MULTIFUNCTIONAL PIPELINE UNITS OF THE DATAFLOW ARCHITECTURE

Milan JELŠINA
Technical University of Košice, Faculty of Elektrotechnical Engineering and Informatics,
Department of Computers and Informatics, Letná 9, 042 00 Košice, Slovakia,
Phone: (+421 55) 602 25 76, E-mail: Milan.Jelsina@tuke.sk

SUMMARY

The paper deals with the parallel computer architecture of the MIMD paradigm being developed on the Department of Computers and Informatics [12], [13], [14], [15], [16], [17], [19], [21]. The main objective of the proposed architecture is, on the data flow (DF) principles to design a model of the parallel computer system for the high-performance of real-time problems programming [8], [9], [18], [22]. Parallel processing in that architecture model is executed by the dynamic multifunctional pipeline structure (19) of coordinating processors, which represents main components of the proposed data flow system.

The concepts of both the architecture layout and the dynamic pipeline implementation are presented in the paper. The contribution deals with the multifunctional pipeline components design of the DF architecture model. The high degree of the instruction-level parallelism [4], [5] is obtained in the proposed architecture. It is supposed that the hardware implementation of the architecture can be used as a specialised accelerator in problem-oriented computer systems with high requirements on the operation speed [9], [10], [11], [18], [19], [20], [23]. This research is supported by VEGA grand project No. 1/9027/2002.

Keywords: *parallel computer architecture, dataflow architecture, pipelining, multifunctional pipeline unit*

1. INTRODUCTION

Data flow (*DF*) computer architectures are based on a *DF* computing model by which any program instruction is ready for execution whenever competent operands become available [1], [2], [11]. The DF model represents a radical alternative to the von Neumann computing model since the execution is driven only by the availability of operands. *DF* computers have the potential for exploiting all the parallelism available in a program, which is obviously represented by the data flow graph (*DFG*). *DFG* can be used as a machine language in data flow system.

Each node in the *DFG* is an instruction (operator) and the arcs indicate the flow of result data, i.e. data tokens (*DT*), from producer to consumer instructions. All instructions are active, waiting for the input data (operands). When all input data of an instruction are available that can be executed, removing the tokens from its input arcs and placing tokens on its output arcs. That approach of the execution instructions represents the basic principle of the conventional DF architecture. Advanced architectures support augmenting the DF computation model with traditional mechanisms, such as multithreading, large-grain computation, data flow with complex machine operations, RISC approach, hybrid approach, etc. See [3], [4], [5].

The model of the data flow architecture under development at the Department of Computers and Informatics stems from the applications multithreading, hybrid principles and functional programming on the DFG level.

2. DATAFLOW PROGRAM EXECUTION

The high level functional language has been proposed for the representation data flow program [12], [15]. Formal description of the functional program in the general form can be found with its detailed description in [13] namely its properties and translation. The architecture of data flow system for the parallel execution of the functional program on that DFG level stems from the definition of the supercombinator-based target block code of the set of functions [12].

The functions definition and the main expression (program) for the shortness holds the form as follows:

$$\begin{aligned}
&d_1\ f_1\ p_{1_1}\ p_{1_2}\ \cdots\ p_{1_{n1}} = e_1\\
&d_2\ f_2\ p_{2_1}\ p_{2_2}\ \cdots\ p_{2_{n2}} = e_2\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&d_k\ f_k\ p_{k_1}\ p_{k_2}\ \cdots\ p_{k_{nk}} = e_k\\
&E\ f\ c_{i_1}\ c_{i_2}\ \cdots\ c_{i_{ni}}
\end{aligned} \tag{1}$$

where

d_i is i - th function body,
f_i - a function name (identifier of the i - th function),
$p_{i_1}\,p_{i_2}\ldots p_{i_{ni}}$ - pattern of parameters consisting of constructors and variables,
e_i - expression constituting the body of the function,
E - main expression of the program,
f - main function of the program,

$c_{i_1} c_{i_2} \dots c_{i_{ni}}$ - constant pattern submitted to the main function *f*.

