 1968. 326p.
- Taxonomy of the computing system architectures. http://www.parallel.ru.
- Flynn M. Very high-speed computing systems // Proc. IEEE. 1966. N 54. P.1901-1909.

- Patent № 2179333 RU “Synergistic computing systems”.
- Patent № 2198422 RU “Asynchronous synergistic computing systems”.
- Streltsov N., Sparso J., Bokov S.,Kleberg S. The Synputer – A Novel MIMD Processor Targeting High Performance Low Power DSP Applications // International Signal Processing Conference, Dallas, 1-3 April, 2003. P. CD-ROM.
- Streltsov N.V. Parallelism realization in the multicellular processor // Transactions of the 3rd International scientific conference “Parallel Computations and Control Problems”. Moscow: The Institute of Control Sciences, 2-4 October, 2006. P.337-347.