The *DFG* of computed function is illustrated in Fig. 1. The block code (supercombinator) of the translated function *f* is fired by the synchronising token which is indicated like a trigger (*t*). By the *ACCEPT* operator the variable value (data token) x_i ($i = 1, 2, ..., n$) inputs into block code of the function $f\ (x_1,\ x_2,\ ...,\ x_n)$. The function is described by expression *E* built up from *DFG* operators. The *ACCEPT* operator also distributes the value of data token to the inputs of corresponding *DFG* operators. The *RETURN* operator backspaces the function value computed to the previous block code in which the matching vector of the function have been created by the *APPLY* operator (in the Fig. 1 is not shown). In such a way backtracking of calculated values of particular expression *E* of the program *P*, is executed. The *APPLY* operator allocates the matching vector (*MV*) in the frame store (FS), resp. structure store (*SS*) as given of the right part in the Fig. 1. Because supercombinators of the target *DFG* can be applied concurrently and the execution of the separated supercombinators can by executed also concurrently, the proposed approach of the program processing effectively supports the parallel execution of the *DFG* on the level of both functions and operators.

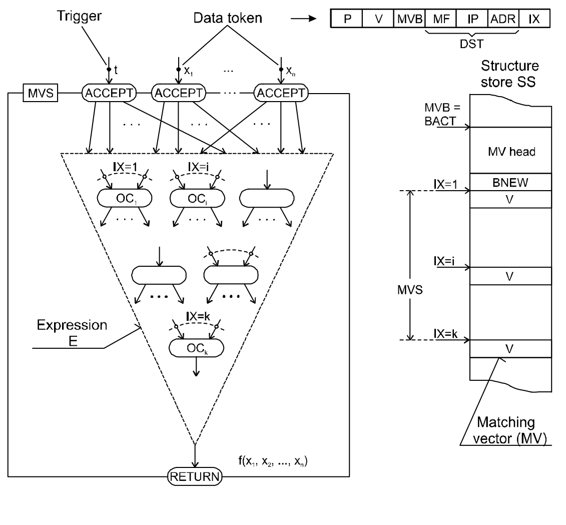


Fig. 1 Organization of function computing via data flow graph operators

More detail the functional program execution by its translation to the data flow program graph at a machine language level is introduced in [13], [15].

Definition of the data token in proposed architecture stems from the next description. The incoming input operand of the function operator (instructions) can initiate its execution when firing rule is fulfilled. The corresponding data token (*DT*) has the following format:

$$\langle DT \rangle ::= \langle P \rangle \langle D \rangle \langle MVB \rangle \langle DST \rangle \langle [IX] \rangle \qquad (2)$$

where

P is an operand priority,

D - input data (data, pointer, trigger) of the operator, $\langle D \rangle ::= \langle T, V \rangle$, where *T* is a data type, *V* - data value ($V = Val\,(D)$),

MVB - matching vector (*MV*) base (address), *MV* being placed in the structure store (*SS*) is intended for input operands matching of function operators,

DST - data token destination, $\langle DST \rangle ::= \langle MF \rangle \langle IP \rangle \ \langle ADR \rangle$, where *MF* is a matching function ($MF \in \{B, M\}$, *B* - bypass, *M* - matching), *IP* is the operator input port of the DT (*L* - left, *R* - right) and *ADR* is an instruction address of the *SS*,

IX - position index of the second operand data in the matching vector located.

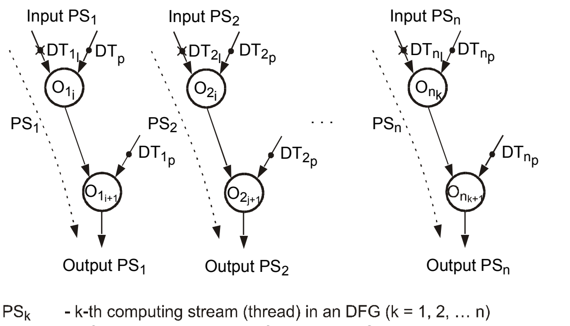
The format of a data flow instruction (*DFI*) being placed in the instruction store (*IS*) by the address (*ADR*) to perform binary operation is as follows:

$$\langle DFI \rangle ::= \langle OC \rangle \langle LI \rangle \langle \{DST,\ [IX]\ \}^n \rangle \qquad (3)$$

where *OC* is operator code, *LI* - literal, which defines the number of following operators, *DST* – destination of the next operator for which the output operation result is input operand (destination operand).

Multithreading execution of the DFG represents the parallel computing process in the superscalar computer environment. Fig. 2 shows *n* computing stream (thread) PS, which can be performed in parallel. That approach creates the supposition for the instruction parallelism (*IPL* – *I*nstruction *L*evel *P*arallelism). Computing streams PS_1, PS_2, ..., PS_n are to be executed, when input operands (tokens DT) of adequate DFG operators (instruction) are available. It means, that for the some *i*nput token DT_{m_i} the corresponding input token DT_{p_i} (the partner of the operand) must be available. That process is holding name as matching operands.

Having illustrated how elementary styles of the proposed data flow can by supported, the following explanation concerns to the implementation of the architectural DF model.



PS_k - k-th computing stream (thread) in an DFG ($k = 1, 2, \dots n$)
DT_l, DT_p- left and right data token of an operator O_m in the k-th computing stream (thread) of the corresponding DFG ($m \in \{1_i \dots 2_j \dots n_k \dots\}$)

Fig. 2 Multithreading execution of the DFG

3. IMPLEMENTATION OF THE DATAFLOW ARCHITECTURE MODEL

The proposed data flow architecture uses advanced pipelining design by which the parallel processing of DFG is executed.

The main component of the proposed dataflow architectural model is shown in on the Fig. 3. Structural organisation of the DF computer architecture model consists of the five base components [14], [16], [19]:

- *CP - Coordinating Processors* are designated for control, co-ordination and processing instruction of a DF program, when any operand is being come on the CP´s input port CP.DI from own output port CP.DO, from the output port CP.DO of other CP by means of the interconnection network, from the data queue unit or from frame store. Structural organisation of CP is designed as a *dynamic multifunction system*, which consists of five pipeline segments: LOAD, FETCH, OPERATE, MATCHING and COPY. State of the operations flow in the CP is indicated by of value setting of the signal CP_free = 0.
- *DQU - Data Queue Unit*, is designated for the storage of the DT, representation operands, which are waiting for matching during of the DF program execution.
- *IS - Instruction Store*, is the memory of the program DF instruction (DFI) in form of the relevant DFG.
- *FS - Frame Store*, is the memory of matching vectors (MV). By its items content CP determine the operands presence for corresponding operations flow. Operator (node) in DFG defines this flow. The format of the MV item in FS holds the form as follows: ⟨*FS*⟩ ::= ⟨*AF*⟩ ⟨*V*⟩, where *AF* (affiliation flag) is an attribute of the operand presence and *V* is a value of the given DT.
- IN - Interconnection Network, is designed for switching of individual CP's together for their data transmission.

The base component of the *DF* architecture is represented by the coordinating processor (*CP*). It is responsible mainly for the organisation and coordination of program instruction processing, which is enforced by the firing rule: an instruction is executable, if one operand of any instruction is incoming from the *DQU* or *CP* output port and the second one is available in the frame store (*FS*). In the opposite case, i. e. second operand is not found in the *FS*, the incoming operand is put to the *SS* (operand matching process).

The *CP* architecture is presented by the pipeline system built up like the multifunctional pipeline unit. Because the *CP* can traverse across its states in the various ordering, the dynamic pipeline is used. A dynamic pipeline can be reconfigured to perform variable functions at a different time. Those allow feedforward and feedback connection of processing stages of the pipeline unit that corresponds to the *CP* state diagram in Fig. 4.

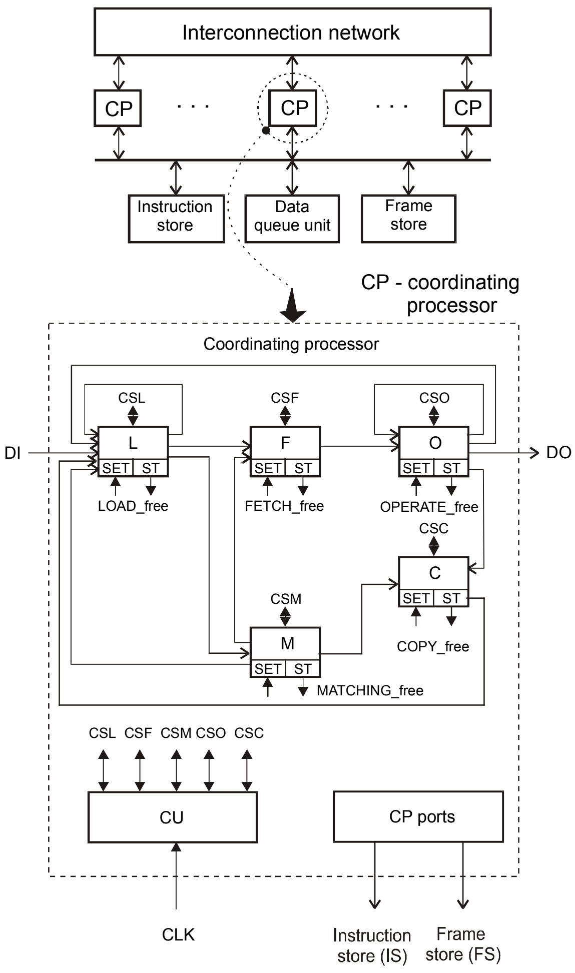


Fig. 3 Component of the dataflow architecture model

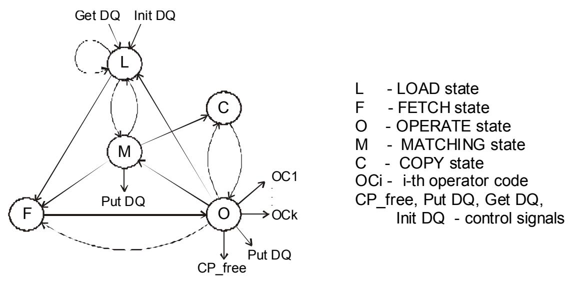


Fig. 4 State diagram of coordinating processor pipeline stages

CP communicates with another DF architecture components by their input/output ports.

The structure of CP input and output ports and their connection to other components of the proposed architecture model (*IS*, *FS*, *DQU*, IN), which take part in tokens processing at the execution of the DFG´s operators, are introduced in the Fig. 5.

All main communications between the CP and its external components, at the multipipeline execution of the DF program (DFG), by this components are realised.

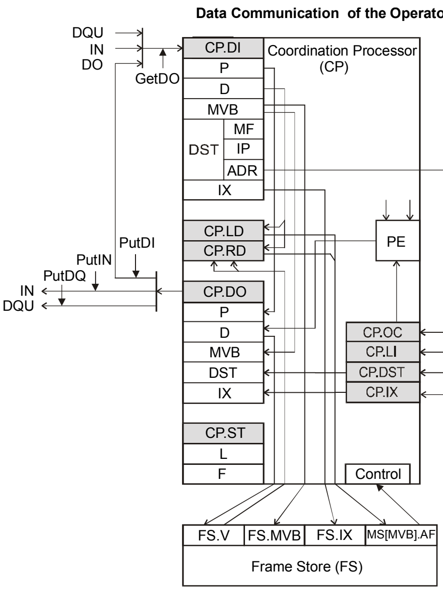


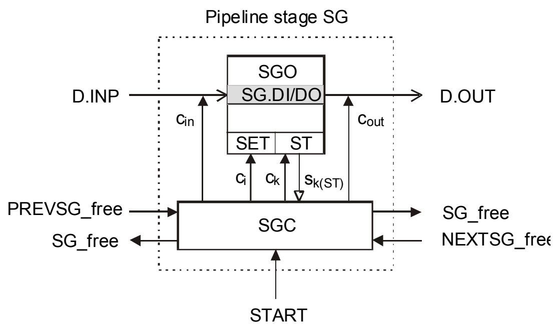
Fig. 5 The structure of CP input and output ports

4. COORDINATING PROCESSOR ARCHITECTURE

At the *DFG operators* (instructions) execution the *CP* can traverse across *L* (*LOAD*), *F* (*FETCH*), *O* (*OPERATE*), *M* (*MATCHING*), *or C* (*COPY*) stages (states). See Fig. 4. The *CP* is introduced to the *LOAD* state, if it is waiting or getting an operand from the *DQU*, from the data output port *CP.DO.D* or from the *IN*, by the *Get DQ* control signal. The signal *Init DQ* control provides initialisation of the DF system at its start). In the *FETCH* state the *CP* activates *IS*, carries out the fetch of the instruction operator code *OC* and handles the next step of the instruction execution. In the *OPERATE* state the *CP* organises the instruction processing in von Neumann manner. The result of the instruction execution representing the input operand of the next instruction is examined at the *MATCHING* state. If the incoming result operand *D* does not find the partner operand on the input port of the next *OCi* operator, no operand matching will be organised. That fact is indicated by the affiliation flag (*AF*) in the corresponding position of the matching vector (*MV*). The opposite case, when the partner operand is available (it is stored in the corresponding item of the SS´s matching vector) the requested operation is executed by the processor element unit (*PEU*). The operation result is available for the next operator execution, if *CP* is free (*CP.ST* = *CP*_free). If *CP* is busy then the operator execution is provided by another free *CP*, which is accessible through the interconnection network (*IN*) and its control (*INC*). In the case, where no *CP* is free (*CP. ST* = $\rceil$ *CP*_free) the output operand (result) is put to the data queue unit (*DQU*) by the *Put DQ* control signal. When the matching is not successful the result of the operation is put on the selected items of the corresponding matching vector by the matching function control signal *MF* = *M*. By the *COPY* state the *OPERATE* state can be decomposed to multiple steps when the operator processing consists of too much inner loops for its execution (the case of some duplication operators).

The pipeline conception of the coordinating processor architecture supports the multifunctional execution of the DF program. Any computing stream (thread) PS_i of a DF program is executed by multifunctional organisation of the structure of pipeline stages *SG* (*SG* – *S*ta*G*e), e.i., *SG* ∈ {*L*, *M*, *F*, *O*, *C*}, those interpret identically named states *ST* (*ST* – *ST*ate), e.i., *ST* ∈ {*L*, *M*, *F*, *O*, *C*} of the coordinating processor, at the logical level (Fig. 3).

CP consists of the both pipeline stages and control units. The control unit (CU) generates control signals (microcomand), which initialise the execution of microoperations in the coordinating processor adequate stages L, F, M, O, C.



SGO - pipeline stage operation
SGC - pipeline stage control
SG.DI/DO - data input and output stage
START - beginning initialization of stage

Fig. 6 General conception of coordinating processor pipeline stages

On the Fig. 6 is presented the general conception of coordinating processor pipeline stages. Every SG consists of both the *operation* (SGO) and *control* (SGC) part. SGC generates control/state signals denoted as LOAD_free, FETCH_free, OPERATE_free, MATCHING_free and COPY_free to indicating accessibility (busy condition) the specific stages. Values of the control/state signals are set at the SGO´s input ports SET and ST. Specialised state signals indicate the testing result of the branch microoperations on the SGO´s output port ST. Apart from those signals, which specify the SG function, CU generates also synchronising signals intended for the clocking of the execution microoperations of coordinating processor stages. Control/state, state and synchronisation signals are distributed in particular SGC.

SGO processes its input data token *DT* = *D.IN* and executes the microoperations in the adequate pipeline stage. Result of that is the setting of its output data token *DT* = *D.OUT*, e.i.:

$$D.\ OUT := f_{SG}(D.\ IN)$$

where

f_{SG} is function pipeline stage SG, which interpret state *ST* ∈ {*L*, *F*, *M*, *O*, *C*}of the CP, when operator of the DFG is executed,

D.IN – input token of the stage *SG* (operand of corresponding operator of the instruction stream PS_i),

D.OUT–output token of the stage *SG* (operation result, which have executed by the operator of the instruction stream PS_i).

Tokens processing in the *SG* is performed on the base of the microprogram control by microcomands (microinstructions), those generates its control part *SGC*.

Each CP´s pipeline stage is controlled by the microprogram. Microprogram control is specified by one of next forms of microcomands:

$$c_i : M / s_j \qquad (4)$$

$$\mathrm{ck} : \mathrm{P} / \mathrm{sn}\ ,\ \overline{P} / \mathrm{sm} \qquad (5)$$

Microinstruction in the form (4) means, that control signal c_i activates the microoperation *M* (for example, assignment of value on one of the variable *FETCH_free* : = 1), after the execution of that, state signal s_j will be set on the value of s_j = 1, on the base of what the *j*-th microcomand c_j will be started in the next step of control microprogram.

Microinstruction in the form (5) means, that control signal c_k activates the branch microoperation, by which is tested the truth of the predicate *P* (for example, evaluation of the component state FETCH, e.i. evaluation of the condition (predicate) *FETCH_free* = 1 or *FETCH_free* = 0). If *P* is truth (*P* = 1), corresponding state signal $\mathrm{s}_{\mathrm{k}(ST)}$ will be set as $\mathrm{s}_{\mathrm{k}(ST)} = s_n = 1$, in consequence of that, the *n*-th microcomand c_n, which initialises of the microoperation M_n execution in the next step of control microprogram. In the opposite case, if P is false (*P* = 0), the corresponding state signal $s_{k(ST)}$ will be set as $\mathrm{s}_{\mathrm{k}(ST)} = s_m = 1$, in consequence of that, the *m*-th microcomand c_m , which initialises of the microoperation M_m execution in the next step of control microprogram.

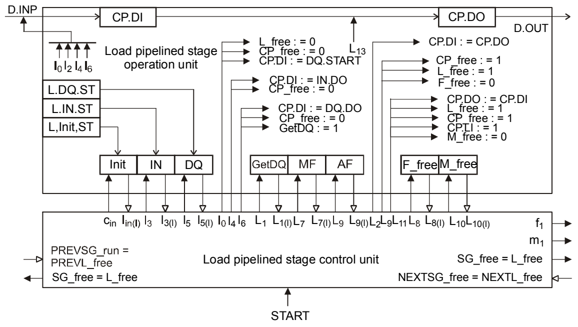


Fig. 7 Internal control, state, and synchronise signals for the LOAD stage

Particular pipeline stages of the CP being decomposed to the SGO and SGC parts, including their internal control, state, and synchronise signals are introduced on the Fig. 7 for the LOAD stage. The control design of other pipeline stages is created analogue.

5. MICROPROGRAM CONTROL OF COORDINATING PROCESSOR PIPELINE SEGMENTS

Microprogram control of CP at the DFG operators execution is performed by the microoperations, which are specified through the reference (4) and (5) for individual stages of the multifunctional organisation of the CP. In general, the initialisation of individual stages microoperations are executed by the control signals $st \in \{l_i, f_j, m_k, o_s, c_r\}$ in the next form:

ST st_0 : SG_free : = 1 / st_1

ST st_{1w} : (PREVSG_*free* = 1) / st_{1w},
(PREVSG_*free* = 0) / st_2

ST st_2 : SG_free : = 0 / st_3

ST st_{3w} : (NEXTSG_free = 1 / st_{beg} ,
(NEXTSG_free = 0 / st_{3w}

ST st_{beg} :
SG_MICROOPERATION

ST st_{end} :

. . . .

ST st_{n} : NEXTSG_*free* : = 1 / st_{next}

where

st_p is a control signal, which initialises the continuous and next microoperation in stage *SG* at the execution of DFG´s operator,

p – control signal index, which presents the order of microoperatios to be executed (subscript *w* of the index *p* describes to the initialisation of the waiting operation, the subscripts *beg*, *end* of the index *p* – to the initialisation of both the start and the end of microoperations set {*SG_OPERATION*}, which represent to the function of the stage *SG_ST* and *next* subscript – to the initialisation of the execution of the microoperation set of the next stage of the CP),

SG_free – the indication of the *SG* stage activity (*SG_free* = 0 ⇒ the stage *SG* is busy, *SG_free* = 1 ⇒ the stage *SG* is idle),

NEXTSG_free – the indication of the next stage activity (*NEXTSG_free* = 0 ⇒ the next stage is busy, *NEXTSG_free* = 1 ⇒ the next stage is idle),

PREVSG_free – the indication of the previous stage activity (*PREVSG_free* = 0 ⇒ the next stage is busy, *PREVSG_free* = 1 ⇒ the next stage is idle).

Microprogram control of the multifunction structure of CP´s stages (*SG*) presents next microprogram, the LOAD part of which is introduced in more detail:

CP state		microinstruction
INIT	c_{in} :	(Init = 1) /l_0, (Init = 0) /s_{in}
LOAD	l_0 :	(*LOAD*_free : = 0 || CP_free : = 0 || CP.DI : = DQU.START) /l_7
LOAD	l_1 :	(GetDO = 1) /l_2, (GetDO = 0) /l_1
LOAD	l_2 :	(CP.DI : = CP.DO) /l_7
LOAD	l_3 :	(IN ≠ Empty) /l_4, (IN = Empty) /l_5
LOAD	l_4 :	(CP.DI : = IN.DO || CP_free : = 0) /l_7
LOAD	l_5 :	(DQU ≠ Empty) /l_6, (DQU = Empty) /l_3
LOAD	l_6 :	(CP.DI : = DQU.DO || CP_free: =0 || GetDO : = 1) /l_7
LOAD	l_7:	(CP.DI.MF = M) /l_{10}, (CP.DI.MF = B) /l_8
LOAD	l_8 :	(*FETCH*_free = 1) /l_9, *(FETCH*_free = 0) /l_8
LOAD	l_9 :	(CP_free : = 1 || *LOAD*_free : = 1 || *FETCH*_free : = 0) /f_l
LOAD	l_{10}:	(*MATCHING*_free = 1) /l_{11}, (*MATCHING*_free = 0) /l_{10}
LOAD	l_{11} :	(CP.DO : = CP.DI || *LOAD*_free : = 1|| || CP_free : = 1 || CP.LI : = 1 || || *MATCHING*_free : = 0) /m_l

FETCH part	(input signals: f_1, f_2, f_3, f_4, f_5, output signals: f_i, o_3,)
OPERATE part	(input signals: o_1, ... o_{22}, output signals: o_i, l_1, m_1, c_1,)
MATCHING part	(input signals: m_1, ... m_{17}, output signals: m_i, f_1, l_1, c_1,)
COPY part	(input signals: c_1, c_2, c_3, c_4, c_5, output signals: c_i, o_9,)

Microoperations to be performed in individual CP´s pipelined stages are introduced in [21] more detail.

Algorithms of control signals generation, which initialise the performance microoperations in the individual CP´s pipelined stages are presented by block diagram, as follows. For example, the block diagram of control signals generation in pipeline stages LOAD is introduced in the Fig. 8.

6. CONCLUSION

The contribution deals with the multifunctional pipeline components design of the data flow architecture model. The concepts of both the architecture layout and the multithread pipeline are presented in the proposed data flow system.

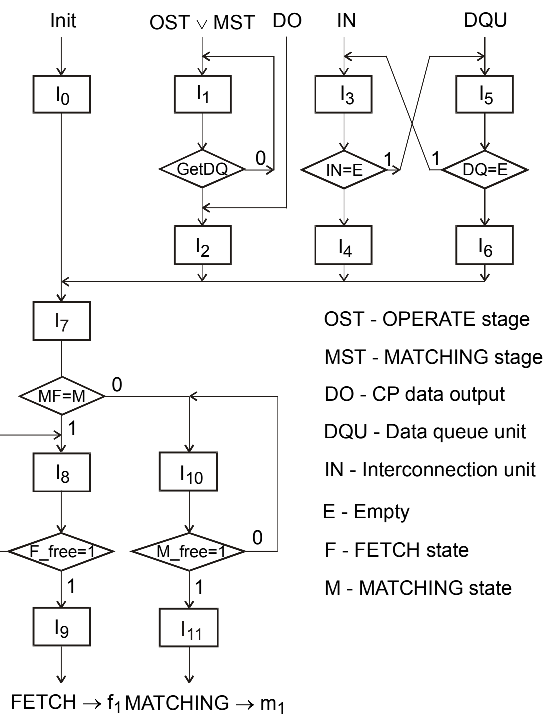


Fig. 8 The block diagram of control signals generation in pipeline stages LOAD

In proposed data flow architecture a basic outline of its structure organisation and microprogram control are emphasised. Principles of parallel processing by multifunctional pipeline structure of the coordinating processors as main components of the data flow architecture is introduced, too. The pipeline approach unifies execution of both single operators and sequence operators of a function to be processed by the *CP*. The pipeline execution of function operators and concurrency executions by the coordinating processors of the proposed data flow system enable the high-performance parallel processing. The used approach reflects the properties of data flow machine-oriented programming language, which is not, however, the topic of this contribution.

The proposed data flow system stems from the following features:

- programming environment on the base functional language on the DFG level,
- fine-grain parallelism of the data flow program graph,
- data flow architecture on the base of the dynamic pipeline coordinating processors by which the parallel processing of the functional program can be executed,
- applications multithreading,
- hybrid principles of the model architecture,
- direct operand matching of data flow graph operators.

It is supposed that hardware implementation of this architecture model, after its completion, can be used as a specialised accelerator in high-performance problem-oriented computer systems.

REFERENCES

[1] Denis, J. B., Misunas, D. P.: A Preliminary Architecture fo a Basic Data Flow Processor. In: Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston 1975, TX 126-132.

[2] Srini, V. P.: An Architectural Comparison of Dataflow System, Computer Vol. 19, 1986, 68-69.

[3] Arvind, Nikhil, R. S.: Executing a Program on the MIT Tagged-Token Dataflow Architecture, Lecture Notes in Computer Science 259, Springer Verlag Berlin, 1987, 1-29.

[4] Wall, D. W.: Limits of Instruction-Level Paralelism. Proc. Fourt. Int. Conf. Achitecture Support for Programming Lanquages and OS, 1991, 176-188.

[5] Butler, M. at all.: Single-Instruction-Stream Parallelism in Greater Then Two. Proc. 18. Annual Int.Symp. Computer Architectures, 1991, 276-289.

[6] Sakai, S. et all: Prototype Implementation of Highly Parallel Dataflow Machine EM-4. Proc. Int. Parallel Processing Symposium, 1991.

[7] Hwang, K.: Advanced Computer architecture. Mc-Graw Hill, Inc., New York 1993, 770 p.

[8] Abram, G., Treinish, L.: An Extended Data-flow Architecture for data Analysis and Visualisation. Proc. of Conf. on Visualisation ´95 (Cat. No. 95CB35835), Atlanta, Ga, USA 1995, 263 – 270.

[9] Jamil, T., Deshmukh, R. G.: Design of a Tokenless Architecture for Parallel Computations Using Associative Dataflow Processor, Proc. of Conf. on IEEE SOUTHEASTCON ´96, Briging Together Education, Science and Technology (Cat. No. 96CH35880), Tampa, FL, USA 1996, 649 - 656.

[10] Depta, J.: Data Flow Architecture for Advanced Process Control. Proc. of Conf. on Computer Software Structures Integrating AI/ KBS Systems in Process Control, A Postprint Volume from the IFAC Workshop, Lund (Sweden) 1996, 21 - 26.

[11] Šilc, J., Robič, B., Ungerer, T.: Asynchrony in Parallel Computing: From Dataflow to Multithreading, Parallel and Distributed Computing Practices, Vol. 1, 1998, 57-83.

[12] Jelšina, M., Kollár, J.: The Dataflow Implementation Environment for Functional Languages. Proc. Of Japan – Central Joint Workshop on Advanced Computing in Engineering, Pultusk (Poland), September 1994, pp.26 – 29.

[13] Kollár, J.: Implementation of Functional language at the Dataflow Computer System, Internal DCI report: DCI 13 – 95, Technical University Košice, 1995.

[14] Jelšina, M.: The Data flow Computer Architecture with Direct Operand Matching. Journal of Electrical Engineering, 46, 1995, 8, 279-285.

[15] Jelšina, M., Krahulík, P, Legnavský, M.: Dataflow architecture for the Parallel Implementation of the Functional Language, Proc. of International Conf. on Information, Communications and Signal Processing, IEEE Singapore Section, Singapore 1997, Vol. 3 of 3, 1452-1456.

[16] Jelšina, M., Legnavský, M.: Parallel Execution of the Program via Multifunctional Pipeline Units of the Data Flow System, Proc. of EC&I'98 Conference, FEI TU, Košice-Herľany 1998,93–99.

[17] Vokorokos, L.: Princípy architektúr počítačov riadených tokom údajov. Copycenter, Košice, 2002, p. 147. ISBN 80-7099-824-5.

[18] Straka, M.: A Contribution to Solving VR tasks in Powerful Parallel Environment. Written essay to doctoral/disertation exams in the field 25-21-9 Computer Tools and Systems. KPI FEI TU, Košice 2002, 66. (in Slovak)

[19] Jelšina, M.: Computer System Architecture. ELFA s.r.o., Košice 2002, 467.

[20] Sobota, B., Spišák, J., Straka, M., Grofčík, M.: Model of Production Process and Virtual Reality Technologies. Automatizace, 43, 2002, 3, 186-189, ISSN 0005-125X. (in Slovak)

[21] Jelšina, M., Vokorokos, L., Sobota, B.: Parallel Computer Architecture of the MIMD Paradigm. Proc. of the III. Internal Scientific Conference of the Faculty of Electrical Engineering and Informatics (III. ISC´2003), FEI TU, May 2003, 35-36.

[22] Hudák, Š., Korečko, Š.: Some Problems of the mFDT Environment Development. Proc. Of International Conference EMES´03, Oradea (Romania), May 2003 (in press).

[23] Jelšina, M. (editor): Parallel Computer System Architecture. Technical report, ELFA s.r.o., Košice 2003 (in preparation).

BIOGRAPHY

Milan Jelšina (Prof., Ing., CSc.), received the Ing (MSc) degree in 1959 in Automation and Telemechanics from LIIZT of Sankt Petersburg. Till 1964 he was employed as engineer at the Projekt Division of the Automation of Railway Trasport Company in Košice. Since 1964 he has been with the Technical University of Košice. He received the CSc (PhD) degree from the Institute of Information and Automation Theory at the Czechoslovak Academy of Science, Prague, in 1969, and was appointed Associative Professor and Full Professor for Technical Cybernetics and Electronic computers in 1972 and 1982, respectively. His research activity was concentrated on hybrid computer systems, computer means of robotic control systems, and microprogramming. Currently he has oriented towards parallel computer architectures and supercomputers, data flow systems and their application at the virtual reality problem solving and at creating of the environment for another advanced information technologies. At the Technical University of Košice he lectures under graduate and graduate courses oriented towards on Computers, Computer Architecture Systems, Parallel Computer Architectures and Supercomputers, and Data Flow Systems. He is preparing courses to be oriented for the Embedded Computer Systems, and Design of Computer Microarchitecture. The results of his pedagogical and scientific activity presents more than 400 science, developed, investigative, project works, monographs, textbooks, articles, and another activities in the area of informatics\ and information technology.